pARMS: A Package for Solving General Sparse Linear Systems on Parallel Computers*

Y. Saad[1] and M. Sosonkina[2]

[1] University of Minnesota, Minneapolis, MN 55455, USA,
saad@cs.umn.edu,
http://www.cs.umn.edu/~saad

[2] University of Minnesota, Duluth, MN 55812, USA,
masha@d.umn.edu,
http://www.d.umn.edu/~masha

Abstract. This paper presents an overview of pARMS, a package for solving sparse linear systems on parallel platforms. Preconditioners constitute the most important ingredient in the solution of linear systems arising from realistic scientific and engineering applications. The most common parallel preconditioners used for sparse linear systems adapt domain decomposition concepts to the more general framework of "distributed sparse linear systems". The parallel Algebraic Recursive Multilevel Solver (pARMS) is a recently developed package which integrates together variants from both Schwarz procedures and Schur complement-type techniques. This paper discusses a few of the main ideas and design issues of the package. A few details on the implementation of pARMS are provided.

1 Introduction

The past decade has seen excellent progress in the use of parallel computing technologies for dealing with real-life engineering applications. This development is due in great part to the maturation of parallel computer hardware and software. In particular, the emergence of standards for message passing languages such as the Message Passing Interface (MPI) [3], is probably the most significant factor in the promotion of parallel computing methodologies. It has become fairly inexpensive and easy to build small clusters of PC-based networks of workstations, making large and expensive supercomputers or massively parallel platforms much less cost-effective. These clusters of workstations as well as most of the current medium size machines are typically programmed in message passing, using the MPI communication library. This trend is likely to persist as many engineers and researchers in scientific areas are now familiar with this mode of programming.

Among the difficulties that remain when using parallel computers to solve industrial problems is the fact that applications are far more complicated to

* Work supported by NSF/ACI-0000443, NSF/INT-0003274, and by the Minnesota Supercomputer Institute

program on distributed memory computers than on traditional scalar computers or shared memory computers. As a result there is a pressing need to develop libraries for dealing with common problems in scientific computing, among which the solution of sparse linear systems is arguably one of the most important.

In this paper, we outline the main features and the design rationale of `pARMS`, a package for solving sparse linear systems on distributed memory parallel computers.

2 Distributed Sparse Linear Systems

The main paradigm used in `pARMS` is that of a distributed sparse linear system. To illustrate this concept, assume we have to solve the sparse linear system

$$Ax = b, \tag{1}$$

where A is a large and sparse nonsymmetric real matrix of size n. The simplest, and clearly not the best, way to distribute the system to processors is to assign blocks of approximately the same number of contiguous equations to the processors. Thus, if there are n equations and P processors, we would assign equations 1 to n/P to Processor 1, those from n/P +1 to processor 2, and so on. We refer to this as a distributed linear system – since the equations and right-hand side are (conformally) distributed among processors. This mapping of the equations to processors is straightforward and there are clearly more efficient ways of partitioning a general sparse linear system using graph partitioners. The question addressed here is to develop iterative techniques for solving such "distributed sparse linear systems".

It is typical in the case of finite element techniques to partition the finite element mesh by a graph partitioner and assign a cluster of elements which represent a physical subdomain to each processor. Each processor then assembles only the local equations associated with the elements assigned to it. In other instances, the linear system is only available in assembled form. Here also a graph partitioner is invoked to determine a good way to map pairs of equations-unknowns to processors. In either case, each processor will end up with a set of equations (rows of the linear system) and a vector of the variables associated with these rows. This natural way of distributing a sparse linear system has been adopted by most developers of software for distributed sparse linear systems (see, e.g., [4,1,6,2]) because it is closely tied to a physical viewpoint.

2.1 Local Representation

Figure 1 shows a 'physical domain' viewpoint of a distributed sparse linear system. Each point (node) belonging to a 'subdomain' is actually a pair representing an equation and an associated unknown. As is often done, we distinguish between three types of unknowns: (1) Interior variables are those that are coupled only with local variables by the equations; (2) Local interface variables are those coupled with non-local (external) variables as well as local variables; and (3) External interface variables are those variables in other processors which are coupled with local variables.

The local equations can be represented as shown in Figure 2. Note that these local equations do not correspond to contiguous equations in the original system. The matrix represented in the figure can be viewed as a reordered version of the equations associated with a local numbering of the equations/unknowns pairs.

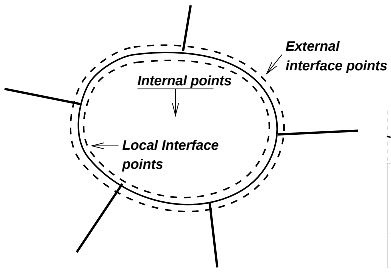

Fig. 1. A local view of a distributed sparse matrix.

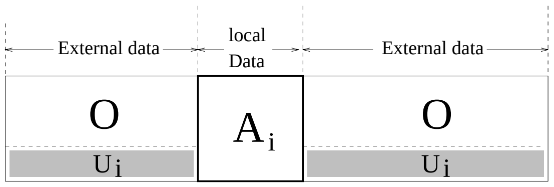

Fig. 2. Corresponding local matrix.

The rows of the matrix assigned to a given subdomain are split into two parts: a *local* matrix A_i which acts on the local variables and an *interface* matrix U_i which acts on remote variables. When performing a matrix-vector product, each processor acts locally on its data. It begins by performing a local matrix-vector product first, and then it receives the remote variables from adjacent processor(s) before completing its part of the matrix-vector product. In most data structures used for the parallel solution of distributed sparse linear systems, the boundary points are separated from the interior points. In addition, it is common for a number of reasons, to list interface nodes last after the interior nodes.

Each local vector of unknowns x_i is thus split in two parts: the subvector u_i of internal nodes followed by the subvector y_i of local interface variables. The right-hand side b_i is conformally split in the subvectors f_i and g_i,

$$x_i = \begin{pmatrix} u_i \\ y_i \end{pmatrix} \ ; \quad b_i = \begin{pmatrix} f_i \\ g_i \end{pmatrix} \ . \tag{2}$$

The local matrix A_i residing in processor i as defined above is block-partitioned according to this splitting, leading to

$$A_i = \begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix} \ . \tag{3}$$

With this, the local equations can be written as follows.

$$\begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix} \begin{pmatrix} u_i \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{j \in N_i} E_{ij} y_j \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix} \tag{4}$$

The submatrix $E_{ij}y_j$ accounts for the contribution to the local equation from the neighboring subdomain number j and N_i is the set of subdomains that are neighbors to subdomain i. The sum of these contributions, seen on the left side of of (4) is the result of multiplying a certain matrix by the external interface variables. It is clear that the result of this product will affect only the local interface variables as is indicated by the zero in the upper part of the second term in the left-hand side of (4). For practical implementations, the subvectors of external interface variables are grouped into one vector called $y_{i,ext}$ and the notation

$$\sum_{j \in N_i} E_{ij} y_j \equiv U_i y_{i,ext}$$

will denote the contributions from external variables to the local system (4). In effect this represents a local ordering of external variables to write these contributions in a compact matrix form. With this notation, the left-hand side of the (4) becomes

$$w_i = A_i x_i + U_{i,ext} y_{i,ext} \tag{5}$$

Note that w_i is also the local part the matrix-vector product Ax in which x is a vector which has the local vector components x_i, $i = 1, \ldots, s$. Matrix-vector product operations can be performed in three stages. First the x_i is multiplied by A_i. Then a communication step takes place in which the external data $y_{i,ext}$ is received. In effect this is an exchange operation, since each processor needs to receive the remote interface variables from other processors. In the third and last stage, the external data is multiplied by the local matrix U_i and the result is added to the current result $A_i x_i$.

Since communication is an important part of the matrix-vector product operation, it is useful in a preprocessing phase to gather the data structure representing the local part of the linear matrix as was just described. It is also important to form any additional data structures required to prepare for the communication that will take place during the solution phase. In particular, each processor needs to know (1) the processors with which it must communicate, (2) the list of interface points and (3) a break-up of this list into pieces of data that must be sent and received to/from the "neighboring processors".

2.2 Existing Software

Several packages for solving sparse linear systems by iterative methods on parallel computers were developed before. Most of these utilize a similar viewpoint of distributed sparse matrices and exploit the domain-decomposition viewpoint. A first version of the Parallel SPARSe LIBrary (PSPARSLIB) [15], a precursor of pARMS, was developed in 1993-1994 using P4, the ancestor of the current MPI, and then CMMD, the communication Library on Thinking Machine's Connection Machine 5. Later, other packages appeared starting in the mid-1990s. Among them we mention PETSc [1], Block-Solve [6], Aztec [4], and ParPre [2]. All these packages offer most of the standard "block", i.e., domain-based, preconditioners based on the additive Schwarz procedure.

Recently, another package called pARMS [7] was introduced. Like earlier packages, pARMS offers the traditional local "Schwarz" type preconditioners already available in PASPARSLIB. In addition, it also provides a truly algebraic multilevel strategy for preconditioning. pARMS can be viewed as a combination of the Algebraic Recursive Multilevel Solver (ARMS) presented in [14] on the one hand and an outgrowth of the PSPARSLIB package on the other. The primary accelerator used in pARMS is FGMRES, the flexible variant of GMRES [9], though a few other choices are also made available. FGMRES, is a right-preconditioned variant of GMRES which allows variations in the preconditioner at each step. Details on the implementations of parallel Krylov accelerators can be found in [11,12,16,10]. This note will focus on design issues and on preconditioners.

3 Preconditioners

There are three classes of preconditioners provided in pARMS . The first, and simplest, is the class of block preconditionings, or variants of Schwarz preconditioners. The second class of preconditioners comprises various types of Schur complement techniques. We discuss these three classes in turn.

3.1 Schwarz Preconditioners

The appeal of Schwarz preconditioners lies in their simplicity. In additive Schwarz, for example, a preconditioning operation consists of computing the current residual, and then performing a local solve which yields the local correction for the local part of the solution. Formally, this is described by the following procedure.

ALGORITHM **31** *Additive Schwarz (Block Jacobi) preconditioning step*

```
1. Obtain external data $y_i$
2. Compute (update) local residual $r_i = (b - Ax)_i$
3. Solve $A_i \delta_i = r_i$
4. Update local solution $x_i = x_i + \delta_i$
```

The solves with the matrices A_i in Line 3 are all performed independently. It is clear that a direct solution method could be used in case each subproblem is small enough. However, it is common to use iterative solvers for these systems since these can still be fairly large.

In domain decomposition methods, however, the accelerator for the global iteration must take into account the fact that the preconditioner may not a constant operator (i.e., that it may change at each step of the outer iteration). In general, we found that iterating to solve each of the sub problems accurately is not cost-effective. Often a simple forward-backward sweep, with ILU factors obtained from an ILUT preconditioner yields the fastest combination. Subdomain partitions may be allowed to overlap. This technique works reasonably well for a small number of overlapping subdomains as was shown in experiments using a purely algebraic form in [8]. The following are the basic options available in the category of additive Schwarz procedures:

`add_ilut`. Additive Schwarz procedure, with or without overlapping, in which ILUT is used as a preconditioner for solving the local systems.

`add_iluk`. Similar to `add_ilut` but uses ILU(k) as a preconditioner instead of ILUT.

`add_arms`. Similar to `add_ilut` but uses ARMS as a preconditioner for local systems.

There are two important variations available for each of these preconditioners. The first is that each of them can allow subdomain overlap. In addition, there are two options for performing the approximate solve $A_i \delta_i = r_i$. The first is simply to apply a forward-backward sweep combination based on the corresponding incomplete factorizations. The second is to solve the local system iteratively using a few steps of GMRES preconditioned with these incomplete factorizations.

3.2 Single-Level Schur Complement Techniques

The local system (4) can be written as

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

where x_i is the vector of local unknowns, $y_{i,ext}$ is the vector of external interface variables, and b_i is the local part of the right hand side. We can eliminate u_i from the local equations (4) to obtain the local Schur complement system:

$$S_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i - E_i B_i^{-1} f_i \equiv g_i', \tag{6}$$

where S_i is the "local" Schur complement

$$S_i = C_i - E_i B_i^{-1} F_i \tag{7}$$

Writing all the local Schur complement equations together results in the global Schur complement system:

$$\underbrace{\begin{pmatrix} S_1 & E_{12} & \dots & E_{1p} \\ E_{21} & S_2 & \dots & E_{2p} \\ \vdots & & \ddots & \vdots \\ E_{p1} & E_{p-1,2} & \dots & S_p \end{pmatrix}}_{S} \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix}}_{y} = \underbrace{\begin{pmatrix} g_1' \\ g_2' \\ \vdots \\ g_p' \end{pmatrix}}_{g'} \tag{8}$$

The important observation here is that the off-diagonal blocks E_{ij} are the same as those used for the additive Schwarz procedure. They are directly available from the data structure used to implement any distributed sparse linear system using a vertex-based partitioning.

The above system can be solved approximately by some procedure to be specified, and then the u_i variables can be computed from substituting y_i in the first equation in (4),

$$u_i = B_i^{-1}\left[f_i - F_i y_i\right]$$

which requires only a local solve. This process, which computes an approximation to the system (4) for an arbitrary right-hand side, defines one step of the preconditioning for the global system.

The various Schur complement procedures provided in `pARMS` are defined from the way in which the system (8) is solved. In the single level Schur complement approaches, this system is solved approximately by some iterative procedure. The simplest of these techniques solve the system (8) approximately by a block Jacobi type procedure, i.e., the system is preconditioned with $diag(S_i)$ and solved by a few steps of GMRES. One issue that remains, is to determine S_i or its factorization. In `pARMS` , the following relation exploited. Assume that the local matrix is partitioned as shown in (3), and let its LU factorization be

$$A_i = \begin{pmatrix} L_{B_i} & 0 \\ E_i U_{B_i}^{-1} & L_{S_i} \end{pmatrix} \begin{pmatrix} U_{B_i} & L_{B_i}^{-1} F_i \\ 0 & U_{S_i} \end{pmatrix}$$

Then,

$$L_{S_i} U_{S_i} = S_i \ .$$

Simple though this relation may appear, it provides a powerful way of obtaining factorizations of the local Schur complements. From any factorization, exact or incomplete, of the local matrix A_i a resulting factorization can be extracted for the local Schur complement S_i. Note that the factorization of A_i is required, so this extraction involves no additional computation.

The following one-level Schur complement preconditioners are available in `pARMS` .

`sch_ilut`. The global Schur complement system is solved with a block-Jacobi preconditioner, in which the diagonal block S_i is replaced by the matrix $L_{S_i} U_{S_i}$ obtained from the trace of the ILUT factorization of A_i corresponding to the interface variable y_i.

`sch_iluk`. This is the same as `sch_ilut` except that ILU(k) is used instead of ILUT.

`sch_arms`. This is the same as `sch_ilut` except that a (local) ARMS factorization is used for A_i instead of ILUT.

The `sch_ilut` preconditioner is the same as the one in [13].

3.3 Multi-level Schur Complement Techniques

Multi-level Schur complement techniques available in `pARMS` are based on exploiting block independent sets, see [14]. The idea is to create another level of partitioning of each subdomain. So within each subdomain, a second level of partitioning is provided by the block-independent set ordering. We refer to the subdomains in this second level partitioning as *sub-subdomains* for lack of a better term. An illustration is shown in Figure 3. Thus, two types of interface points are created. *Inter-domain* interface points are those points that have coupling with nodes outside of the current processor. These are traditionally termed "interface nodes". *Local* interface points are interface points between the sub-subdomains. Their couplings are all local to the processor and so these points do not require communication. Note that sub-subdomains could be obtained by a standard partitioner. However, as was mentioned above, in the current implementation they result from a *block independent set* reordering strategy utilized by ARMS [14].

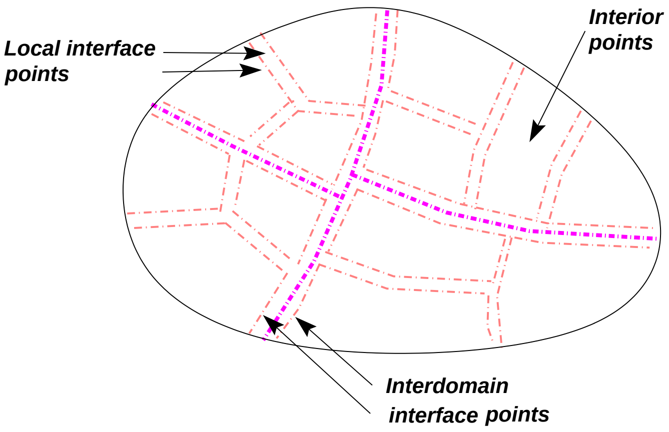

Fig. 3. A two-level partitioning of a domain

In order to explain the multilevel techniques used in `pARMS`, it is necessary to discuss the sequential multilevel ARMS technique. In the sequential ARMS, the matrix coefficient at the l-th level is reordered in the from

$$P_l A_l P_l^T = \begin{pmatrix} B_l & F_l \\ E_l & C_l \end{pmatrix} \quad (9)$$

where P_l is a "block-independent-set permutation", which can be obtained in a number of ways, and whose goal is to produce a matrix B_l that is block-diagonal. At the zero-th level ($l = 0$) the matrix A_l is the original coefficient matrix of the linear system under consideration. The above partitioned matrix is then approximately factored as

$$P_l A_l P_l^T \approx \begin{pmatrix} L_l & 0 \\ E_l U_l^{-1} & I \end{pmatrix} \times \begin{pmatrix} U_l & L_l^{-1} F_l \\ 0 & A_{l+1} \end{pmatrix} \tag{10}$$

where I is the identity matrix, L_l and U_l form the LU (or ILU) factors of B_l, and A_{l+1} is an approximation to the Schur complement with respect to C_l,

$$A_{l+1} \approx C_l - (E_l U_l^{-1})(L_l^{-1} F_l). \tag{11}$$

During the factorization process, approximations to the matrices $G_l \equiv E_l U_l^{-1}$ and $W_l \equiv L_l^{-1} F_l$ are computed for obtaining the Schur complement (11) but these two matrices are discarded after A_{l+1} is computed.

To define a recursive multilevel strategy, note that all that is needed is to define a procedure for solving the reduced system obtained by eliminating the unknown associated with the block B_l. This leads to a certain reduced system with the coefficient matrix A_{l+1}. Now recursivity is invoked and this system is partitioned again in the form (9) in which l is replaced by $l+1$. At the last level, the Schur complement system is solved using GMRES preconditioned with ILUT [11]. In the parallel version of ARMS, the same overall strategy is used except that now the last Schur complement system contains block-independent sets that lie in different subdomains, as well as the inter-domain points.

Consider a one-level `pARMS` for simplicity. In the first level reduction, the matrix A_1 that is produced, will act on all the interface variables, whether local or inter-domain of the zeroth-level. Thus, a one-level `pARMS` would solve for the unknowns associated with these variables – by some technique to be defined – and then obtain the interior variables in each processor. This second substitution phase does not involve communication. The "technique to be defined" which was just mentioned, will solve the Schur complement system associated with all the interface variables of the first level. These interface variables include the inter-domain interface points as well as the local interface points found at the zeroth level. We refer to the related Schur complement as the "expanded Schur complement". `pARMS` provides two distinct techniques for solving the global Schur complement system, listed next.

`sch_gilu0`. The expanded Schur complement system is solved with GMRES preconditioned with a parallel ILU(0), see, for example [11]. The parallel ILU(0) preconditioning requires a global order (referred to as a schedule in [5]) in which to process the nodes. A global multicoloring of the domains is used for this purpose as is often done with global ILU(0).

`sch_sgs`. Here the symmetric block Gauss-Seidel preconditioner is used to solve the global Schur complement system.

Consider now a multilevel technique. If a second level reduction is applied then the above can be repeated with the provision that the inter-domain interface points will not become *local*, i.e., the block independent reordering should simply bypass these points to ensure that they remain inter-domain points for the next level. In this case there is strictly no change to the above, except for the fact that there are now two levels to descend before reaching the last Schur complement system (expanded system) to be solved by one of the above techniques.

In the above technique, the actual inter-domain points will remain that way throughout the levels. A second method not implemented as yet, consists of adding a different type of reduction after a few levels of reduction of the first type have been performed. In the second type of reduction, group-independent sets across subdomains will be sought. This third class of Schur complement techniques will considered in the future.

4 A Few Implementation Details

The implementation of `pARMS` builds on the earlier package PSPARSLIB. However, a major difference with PSPARSLIB is that most of the code is implemented in C instead of Fortran. The selection of C versus FORTRAN 90, was motivated mostly by the desire to integrate `pARMS` with existing packages (including ARMS) which are gradually moving away from FORTRAN.

The package is articulated around several objects, represented by C structs. We briefly discuss a few of these in order to give an idea of the overall organization of the package.

The communication object is associated with a given distributed matrix (see next) and comprises such things as: the number and list of neighboring processors, their color (when required), the list of ghost nodes, the overlap array (number of processors to which each interface variable belongs). Some of the other variables are the processor label (`myproc`), the number of local variables (`nloc`), and the number of internal variables (`nbnd`). Note that several of these parameters and arrays were also available in PSPARSLIB.

The distributed matrix object defines the data structure for the distributed matrix. It includes a communication object for the distributed matrix, a Hash struct, and a sparse matrix storage format struct. The sparse matrix struct is defined to allow the capability to handle several storage schemes, though currently only the Compressed Sparse Row scheme is supported. It includes a number of functions pointers required to deal with the distributed matrix. Most important among these are the `bdry` and `setup` functions inherited from PSPARSLIB. These two functions are called to create the communication object (`bdry`) and the local matrix (`setup`) at the start. Another function, used in the preprocessing phase, is `getmap` which determines the node (original labels) to processor mappings.

The Preconditioning object contains mostly function pointers which allow to call the selected preconditioner constructors and the associated operations. So far a selection of 13 different basic preconditioners are available. Among these, variations can be obtained by changing parameters. Two associated structs are `prepar` and `ipar` which define the parameters for the preconditioner construction and operations. Once the preconditioning method `meth` is selected, the `CretePrecon` function is then invoked, as for example, in

```
        CreatePrecon(distMat, &precon, meth, prepar, ipar)
```

This has the effect of setting the `precon` struct and setting up the preconditioner for the distributed matrix `distMat`, according to the method and parameters selected.

The following lines are extracted from a sample main program and show the most important function calls made to set-up a distributed sparse linear system and to solve it with `pARMS` .

```
/*---creates the distr. mat. struct.  */
1.  CreateMat(&distMat, "csr");
2.  ... Partition the global system. [or do parallel
    ... assembly]. Let (b, jb, ib) be the local matrix
    ... in CSR format (global labeling).
/*--- copy this partial csr matrix to dstMat structure
3.  CopCsrToDm(distMat, b, jb, ib);
/*---create node to processor map     */
4.  getmap(distMat, maptmp,mapptr,&n);
/*---create the interface information*/
5.  bdry(distMat);
/*---build the local data structures  */
6.  setup(distMat) ;
7.  ... A few similar operations follow for ‘‘vec’’
    ... objects right-hand-side and solution
/*--- create preconditioner           */
8.  CreatePrec(distMat,&precon,meth,prepar,ipar);
/*--- call FGMRES to solve            */
9.  fgmresd(distMat,precon,ipar,rhs,sol);
```

In Line 2, the system is read by one processor then partitioned and distributed to other processors. Alternatively, some partitioning of the physical domain is performed and the local equations (with original global labeling) are obtained in parallel. Also some of the parameters such as those in `meth, ipar, prepar` are read or set in the main program. We have mentioned that there are 13 preconditioners available in `pARMS` . `pARMS` also includes three different accelerators: `fgmres, bcgstab` and `dgmres`. The last of these is the deflated GMRES algorithm which uses eigenvalue deflation.

The techniques described in this paper have all been tested in [7] which reports a number of experiments on an IBM SP and Origin 3800, including comparisons with a direct solver, and parallel efficiency studies. The paper and soon-to-be-released package can be obtained from `http://www.cs.umn.edu/ saad`.

Acknowledgment. The authors would like to acknowledge the work of Zhongze Li and Brian Suchomel who developed the biggest parts of `pARMS` and the sequential ARMS respectively.

References

1. S. Balay, W. D. Gropp, L. Curfman McInnes, and B. F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.24, Argonne National Laboratory, 1999.
2. 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.
3. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT press, 1994.
4. 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.
5. D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. Technical Report (preprint), Old-Dominion University, Norfolk, VA, 2000.
6. 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.
7. Z. Li, Y. Saad, and M. Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. Technical Report umsi-2001-100, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2001.
8. G. Radicati di Brozolo and Y. Robert. Parallel conjugate gradient-like algorithms for solving sparse non-symmetric systems on a vector multiprocessor. *Parallel Computing*, 11:223–239, 1989.
9. Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. on Sci. and Stat. Comput.*, 14:461–469, 1993.
10. Y. Saad. Krylov subspace methods in distributed computing environments. In M. Hafez, editor, *State of the Art in CFD*, pages 741–755, 1995.
11. Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS publishing, New York, 1996.
12. 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.
13. Y. Saad and M. Sosonkina. Distributed schur complement techniques for general sparse linear systems. *J. Scientific Computing*, 21(4):1337–1356, 1999.
14. Y. Saad and B. Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. Technical Report umsi-99-107-REVIS, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2001. Revised version of umsi-99-107.
15. Y. Saad and K. Wu. Parallel sparse matrix library (P_SPARSLIB): The iterative solvers module. Technical Report 94-008, Army High Performance Computing Research Center, Minneapolis, MN, 1994.
16. Y. Saad and K. Wu. Design of an iterative solution module for a parallel sparse matrix library (P_SPARSLIB). In W. Schonauer, editor, *Proceedings of IMACS conference, Georgia, 1994*, 1995.