P-SPARSLIB: A PORTABLE LIBRARY OF DISTRIBUTED MEMORY SPARSE ITERATIVE SOLVERS

Yousef Saad* Andrei V. Malevsky †

September 1, 1995

*University of Minnesota, Department of Computer Science, 200 Union Street S.E., Minneapolis, MN 55455 USA; saad@cs.umn.edu.

†Université de Montréal et Centre de Recherche en Calcul Appliqué, 5160, boul. Décarie, bureau 400, Montréal (Québec) H3X 2H9, Canada; malevsky@cerca.umontreal.ca.

Abstract

Domain Decomposition techniques constitute an important class of methods which are especially appropriate in a parallel computing environment. However, few general purpose computational codes based on these techniques have been developed so far. In this paper, we propose one such solver developed around the idea of 'distributed sparse matrices'. We explore issues related to data structures for distributed sparse matrices, simple matrix operations, as well as the implementations of the iterative solution kernels and analyze their performance on different architectures. We also describe a preliminary version of a portable library of parallel sparse iterative solvers, P-SPARSLIB, which encompasses the proposed concepts.

Keywords: Sparse matrix, Domain decomposition, Iterative solver, Parallel computer, Distributed memory

1 Introduction

Domain decomposition (DD) has emerged as a fairly general and convenient paradigm for solving Partial Differential Equations (PDEs) on parallel computers (see, e.g. [Quarteroni 1994]). Typically, a physical domain is partitioned into several sub-domains and some technique is used to recover the global solution by a succession of solutions of independent subproblems associated with the subdomains. Each processor handles one or several subdomains in the partition and then the partial solutions are combined, typically over several iterations, to deliver an approximation to the global system. All DD techniques rely on the fact that each processor can do a large part of the work independently, and the work to construct the global solution from the partial solutions is not too expensive relative to that of obtaining these partial solutions.

In order to implement DD techniques efficiently and for realistic problems we need to keep in mind the following requirements.

1. In order to be able to deal with the most complex problems that arise in realistic applications, it is vital to provide automatic tools for performing the many tasks that would otherwise make the implementation impractical.

2. Flexibility is critical in a general purpose DD implementation. A general purpose DD library must be able to accommodate various data structures, iterative solvers, preconditioners, and be adaptable to new computer architectures.

3. The message passing programming paradigm represents one of the best possible environments for DD techniques. In addition, DD techniques may also benefit from a heteregenous computing environment.

Consider the first item in the above list. In order to implement a domain decomposition approach we need a number of numerical and non-numerical tools for performing the preprocessing tasks required to partition a domain and map it into processors, as well as to set up the various data structures. It is important to be able to perform such tasks automatically. The dependency between the unknowns in the problem is often conveniently represented by a graph. One of the basic tasks to be performed is to find a partition of the graph into subgraphs whose union is equal to the original graph. Ideally,

the subdomains should have roughly equal size, although criteria of load balancing other than subdomain size can also be used. Once the graph partitioning is done, we must find a good subdomain to processor mapping to minimize the inter-processor communications. A general recipe for this task is difficult to find, since it is architecture- and problem-dependent. The library must include preprocessing routines to set-up the data structures and prepare for the iterations.

There is often a trade-off between the second and the third items in the above list of requirements, namely between flexibility and performance. The functions whose performance is critical for the overall speed of a code must often be tuned-up for a particular environment. To achieve good performance it is necessary to isolate these routines in order to adapt them to changes in computer architecture and programming environment. For this reason, a general purpose library must have a hierarchical structure. All the functional routines (iterative solvers and preconditioners) must not depend on a particular data structure or a message-passing library. They must instead rely on a lower level set of basic kernels and tools. We have employed a 'reverse communication mechanism' in order to make the major components of the library free of any particular data structures. In addition we have aimed at making the code as machine-independent as possible in order to keep the bulk of the routines reusable if an architecture or a message-passing paradigm changes. We have avoided to embrace any of the proposed message-passing standards such as PVM [Beguelin et al. 1993] or MPI [Gropp et al. 1994] throughout the code, but have rather isolated the communications routines in order to be able to port them with minor efforts and to utilize the vendor-supplied software and hardware solutions. We have assembled the communications routines in a toolkit which together with the basic linear algebra routines (BLAS) serves as a module of 'basic kernels', both computatutional and communicational, for the library.

Loosely coupled parallel computers provide a perfect framework for implementing DD-type algorithms since processors can perform fairly substantial coarse grain tasks such as local solutions, between synchronization points. In addition there is a large number of possible combinations of different algorithms to choose from. As will be seen, it is rather easy to develop preconditioners specifically for distributed sparse matrices. On the other hand, this added flexibility does not come for free, and programming is typically more complex.

2 Graph partitioning concepts

Here we take a slightly more general viewpoint than that commonly used in the PDE framework, and we consider general sparse linear systems as the starting point. The dependency between the unknowns in the linear system is often conveniently represented by an adjacency graph. The nodes of the graph may represent vertices of a physical mesh for a discretized PDE, and we will sometimes call the unknowns 'nodes' or 'vertices'. The matrix can also originate from an application other than PDEs, e.g. electrical networks, or queuing models.

One of the first tasks to be performed when solving a sparse linear system in parallel is to partition the linear system and map it into the processors. This can be achieved by partitioning the adjacency graph of the matrix into p subgraphs, whose union is equal to the original graph. We are given a graph (V, E) where the vertex set V represents the set of unknowns and the edge set E represents the connectivity between these unknwons as defined by the sparse matrix. We partition the vertex set V into p subsets $V_1, \ldots, V_p$ This results in a partition of the original graph in p subgraphs, (V_i, E_i). The edge sets E_i describe the connectivity between vertices belonging to V_i to other nodes, possibly belonging to subsets other than V_i. For simplicity, we assume that a vertex subset V_i. and its subgraph are associated with a processor i of a distributed-memory computer. 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:

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

When all the subsets V_i are not pairwise disjoint (i.e. some unknowns belong to more than one subset) the term 'partition' conflicts with common usage, but we will use the term 'overlapping partition' in this case. Efficient partitioning or node-to-processor mapping is often a problem-dependent task, but a number of heuristics for general sparse matrices have been developed (see, e.g., [Pothen et al. 1990; Goehring and Saad 1994]).

The most general way of describing a node-to-processor mapping is to set up a list for each processor, containing all the nodes that are mapped to that processor. A graph partitioner may provide such a set of lists in a standard pointer-list representation of the following form.

PTR =	1	5	7	9	13							
LST =	1	2	5	6	3	4	9	10	7	8	11	12

The LST array lists all the vertices of subdomain 1 followed by those of subdomain number 2, etc... PTR is a pointer array the entries of which point to the beginning of the sublist for each subdomain in the array list. Thus, in the above example, the vertices 1, 2, 5, 6, belong to subdomain number 1, vertices 3, 4, to subdomain 2, vertices 7, 8, 11, 12 to subdomain 3, vertices 9, 10, to subdomain. This corresponds to the illustration in Fig. 1.

There are two issues which can be raised relative to these mappings. First, we wish to be able to find good partitionings of the original graph into subgraphs. We use techniques requiring only graph theory – since we do not necessarily have coordinates of the vertices. The goal is to find a partition of the graph which achieves a good load balancing of the work among the processors and would give a small ratio of communication over computation with external processors during the iterative procedure. A number of strategies have been described in the literature the best known of which is the Recursive Spectral Bisection algorithm [Pothen et al. 1990] and its many variations, see, e.g., [Hendrickson and Leland 1992; Leet et al. 1993]. There are also a number of simpler and less expensive heuristics which do work fairly well in practice, see for example [Goehring and Saad, 1994].

The second problem is to find a good mapping of the subdomains or subgraphs to the processors, once a graph partitioning is found. This subgraphs-to-processor mapping can generally be architecture-dependent. However, the partitioning algorithm can clearly take advantage of a measure of how good a given partitioning may be, by using different weight functions for the vertices, for vertex-based partitionings. We could also search for a good mapping which will minimize communication costs given some knowledge on the architecture. Our motivation here is that it is far easier to first find an architecture-independent partition, and then, in a second phase, find the proper mapping to the given architecture. Currently, many parallel computers are built with the goal of attempting to make the system emulate a fully connected computer by minimizing the difference in performance between different mappings.

We can define a binary relation between te subdomains to translate the existence of edges that leave from a subdomain and reach a node in another subdomain. This clearly defines a graph, referred to as a quotient-graph

or Q-graph, associated with the decomposition. Nodes of the Q-graph will represent subdomains (subgraphs) and there is an edge from one vertex to another in the Q-graph when there is at least one edge in the original graph from a node of the first subdomain to a node of the second subdomain.

Given a map $\{V_i\}_{i=1,\ldots,s}$ of a graph $G = (V, E)$ we define a graph $G_Q = (V_Q, E_Q)$ whose vertices labeled $i = 1, \ldots, s$ represent the subsets $V_i, i = 1, \ldots, s$, and whose edge set is

$$E_Q = \{(i, j), i, j \in V_Q, \ \ s.t. \ \exists v \in V, w \in V, (v, w) \in E\}$$

Thus, the Q-graph associated with the graph partitioning illustrated in Fig. 1 is shown in Fig. 2. The definition of a Q-graph given here is similar to the common definition of quotient graphs in [George and Liu 1981] except that we allow subgraphs to have common nodes. The Q graph indicates which subdomains or subgraphs need to exchange data with one another in an iterative solution procedure. It is therefore important to extract this information in the preprocessing phase which preceeds the execution of an iterative method for solving a distributed sparse linear system.

We have defined the above partitioning in terms of vertices (nodes) being assigned to processors. We may in some cases, wish to assign edges of the graph to processors, or even elements in a finite-element type approach. There is no conceptual difference between these mappings, since edge-based partitionings can be viewed as vertex-based partitionings on the dual graph. Recall that a dual graph of a graph $G = (V, E)$, is defined as a graph whose vertex set is E, and whose edge set is the set of pairs of edges in the original graph sharing a vertex.

3 Distributed sparse matrices

Sparse matrix-by-vector multiplications are among the most expensive operations in an iterative solver, and their performance may strongly influence the overall speed of the scheme. In this section, we describe the data structures or 'formats' used for storing distributed sparse matrices. We will also show how to perform matrix-vector products in these formats. The goal of the 'distributed sparse matrix' data structure is to minimize inter-processor communication and allow an overlap between computations and communications whenever possible. The proposed communication formats only assume that the matrix is distributed row-wise among the processors, but do not

specify the local storage mode. The same communication formats can be combined with local matrices stored in the Compressed Sparse Row (CSR), Block Sparse Row (BSR), Jagged Diagonal format (JAD), or even dense storage mode.

Assume that we have a convenient partitioning of the graph, as defined by a certain node (vertex) to processor mapping. We distinguish the following four different classes of nodes relatively to a subdomain (processor) i:

1. internal nodes;
2. local interface nodes;
3. external interface nodes;
4. nodes not connected to a processor i.

A node is *internal* to a subdomain i, (or, equivalently to processor i holding the subset V_i) if it is connected only to the elements of V_i. The *local interface nodes* are connected to elements of other subsets. The *external interface nodes* are the nodes which belong to the other processors but are connected to vertices of V_i. All three types of nodes must be represented in a local data structure in order to perform matrix-by-vector products efficiently. These definitions are illustrated in Figure 3. With these definitions in mind, we now need to set up a local data structure in each processor for a distributed matrix which will allow us to carry out the basic operations such as a matrix-by-vector product. First, we recall that we are assuming that *all vectors associated with any solution procedure, including solution vectors, right hand-sides, and Krylov vectors, are partitioned conformally.* In other words the components of a global vector x whose indices belong to V_i are mapped to Processor (subdomain) i. Thus, to add two global vectors it suffices to add all the local representations of these two vectors. To perform dot product, we need to perform a dot product for each pair of these local representations and then sum-up all the local results. If row number k is mapped into processor i then so is the unknown k, i.e. the matrix is distributed row-wise across the processors according the distribution of the variables. The first step in setting up the local data-structure prior to executing an iterative algorithm for a distributed sparse matrix is to have each processor determine the set of all other processors with which it must exchange information when performing matrix-vector products. Although these are not necessarily physical neighbors, they hold subdomains that are

adjacent to the subdomain that is mapped to them. For simplicity, and by analogy with the name used in the actual codes that are run on each processor, we will refer to the label of a given processor in which a copy of the (same) code is executed on each processor as *myproc*.

The information needed to find these neighboring processors is contained in the global node-to-processor mapping array described in the previous section. For simplicty we will assume for this description that there is no overlap, i.e., any node j belongs to only one processor, which we denote by $map(j)$. The local rows are inspected one by one and for each nonzero a_{ij} with $map(j) \neq myproc$, where *myproc* is the label of the current processor, we add $map(j)$ to the list of neighboring processors if it is not already listed. In the overlapping case, $map(j)$ can actually be a set having more than one element, so the linked list structure for node j must be crossed each time. We store the labels of the neighboring procesors in an array $proc(1 : nproc)$ where *nproc* is the number of processors found.

In this phase, each processor *myproc* will also determine its external interface nodes, or the nodes which belong to the neighboring processors and are coupled with the local interface nodes of that processor. When performing a matrix-by-vector product, neighboring processors must exchange values of their adjacent interface nodes. In order to perform this data exchange operation efficiently, it is important to group these nodes processor by processor. Thus, we first list all those nodes which must be sent to $proc(1)$, followed by those to be sent to $proc(2)$ etc.. Two arrays are used for this purpose, one called ix which lists the nodes as indicated above and a pointer array *ipr* which points to the beginning of the list for $proc(i)$. At the end of the preprocessing step each processor `myproc` must have the following information.

1. `nproc` – The number of all adjacent processors, i.e., processors with which processor `myproc` will be exchanging information.

2. `proc(1:nproc)` – List of the `nproc` adjacent processors.

3. `ix` – The list of local interface nodes, i.e., nodes whose values must be exchanged with neighboring processors. The list is organized proccessor by processor in order to perform the data exchange efficiently using a pointer-list data structure.

4. `ipr` – The pointer to the beginning of the list in array `ix` of each of

`nproc` neighboring processors.

This information is extracted by examining the adjacency graph as well as the partitioning. It is performed in each processor independently if the adjacency graph is available in each node.

4 Distributed matrix-by-vector multiplication

In order to perform a matrix-by-vector product with a distributed matrix, we need to multiply the matrix consisting of rows that are local to a given processor by a distributed vector. Some components of the vector will be local, but some components, namely values at the external interface nodes, must be moved to the current processor from the adjacent subdomains (processors). Let A_{loc} be the local matrix, i.e., the (rectangular) matrix consisting of all the rows that are mapped to *myproc*. We will call B_{loc} the 'diagonal block' of A_{loc}, or the submatrix of A_{loc} whose nonzero elements a_{ij} are such j is a local variable. Note that B_{loc} is a square matrix of size $nloc \times nloc$ where $nloc$ is the number of unknowns residing on *myproc*. Similarly, we will call B_{ext} the 'off-diagonal' block, i.e., the submatrix of A_{loc} whose nonzero elements a_{ij} are such that j is not a local variable. This structure of a distributed matrix is illustrated in Fig. 4.

To perform a matrix-vector product, we must perform the following steps:

1. multiply the diagonal block B_{loc} by the local variables;
2. bring in the external variables (components of the distributed vector at the external interface nodes);
3. multiply the off-diagonal block B_{ext} by these external variables and add the result to that obtained from the first multiplication.

Note that the steps 1 and 2 can be performed simultaneously. A processor can be multiplying B_{loc} by the local variables while waiting for the external variables to be received.

A section of the code to perform a matrix-by-vector product as we implemented it is as follows.

```
call MSG_bdx_send(nloc,x,y,nproc,proc,ix,ipr,ptrn)
```

```
call amux(nloc,x,y,aloc,jaloc,ialoc)
call MSG_bdx_receive(nloc,x,y,nproc,proc,ix,ipr,ptrn)
nrow = nloc - nbnd + 1
call amux1(nrow,x,y(nbnd),aloc,jaloc,ialoc(nloc+1))
```

In the above code segment, `MSG_bdx_send` and `MSG_bdx_receive` are communication routines to be described in the next section, `amux` ($y = Ax$) and `amux1` ($y = y + Ax$) are the local sparse matrix-by-vector multiplication routines for the CSR format. Here, `nloc` is the number of nodes mapped to *myproc*; `nbnd` is the pointer to the start of the interface nodes; `aloc, jaloc, ialoc` are the data structures for the two matrices B_{loc} and B_{ext} stored together as one matrix in a CSR format. The call to `amux` performs the operation $y := B_{loc}x_{loc}$. The call to `amux1` performs $y := y + B_{ext}x_{ext}$. Notice that the data for the matrix B_{ext} is simply appended to that of B_{loc}, a standard technique used for storing a succession of sparse matrices. The B_{ext} matrix acts only on the subvector of x which starts at location *nbnd* of x. The size of the B_{ext} matrix is $nrow = nloc - nbnd + 1$.

In the above example we have used the CSR format for the purpose of illustration only. In fact, we can store the matrices B_{loc} and B_{ext} in any format that will yield good performance, while keeping the overall strategy the same. A local matrix-by-vector multiplication routine can be optimized to take advantage of a particular architecture and employ local BLAS-type functions. It is important to reorder A_{loc} in such a way as to have all the internal nodes followed by the interface nodes. There are several advantages of this ordering, one of which being that it facilitates implementations Schur complement type approaches which iterate with interface points only.

5 Message-passing tools

Operations with matrices and vectors on distributed-memory architectures require data exchange between the processors. The iterative solvers utilize only a limited subset of a message-passing library. In fact, there are only two operations in the body of a solver which employ inter-processor communications. These are the *data exchange* between the boundaries of the subdomains and *the distributed dot product* of two vectors. The first of these two operations occurs in the distributed matrix-by-vector products and in the standard preconditioning operations. The dot products are required at each step of a Krylov subspace procedure. These two communication kernels

require special attention if we wish to achieve good performance. The boundary exchange and distributed dot product routines together with some auxiliary routines have been assembled together in module called the *message-passing toolkit*. So far, we have implemented three versions of the toolkit, the CM5 version (based on the CM5 message-passing library, CMMD), the CRAY-T3D version (employing both CRAY-PVM and SHMEM routines), and the PVM version (based on the PVM 3.2 distributed by the Oak Ridge National Laboratory).

In order to perform the boundary exchange efficiently, we have exploited two features of message-passing: the asynchronous message-passing capabilities and the redundancy of communications. An asynchronous message-passing means that a processor can send data into the network and continue to perform some work without waiting for the data to actually arrive to its destination. Asynchronous message-passing can be crucial in order to achieve good performance on some architectures. For instance, only one message can be transmitted at any time by the Ethernet network serving as a bus for a workstation cluster, and the case of synchronous (blocking) message-passing all the other processors would be idle while waiting for a pair of them to finish the data exchange. The matrix-by-vector multiplication code from the previous section calls a boundary exchange routine, `MSG_bdx_send`. It sends the boundary information out according to the the tables `nproc,proc,ix,ipr` described in the above section. The boundary information is needed only for the second matrix-by-vector multiplication. The first multiplication involves only internal nodes, and therefore can be done without waiting for the interface data to arrive. In this example, the call to `MSG_bdx_receive` ensures that all the inter-processor communications have been completed prior to the matrix-by-vector multiplication for the interface nodes. The computation-communication overlap has increased the speed of the distributed matrix-by-vector multiplication routine almost by a factor of two on the CRAY-T3D massively parallel processor.

The data exchange between the boundaries of subdomains often follows a repeated pattern. The redundancy can be exploited in some cases. The parameter `ptrn` passed to `MSG_bdx_send` in the code example from the previous section specifies a communication pattern to use. The first call to `MSG_bdx_send` with a certain pattern `ptrn` creates a communication channel between the participating processors. The processors exchange the addresses of their send/receive buffers, length and type of message. Each subsequent

call to `MSG_bdx_send` with the same `ptrn` utilizes the corresponding pattern. In the CM-5 version, we have employed the virtual channels functions provided in CMMD, the CM-5 message-passing library, which allow direct link between processors and minimize the handshaking overhead involved in point-to-point message passing. Once a pattern is established, the processors can communicate without extra handshakes, and the latency decreases from 85 microsec (point-to-point communication) to 30-35 microsec (virtual channel). In the CRAY-T3D version, the receiving processor initially sends the address where the interface data must be put to the sender. At each subsequent call to `MSG_bdx_send`, the sender knows a location in the receiver's memory to place the data. Then a single call to a low-level function (a SHMEM routine) puts the data into the necessary place. The advantages of establishing communication channels originate from the features specific to message-passing libraries, and there is no general recipe for programming the channel functions.

A matrix-by-vector multiplication is usually the most time-consuming part of an iterative solver. In addition to the cost of communication, a sparse distributed matrix-by-vector product has the overhead from the operations with indices. These index operations sometimes generate cache misses and RAM page faults which also impede performance of the matrix-by-vector multiplication. We have estimated performance of the distributed matrix-by-vector product kernel for different architectures. The execution rate is given in the Fig. 5. In the tests, the matrix was generated by a 5-point 2D finite-difference stencil and stored in the CSR format. The local matrix-by-vector multiplication for this storage format was implemented in FORTRAN 77, and one can anticipate better performance on the CRAY-T3D with optimized local matrix-by-vector product kernels [Li 1995]. A 32 processor configuration was used for each architecture. Time was measured by a wallclock timer in a single user mode. The CM5 code does not utilize the vector units, and performance of its local operations is equivalent to the speed of SUN SPARC 2 processor (SPARC 10 for the CM5E). The dot product belongs to a family of operations known as 'global reductions' where the data are gathered from the processors, combined following a certain rule, and then broadcast back to the processors. The best strategy for global reductions depends on a network topology. Many parallel computer manufacturers are starting to provide hardware and software support for performing global reduction operations efficiently. For instance,

the reductions are performed on the CM-5 by a separate low-bandwidth and low-latency network. Global reductions are included in the MPI standard [Gropp et al. 1994] and in the CRAY-T3D message-passing library (SHMEM). However, the global reductions must be coded employing the basic point-to-point communications for a more general message-passing paradigm, such as PVM. A global reduction can be performed in $\log_2 N$ steps (N is the number of processors involved) by subsequent binary dissection with a broadcast to all at the end. The global reductions used by the sparse iterative solvers require only a single number to be gathered from all the processors after they perform the reduction on a local vector. Thus, a network latency rather than a bandwidth determines performance of the global reductions.

We have also tested performance of the distributed dot product kernel for different architectures. The execution rate is given in the Fig. 6. As for the matrix-by-vector product kernel, a 32 processor configuration was used for each architecture with the execution time measured by a wallclock timer in a single user mode.

6 Preconditioned Krylov subspace algorithms

The computational requirements of the various conjugate gradient like algorithms that have been developed are essentially identical. The GMRES algorithm was introduced in [Saad and Schultz 1986] for solving general sparse nonsymmetric linear systems. Here, we would like to illustrate the implementation of these methods with only one such technique, namely the Flexible variant of the GMRES algorithm (FGMRES) [Saad 1993]. The FGMRES allows the preconditioner to vary from step to step. In our context, we would like to be able to use any secondary iterative procedure as a preconditioner, a feature which is quite helpful in DD methods or in any parallel computing implementation. In the simplest case, if a a block-Jacobi iteration is used as a preconditioner (additive Schwarz), in which the blocks correspond to the different subdomains, then we solve each system associated with a subdomain by an iterative process. In the standard 'non-flexible' techniques, these inner solutions must be 'exact' or highly accurate in each subdomain. With FGMRES and other flexible techniques this does not have to be the case. FGMRES even allows the inner preconditioning steps to be completely asynchronous, a feature which may help minimize

communication and synchronization costs in a parallel approach.

Consider a linear system of the form

$$Ax = b, \tag{2}$$

where A is a large sparse nonsymmetric real matrix of size N. The GMRES algorithm is a technique which minimizes the 2-norm of the residual vector $b - Ax$ over x in the Krylov subspace

$$K_m = \text{Span}\{r_0, Ar_0, \ldots, A^{m-1}r_0\},$$

where r_0 is the initial residual vector $b - Ax_0$. When a preconditioner M is applied to the right of the above linear system, we implicitly solve the preconditioned linear system

$$(AM^{-1})(Mx) = b. \tag{3}$$

instead of 2. FGMRES allows the right preconditioner M to be different at each step j. The algorithm is desribed next.

1. **Start:** Choose x_0 and a dimension m of the Krylov subspaces. Define an $(m+1) \times m$ matrix $\bar{H}_m$ and initialize all its entries $h_{i,j}$ to zero.

2. **Arnoldi process:**

   (a) Compute $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$ and $v_1 = r_0/\beta$.

   (b) For $j = 1, ..., m$ do

   - Compute $z_j := M_j^{-1} v_j$
   - Compute $w := Az_j$
   - For $i = 1, \ldots, j$, do $\left\{ \begin{array}{l} h_{i,j} := (w, v_i) \\ w := w - h_{i,j} v_i \end{array} \right.$
   - Compute $h_{j+1,j} = \|w\|_2$ and $v_{j+1} = w/h_{j+1,j}$.

   (c) Define $Z_m := [z_1, ..., z_m]$.

3. **Form the approximate solution:** Compute $x_m = x_0 + Z_m y_m$ where $y_m = \text{argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$ and $e_1 = [1, 0, \ldots, 0]^T$.

4. **Restart:** If satisfied stop, else set $x_0 \leftarrow x_m$ and goto 2.

The Arnoldi loop constructs an orthogonal basis of the preconditioned subspace

$$K_m = \text{Span}\{v_1, AM_1^{-1}v_1, \ldots, AM_{m-1}^{-1}v_{m-1}\}$$

by a modified Gram-Schmidt process, in which the new vector to be orthogonalized is defined from the previous vector in the process.

Note that if the preconditioner is constant, i.e., if $M_j = M$ for $j = 1, \ldots, m$ then the method is equivalent to the standard GMRES algorithm, right-preconditioned with M. The approximate solution x_m obtained from this modified algorithm minimizes the residual norm $\|b - Ax_m\|_2$ over $x_0 + \text{Span}\{Z_m\}$. In addition, if at a given step k, we have $Az_k = v_k$ (i.e., if the preconditioning is 'exact' at step k) and if the $k \times k$ Hessenberg matrix $H_k = \{h_{ij}\}_{i,j=1,\ldots,k}$ is nonsingular then the approximation x_k is exact.

To further enhance flexibility, we found it extremely helpful to include an additional feature referred to as a 'reverse communication mechanism' whose goal is to avoid passing data structures to the iterative solver [Ashby and Seager 1990]. The passing of a matrix can be a heavy burden on the programmer since it is nearly impossible to find a data structure that will be suitable for all possible cases. The solution is not to pass the matrices in any form. Whenever a matrix-by-vector product or a preconditioning operation is needed, we can simply exit the subroutine and have the subroutine caller perform the desired operation. The calling program should call the iterative routine again, after placing the result of the matrix-vector operation in one of the vector arguments of the subroutine.

The FGMRES routine must return a parameter indicating the type of operation requested. Thus, a typical execution of a flexible GMRES routine with reverse communication would be as follows:

```
   icode = 0
1  continue
   call fgmres(n,im,rhs,sol,i,v,w,wk1,wk2,eps,maxits,io,icode)
   if (icode .eq. 1) then
     call  precon(n,wk1,wk2) <-- preconditioning operation
     goto 1
   else if (icode .eq. 2) then
     call  matvec(n,wk1,wk2) <-- matrix vector product
     goto 1
   endif
```

The icode parameter in the above program segment, is an indicator of the type of operation needed by the subroutine. If it is set to one then we need to apply a preconditioning operation to the vector $wk1$, put the result in $wk2$ and call FGMRES again. If it is equal to two then we need to multiply the vector $wk1$ by the matrix A, then put the result in $wk2$ and call FGMRES again. Reverse communication enhances the flexibility of the FGMRES routine enormously. For example, when changing preconditioners, we may iterate on a coarse mesh and do the necessary interpolations to get the result at a given step and then iterate on the fine mesh in the following step. This can be done without having to pass any data regarding the matrix or the preconditioner to the FGMRES accelerator. Since the matrix-by-vector multiplication has been taken away from the body of FGMRES routine, the only communication routine it calls is the distributed dot product of two vectors. The rest of operations can be done independently and require no synchronization.

7 Structure of the P-SPARSLIB library

We have implemented the above ideas in a software library for parallel sparse matrix computations, named P-SPARSLIB. It consists of four parts:

1. accelerators (GMRES, FGMRES, CG, etc.);
2. preprocessing tools;
3. preconditioning routines;
4. message-passing tools.

The accelerators together with the preconditioners constitute the functional layer of the library. Except in special instances, Krylov subspace methods, whether for standard or distributed matrices, will work poorly without preconditioning. The preconditioners module consists of a number of 'standard' options for preconditioning distributed sparse matrices, such as overlapping block Jacobi (overlapping additive Schwarz), multicolor block SOR (overlapping multicolor multiplicative Schwarz), distributed ILU(0), approximate inverse preconditioners, etc. These modules will be described in forthcoming paper.

The message-passing tools with the local BLAS-1 routines form the lowest level of the library. The message-passing toolkit consists of the boundary

information exchange routine, distributed dot product, send/receive routines used at the preprocessing stage, and some auxiliary functions such as timers, machine configuration inquiries, synchronization tools. This routines utilize an underlying message-passing library (CMMD, PVM, SHMEM). The rest of the modules is completely machine-independent and would work in both distributed and shared-memory environment. Most of the routines have been written in FORTRAN 77 with a few C modules. The P-SPARSLIB routines are available on the Internet from
`http://www.cs.umn.edu/research/darpa/p_sparslib/psp-abs.html`.

8 Conclusions

With the currently available hardware, the overall performance of the library the local is predominantly controled by the BLAS-1 routines. We have estimated the cost of inter-processor communications by running the example mentioned earlier without the data exchange between the processors. The communication overhead for the massively parallel computers equipped with fast networks (CRAY-T3D and CM-5) was rather moderate, and did not exceed 10% for all the grids tested. This fact shows that it is the rate of local arithmetic which must be chiefly improved. We have employed a FORTRAN 77 version of the BLAS-1 routines distributed with the LINPACK library [Dongarra et al. 1979]. Vendor-supplied, optimized BLAS-1 routines may greatly enhance performance of the P-SPARSLIB modules. We have also measured performance of the matrix-by-vector and dot product kernels on a cluster of IBM RISC 6000 workstations connected with the ATM network. We found that despite an impressive rate of local arithmetics, a cluster of workstations can hardly be used to run P-SPARSLIB because of a very high latency of communications which resulted in 500% communication overhead.

The work remaining to make the library useful for solving real life problems, is to develop more preconditioning options, and possibly to establish one or a few standard data structures for sparse matrices in order to facilitate the use of parts of the library.

Acknowledgments

Work supported in part by ARPA under grant number NIST 60NANB2D1272, and in part by NSF under grant number CCR-9214116. The authors would like to acknowledge the support of the Minnesota Supercomputer Institute and the National Center for Supercomputing Applications in Urbana-Champaign which provided the computer facilities and an excellent research environement to conduct this research.

References

[1] Ashby, S.F. and Seager, M. 1990. A proposed standard for iterative linear solvers. *Technical report, Lawrence Livermore National Laboratory*, Livermore, CA.

[2] Beguelin, A., Dongarra, J., Geist, A., Manchek, R., Moore, K., and Sunderam, V. 1993. Tools for heterogeneous network computing, In: *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, Philadelphia, PA, pp. 854-862.

[3] Dongarra, J.J., Bunch, J.R., Moler, C.B., and Stewart, G.W. 1979. *LINPACK Users' Guide*, SIAM, Philadelphia, PA.

[4] George, J.A. and Liu, J.W. 1981. *Computer solution of large sparse positive definite systems*, Prentice-Hall, Englewood Cliffs, NJ.

[5] Goehring, T. and Saad, Y. 1994. Heuristic algorithms for automatic graph partitioning. *Technical Report UMSI 94-29, University of Minnesota Supercomputer Institute*, Minneapolis, MN.

[6] Gropp, W., Lusk, E., and Skjellum, A. 1994 *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA.

[7] Hendrickson, B. and Leland, R. 1992. An improved spectral graph partitioning algorithm for mapping parallel computations. *Technical Report SAND92-1460, UC-405, Sandia National Laboratories*, Albuquerque, New Mexico.

[8] Leet, C.A., Peyton, B.W., and Sincovec, R.F. 1993. Toward a parallel recursive spectral bisection mapping tool. *Technical report, Oak Ridge National Laboratory*, Knoxville, TN.

[9] Li, G. 1995 A block variant of the GMRES method on distributed memory multiprocessors, submitted to *Supercomuting'95*.

[10] Pothen, A., Simon, H.D., and Liou, K.P. 1990. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix. Anal. Appl.*, 11:430–452.

[11] Quarteroni, A. (ed.) 1994. *Domain decomposition methods in science and engineering. Proceedings of the 6th International Conference on Domain Decomposition*, AMS, Providence, RI.

[12] Saad, Y. and Schultz, M.H. 1986. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7: 856-869.

[13] Saad, Y. 1993. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Statist. Comput.*, 14: 461-469.

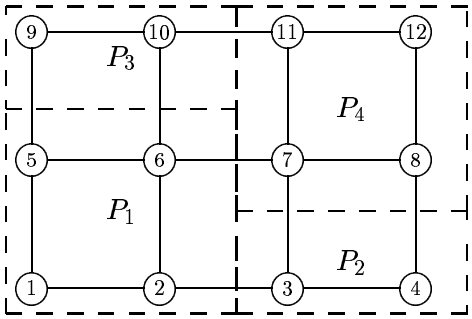

Figure 1: Mapping of a simple 4×3 mesh to 4 processors.

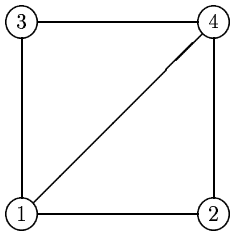

Figure 2: Q-graph for the mapping in Fig. 1.

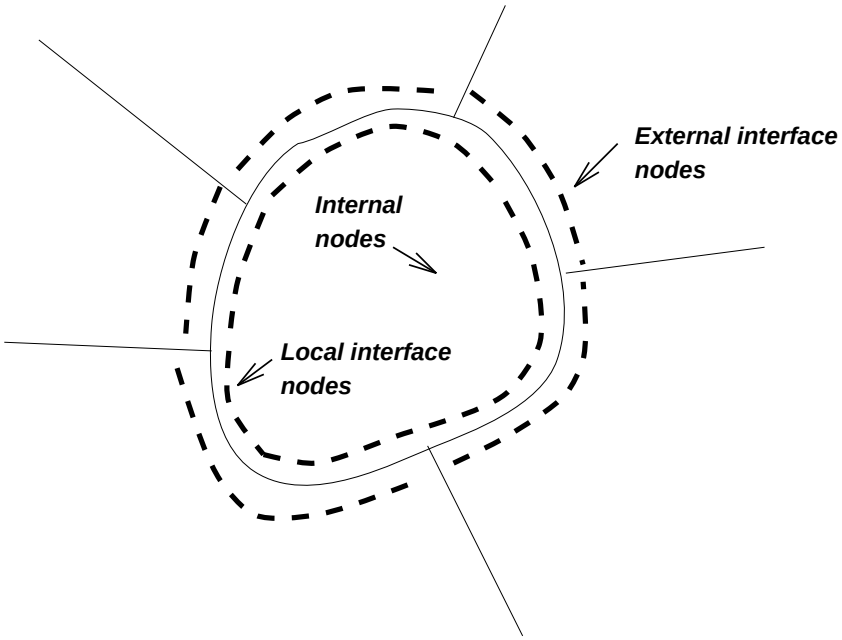

Figure 3: The local view of a distributed sparse matrix.

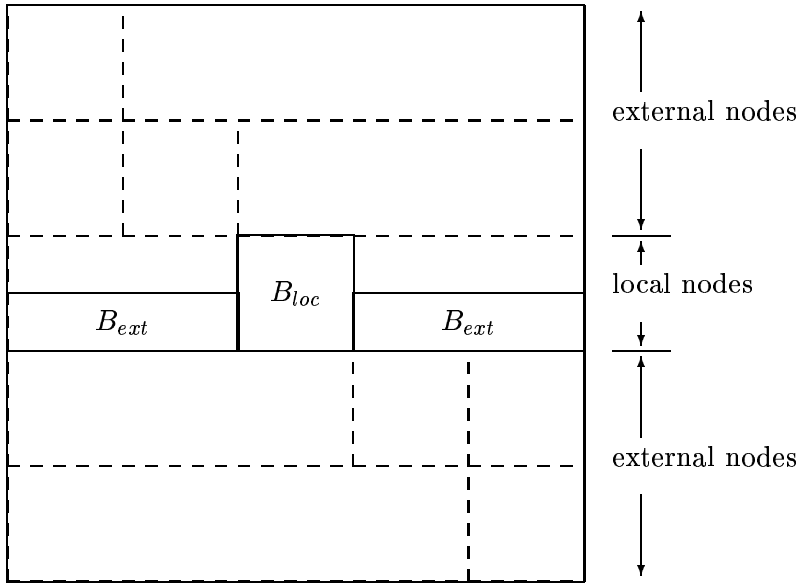

Figure 4: A global view of a distributed sparse matrix A for a simple row-wise partitioning.

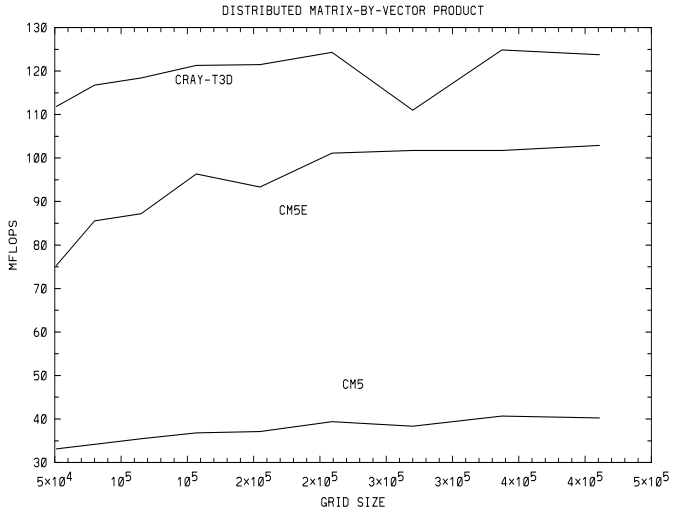

Figure 5: Performance of the sparse distributed matrix-by-vector product kernel.

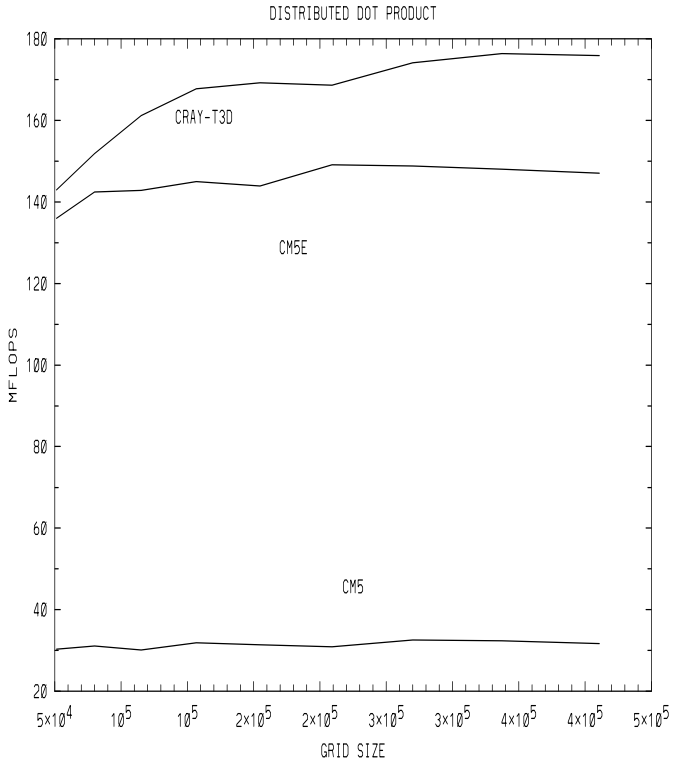

Figure 6: Performance of the distributed dot product kernel.