Iterative Solution of General Sparse Linear Systems on Clusters of Workstations[*]

Gen-Ching Lo and Yousef Saad
Department of Computer Science, and
Minnesota Supercomputer Institute
University of Minnesota
Minneapolis, MN 55455

May 27, 1996

Abstract

Solving sparse irregularly structured linear systems on parallel platforms poses several challenges. First, sparsity makes it difficult to exploit data locality and this is true for both distributed and shared memory environments. Second, it is difficult to find efficient ways to precondition the system. For example, preconditioning techniques that have a high degree of parallelism often lead to slower convergence than their sequential counterparts. Finally, a number of other 'global' computational kernels such as inner products can outweigh any gains due to parallelism, and this is especially true on workstation clusters where latency times may be high. In this paper we discuss these issues and report on our experience with PSPARSLIB, an on-going project for building a library of parallel iterative sparse matrix solvers.

1 Introduction

In the past few years, there has been a flurry of activity on the use of distributed memory computers to solve challenging scientific problems. Particularly noteworthy is the recent surge of interest in the use of workstation clusters. Connecting a small number of workstations by fast communication networks is increasingly gaining acceptance as a low-cost and effective alternative to large supercomputer engines. However, in using iterative methods to solve large sparse linear systems on workstation clusters several problems emerge. First, if few processors are used then any gains that are made from parallelism can easily be outweighed by overhead, particularly communication. This is not untypical of parallel processing in general but the problem is more acute with workstation clusters because communication costs can be quite high. Second, inner products and other global operations which normally cause few problems can now be the source of serious bottlenecks. The reasons are similar: the communication overhead in performing the global sum of the smaller inner products can overwhelm the actual time to carry out the arithmetic. On the other hand, workstation clusters present several advantages over traditional distributed computers. One of these advantages is that memory is no longer a serious limitation. This is in contrast with massively parallel computers where the global memory is often very large but the local memory attached to each individual processor is small, being limited by physical

[*]Work supported in part by ARPA under grant number NIST 60NANB2D1272 and in part by the National Science Foundation under grant NSF/CCR-9214116

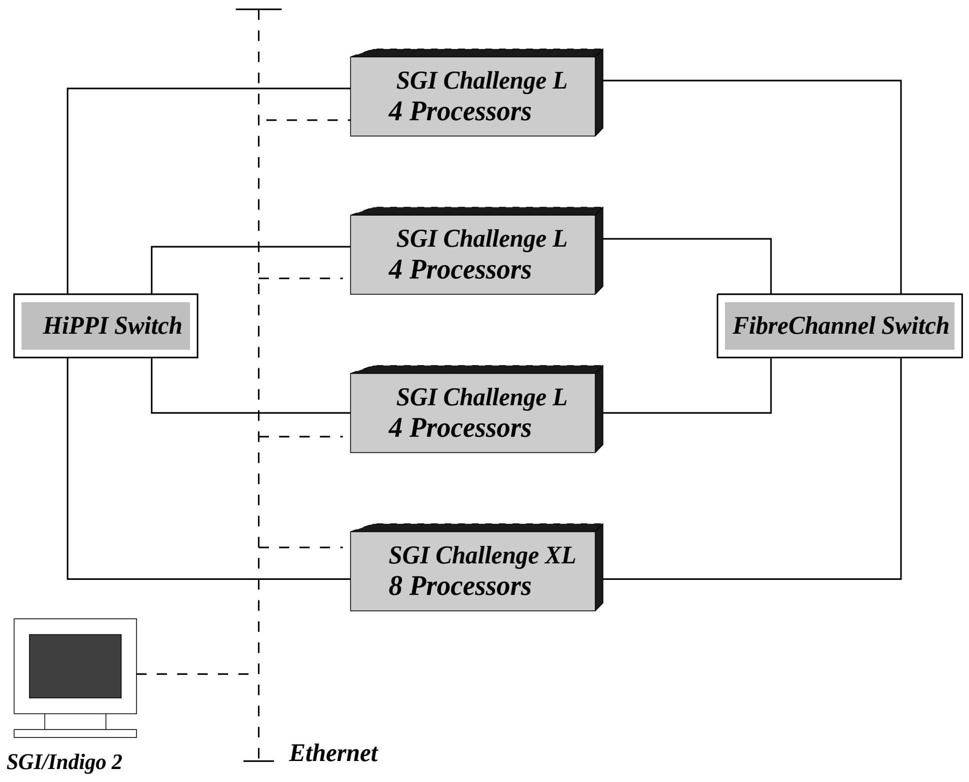

Figure 1: A workstation cluster configuration.

constraints. For example, in our earlier experiments on the Thinking Machines CM-5 we found that in order to obtain satisfactory speed-ups, the problems to be solved would have to be very large, so large in fact that they would not fit in memory given the relatively small size of local memory available. This is not the case on workstation clusters.

A typical workstation configuration is shown in Figure 1. In this particular case, each workstation is a 4-processor computer having a memory size of 512MB except for the rack-mounted Challenge XL, which has 8 processors and 2 GB of RAM. Each processor is a R4400 processor with 4MB cache. Codes within each node can be programmed either in shared memory mode, or using message passing. We used MPI to communicate between nodes within each workstation or between nodes of the cluster.

The HiPPI and Fibre-Channel switches shown are switches which permit high-speed communication between the 4 workstations. Communication speeds based on these new technologies are constantly gaining ground. One of the limiting factors here is not the bandwidth (the speed of the links themselves) but the latency. In many cases, data movement starts taking place only after several layers of software – corresponding to different faces of a common protocol – have been traversed resulting in relatively long delays.

The outline of the paper is as follows. The paper starts by discussing distributed sparse linear systems in general terms and addresses issues related to data structures and matrix-vector operations. Section 3 gives some details on the implementation of Krylov subspace methods such as GMRES and the orthogonalization procedure. Section 4 presents a few preconditioning techniques. In Section 5 a general framework for Schur complement techniques is presented. Section 6 presents numerical experiments and the last section draws a few concluding remarks.

2 Distributed Sparse Linear Systems

We consider a linear system of the form

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

where A is a large sparse nonsymmetric real matrix of size n. When mapping such a system into a distributed memory parallel computer, it is most natural to assign pairs of equations-unknowns the same processor. Thus, each processor will hold a set of equations (rows of the linear system) and a vector of the variables associated with these rows. The assignment of the equation-unknown pairs to processors, can be determined with the help of a graph partitioner or *ad hoc* from knowledge of the problem. Here, it is assumed for simplicity that each processor is assigned only one subgraph (or subdomain, in the PDE literature). This natural way of distributing a sparse linear system is fairly general and is closely related to the physical viewpoint.

Rather than starting from the standard natural ordering used in the sequential setting, it is important to regard the system as a distributed object and try to develop techniques for the global system using the distributed data structure. It is crucial when developing these techniques to set up the local equations as well as the dependencies of local variables from external variables. A preprocessing phase is required to determine this information as well as some other information required during the iteration phase.

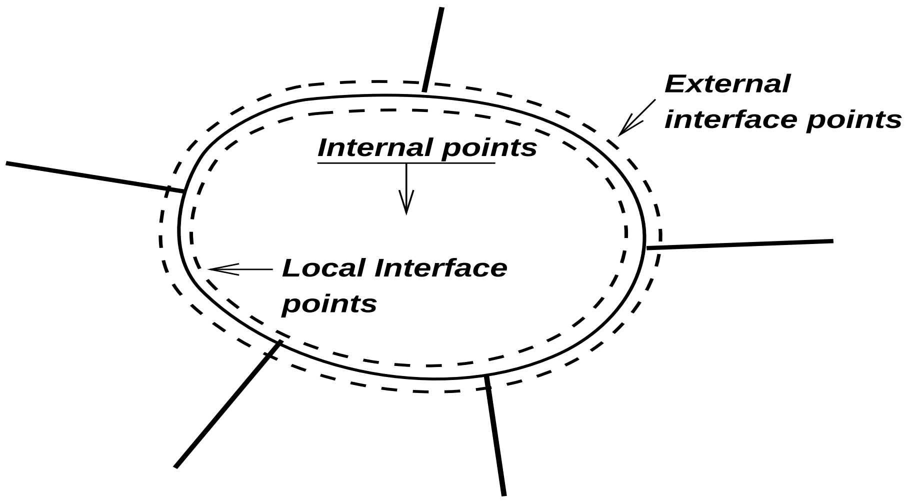

Figure 2: A local view of a distributed sparse matrix.

2.1 The Local Data Structure

Figure 2 is an illustration of the 'physical domain' viewpoint of a sparse linear system adopted in PSPARSLIB. This representation borrows from the domain decomposition literature – so the term 'subdomain' is often used instead of the more proper term 'subgraph'. Each point (node) belonging to a 'subdomain' is actually a pair representing an equation and an associated unknown. It is important to 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

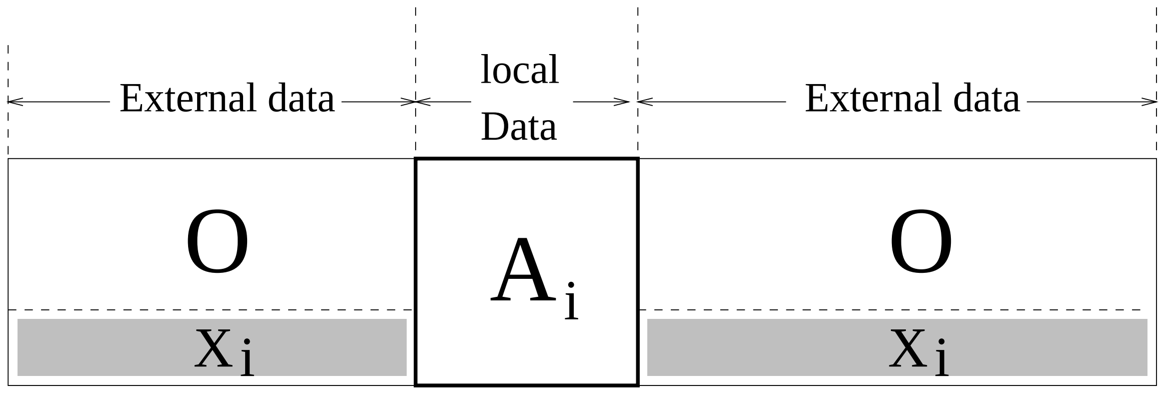

Figure 3: Representation of the local system.

local variables; and (3) External interface variables are those variables in other processors which are coupled with local variables.

Along with this figure, we can represent the local equations as shown in Figure 3. The local equations do not in any way correspond to contiguous equations in the original system. The matrix represented in the figure can be viewed as a reordered version of the equations which uses a local numbering of the equations/unknowns pairs.

As can be seen in the figure, the rows of the matrix assigned to a certain processor, say processor k, have been split into two parts: a *local* matrix A_i which acts on the local variables and an *interface* matrix X_i which acts on remote variables. These remote variables must be first received from other processor(s) before the matrix-vector product can be completed in these processors. A key feature of the data structure is the separation of the boundary points from the interior points. The interface nodes are always listed last after the interior nodes. This 'local ordering' of the data presents several advantages, including more efficient interprocessor communication, and reduced local indirect addressing during matrix-vector products. The zero blocks shown are due to the fact that local internal nodes are not coupled with external nodes.

Thus, each local vector of unknowns x_i is 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,

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

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

$$A_i = \left(\begin{array}{c|c} B_i & E_i \\ \hline F_i & C_i \end{array} \right) \; . \tag{2}$$

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

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

The term $E_{ij}y_j$ is 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 (3) 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 (3). For practical implementations, the subvectors of external interface variables are grouped into one vector called $y_{i,ext}$ and the notation

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

will be used to denote the contributions from external variables to the local system (3). 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 (3) becomes

$$w_i = A_i x_i + X_{i,ext} y_{i,ext} \tag{4}$$

Note that w_i is also the local part the matrix-by vector product Ax in which x is a vector which has the local vector components x_i, $i = 1, \ldots, s$.

To facilitate matrix operations and communication, an important task is to gather the data structure representing the local part of the linear matrix as was just described. In this preprocessing phase it is also important to form any additional data structures required to prepare for the intensive 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".

The complete description of the data structure associated with this boundary information is given in [17] along with additional implementation details.

2.2 Matrix-vector products

Consider now the matrix-vector operation for a distributed matrix. This is an essentially local operation which takes a distributed vector x and produces the result $w = Ax$, distributed conformally to the mapping of all vectors. Each processor i will produce w_i, the local part of the result w. The matrix-vector product is carried out by implementing equation (4). First, the external data $y_{i,ext}$ needed in each processor is obtained. The matrix-vector product with the matrix A_i on the local data x_i can be carried out at the same time that this communication step is being performed. Then the matrix-vector product with the matrix X_i on the external data $y_{i,ext}$ can be carried out and the result is added to the result obtained from $A_i x_i$. Thus, a matrix-vector product can be performed by the following sequence of operations

1. multiply the local matrix A_i by the local variables;
2. receive the external variables;
3. multiply these external variables by the external matrix X_i associated with them and add the result to that obtained from the first multiplication.

Note that steps 1 and 2 can be performed simultaneously. A processor can be multiplying A_i by the local variables while waiting for the external variables to be received. For additional details on the matrix-vector product operation see [17].

3 Distributed Krylov accelerators

The main operations in a standard Krylov subspace acceleration are (1) vector updates, (2) dot-products, (3) matrix-vector products and (4) preconditioning operations. If we exclude the matrix-vector products and preconditioning steps, the rest of the operations in an algorithm such as GMRES are mainly operations to orthogonalize sets of vectors, as well as vector updates. It is assumed that all the vector quantities are split in the same fashion a global SAXPY of two vectors across p processors, consists of p independent saxpy's. In contrast, a global dot product requires a global sum of the separate dot products of the subvectors in each processor. The dot products are mainly used in orthogonalizing sets of Krylov vectors and this constitutes one of the potential bottlenecks in a parallel implementation of GMRES procedures. This is discussed in detail in Section 3.3.

3.1 FGMRES

The main Krylov accelerator used in this paper is the flexible variant of GMRES [18] known as FGMRES [12]. This is a right-preconditioned variant that allows the preconditioning to vary at each step. Since the preconditioning operations require solving systems associated with entire subdomains it becomes important to allow the preconditioner itself to be an iterative solver. This means that the GMRES iteration should allow the preconditioner to vary from step to step within the inner GMRES process. One variant of GMRES which allows this is called the flexible variant of GMRES (FGMRES) [12]. It is derived by observing that in the last step of the standard GMRES algorithm, the approximate solution is formed as a linear combination of the preconditioned vectors $z_i = M^{-1}v_i, i = 1, \ldots, m$, where the v_i's are the Arnoldi vectors [18]. Since these vectors are all obtained by applying the same preconditioning matrix M^{-1} to the v's, we need not save them. We only need to apply M^{-1} to the linear combination of the $v's$. If the preconditioner varies at every step, then we need to save the 'preconditioned' vectors $z_j = M_j^{-1}v_j$ to use them when computing the approximate solution. For further details on the algorithm, see [12].

3.2 Reverse Communication

An additional feature of our implementation of FGMRES is that we use "reverse communication", a mechanism whose goal is to avoid passing data structures to the accelerator. When calling a standard FORTRAN subroutine implementation of an iterative solver, we normally need to pass a list of arguments related to the matrix A and to the preconditioner. This can be a burden on the programmer because of the rich variety of existing data structures. The solution is not to pass the matrices in any form. When a matrix – vector product or a preconditioning operation is needed, the subroutine exits and the calling routine performs the desired operation and then calls the subroutine again, after placing the desired result in one of the vector arguments of the subroutine.

An important consequence of this implementation is that data structures associated with the matrix and preconditioner are not needed in the calling sequence of the FGMRES routine. Among the other operation in FGMRES, only the dot product requires global communication. The dot product operation can be done with a 'global reduction' operation, which is often provided in message-passing communication libraries.

3.3 Arnoldi Orthogonalization

One of the potential bottlenecks in a parallel implementation of GMRES is the orthogonalization of the Krylov vectors in the Arnoldi procedure. As will be seen, the parallel performance of GMRES can be strongly affected by the (parallel) performance of the orthogonalization procedure used, especially as the number of processors increases.

In sequential implementations, orthogonalization is typically carried out by a Modified Gram-Schmidt Orthogonalization (MGSO) procedure which is usually sufficient. To orthogonalize a given vector w against j already orthogonal vectors $v_1, \ldots, v_j$, MGSO uses the following loop,

```
1. for $i = 1, \ldots, j$ do
2.     Compute $h := (w, v_i)$
3.     Compute $w := w - hv_i$
4. EndDo
5. Compute $\|w\|_2$ and define $v_{j+1} := w/\|w\|_2$.
```

It is typical to add a reorthogonalization step when loss of orthogonality is deemed severe based on a test. A well-known difficulty with MGSO in a parallel computing environment is that each of the inner products in line 2, must be done in sequence. Each global inner product requires one global communication and this counts as one synchronization point in the procedure. Adding the inner product in line 5, this means that we have exactly $j + 1$ synchronization points which may be rather expensive, particularly when the vectors are short.

One remedy is to use the Classical Gram-Shmidt (CGSO) Orthogonalization which reduces the number of synchronization points from $j + 1$ to just 2. The main loop is as follows.

```
1. For $i = 1, \ldots, j$ do
2.     Compute $h_{ij} := (w, v_i)$
3  EndDo
4. For $i = 1, \ldots, j$ do
5.     Compute $w := w - h_{ij}v_i$
6. EndDo
7. Compute $\|w\|_2$ and define $v_{j+1} := w/\|w\|_2$.
```

The first of the two synchronization points is associated with the inner products computed by the first loop which can all be computed in parallel. The second is in the inner product in line 7. For the larger Krylov subspace sizes, CGS without reorthogonalization tends to perform poorly. Smaller block sizes could be used to reduce loss of orthogonality but this may slow down convergence for harder, larger, problems. Ultimately, this is linked to the quality of the preconditioner, since a good preconditioner reduces the number of steps to achieve convergence, allowing the use of very small Krylov subspaces.

It is possible to reduce the synchronization points in CGSO further to only one by reordering the computations. This 'synchronized version' will be called Synchronized Classical Gram-Schmidt Orthogonalization (SCGSO). It is based on the observation that the normalization of w in Line 7 of CGSO be postponed until the next step of the algorithm – keeping in mind that the vector v_{j+1} is not orthonormal. The algorithm is as follows.

ALGORITHM **3.1** *SCGSO*

```
1. Compute $h_{ij} := (w, v_i)$ for $i = 1, \ldots j$ and $t_j = (v_j, v_j)$.
2. Do $i = 1, \ldots, j-1$ : $h_{ij} := h_{ij}/\sqrt{t_j}$; Compute $h_{jj} := h_{jj}/t_j$
3. $v_j := v_j/\sqrt{t_j}$
4. For $i = 1, \ldots, j$ do
5.     Compute $w := w - h_{ij}v_i$
6. EndDo
7. Define $v_{j+1} := w$
```

One drawback of the above algorithm is that when reorthogonalization is to be undertaken, then the overhead is high. The vector v_j in Line 1, must be reorthogonalized against all $v_i's$, for $i = 1, \ldots, j-1$. This cost is expected. However, a difficulty is that in the traditional Arnoldi routines, the vector w typically depends on v_j, in fact in the unpreconditioned version, it is simply defined as $w = Av_j$. This would mean that $w = Av_j$ would have to be recomputed as well as its inner products with the v_i's. This is far more expensive than in traditional reorthogonalization procedures. However, in the context of FGMRES, variations in the vector v_j are seen as perturbations which can be viewed as part of the variations in the preconditioner. Also, since we typically perform a small number of steps, reorthogonalization is rarely an issue, and near orthogonality is often sufficient. A few experiments with the various orthogonalization procedures are reported in Section 6.2.

4 Preconditioning

The main preconditioners considered in this paper are based on Domain Decomposition ideas. In Domain Decomposition methods the equations are solved by means of a succession of solutions of local residual equations at each step. Domain Decomposition preconditioners are essentially block preconditioners in which blocking is based on the domains. Variants of block Jacobi and block Gauss-Seidel preconditioners are considered. The Schur complement variants of these techniques will be introduced in a separate section.

4.1 Distributed block Jacobi preconditioning

Figure 2 will be used again to illustrate a few key points. Preconditioners can be derived from Domain Decomposition techniques, the simplest of which is the so-called additive Schwarz procedure. This form of block Jacobi iteration, in which the blocks refer to systems associated with entire domains, is sketched next.

Algorithm **4.1** *Block Jacobi Iteration (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 $x_i = x_i + \delta_i$
```

It is interesting to observe that the required communication, as well as the overall structure of the routine, is identical with that of Matrix – Vector products.

To solve the systems which arise in line 3 of the above algorithm, a standard (sequential) ILUT preconditioner [14] combined with GMRES for the solves associated with the blocks is used. A factor which can affect convergence is the tolerance used for the inner solve. As accuracy increases the number of outer steps may decrease. However, since the cost of each

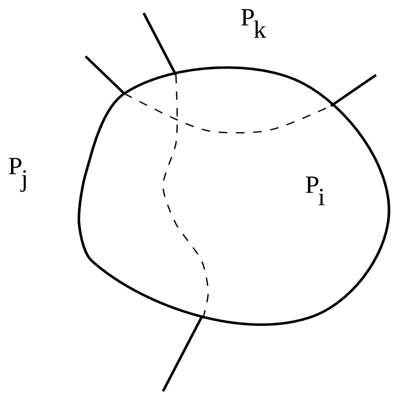

Figure 4: Overlapping domains

inner solve increases, this often offsets any gains made from the reduction in the number of outer steps to achieve convergence. This is illustrated in the numerical experiments section.

Of particular interest in this context are the overlapping Jacobi methods. In the domain decomposition literature [1, 3, 6, 11] it is known that overlapping is a good strategy to reduce the number of steps. There are however several different ways of implementing overlapping block Jacobi iterations. The illustration of Figure 4 will help understand the options. For simplicity, only three subdomains are shown. Some of the overlapping data in domain P_i will have two or three versions. For example the data in the overlapping triangle-shaped subregion will overlap three times and therefore it has three versions, one for P_k, one for P_j and the local version associated with P_i. When exchanging data during the iteration phase, we can either (1) replace the local version of the data by its external version or (2) use some average of the data. The advantage of averaging is that the vectors used to iterate are the same.

4.2 SOR and SSOR preconditioners

A block Gauss-Seidel iteration can be easily carried out as a sequence of annihilations or eliminations of the residual components of the system which are local to the processor. Each elimination provides a correction to these local variables of the unknown vector. These variables are then updated as well as the global residual vector. In order to implement this, all that is required is a global order in which to perform these eliminations as well as some global stopping criterion.

The global ordering can be based on an arbitrary labeling of the processors provided two neighboring domains have a different label. The most common global ordering is a multi-coloring of the domains, which maximizes parallelism [6, 5, 4, 13, 19].

Thus, if the domains are colored and the global ordering of the domains is the ordering defined by the colors, the Gauss-Seidel iteration as executed in each processor would be as follows:

ALGORITHM **4.2** *Multicolor Block Gauss-Seidel Iteration*

```
1. Do $col = 1, \ldots, numcols$
2.     If $(col.eq.mycol)$ Then
3.         Obtain external data $y_{i,ext}$
4.         Update local residual $r_i = (b - Ax)_i$
```

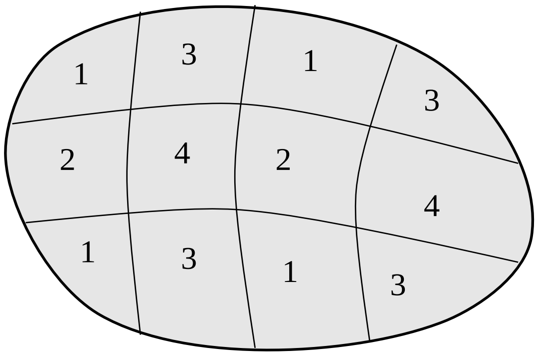

Figure 5: Multicoloring of the subdomains for a multicolor Gauss-Seidel sweep.

```
5.         Solve $A_i \delta_i = r_i$
6.         Update solution $x_i = x_i + \delta_i$
7.     EndIf
```

Algorithm 4.2 is executed on each processor and a convergence test on the global residual or some measure of the error must be included. This block Gauss-Seidel algorithm is the simplest form of the Multiplicative Schwarz procedures used in domain decomposition techniques [2, 9, 10, 7]. Many variations are possible, including overlapping of the domains, inaccurate solves in step 5, inclusion of a relaxation parameter ω, etc.

Normally, after a step is done with an active color, the processors of this active color need only send data to the (inactive) neighboring processors. They need not receive any new data from them since their interface data have not changed (and they themselves will become inactive in the next color step). This can reduce communication times at the expense of a more complicated code. Our implementations do not take advantage of coloring for communication, i.e., after each color step, all boundary data is exchanged.

One problem with multicoloring is that as the domains associated with a given color is active, all other colors will be inactive. As a result it is typical to obtain only $1/numcol$ efficiency if *numcol* is the number of colors. To alleviate this problem somewhat, we can further block the local variables into two blocks: interior and interface variables. Then the global SOR iteration is performed with this additional blocking.

In effect, each local matrix A_i is split as

$$A_i = \begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix} = \begin{pmatrix} B_i & O \\ O & C_i \end{pmatrix} + \begin{pmatrix} 0 & E_i \\ F_i & 0 \end{pmatrix} \tag{5}$$

The B_i part corresponds to internal nodes. The global diagonal blocking is now associated with the block diagonal matrices in the first part of the right-hand side of (5). In a block-Gauss-Seidel iteration, the equations associated with the interior variables are solved first. Then a loop similar with the color loop of Algorithm 4.2 takes place for the interface variables only. This 2-level block block Gauss-Seidel iteration is as follows,

Algorithm **4.3** *Two-level Gauss-Seidel iteration*

```
1. Solve $B_i \delta_{i,u} = r_{i,u}$
2. $u_i := u_i + \delta_{i,u}$
3. Do $col = 1, \ldots, numcols$
4.     If $(col.eq.mycol)$ Then
```

```
5.          Obtain external data $y_{i,ext}$
6.          Update y-part of residual $r_{i,y}$
7.          Solve $C_i \delta_{i,y} = r_{i,y}$
8.          Update interface unknowns $y_i = y_i + \delta_{i,y}$
9.      EndIf
```

In our current implementation the exchange of data is the same as the one for multicolor SOR and for block Jacobi.

The advantage of this procedure over the standard multicolor Gauss-Seidel iteration is that the bulk of the computational work in each domain, which corresponds to the solves with the internal variables, is done in parallel. Loss of parallelism comes from from the color loop which involves only solves with interfaces, which is of lower complexity.

5 Schur complement techniques

Schur complement techniques refer to methods which iterate on the interface unknowns only, implicitly using internal unknowns as intermediate variables. A global system involving these interface unknowns can be easily obtained by eliminating internal variables from the local equations, see [16]. Here we will focus on a general strategy for deriving Schur complement techniques associated with arbitrary global fixed point iterations.

Consider the simplest case of a block-Jacobi iteration described earlier. The Schur complement system is derived by eliminating the variable u_i from the system (3) extracting from the first equation $u_i = B_i^{-1}(f_i - E_i y_i)$ which yields, upon substitution in the second equation,

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

in which S_i is the 'local' Schur complement:

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

The equations (6) for all subdomains i altogether constitute a system of equations which involves only the interface points y_j, $j = 1, 2, \ldots, s$ and which has a natural block structure associated with these vector variables. The diagonal blocks in this system, namely the matrices S_i, are dense in general but the off-diagonal blocks E_{ij} are sparse. As is known, with a consistent choice of the initial guess, a block-Jacobi (or Gauss-Seidel) iteration with the reduced system is equivalent with a block Jacobi iteration on the global system, see, e.g., [16]. A block Jacobi iteration on the global system takes the following local form:

$$\begin{aligned}
x_i^{(k+1)} &= x_i^{(k)} + A_i^{-1} r_i^{(k)} \\
&= x_i^{(k)} + A_i^{-1} \left(b_i - A_i x_i^{(k)} - \begin{pmatrix} 0 \\ \sum_{j \in N_i} E_{ij} y_j^{(k)} \end{pmatrix} \right) \\
&= A_i^{-1} \begin{pmatrix} f_i \\ g_i - \sum_{j \in N_i} E_{ij} y_j^{(k)} \end{pmatrix} \\
&= \begin{pmatrix} * & * \\ -S_i^{-1} F_i B_i^{-1} & S_i^{-1} \end{pmatrix} \begin{pmatrix} f_i \\ g_i - \sum_{j \in N_i} E_{ij} y_j^{(k)} \end{pmatrix}
\end{aligned}$$

Here a $*$ denotes a nonzero block whose actual expression is unimportant. The important observation is that the y iterates satisfy an independent relation of the form,

$$y_i^{(k+1)} = S_i^{-1}\left[g_i - F_i B_i^{-1} f_i - \sum_{j\in N_i} E_{ij} y_j^{(k)}\right] \tag{8}$$

If we call g_i' the right-hand side of the reduced system (6) then (8) can be rewritten as

$$y_i^{(k+1)} = y_i^{(k)} + S_i^{-1}\left[g_i' - S_i y_i^{(k)} - \sum_{j\in N_i} E_{ij} y_j^{(k)}\right] \tag{9}$$

which is nothing but a Jacobi iteration on the Schur complement system.

In summary, the sequence of the y-part of the Jacobi vectors on the global system can be viewed as a sequence of Jacobi iterates for the Schur complement system. A similar result holds for the Gauss-Seidel iteration as well. From a global viewpoint, we have a *primary* iteration for the global variable of the form,

$$x^{(k+1)} = Mx^{(k)} + c \tag{10}$$

and the vectors of interface variables y associated with these iterates satisfy an iteration of the form,

$$y^{(k+1)} = Gy^{(k)} + h\ . \tag{11}$$

The matrix G is not known explicitly but it is easy to advance the iteration by one step from an arbitrary (starting) vector v, meaning that it is easy to compute $Gv + h$ for any v.

Now the idea is to accelerate the sequence $y^{(k)}$ with a Krylov subspace algorithm such as GMRES. One way to look at this acceleration procedure is that we are attempting to solve the system

$$(I - G)y = h \tag{12}$$

To solve the above system with a Krylov-type method an initial guess and corresponding residual are needed. Also, the Krylov iteration requires a number of matrix-vector product operations. The right-hand side h can be obtained from one step of the iteration (11) computed for the initial vector 0, i.e.,

$$h = (G \times 0 + h)$$

Given the initial guess $y^{(0)}$ the initial residual $s^{(0)} = h - (I - G)y^{(0)}$ can be obtained from

$$s^{(0)} = h - (y^{(0)} - Gy^{(0)}) = y^{(1)} - y^{(0)}$$

Matrix-vector products with $I - G$ can be obtained from one step of the original iteration. To compute $w = (I - G)y$ proceed are as follows,

1. Perform one step of the primary iteration $\binom{u'}{y'} = M\binom{0}{y} + c$;
2. set $w := y'$;
3. Compute $w := y - w + h$

This strategy allows to derive a Schur complement technique for any primary fixed-point iteration on the global unknown. Among the possible choices are the Jacobi, and SOR iterations, with and without overlap, as well as iterations derived (somewhat artificially) from ILU preconditioning techniques. The main advantages of this viewpoint are the generality and flexibility of the formulation.

6 Numerical Experiments

In this section, we report on some results obtained for solving distributed sparse linear systems on an IBM SP2 with 10 nodes, an IBM cluster of 8 workstations, and an SGI Challenge cluster. The IBM SP2 available to us has a maximum of 10 processors. The SGI challenge workstation cluster consists of three 4-processor Challenge L workstations one 8-processor Challenge XL workstation. The processors on the Challenge L and XL are the same (R4400) but the memory configurations are different. Communication between different SGI cluster workstations (resptively IBM RS workstations) is performed with a Fibre-Channel switch (respectively, an ATM switch) using the MPI communication library. The SP-2 nodes communicate with a High-Performance switch.

6.1 The test problems

Seven test matrices have been used, with sizes ranging from fairly small to large. Table 1 shows the sizes and number of nonzero elements of these matrices along with some information on their pattern. A 'Sym' symbol indicates that the sparsity pattern is symmetric and a 'NonSym' symbol indicates a nonsymmetric pattern. When the pattern of a matrix is nonsymmetric then its symmetrized version is used for the purpose of partitioning. The matrices are sometimes scaled (rows then columns are scaled by their 2-norms) and this is also indicated on the 'Scaling' column. The first two of these matrices are from the Harwell-Boeing collection. The EX matrices are from the FIDAP subcollection of matrices and the last two are from a NASA subcollection [1].

Matrices	Dimension	NNZ	Scaling	Pattern
Harwell-Boeing:				
PORES2	1224	9613	No	NonSym
SHERMAN5	3312	20793	No	NonSym
FIDAP:				
ex20.mat	2203	69981	Yes	Sym
ex27.mat	974	40782	Yes	Sym
ex37.mat	3565	67591	No	Sym
SIMON:				
RAEFSKY3	21200	1488768	No	Sym
VENKAT01	62424	1717792	No	Sym

Table 1: The test matrices

All problems are tested in the following manner. The matrix is read on one workstation. Then the graph partitioning is done on this workstation and the matrix and right-hand side are then distributed. The distributed sparse system solver is then invoked to solve the resulting distributed sparse linear system.

6.2 Experiments with Arnoldi Orthogonalization

Table 2 compares MGSO, CGSO, and SCGSO on the SGI cluster with two different machine configurations. In the four workstations case, using four processors means a configuration

[1]All these matrices are available via anonymous ftp

of one processor per machine and using eight processors means a configuration of two processors per machine.

1 workstation				
Matrix	**Procs**	**MGSO**	**CGSO**	**SCGSO**
RAEFSKY3	1	107.8	107.8	109.5
	4	87.3	53.1	46.7
	8	102.6	44.8	36.0
VENKAT01	1	335.1	357.0	358.2
	4	170.2	116.5	111.5
	8	138.8	80.1	69.3
4 workstations				
RAEFSKY3	1	107.8	107.8	109.5
	4	167.3	110.5	70.9
	8	184.6	117.0	67.1
VENKAT01	1	335.1	357.0	358.2
	4	261.5	272.0	225.8
	8	208.8	159.4	96.1

Table 2: MGSO, CGSO, and SCGSO on the SGI cluster

These examples show that when m is large and more processors are used then the Modified Gram Schmidt Orthogonalization (MGSO) may generate long synchronization delays and lead to poor performance. They also indicate that the impact may be more damaging for small size problems. The Synchronized Classical Gram-Schmidt orthogonalization procedure (without reorthogonalization) is the overall winner. However, if a large Krylov subspace dimension must be used, the compromise offered by the classical Gram-Schmidt with reorthogonalization may be safer. Reorthogonalization may be performed only when needed, as determined by a test suggested by Daniel et al. [8].

6.3 Different cluster configurations

When a fixed number of processors (e.g. 16) is available, there are many different ways to configure a cluster. We can put all 16 processors into one box or have 8 in each of 2 boxes or 4 in each of 4 boxes. It is expected that computations done in the same box will be faster. However, it is important to have an idea on how much is lost when a computation is done across different boxes versus when it is done on a single computer. Many factors are at play, including the speed of the internal bus and the memory configuration (4-way interleave versus 2-way interleave for example), the speed and latency of the high-speed network, etc.. The experiments which follow give one such comparison. When preconditioning is used, for different numbers of processors, different preconditioners result. In obtain to obtain comparable data, we use GMRES without preconditioning.

Table 3 shows the performance of GMRES(15) for the matrices RAEFSKY3 and VENKAT01. The header 'SGI k Wst' indicates the type and number of workstations used. The processors indicated in the next line are divided equally (when applicable) among the workstations. For example, for the '4 Wst' configuration with 8 processors, each workstation has 2 processors. and for the '4 Wst' configuration with 4 processors, each workstation has 1

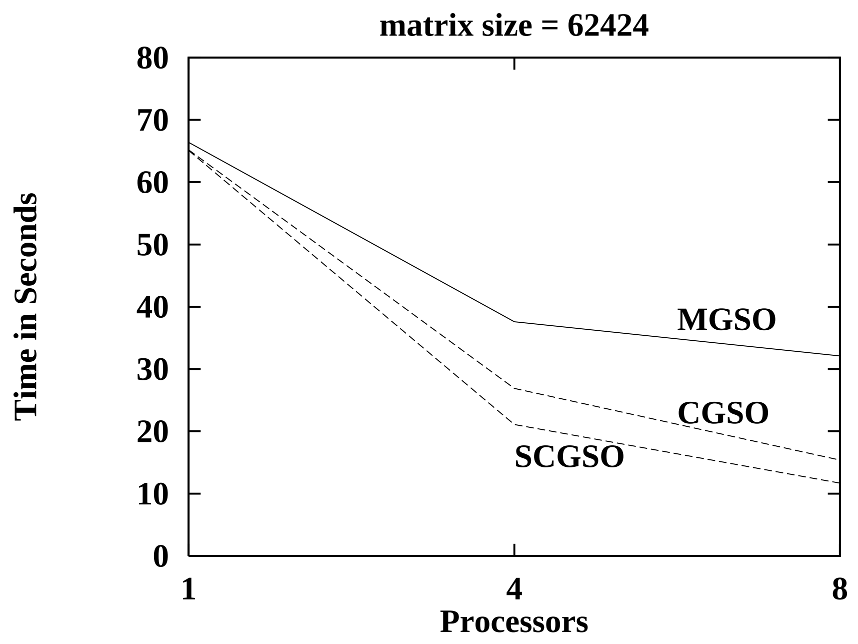

Figure 6: MGSO, CGSO, and SCGSO on IBM SP2

processor. The table also shows the speed-up and efficiency for the non-trivial cases of 4 and 8 processors. As can be seen, there is a loss of efficiency when going out of a workstation. The loss is not too damaging for the larger problems but other experiments show that it could be dramatic for smaller problems. The efficiency of 57% is achieved on 8 nodes on the SP2 as opposed to 40% on 8 processors distributed on 4 Challenges, and 70% on a single box 8-processor Challenge.

	SGI 1 WSt			SGI 4 WSts			IBM SP2		
	1	4	8	1	4	8	1	4	8
RAEFSY3.	598.8	176.7	105.4	598.8	238.8	184.3	118.1	40.4	25.5
Speed-up:		3.38	5.67		2.50	3.24		2.92	4.62
Efficiency:		84%	70%		62%	40%		73%	57%
VENKAT01.	957.3	278.7	160.0	957.3	455 .6	249.9	190.1	73.3	41.7
Speed-up:		3.43	5.98		2.10	3.83		2.59	4.55
Efficiency:		85%	74%		52%	47%		64%	56%

Table 3: Execution times, speed-ups, and efficiencies for performing 1500 steps of GMRES with various cluster configurations

6.4 Experiments with block Jacobi preconditioning

We now compare the results obtained with the distributed block Jacobi preconditioner using the three different overlapping options as described in Section 4.1. These results are summarized in Table 4. In the table, Jaco_no stands for block Jacobi with no overlapping, Jaco_ov_av for block Jacobi with overlapping and averaging of the overlapping data, and Jaco_ov for block Jacobi with overlapping and exchange of overlapped data. In the table *its* is the number of FGMRES iterations. The comparison shows that overlapping can reduce the total number of iterations. A comparison of the results shows that Jaco_ov is usually faster than the other two options.

Matrix	**Method**	**Number of Processors**									
		1		2		4		8		16	
		its	time	its	time	its	time	its	time	its	time
SHERMAN5	Jac_no	6	0.42	6	0.42	19	1.16	31	2.32	80	27.08
	Jac_ov_av	7	0.50	7	0.49	12	0.72	16	1.44	23	8.10
	Jac_ov	7	0.49	7	0.49	12	0.70	16	1.22	30	10.4
RAEFSKY3	Jac_no	4	12.7	9	19.4	11	11.7	13	6.6	8	3.6
	Jac_ov_av	4	14.9	5	10.4	6	8.72	7	6.26	6	3.98
	Jac_ov	4	14.8	5	9.71	7	8.16	7	4.05	7	3.22
VENKAT01	Jac_no	5	30.7	13	45.3	14	24.0	16	15.2	16	12.0
	Jac_ov_av	5	30.9	8	28.4	9	28.8	11	16.8	11	12.7
	Jac_ov	5	31.0	9	30.8	9	16.4	11	11.3	11	8.25

Table 4: Comparison of FGMRES with distributed block Jacobi preconditioner and three different domain overlapping strategies

The next results are obtained on the IBM SP2. Table 5 shows how requiring more accuracy in the inner solver can affect the outer solver. Both number of steps and time are shown. The main conclusion from the table is that in most cases, it does not seem to pay to perform more inner iterations.

Matrix size	inner its	**Number of Processors**			
		1	2	4	8
		its time	its time	its time	its time
ex37.mat	1	5 0.17	5 0.15	6 0.14	7 0.25
	3	5 0.14	5 0.13	6 0.11	7 0.12
	5	5 0.14	5 0.13	6 0.11	7 0.11
RAEFSKY3	1	9 4.52	8 2.26	7 1.14	7 0.73
	3	6 3.92	5 1.75	7 1.60	6 0.71
	5	4 2.77	4 1.59	7 2.07	6 0.91
	10	4 3.96	4 2.08	7 2.39	6 1.31

Table 5: Comparison on the IBM SP2, of various choices for the number of steps in the inner solver

6.5 Experiments with block Gauss-Seidel preconditioner

Table 6 shows a comparison of multicolor Gauss-Seidel and block Jacobi preconditioners on the IBM SP-2. On each subdomain the systems are solved ILU solves, with LU factors obtained from an ILUT factorization. The msor_s preconditioner refers to the two-level multicolor Gauss-Seidel preconditioner in which the interface data is solved for after the

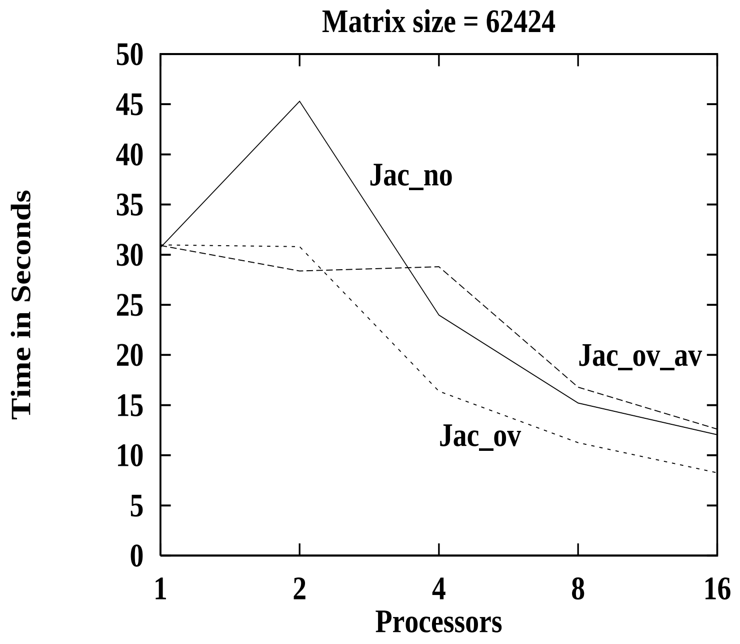

Figure 7: Timing for overlapped block Jacobi on the SGI cluster

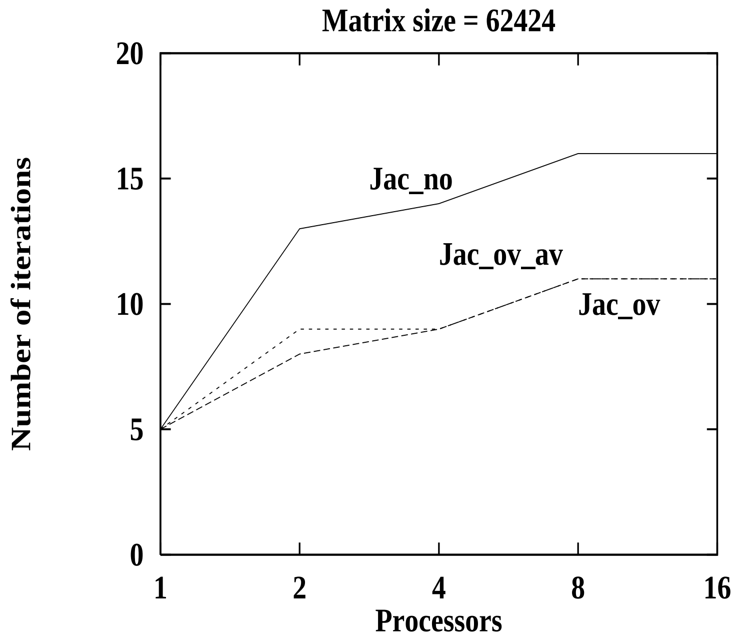

Figure 8: Iteration count for overlapped block Jacobi on the SGI cluster

interior data. As can be seen this approach is very close to the block Jacobi iteration. As was already explained, the Multicolor SOR scheme used here involves a high of idle time corresponding to the fact that one color is active at any one time.

Matrix	**Method**	**Number of Processors**							
		1		2		4		8	
		its	time	its	time	its	time	its	time
RAEFSKY3	Jac_ov	11	11.71	10	5.68	10	3.46	10	1.88
	msor	11	9.16	10	6.42	12	5.80	10	3.27
	msor_s	11	9.20	10	4.89	10	3.47	10	2.66
VENKAT01	Jac_ov	20	30.05	20	15.45	21	9.42	25	6.27
	msor	20	23.84	21	18.32	22	16.96	25	10.63
	msor_s	20	23.91	21	13.74	22	10.60	24	6.73

Table 6: Block Jacobi versus Multicolor Gauss-Seidel on the IBM SP-2.

6.6 Experiments with Schur complement techniques

Before discussing the results obtained with the Schur complement technique, it is worth pointing out that these techniques are often implemented in conjunction with direct solvers. These solvers are invoked either to compute the actual Schur complement matrix, in forming the Schur complement system, or for solving the successive linear systems which arise during the iterative process. If an iterative process is to be used instead of a direct solver, it is important to note that the solves involved with the Schur complement iteration process must be accurate. This represents the main weakness of Schur complement techniques. The experiments we performed confirm this fact. As is shown in Table 7, the Schur complement approach is not competitive with the standard primary preconditioners from which they are defined. In the table 'itsgmr=1' represents a scheme in which only one step of the primary iteration is performed, while 'itsgmr=5' represents the scheme in which five (at most) steps are taken. The 'fgmr' columns represent the times spent in FGMRES alone and 'tot' is the total time. It is observed that there are significant savings in the orthogonalization times (FGMRES) due to the shorter vectors involved in the FGMRES iteration. However these times are only a small percentage of the total time. In fact, in the Schur complement techniques, most of the time is spent in the accurate solutions related to the primary preconditioning scheme.

6.7 A performance comparison on different machines

Figure 9 shows the times achieved for various machines to solve a linear system with the VENKAT01 matrix. We must mention that the y coordinate in the plot is not to scale. The Cray time and the Sun time are provided only as reference points. For example, the time achieved on the CRAY C-90 is on one processor and *the code has not been optimized for vectorization.* Since the ILUT code used is fairly scalar in nature, it is not too surprising that one IBM RS 6000 processor achieves a similar speed. The performances shown for the SGI cluster are for all processors in one workstation. Notice how the gains in speed-up become smaller of the SP-2 after the number of processors exceeds 4. One the IBM workstation cluster the gains are not as good initially but improve steadily as the number

	block Jacobi						Schur block Jacobi					
	itsgmr = 1			itsgmr = 5			itsgmr = 1			itsgmr = 5		
Procs	its	fgmr	tot	its	fgmr	tot	its	fgmr	tot	its	fgmr	tot
1	13	.52	10.05	5	.122	6.67	1	.0	4.94	1	.0	21.38
2	12	.29	5.57	5	.07	4.04	10	.016	30.23	3	.043	45.38
4	12	.19	3.27	6	.059	2.90	11	.061	19.30	3	.024	27.29
8	17	.19	2.65	9	.067	2.53	12	.066	11.87	4	.025	19.94
10	16	.13	1.97	10	.073	2.21	11	.069	7.32	4	.036	12.22

Table 7: Schur complement techniques for the test matrix VENKAT01 on the IBM SP2

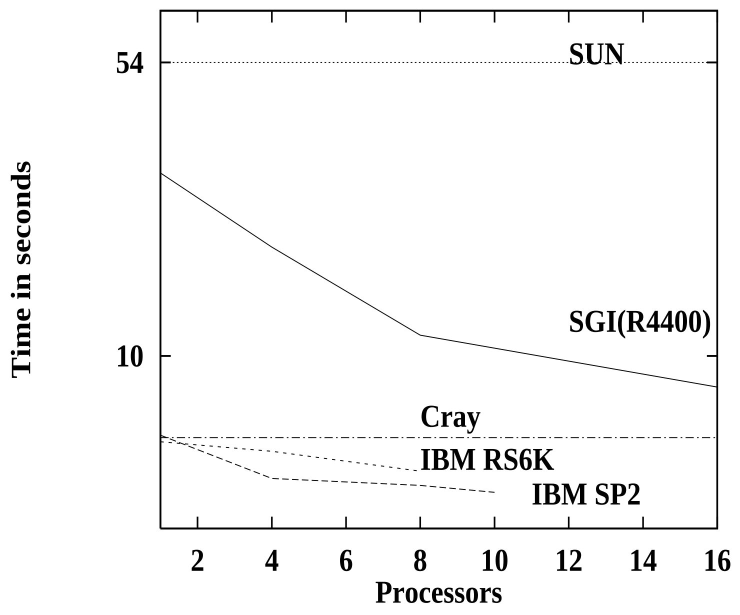

Figure 9: Performance comparison for different machines

of processors increases. Overall, the SP-2 achieves a fairly good performance for the solution of unstructured sparse linear systems of equations.

6.8 Where is the time spent?

Figures 10 presents a breakdown of the times spent in a typical solution. The results are for the matrix VENKAT01, using a relative tolerance of $\epsilon = 10^{-6}$ and a Krylov subspace dimension of $m = 15$. The figure shows that all contributions decrease as the number of processors increases at the exception of the FGMRES time (dominated by orthogonalization) which moves up slightly in going from 8 to 16 processors.

7 Conclusion

Thanks to the availability of communication libraries such as MPI and PVM, and inexpensive network technologies, workstation clusters have recently become one of the most

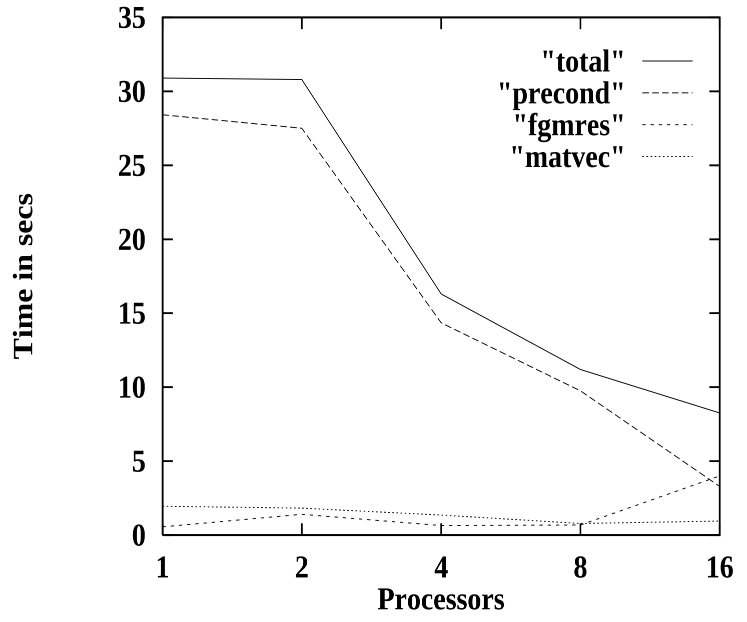

Figure 10: Contributions to total execution time for a distributed block Jacobi preconditioner with overlapping

promising paradigms for high performance computing. Our main conclusion from the experiments in this work is that workstation clusters can be effectively used to solve very large sparse linear systems. Small systems can be handled more efficiently on a single workstation. The break-even point depends on many factors and is a function of the architecture parameters and the iterative solution techniques used. For example, any improvements in latency and bandwidth will allow us to solve smaller problems more efficiently, thus moving down the break-even point. For very large problems, communication becomes less of an issue as it is small relative to computation. For these problems, avoiding idle time and achieving a more effective utilization of the memory and the processors becomes more important.

Of all the preconditioning options we have tried, the overall winner is the overlapping Additive Schwarz (overlapping block Jacobi). Within each subdomain an effective iterative solver can be used. However, we found that using one step of an ILU solve with an accurate ILUT factorization is usually less expensive. Other preconditioning options exist, see e.g. [15], which have not been tested here. As the number of processors increases these alternatives may become preferable.

References

[1] P. E. Bjørstad. Multiplicative and Additive Schwarz Methods: Convergence in the 2 domain case. In Tony Chan, Roland Glowinski, Jacques Périaux, and O. Widlund, editors, *Domain Decomposition Methods*, Philadelphia, PA, 1989. SIAM.

[2] P. E. Bjørstad and O. B. Widlund. Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM Journal on Numerical Analysis*, 23(6):1093–1120, 1986.

[3] P. E. Bjørstad and O. B. Widlund. To overlap or not to overlap: A note on a domain decomposition method for elliptic problems. *SIAM Journal on Scientific and Statistical Computing*, 10(5):1053–1061, 1989.

[4] X. C. Cai and Y. Saad. Overlapping domain decomposition algorithms for general sparse matrices. *Numerical Linear Algebra with Applications*, 1996. To appear.

[5] X. C. Cai and O. Widlund. Multiplicative Schwarz algorithms for some nonsymmetric and indefinite problems. *SIAM Journal on Numerical Analysis*, 30(4), August 1993.

[6] Xiao-Chuan Cai, William D. Gropp, and David E. Keyes. A comparison of some domain decomposition and ILU preconditioned iterative methods for nonsymmetric elliptic problems. *J. Numer. Lin. Alg. Appl.*, June 1993. To appear.

[7] T. F. Chan and T. P. Mathew. Domain decomposition algorithms. *Acta Numerica*, pages 61–143, 1994.

[8] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization. *Math. Comput*, 30:772–795, 1976.

[9] Maksymilian Dryja and Olof B. Widlund. Towards a unified theory of domain decomposition algorithms for elliptic problems. In Tony Chan, Roland Glowinski, Jacques Périaux, and Olof Widlund, editors, *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations, held in Houston, Texas, March 20-22, 1989.* SIAM, Philadelphia, PA, 1990.

[10] Maksymilian Dryja and Olof B. Widlund. Some recent results on Schwarz type domain decomposition algorithms. In Alfio Quarteroni, editor, *Sixth Conference on Domain Decomposition Methods for Partial Differential Equations.* AMS, 1993. Held in Como, Italy, June 15–19,1992. To appear. Technical report 615, Department of Computer Science, Courant Institute.

[11] William D. Gropp and Barry F. Smith. Experiences with domain decomposition in three dimensions: Overlapping Schwarz methods. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1992. To appear in the Proceedings of the Sixth International Symposium on Domain Decomposition Methods.

[12] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific and Statistical Computing*, 14:461–469, 1993.

[13] Y. Saad. Highly parallel preconditioners for general sparse matrices. In G. Golub, M. Luskin, and A. Greenbaum, editors, *Recent Advances in Iterative Methods, IMA Volumes in Mathematics and Its Applications*, volume 60, pages 165–199, New York, 1994. Springer Verlag.

[14] Y. Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[15] Y. Saad. ILUM: a parallel multi-elimination ILU preconditioner for general sparse matrices. *SIAM Journal on Scientific Computing*, 1996. To appear.

[16] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS publishing, New York, 1996.

[17] 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, Sept. 1995*, 1995.

[18] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[19] J. N. Shadid and R. S. Tuminaro. A comparison of preconditioned nonsymmetric krylov methods on a large-scale mimd machine. *SIAM J. Sci Comput.*, 15(2):440–449, 1994.