Krylov Subspace Methods in Distributed Computing Environments

Yousef Saad\textsuperscript{1}

\textit{Department of Computer Science, University of Minnesota, Minneapolis, USA}

1 Introduction

In the last few years, substantial progress has been made in replacing direct solution methods for solving the large sparse linear systems that arise from Fluid Dynamics equations by iterative solvers. Indeed, a general current consensus seems to be that for three-dimensional models and 2-dimensional PDE’s with several degrees of freedom per grid point, iterative methods are essentially mandatory, mainly because of the enormous memory requirements of direct methods in such cases. Among standard iterative methods, those based on Krylov subspace techniques coupled with suitable preconditioners are generally considered to offer the best compromise between efficiency and robustness. In addition to their overwhelming advantage over direct methods, in terms of memory and computational cost, iterative methods are also attractive because of the simplicity with which they can be adapted to high performance computers.

Parallel computing in various forms is progressively emerging as the only viable means of making acceptable gains in computational speeds in the future. As models are becoming more complex, the demand on computational speeds is unsustainable with standard technologies. Parallel processing comes with new challenges as well as difficult choices to make. In particular, one important issue which is still currently debated is related to the choice of the programming paradigm and that of an architectural model. Specifically, there is little consensus as to whether MIMD distributed memory message passing models are to be favored or whether we can do as well with data parallel (SIMD) models. One possible trend that seems to be emerging is that a combination of both paradigms may be desirable. One should be able to exploit the power of individual processors, which often are vector chips (Connection Machine CM-5 and Cray’s projected MPP machine) or processor arrays (Connection Machine CM-2). These local computations are best expressed in FORTRAN-90/ HPFF style FORTRAN, for example. At the same time, one would like to be able to benefit from

\textsuperscript{1}This research was supported by NIST under grant number 60NANB2D1272 and by the Army Research Office under grant number DAAL03-89-C-0038.
the flexibility provided by the message passing paradigm, which allows the exchange of data between processors or clusters of processors. In the sequel we will refer to the message passing paradigm as “Distributed Computing”. This includes not only multiprocessor computers such as the CM-5, but also networks of heterogeneous computers.

As was already mentioned the main advantage of distributed computing is the flexibility that it provides. One of its disadvantages, is that it is generally not suitable for fine grain parallelism because of the usually high latency times required to move data. Another disadvantage is that communication between processors must be coded explicitly which results on an additional burden to the programmer. A number of numerical techniques in CFD can take advantage of Distributed Computing, but not of data parallel computing. Perhaps the best illustration of this is when using a domain decomposition approach. An irregular domain can be partitioned into a number of subdomains, usually one subdomain per processor, and then some iterative technique can be used over the global domain to solve the original equations. The iterative method will only require exchange of interface values between neighboring subdomains and some exchange of global information such as (global) dot-products for example.

In this paper we will consider the issues raised in a practical implementation of a distributed Krylov subspace solver and provide some solutions. Some of the techniques and tools described here are in their early stages of development. The main approach used to develop a distributed iterative solver is essentially domain decomposition. However, as will be explained, domain decomposition is far more general than just a successful technique for solving Partial Differential Equations. If we take a slightly more general viewpoint, essentially using graph representations, we find that the domain decomposition concept can be helpful in developing such tools as ‘distributed’ Incomplete LU factorizations for general sparse matrices, as well as distributed sparse direct solvers. In this paper we will restrict ourselves to discussing the main ideas in this context. Readers are referred to [3] and [16] for additional details.

2 Preconditioned Krylov Subspace Methods

We consider a linear system of the form

\[ Ax = b, \]  

where \( A \) is a large sparse nonsymmetric real matrix of size \( N \). A number of projection processes on so-called Krylov subspaces have been proposed in recent years to solve such systems, see for example, [1, 7, 9, 8, 11, 10, 14, 22, 20, 21]. Thus, the GMRES algorithm introduced in [21] for solving general sparse nonsymmetric linear systems of equations 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 \( b - Ax_0 \).
Preconditioning consists of transforming the above system, e.g., by multiplying it through with a certain matrix \( M^{-1} \), into one that will be easier to solve by a Krylov subspace method, i.e., one which is likely to require fewer steps to converge. Thus, when the preconditioner \( M \) is applied to the right, we will solve instead of (1), the preconditioned linear system
\[
(AM^{-1})(Mx) = b.
\]

It is important in many applications to be able to allow the preconditioner to vary from step to step within the inner GMRES process [15]. Such a variant is derived by simply 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 [21]. 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. The resulting ‘flexible’ variant of GMRES is described below.

**Algorithm 2.1 Flexible variant of preconditioned GMRES (FGMRES)**

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 \( j = 1, \ldots, m \) do
      
      • Compute \( z_j := M_j^{-1}v_j \)
      
      • Compute \( w := Az_j \)
      
      • For \( i = 1, \ldots, j \), do
         \[
         \begin{cases} \hspace{1cm} h_{i,j} := (w, v_i) \\ w := w - h_{i,j}v_i \end{cases}
         \]
      
      • Compute \( h_{j+1,j} = \|w\|_2 \) and \( v_{j+1} = w/h_{j+1,j} \).
   
   (c) Define \( Z_m := [z_1, \ldots, z_m] \).

3. **Form the approximate solution:** Compute \( x_m = x_0 + Z_m y_m \) where
   \( y_m = \arg\min_y \|\beta e_1 - \bar{H}_m y\|_2 \) 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 simply constructs an orthogonal basis of the preconditioned sub-space \( \text{Span}\{v_1, AM_i^{-1}v_1, \ldots, AM_m^{-1}v_m\} \) 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 \( M_j = M \) for \( j = 1, \ldots, m \) then the method is clearly 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 \( x_0 + \text{Span}\{Z_m\} \), [15]. 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.

There are many possible applications of the added flexibility provided by FGMRES. 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 domain decomposition methods or in any parallel computing implementation. FGMRES even allows the inner preconditioning steps to be completely asynchronous, a feature which can help minimize communication and synchronization costs in a parallel approach.

To further enhance flexibility, our implementation of FGMRES includes an additional feature referred to as a “reverse communication mechanism” whose goal is to avoid data structures. 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 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-vector product or a preconditioning operation is needed, we can simply exit the subroutine and have the subroutine caller perform the desired operation and call the subroutine again, after placing the desired result in one of the vector arguments of the subroutine. We need a code parameter to help determine the type of operation that is requested by FGMRES. Thus, a typical execution of a flexible GMRES routine with reverse communication would be as follows:

```plaintext
  icode = 0
  continue
  call fgmres (n,im,rhs, sol, i, vv, w, wk1, wk2, eps, maxits, iout, icode)
  if (icode .eq. 1) then
    call precon(n, wk1, wk2) <--- user's preconditioning operation
    goto 1
  else if (icode .eq. 2) then
    call matvec (n,wk1, wk2) <--- user's 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 $w_{k1}$, put the result in $w_{k2}$ and call FGMRES again. If it is equal to two then we need to multiply the vector $w_{k1}$ by the matrix $A$, then put the result in $w_{k2}$ 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 in $w_{k2}$ in 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.

3 Implementation on Distributed Computers

We start with a Partial Differential Equation problem on a domain which has already been decomposed in some desirable way into a number of subdomains as is illustrated in figure 3.1. Here it is assumed that a domain decomposition tool has been used to partition the domain into $s$ subdomains of roughly the same size. Given the potentially very complex and large meshes, it is important to design software tools to perform this partitioning ‘automatically’ and this issue will examined in Section 4. Let us assume that each subdomain is assigned to a different processor. On some machines it is important to assign the subdomains carefully so as to minimize communication costs in the iteration phase. This subdomain to processor mapping must again be done automatically for efficiency, but we will not examine this issue in this paper.
Once the decomposition is achieved and the mapping completed, we would like to solve the original equations involving the nodes in all subdomains. A typical iterative approach is to use a preconditioned Krylov subspace process which will require the following operations during the iteration phase.

1. Vector updates and dot-products;
2. Matrix vector multiplications;
3. Preconditioning operations.

We will assume that all vectors are partitioned and mapped in the same manner, all variables associated with a given mesh point being assigned to the same processor to which this mesh point is assigned in the domain decomposition mapping illustrated above. We now consider the above operations in turn.

### 3.1 Vector updates and dot products

Vector updates are operations of the form
\[ y(1:n) = y(1:n) + a \cdot x(1:n) \]

where \( a \) is a scalar and \( y \) and \( x \) two vectors. Since the vectors \( x \) and \( y \) are distributed in the same manner among the processors, this vector operation will be translated into \( p \) independent vector updates, requiring no communication. Specifically, if \( nloc \) is the number of variables local to a given processor, this processor will simply execute a vector loop of the form

\[ y(1:nloc) = y(1:nloc) + a \cdot x(1:nloc) \]

and all processors will execute a similar operation simultaneously.

Unlike vector updates, dot products require global communication between processors. We would like first to be more specific regarding the desired functionality of the ‘distributed dot product’ operation. We need to compute the inner product \( t = x^T y \) of two distributed vectors \( x \) and \( y \) and then make the result \( t \) available in each processor, since this result is typically needed to perform vector updates in each node. For a large number of processors, this sort of operations may be expensive in terms of communication costs. However, many parallel computer manufacturers have become aware of their importance and are starting to provide hardware and software support for performing global reduction operations efficiently. These reduction operations include global sums, global max/min calculations, etc.. A commonly adopted calling convention is to provide one subroutine for all these operations, e.g., ‘reduce’, and pass the type of operation (add, max, min, multiply, ...) as a parameter. With this, a distributed dot-product function can be programmed roughly as follows.

```fortran
function distdot(nloc, x, incx, y, incy)
    integer nloc
    real*8 x(nloc), y(nloc)
    tloc = DDOT(nloc, x, incx, y, incy)
    distdot = REDUCE(tloc,'add')
    return
endfunction
```

In the above function ddot performs the usual BLAS-1 dot product operation. The reduce operation is called with ‘add’ as the parameter for the operation-type, and as a result it will sum all the variables ‘tloc’ from each processor and put the result in the variable distdot in each processor.

At this point we can make the following important observation. If reverse communication is used, no matrices are explicitly invoked in the FGMRES routine and the only difference between a sequential and a parallel implementation of FGMRES lies in the dot product. A standard dot-product (function ddot) is to be replaced by a global one (function distdot). As was shown above the calling sequences associated with the two types of dot products are identical. As a result, the parallel and the sequential versions of FGMRES will be very similar. This is achieved by (1) hiding the communication involved in the dot product from the user and (2) hiding the data structures related to the matrices thanks to the reverse communication implementation.
3.2 Matrix – Vector products

We begin by describing the data structure we need in each processor in order to be able to perform a global matrix-vector product for a distributed sparse matrix.

\[ A_{loc} = \]

\[ \begin{array}{c}
\text{Internal points } (x_{\text{int}}) \\
\text{Local interface points } (x_{\text{int}})
\end{array} \]

\[ B_{ext} = \]

\[ \text{External interface matrix} \]

**Figure 3.2** The local matrices and data structure associated with each subdomain.

We first need to determine how to map the matrix elements to the processors. Our simplest option is to duplicate the matrix in each processor but this is clearly wasteful. Instead, we can simply assume that the matrix is distributed row-wise across the processors according the distribution of the variables, i.e., row number \( i \) is located in the same processor where the unknown \( i \) has been assigned. In order to perform a matrix-vector product, we need to operate with these rows on the local variables and a few variables that are external to the processor. These external variables correspond to interface points belonging to neighboring subdomains. In fact we can decompose the matrix-by-vector product into two operations, one involving only the local variables and the other involving external variables. We thus have two local matrices to construct in the preprocessing phase. A square matrix \( A_{loc} \) which involves the local variables, and a matrix \( B_{ext} \) which involves the external points. To perform a global matrix-vector product, each processor must obtain the external variables from the neighboring processors in a certain order, multiply them by the matrix \( B_{ext} \) and add the result to that obtained from the multiplication by \( A_{loc} \). With this decomposition the global matrix – vector product can be implemented as indicated in Algorithm 3.1 below. In what follows, \( x_{loc} \) is a vector of variables that are local
to a given processor. The components corresponding to the local interface points, ordered to be the last components in $x_{loc}$ in our data structure, are called $x_{bnd}$. The external interface points, listed in a certain order, constitute a vector which is called $x_{ext}$. The matrix $A_{loc}$ is a sparse $nloc \times nloc$ matrix which represents the restriction of $A$ to the local variables $x_{loc}$. The matrix $B_{ext}$ operates on the external variables $x_{ext}$ to give the correction which must be added to the vector $A_{loc}x_{loc}$ in order to obtain the desired result $(Ax)_{loc}$.

**Algorithm 3.1 Distributed Sparse Matrix Product Kernel**

1. **Communicate:** Exchange interface data. Scatter $x_{bnd}$ to neighbors and gather $x_{ext}$ from neighbors.

2. **Local matrix – vector product:** $y = A_{loc}x_{loc}$

3. **External matrix – vector product:** $y = y + B_{ext}x_{ext}$.

We should point out that steps 1 and 2 are independent and can be overlapped. A preprocessing routine is required to obtain the matrices $A_{loc}$ and $B_{ext}$ and determine the interface points from the neighboring processors, in the order in which $B_{ext}$ will operate on them. Note also that the matrix by vector products in steps 2 and 3 can use any convenient data structure that will improve efficiency by exploiting the local architecture. For this reason, the work being done elsewhere on defining a user-level sparse BLAS can be fully exploited [5].

### 3.3 Preconditioning operations

With the domain-decomposition data mapping outlined above, the usual ILU factorization cannot easily be exploited. Fortunately, several alternatives exist and we would like to describe just a few of them.

**ILU for subdomain ordered equations.** The only reason why the usual ILU is not easy to implement for the domain decomposition data mappings described before, is that we often use an ordering of the equations that is inconsistent with the mapping. Often, ILU factorizations are defined for the original natural ordering of the equations. However, if we reorder the equations in such a way that the unknowns in the same processor are number contiguously, then the ILU factorization corresponding to this ordering is quite easy to implement. Indeed let us conceptually number the equations so that all the *internal points* of all subdomains are numbered first, followed by all interface points. We list the internal points of Subdomain 1, followed by those of Subdomain 2, ..., followed by those of Subdomain $s$. When all internal points have been listed we list the interface points of Subdomain 1, followed by those of Subdomain 2, etc. With this numbering, the global matrix will have the following structure, for the case $s = 4$. 


\[ A = \begin{pmatrix}
A_1 & F_1 \\
A_2 & F_2 \\
A_3 & F_3 \\
A_4 & F_4 \\
E_1 & E_2 \\
E_3 & E_4 \\
D & D
\end{pmatrix} \]  

(3)

It is simple to see that if we wish to perform an ILU factorization on this matrix, we must start by performing an ILU factorization for each of the local matrices \( A_i \), which can be done independently and without any communication. The rest of the factorization will require already computed rows of \( L \) and \( U \) from other processors. The implementation of a corresponding ILU(0) causes no major difficulty. We note that since the matrix has been reordered, the quality of this preconditioner may be quite different from that associated with the original ordering. However, it has been our experience that when more fill-in is used the preconditioner improves and the difference in quality narrows, and may even reverse in favor of the more parallel version [19].

**Multi-color \( k \)-step SOR/SSOR iterations** There are a few simple greedy heuristic which allow to color the subdomains in such a way that no two neighboring subdomains share the same color. This is a graph coloring problem and the reader is referred to [17], for example, for algorithms and some references. Once the subdomains have been colored one can use a form of Multi-Color SOR or SSOR relaxation to precondition the global equations. In each processor, a one-step SOR iteration takes the form

**Algorithm 3.2 Multi-color SOR preconditioning**

For \( k = 1, 2, \ldots, n\text{colors} \) Do
If \( k = \text{color(mynode)} \) then
\[ x_{\text{loc}}^k := x_{\text{loc}}^{k-1} + A_{\text{loc}}^{-1} (b - Ax_{\text{loc}}^{k-1}) \]
endif
enddo

In the above algorithm, \( \text{mynode} \) represents the processor number in which the code is being executed. Also, \( (b - Ax)_{\text{loc}} \) represents the local part of the global residual vector associated with the approximation \( x \). In fact the calculation of \( (b - Ax)_{\text{loc}} \) will require a global matrix-by-vector product which was described in the previous section. The superscript \( k \) related to the color indicates that we cannot modify \( x_{\text{loc}} \) at step \( k \) before we have completed the modification at step \( k - 1 \). An \( s \) step SOR iteration would simply consist of adding an outer loop to the above algorithm.

In [2] an idea based on multicolor SOR was studied. It was termed ‘a multiplicative’ Schwartz procedure and was found to be one of the best ‘practical’ preconditioners tested by the authors. Note that we can also define a block-Jacobi - type preconditioner similarly and that overlapping domains can also be used to enhance the performance of such preconditioners [2].

10
Preconditioned iterations for interface points  An interesting observation which can be made is that it is possible to write equations for interface points alone. One advantage of this approach is that the memory required in the Krylov subspace technique can be dramatically reduced because there are usually far fewer interface points than there are internal points. Since local memory availability is often tight in multiprocessor systems there is certainly a big appeal in using FGMRES on the interface points only rather than all the points. Let \( A_{loc,i} \) be the local matrix residing in processor \( i \) as defined before and let us block-partition it according to the internal variables \( x_{int} \) followed by the (local) interface variables \( x_{bnd} \):

\[
A_{loc,i} = \begin{pmatrix}
A_i & B_i \\
C_i & D_i
\end{pmatrix}
\]  

(4)

If we call \( x_i \) the vector of the internal variables, and \( y_i \) the vector of the interface variables, we can write all local equations as follows.

\[
A_i x_i + B_i y_i = f_i \\
C_i x_i + D_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i
\]

The term \( E_{ij} y_j \) is the contribution to the equation from the neighboring subdomain number \( j \) and \( N_i \) is the set of subdomains that are neighbors to subdomain \( i \). Assuming that \( A_i \) is nonsingular, we can eliminate the variable \( x_i \) from this system by extracting from the first equation \( x_i = A_i^{-1}(f_i - B_i y_i) \) which yields, upon substitution in the second equation,

\[
S_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i - C_i A_i^{-1} f_i
\]

(5)

in which \( S_i \) is the ‘local’ Schur complement:

\[
S_i = D_i - C_i A_i^{-1} B_i
\]

(6)

The equations (5) 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. We need to find ways of preconditioning such a system. We start by noting that the multicoloring SOR or SSOR idea described in the previous subsection can be exploited. In fact, the same graph coloring can be used since interface points are subsets of the local points. In the course of a multicolor Block-SOR iteration we only need to know how to solve a linear system with the diagonal block \( S_i \). For this purpose it is helpful to interpret the Schur complement. Let us call \( P \) the canonical injection matrix from the local interface points to the local nodes. If we have \( n_i \) points
locally and if $m_i$ is the number of the local interface points, $P$ is an $n_i \times m_i$ matrix whose columns are the last $m_i$ columns of the $n_i \times n_i$ identity matrix. Then it is easy to see that

$$S_i = (P^T A_{i}^{-1} P)^{-1}.$$  

(7)

If $A_{\text{loc},i} = LU$ is the LU factorization of $A_{\text{loc},i}$ then it can be easily verified that

$$S_i^{-1} = P^T U^{-1} L^{-1} P = P^T U^{-1} P P^T L^{-1} P,$$

(8)

which indicates that in order to operate with $P^T L^{-1} P$ we simply need to solve with the last $m_i \times m_i$ principal submatrix of $L$. Similarly for $P^T U^{-1} P$ which requires only a back-solve with the last $m_i \times m_i$ principal submatrix of $U$. Therefore, we only need the LU factorization of $A_{\text{loc},i}$ in order to be able to solve a system with the matrix $S_i$. More interestingly we can exploit approximate solution methods associated with incomplete factorizations of $A_{\text{loc},i}$.

4 Tools for Automatic Domain Decomposition

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 decompose a domain and map it into processors, as well as to set up the various data structures. As was mentioned earlier, it is quite important to be able to perform such tasks automatically. One of the basic tasks to be performed is to find a partition array, i.e., an array $(p_1, p_2, \ldots, p_n)$ in which $p_i$ is the subdomain number to which variable $i$ belongs. Ideally, the subdomains should have roughly equal size, although criteria of load balancing other than subdomain size can also be used. Among the tools that must be developed in any software library for implementing domain decomposition algorithms we can mention: (1) partitioning a domain (in fact a graph) into $s$ subgraphs of similar size; (2) building a coarse mesh given the original fine mesh (see justification later); (3) finding a good subdomain to processor mapping (architecture dependent), once the partition array is found; and (4) coloring the subdomains such that two neighboring subdomains have different colors.

Next, we describe a simple algorithm for parallel automatic domain decomposition. Recall that a level-set traversal usually starts with a vertex and then defines the next level-set to be traversed as the set of all unmarked neighbors of the vertices in the current level-set. The vertices in a level set are marked as soon as they are visited. We can now generalize this concept slightly. First we can start with not only one node but several independent vertices at once. This will be useful if the initial set of points are spread apart. The second modification to the traversal is that the mark that is put on each node that has been visited is now a label. In addition the vertices will inherit the labels of their parents in the traversal process.

Instead of describing the fully parallel processes, i.e., the host program and the node program, it is much simpler to present a version that is sequential with respect to the domains. The parallel version is easily derived from this sequential algorithm.
\textbf{Algorithm 4.1 Automatic Domain Decomposition: greedy approach}

1. \textbf{Start:}
   Find an initial set of `coarse mesh' vertices $v_1, \ldots, v_{\text{ndom}}$
   For $i = 1, 2, \ldots, \text{ndom}$ Do $\text{label}(v_i) := i$.
   Define $\text{levset} := \{v_1, \ldots, v_{\text{ndom}}\}$ and $\text{nodes} = \text{ndom}$
2. \textbf{Loop: While (nodes < n) Do}
   $\text{Next levset} = \emptyset$
   For each $v_j$ in $\text{levset}$ Do
     for each neighbor $v_k$ of $v_j$ s.t. $\text{label}(v_k) = 0$ Do
       $\text{Next levset} := \text{Next levset} \cup \{v_k\}$
       $\text{label}(v_k) := \text{label}(v_j)$
       $\text{nodes} = \text{nodes} + 1$
     EndFor
   EndFor
   $\text{levset} := \text{Next levset}$
   EndWhile

The algorithm starts with one node in each processor then expands by adding level-sets until all points are labeled. At the end, all nodes having the same label will constitute a subdomain. We must assume here that the initial graph is connected or that there is at least one starting node in each connected component. The parallel version of this algorithm consists of assigning each starting node to a different processor, then expanding the level sets independently. At some point there will be conflicts, i.e., two processors will attempt to ‘acquire’ the same node which belongs to two level sets originating from two different starting nodes. In such cases, the host program must arbitrate. Our current implementation simply uses a first-come first served rule, but there are several possible improvements which are not considered here. The process is illustrated in figure 4.1 for a $15 \times 15$ 5-point grid. We point out that even though the grid is regular, this is not exploited by our program.
Figure 4.1 The graph decomposition algorithm for a $15 \times 15$ mesh.
The success of the procedure rests on a good distribution of the starting nodes. We wish to have points that are uniformly spread-out within the whole mesh or graph. If a coarse mesh is already available from the discretization then the nodes of this mesh can be taken as the initial nodes. Otherwise, a procedure for automatically obtaining a good initial distribution of points can be used. One such procedure has been recently developed and tested. The procedure recursively finds independent sets and defines coarser meshes on them. We will give a brief description of one step of the recursive coarsening process. First, we find a maximal independent set in the graph \([18]\). Referring to the adjacency graph \( G = (V, E) \) of the matrix, an independent set \( S \) is a subset of the vertex set \( V \) such that

\[
\text{if } x \in S \text{ then } (x, y) \in E \text{ or } (y, x) \in E \Rightarrow y \notin S
\]

This is to say that elements of \( S \) are not allowed to be coupled with other elements of \( S \) by incoming or outgoing edges. Finding such sets is relatively easy. An independent set is maximal if it cannot be augmented by elements in its complement to form a larger independent set. Independent set orderings have been mainly used for deriving parallel direct solution techniques for sparse linear systems\([12, 13, 4]\) and multifrontal techniques\([6]\) can be viewed as a particular case.

A simple greedy procedure for finding an independent set \( S \) is to essentially traverse the nodes in a given order and if a node is not already marked we select it as a new member of \( S \) and then mark it and all its nearest neighbors.

**Algorithm 4.2 Greedy algorithm for independent set ordering**

1. Let \( S = \phi \).
2. For \( j = 1, 2, \ldots, n \) Do:
   If node \( j \) is not marked then
   \( S = S \cup \{j\} \).
   Mark \( j \) and all its nearest neighbors.
endif
enddo

Here the nodes are traversed in the natural order \( 1, 2, \ldots, n \) but we can traverse them in any permutation \( \{i_1, \ldots, i_n\} \) of \( \{1, 2, \ldots, n\} \). Heuristics are proposed in\([18]\) to find good orders of traversal, i.e., traversals that yield large independent sets.

Once an independent set is found we need to define a coarse mesh on the set \( S \). This is simply a graph that connects the vertices in \( S \) by attempting to translate the couplings that existed in the finer mesh. In words, we connect two coarse points if there is a path of length two in the original mesh that connects these points. This briefly describes one step of the coarsening procedure. The procedure is repeated recursively until we are satisfied with the number of coarse points obtained. For example, in Figure 4.1 the 10 mesh points used to start the partition procedure illustrated there are obtained from three coarsening steps. The initial matrix has size \( n = 225 \), then the first coarse grid matrix has size \( n_1 = 113 \) the second has size \( n_2 = 32 \) and finally the last has size \( n_3 = 10 \).
5 Conclusion

We have given an overview of some of the implementation and algorithmic issues related to the development of preconditioned Krylov subspace methods on Distributed Computing environments. We have advocated the use of Krylov subspace techniques that allow variable preconditioners and we favored reverse communication implementations for added flexibility. In addition, we have also adopted the concept of ‘graph decomposition’ as a general technique for performing distributed sparse matrix computations. Software tools to aid in implementing such techniques have been developed and we have shown a few of them. As was demonstrated in the previous section, techniques from computer science, specifically graph theory, can be extremely helpful in developing efficient heuristics in the general context of parallel algorithms. If parallel processing is to be successful in an area such as CFD, it is vital that automatic tools based on such heuristics be developed. Another aspect which we touched upon was the development of alternative preconditioners. We have described just a few such preconditioners but clearly much remains to be done as the traditional efficient preconditioning techniques are essentially not applicable.

References


17