Toward the Design of a Parallel, Linear Algebraic System

W. D. Turner and J. E. Flaherty

Scientific Computation Research Center
and Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180

Abstract

We describe extensions to general, serial linear algebraic systems sufficient to provide for their use in a parallel, iterative context. Some base assumptions about the structure of the underlying problem and problem domain, as well as the initial distribution of the parallel structures are described to provide a framework for the discussion of implementation details; however, it is felt that the design is sufficiently flexible that it could easily be applied to other implementations.

Key words: *linear algebra, iterative methods, generic programming, parallel computation*

1 Introduction

The design of a mathematical algorithm can be both time consuming and difficult. Care must be taken to ensure that the computer program accurately reflects the underlying mathematics, is robust over the allowable inputs, and is efficient. Large-scale projects often represent an investment of months or years. Despite this effort, programs are often fragile and require significant rework for even minor changes to the underlying assumptions or data structures.

As an example, consider the field of computational linear algebra regarding the solution of the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}.$$

For small problems, the coefficient matrix $\mathbf{A}$ can easily be stored as a dense, two-dimensional array of values and the solution $\mathbf{x}$ obtained by direct methods. For larger problems, especially those arising from a discretization process such as finite elements, a banded or skyline matrix form may be more appropriate for use with a direct method. As the problem grows larger, an iterative method may become the most efficient solution strategy. Larger still and a preconditioner may be necessary with attendant changes to the underlying matrix form. When the problem size becomes large enough, parallel computation may be needed for performance, and various parallel data distributions have to be considered. Each of these changes generally implies a separate cycle of implementation and testing.

Changing system size is not the only problem. Interfacing discretization code of one design with algebraic solution code of another can be a challenge. Even use of an established solution library such as LAPACK [1] or PETSc [2], can involve either writing a translation between formats, or a rewrite of the discretization code to conform to the new interface. The added effort does not further the scientific goals, but merely serves to enhance the efficiency of the underlying linear algebra.

In this paper we address some of these issues with a focus on the design of parallel, iterative, linear algebraic solvers that are independent of the underlying serial data structures and parallel data distribution. In this, our approach is similar to that of the Matrix Template Library (MTL) [12, 13]; however, our design provides a mathematical basis for the analysis of parallel algorithms. It is flexible and allows any matrix or vector type written to the standard to be interchanged within a parallel framework. Moreover, given a parallel data distribution and a "connector" class that defines the data communication pattern, our design provides transparent or nearly transparent transition from serial to parallel execution. Finally, while our design focuses on parallel issues, the standards are applicable to serial problems and serial data structures written to the standard are also interchangeable.

We define our assumptions for the parallel data distribution in term of a mathematical framework in Section 2. We use this framework to analyze several of the building blocks for iterative algorithms in Section 3. In Section 4, we broadly define a set of parallel operations sufficient to implement the analysis and in Section 5 we complete the definition by specifying the necessary routines. Finally, in Section 6 we offer some closing thoughts and suggestions for future investigations.

2 Preliminary Considerations

Consider a matrix

$$\mathbf{A} = \mathbf{A}_1 + \mathbf{A}_2 + \ldots + \mathbf{A}_p, \tag{1}$$

where each of the component matrices $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_p$ are resident on a different processor $k \in [1, \ldots, p]$ of a parallel computer. Each array $\mathbf{A}_k$ typically arises from a discretization process on a spatially distributed mesh. As such, we expect each array to be sparse and computable using only local operations. The component matrices have not been reorganized, and, in particular, $\mathbf{A}_i \cap \mathbf{A}_j \neq \emptyset$ if $\Omega_i \cap \Omega_j \neq \emptyset$, where Ω_k is the portion of spatial domain Ω resident on processor k.

Further, consider a parallel vector

$$\mathbf{x} = \mathbf{x}_1 \cup \mathbf{x}_2 \cup \ldots \cup \mathbf{x}_p$$

which is distributed across the processors in the same manner as $\mathbf{A}$. Assume that any index r which is nonzero in two vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, $i \neq j$, has the same value in both. With this assumption, $\cup$ can be formally defined for an arbitrary component $y[r]$ of $\mathbf{y} = \mathbf{x}_i \cup \mathbf{x}_j$ as

$$y[r] = \begin{cases} x_i[r], & \text{if } x_j[r] = 0 \\ x_j[r], & \text{otherwise} \end{cases}.$$

As a simple example consider a two-dimensional physical domain distributed across two processors as shown in Figure 1. The degrees of freedom (DOFs) at element vertices are indexed as shown. The nonzero structure of the corresponding stiffness matrix $\mathbf{A}$ can be expressed in terms of the component

matrices as

$$
\mathbf{A} = \begin{bmatrix}
x & x & x & & & & x & x & \\
x & x & x & & & & & & \\
x & x & x & & & & & x & x \\
 & & & y & y & y & y & y & \\
 & & & y & y & y & & & \\
 & & & y & y & y & & y & y \\
x & & & y & & & z & z & \\
x & & x & y & & y & z & z & z \\
 & & x & & & y & & z & z
\end{bmatrix} = \mathbf{A}_1 + \mathbf{A}_2 =
$$

$$
= \begin{bmatrix}
x & x & x & & & x & x & \\
x & x & x & & & & & \\
x & x & x & & & & x & x \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
x & & & & & x & x & \\
x & & x & & & x & x & x \\
 & & x & & & & x & x
\end{bmatrix} + \begin{bmatrix}
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & y & y & y & y & y & & \\
 & y & y & y & & & & \\
 & y & y & y & & y & y & \\
 & y & & & y & y & & \\
 & y & & y & y & y & y & \\
 & & & y & & y & y &
\end{bmatrix}.
$$

Each x represents a contribution that can be calculated using only the elements resident on processor 1; each y represents a contribution that can be calculated using only the elements resident on processor 2; and each z represents a contribution that can only be be calculated as the sum of x and y components form processor 1 and processor 2, respectively.

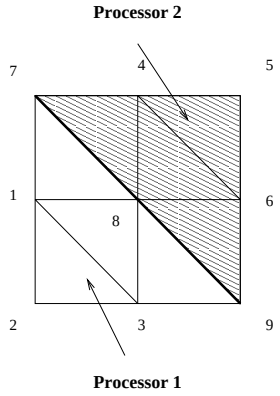

Figure 1: Two-dimensional finite element mesh on two processors.

The component matrices $\mathbf{A}_1$ and $\mathbf{A}_2$ are such that their row and column indices can easily be divided into two sets corresponding to those entries which must be zero and those which may be nonzero. Let n be the dimension of $\mathbf{A}$, and let n_{nz_k} be the number of nonzero rows (columns) in each $\mathbf{A}_k$, $k = 1, \ldots, p$. In a fashion similar to [4] we define an $n \times n_{nz_k}$ *condenser matrix* $\mathbf{C}_k$ such that

$$
\tilde{\mathbf{A}}_k = \mathbf{C}_k^T \mathbf{A}_k \mathbf{C}_k,
$$

where the matrices $\tilde{\mathbf{A}}_k$ contain the (possibly permuted) nonzero rows and columns of $\mathbf{A}_k$. Since $\mathbf{C}_k$ is a permutation matrix,

$$\mathbf{A}_k \quad = \quad \mathbf{C}_k \tilde{\mathbf{A}}_k \mathbf{C}_k^T.$$

For our two-processor example of Figure 1

$$\mathbf{C}_1 \quad = \quad \begin{bmatrix} 1 & & & & & & & & \\ & 1 & & & & & & & \\ & & 1 & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & 1 & & & & \\ & & & & & 1 & & & \\ & & & & & & 1 & & \end{bmatrix},$$

and

$$\tilde{\mathbf{A}}_1 \quad = \quad \begin{bmatrix} x & x & x & x & x & \\ x & x & x & & & \\ x & x & x & & x & x \\ x & & & x & x & \\ x & & x & x & x & x \\ & & x & & x & x \end{bmatrix}.$$

Typically, the matrices $\mathbf{C}_k$ are not generated as their action on the component matrices $\mathbf{A}_k$ can be obtained by renumbering the DOFs locally prior to assembly. A simple mapping between the new, local numbering and the original, global numbering is then sufficient for conversion between the two systems. Formally, we can re-write (1) as

$$\mathbf{A} \quad = \quad \mathbf{C}_1 \tilde{\mathbf{A}}_1 \mathbf{C}_1^T + \mathbf{C}_2 \tilde{\mathbf{A}}_2 \mathbf{C}_2^T + \ldots + \mathbf{C}_p \tilde{\mathbf{A}}_p \mathbf{C}_p^T.$$

Using this notation, we can also define the sequence of vectors

$$\mathbf{x}_k \quad = \quad \mathbf{C}_k \mathbf{C}_k^T \mathbf{x}, \quad 1 \leq k \leq p,$$

from global vector $\mathbf{x}$, where the sequence $\mathbf{x}_k$ meets our requirements for a distributed vector. Applying $\mathbf{C}_1$ above to a vector $\mathbf{x}$ gives

$$\mathbf{x}_1 = \mathbf{C}_1 \mathbf{C}_1^T \mathbf{x} = \begin{bmatrix} \alpha & \alpha & \alpha & 0 & 0 & 0 & \beta & \beta & \beta \end{bmatrix}^T.$$

We can define three regions of $\mathbf{x}_k$ corresponding to those DOFs internal to Ω_k (α region), the components completely external to Ω_k (zero region), and those components in the intersection of Ω_k and one or more sub-regions Ω_i, $1 \leq i \leq p, i \neq k$ (β region). Further, using the condenser matrices $\mathbf{C}_k$ we can define

$$\tilde{\mathbf{x}}_k = \mathbf{C}_k^T \mathbf{x}_k = \mathbf{C}_k^T \mathbf{C}_k \mathbf{C}_k^T \mathbf{x} = \mathbf{C}_k^T \mathbf{x}, \quad 1 \leq k \leq p,$$

which removes the zero region from the distributed vector. For our example,

$$\tilde{\mathbf{x}}_1 \quad = \quad \mathbf{C}_1^T \mathbf{x} = \begin{bmatrix} x & x & x & y & y & y \end{bmatrix}^T.$$

For use in vector-vector operations, it is also convenient to define a second set of operators $\mathbf{O}_k$, $1 \leq k \leq p$, such that

$$\begin{aligned} \hat{\mathbf{x}}_k &= \mathbf{O}_k^T \mathbf{x}, & (2) \\ &= \mathbf{O}_k^T \mathbf{C}_k \mathbf{C}_k^T \mathbf{x}, & (3) \\ &= \mathbf{O}_k^T \mathbf{C}_k \tilde{\mathbf{x}}_k. & (4) \end{aligned}$$

Further,

$$\mathbf{x} = \sum_{i=1}^{p} \mathbf{O}_i \hat{\mathbf{x}}_i, \quad (5)$$

and

$$\mathbf{O}_i \hat{\mathbf{x}}_i \bigcap \mathbf{O}_j \hat{\mathbf{x}}_j = \emptyset, \quad i \neq j. \quad (6)$$

Whereas the matrices $\mathbf{C}_k$ selected all elements of a vector $\mathbf{x}$ corresponding to DOFs resident on processor k, the matrices $\mathbf{O}_k$ select a subset of the vector elements to which they have been granted "ownership." For all DOFs in the interior of a sub-domain Ω_i, ownership of the corresponding vector element is granted to processor i. For all DOFs on the boundary between l processors $i_1, i_2, \ldots, i_l$, ownership is granted to exactly one processor i_k, $1 \leq k \leq l$.

For the example, we could define

$$\mathbf{O}_1 = \begin{bmatrix} 1 & & & & & & & & \\ & 1 & & & & & & & \\ & & 1 & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & 1 & & & \\ & & & & & & & & \\ & & & & & & & & \end{bmatrix}, \quad \mathbf{O}_2 = \begin{bmatrix} & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ 1 & & & & & & & & \\ & 1 & & & & & & & \\ & & & 1 & & & & & \\ & & & & & & & & \\ & & & & & & 1 & & \end{bmatrix},$$

where processor 1 has been granted ownership of DOFs $\{1, 2, 3, 7\}$; and processor 2 has been granted ownership over DOFs $\{4, 5, 6, 8, 9\}$. Other divisions of the boundary DOFs $\{7, 8, 9\}$ are equally valid, including granting ownership of all boundary DOFs to a single processor.

Finally, we define two additional operators designed to extract those DOFs that are on the boundary between two or more processors. Let the matrix $\mathbf{B}_i$ be defined to extract those DOFs which are on the boundary of processor i and all other processors j , $j = 1, \ldots, n$, $j \neq i$, and let $\bar{\mathbf{B}}_i$ extract those DOFs in the interior of processor i.

Using the notation of (2) through (6), we have

$$\begin{aligned} \mathbf{x}_{\mathbf{B}_k} &= \mathbf{B}_k^T \mathbf{x}, = \mathbf{B}_k^T \mathbf{C}_k \mathbf{C}_k^T \mathbf{x}, = \mathbf{B}_k^T \mathbf{C}_k \tilde{\mathbf{x}}_k, \\ \mathbf{x}_{\bar{\mathbf{B}}_k} &= \bar{\mathbf{B}}_k^T \mathbf{x}, = \bar{\mathbf{B}}_k^T \mathbf{C}_k \mathbf{C}_k^T \mathbf{x}, = \bar{\mathbf{B}}_k^T \mathbf{C}_k \tilde{\mathbf{x}}_k, \end{aligned}$$

and

$$\mathbf{x} = \sum_{i=1}^{p} \mathbf{B}_i \mathbf{x}_{B_k} + \sum_{i=1}^{p} \bar{\mathbf{B}}_i \mathbf{x}_{\bar{\mathbf{B}}_k}.$$

Several observations can be made regarding the matrices $\bar{\mathbf{B}}_i$, $\mathbf{B}_i$, and $\mathbf{C}_i$:

- The interior DOFs on one processor cannot be interior DOFs on a different processor, giving

$$\bar{\mathbf{B}}_i^T \bar{\mathbf{B}}_j \quad = \quad \mathbf{0}, \quad i \neq j. \tag{7}$$

- The interior DOFs on one processor cannot be boundary DOFs on any processor, giving

$$\bar{\mathbf{B}}_i^T \mathbf{B}_j \quad = \quad \mathbf{0}, \tag{8}$$
$$\mathbf{B}_i^T \bar{\mathbf{B}}_j \quad = \quad \mathbf{0}. \tag{9}$$

- Since the DOFs local to a given processor i must contain both the DOFs on the boundary between i and other processors, as well as those DOFs on the interior of i,

$$\mathbf{C}_i \mathbf{P}_i \quad = \quad \left[\begin{array}{c|c} \bar{\mathbf{B}}_i & \mathbf{B}_i \end{array} \right], \tag{10}$$
$$\mathbf{C}_i \quad = \quad \left[\begin{array}{c|c} \bar{\mathbf{B}}_i & \mathbf{B}_i \end{array} \right] \mathbf{P}_i^T, \tag{11}$$

  where $\mathbf{P}_i$ is an appropriately sized permutation matrix chosen to reorder the columns of $\mathbf{C}_i$.

Again, using our example, we have

$$\mathbf{B}_1 = \mathbf{B}_2 = \begin{bmatrix} & & \\ & & \\ & & \\ & & \\ 1 & & \\ & 1 & \\ & & 1 \end{bmatrix}, \quad \bar{\mathbf{B}}_1 = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \\ & & \\ & & \\ & & \\ & & \end{bmatrix}, \quad \bar{\mathbf{B}}_2 \quad = \quad \begin{bmatrix} & & \\ & & \\ & & \\ 1 & & \\ & 1 & \\ & & 1 \\ & & \end{bmatrix},$$

3 Analysis

Our motivation is multilevel, preconditioned, iterative algebraic solution procedures for adaptive discretization methods on nonuniform domains [15]. When developing parallel algebriac solution procedures, we identify five operations that must be efficient:

1. a matrix-vector product,
2. a vector-vector dot product,
3. a saxpby,
4. a sub-matrix approximate solve [15], and
5. the generation of coarse levels for spatially refined domains.

The first four tasks are analyzed in terms of the parallel distribution described in Section 2. The fifth task involves considerations of the mesh distribution and parallel spatial structures that will be considered in a separate analysis [16].

3.1 Parallel Matrix-Vector Product

Consider the parallel evaluation of

$$\mathbf{y} \;=\; \mathbf{A}\mathbf{x} \tag{12}$$

or

$$\mathbf{y} \;=\; \sum_{i=1}^{p} \mathbf{A}_i\mathbf{x} \;=\; \sum_{i=1}^{p} \mathbf{C}_i\tilde{\mathbf{A}}_i\mathbf{C}_i^T\mathbf{x} \;=\; \sum_{i=1}^{p} \mathbf{C}_i\tilde{\mathbf{A}}_i\tilde{\mathbf{x}}_i \;=\; \sum_{i=1}^{p} \mathbf{C}_i\breve{\mathbf{y}}_i.$$

Assuming that $\mathbf{A}$ and $\mathbf{x}$ have been distributed as described in Section 2, the vector components $\breve{\mathbf{y}}_i$ can be computed locally on each processor i without additional communication.

Now consider the calculation of the distributed component of $\mathbf{y}$ local to processor j,

$$\tilde{\mathbf{y}}_j \;=\; \mathbf{C}_j^T\mathbf{y} \;=\; \mathbf{C}_j^T\sum_{i=1}^{p} \mathbf{C}_i\breve{\mathbf{y}}_i \;=\; \sum_{i=1}^{j-1} \mathbf{C}_j^T\mathbf{C}_i\breve{\mathbf{y}}_i + \breve{\mathbf{y}}_j + \sum_{i=j+1}^{p} \mathbf{C}_j^T\mathbf{C}_i\breve{\mathbf{y}}_i.$$

By construction, $\mathbf{C}_j^T\mathbf{C}_i\mathbf{z} = \mathbf{0}$, $\Omega_i \cap \Omega_j = \emptyset$. When $\Omega_i \cap \Omega_j \neq \emptyset$, the matrix $\mathbf{C}_j^T\mathbf{C}_i$ provides a translation operator for shared DOFs from the local numbering on processor i to the local numbering on processor j. As written, the operation: (i) translates from the local numbering on processor i to the global numbering and (ii) translates from the global numbering to the local numbering on processor j. This explicit approach is not necessary provided its action on $\mathbf{z}$ can be obtained.

This leads to a two-phase approach for calculating (12) in parallel. First, each processor i calculates $\breve{\mathbf{y}}_i$ simultaneously in parallel using the existing serial matrix-vector product code. Then processor i obtains from each processor j, $\Omega_i \cap \Omega_j \neq \emptyset$, a vector of information corresponding to values in the shared region. These values, after proper translation, are added to $\breve{\mathbf{y}}_i$ giving $\tilde{\mathbf{y}}_i$. On completion, $\mathbf{y} = \cup_{i=1}^{p}\mathbf{C}_i^T\tilde{\mathbf{y}}_i = \cup_{i=1}^{p}\mathbf{y}_i$ meets the criteria for a distributed vector.

3.2 Parallel Vector-Vector Dot Product

Consider the parallel evaluation of

$$a \;=\; \mathbf{y}^T\mathbf{x} \tag{13}$$

$$\begin{aligned}
&= \;(\sum_{i=1}^{p} \mathbf{O}_i\hat{\mathbf{y}}_i)^T(\sum_{i=1}^{p} \mathbf{O}_i\hat{\mathbf{x}}_i)\\
&= \;(\sum_{i=1}^{p} \hat{\mathbf{y}}_i^T\mathbf{O}_i^T)(\sum_{i=1}^{p} \mathbf{O}_i\hat{\mathbf{x}}_i)\\
&= \;\hat{\mathbf{y}}_1^T\mathbf{O}_1^T\sum_{i=1}^{p} \mathbf{O}_i\hat{\mathbf{x}}_i + \hat{\mathbf{y}}_2^T\mathbf{O}_2^T\sum_{i=1}^{p} \mathbf{O}_i\hat{\mathbf{x}}_i + \ldots + \hat{\mathbf{y}}_p^T\mathbf{O}_p^T\sum_{i=1}^{p} \mathbf{O}_i\hat{\mathbf{x}}_i.
\end{aligned}$$

By construction

$$\mathbf{O}_j^T\mathbf{O}_k \;=\; \begin{cases} \mathbf{I} & j = k \\ \mathbf{0} & j \neq k \end{cases};$$

hence,

$$a = \sum_{i=1}^{p} (\hat{\mathbf{y}}_i^T \hat{\mathbf{x}}_i).$$

The terms of the summation can be calculated completely in parallel on the individual processors i using existing serial dot product code, while the summation is a global reduction operation over all processors.

3.3 Parallel SAXPBY

Consider the parallel evaluation of

$$\mathbf{z} = a\mathbf{x} + b\mathbf{y}, \tag{14}$$

for parallel vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$. We rewrite (14) as

$$\begin{aligned}
\mathbf{z} &= \bigcup_{i=1}^{n} a\mathbf{x}_i + \bigcup_{i=1}^{n} b\mathbf{y}_i \\
&= \bigcup_{i=1}^{n} \mathbf{C}_i^T a\tilde{\mathbf{x}} + \bigcup_{i=1}^{n} \mathbf{C}_i^T b\tilde{\mathbf{y}} \\
&= \bigcup_{i=1}^{n} \mathbf{C}_i^T (a\tilde{\mathbf{x}} + b\tilde{\mathbf{y}}) \\
&= \bigcup_{i=1}^{n} \mathbf{C}_i^T \tilde{\mathbf{z}} \\
&= \bigcup_{i=1}^{n} \mathbf{z}_i.
\end{aligned}$$

Each of the factors $(a\tilde{\mathbf{x}} + b\tilde{\mathbf{y}})$ can be calculated in parallel on the various processors i, and the resulting parallel vector $\mathbf{z}$ is distributed correctly without any parallel communication.

3.4 Parallel Sub-Matrix Approximate Solve

We were careful to analyze the previous parallel operators in terms of the serial operation and the parallel distribution and to exactly duplicate the effects of the serial operation. However, for more complex operations such as preconditioners, this may not be the most computationally efficient approach. Each complex algorithm can be analyzed individually; however, as an example consider the SOR family of preconditioners. A forward SOR preconditioning pass [3, 11] can be written as

$$\mathbf{z} = SOR_{(\mathbf{A},\omega)}(\mathbf{b}) \tag{15}$$

$$= \omega(\mathbf{D_A} - \omega\mathbf{L_A})^{-1}\mathbf{b}, \tag{16}$$

where $\mathbf{D_A}$ is the diagonal and $-\mathbf{L_A}$ the strictly lower triangular part of $\mathbf{A}$. (We can also define $-\mathbf{U_A}$ as the strictly upper triangular part of $\mathbf{A}$ and write $\mathbf{A} = \mathbf{D_A} - \mathbf{L_A} - \mathbf{U_A}$.)

For exposition, define an $n \times n_{\mathbf{B}}$ matrix $\mathbf{K}$, where n is the number of rows in A and $n_{\mathbf{B}}$ is the total number of DOFs on inter-processor boundaries, and define the entries k_{ij} as

$$k_{ij} = \begin{cases} 1 & \text{if } i \text{ is the } j^{th} \text{ boundary DOF,} \\ 0 & \text{otherwise} \end{cases}. \tag{17}$$

The analogous $n \times n_{\bar{\mathbf{B}}}$ matrix $\bar{\mathbf{K}}$ for the interior DOFs can more simply defined as

$$\bar{\mathbf{K}} = \left[\bar{\mathbf{B}}_1 \,\middle|\, \bar{\mathbf{B}}_2 \,\middle|\, \cdots \,\middle|\, \bar{\mathbf{B}}_p \right]. \tag{18}$$

The operations $\mathbf{x_K} = \mathbf{K}^T\mathbf{x}$ and $\mathbf{x}_{\bar{\mathbf{K}}} = \bar{\mathbf{K}}^T\mathbf{x}$ extract the boundary degrees of freedom into $\mathbf{x_K}$ and the DOFs on processor interiors into $\mathbf{x}_{\bar{\mathbf{K}}}$.

Now consider the following set of equations,

$$\mathbf{z}_{\bar{\mathbf{B}}} = \omega\bar{\mathbf{K}}(\bar{\mathbf{K}}^T(\mathbf{D_A} - \omega\mathbf{L_A})\bar{\mathbf{K}})^{-1}\bar{\mathbf{K}}^T\mathbf{b} \tag{19}$$

$$\mathbf{c} = \mathbf{b} - \mathbf{K}\mathbf{K}^T\mathbf{A}\bar{\mathbf{K}}\bar{\mathbf{K}}^T\mathbf{z}_{\bar{\mathbf{B}}} \tag{20}$$

$$\mathbf{z_B} = \omega\mathbf{K}(\mathbf{K}^T(\mathbf{D_A} - \omega\mathbf{L_A})\mathbf{K})^{-1}\mathbf{K}^T\mathbf{c} \tag{21}$$

$$\mathbf{z} = \mathbf{z}_{\bar{\mathbf{B}}} + \mathbf{z_B}. \tag{22}$$

Assuming that the operations are performed in sequence, (19) - (22) represent the application of a forward SOR pass as in (16) to a reordered system of equations where unknowns on the interior of the processor domains appear earlier in the pass than those on the boundaries.

We will analyze the equations individually. First rewrite (19) in terms of the parallel decomposition as

$$\mathbf{z}_{\bar{\mathbf{B}}} = \omega\bar{\mathbf{K}}(\sum_{i=1}^{p}(\bar{\mathbf{K}}^T(\mathbf{D}_{\mathbf{A}_i} - \omega\mathbf{L}_{\mathbf{A}_i})\bar{\mathbf{K}}))^{-1}\bar{\mathbf{K}}^T\mathbf{b} \tag{23}$$

$$= \omega\bar{\mathbf{K}}(\sum_{i=1}^{p}(\bar{\mathbf{K}}^T\mathbf{C}_i(\mathbf{D}_{\bar{\mathbf{A}}_i} - \omega\mathbf{L}_{\bar{\mathbf{A}}_i})\mathbf{C}_i^T\bar{\mathbf{K}}))^{-1}\bar{\mathbf{K}}^T\mathbf{b} \tag{24}$$

Using (7), (8),(9), (11) and (18), we derive

$$\bar{\mathbf{K}}^T\mathbf{C}_i = \begin{bmatrix} \mathbf{0} \\ \vdots \\ \mathbf{I}_{n_{\bar{\mathbf{B}}_i}} \\ \vdots \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{n_{\bar{\mathbf{B}}_i}} & \mathbf{0} \end{bmatrix} \mathbf{P}_i^T, \tag{25}$$

where $\mathbf{I}_{n_{\bar{\mathbf{B}}_i}}$ is the identity matrix having rank equal to the number of interior DOFs on processor i. For convenience, set

$$\bar{\mathbf{Y}}_i = \begin{bmatrix} \mathbf{0} \\ \vdots \\ \mathbf{I}_{n_{\bar{\mathbf{B}}_i}} \\ \vdots \\ \mathbf{0} \end{bmatrix} \tag{26}$$

$$\bar{\mathbf{Z}}_i = \begin{bmatrix} \mathbf{I}_{n_{\bar{\mathbf{B}}_i}} & \mathbf{0} \end{bmatrix} \mathbf{P}_i^T. \tag{27}$$

Substituting into (25) gives

$$\mathbf{z}_{\bar{\mathbf{B}}} = \omega\bar{\mathbf{K}}(\sum_{i=1}^{p}(\bar{\mathbf{Y}}_i\bar{\mathbf{Z}}_i(\mathbf{D}_{\bar{\mathbf{A}}_i} - \omega\mathbf{L}_{\bar{\mathbf{A}}_i})\bar{\mathbf{Z}}_i^T\bar{\mathbf{Y}}_i^T))^{-1}\bar{\mathbf{K}}^T\mathbf{b} \tag{28}$$

We note that the factor $\bar{\mathbf{Z}}_i(\mathbf{D}_{\bar{\mathbf{A}}_i} - \omega\mathbf{L}_{\bar{\mathbf{A}}_i})\bar{\mathbf{Z}}_i^T$ reduces the $n_i \times n_i$ system of equations (where n_i represents the number of DOFs resident on processor i), to an $n_{\bar{\mathbf{B}}_i} \times n_{\bar{\mathbf{B}}_i}$ system where only the DOFs in the interior of the processor domain are represented. Extending outward, the $\bar{\mathbf{Y}}_i$ terms translate this logically dense block to a final place in the overall system. Since the various blocks $\bar{\mathbf{B}}_i$ are orthogonal, the matrices $\bar{\mathbf{Y}}_i$ are also orthogonal and the overall result of the summation is a block diagonal matrix. We can, thus, rewrite (28) as

$$\mathbf{z}_{\bar{\mathbf{B}}} = \omega \sum_{i=1}^{p} (\bar{\mathbf{K}}\bar{\mathbf{Y}}_i(\bar{\mathbf{Z}}_i(\mathbf{D}_{\bar{\mathbf{A}}_i} - \omega\mathbf{L}_{\bar{\mathbf{A}}_i})\bar{\mathbf{Z}}_i^T)^{-1}\bar{\mathbf{Y}}_i^T\bar{\mathbf{K}}^T)\mathbf{b} \tag{29}$$

whose local component on processor j,

$$\begin{aligned}
\tilde{\mathbf{z}}_{\bar{\mathbf{B}}_j} &= C_j^T \mathbf{z}_{\bar{\mathbf{B}}} \\
&= \omega \sum_{i=1}^{p} (\bar{\mathbf{Z}}_j^T\bar{\mathbf{Y}}_j^T\bar{\mathbf{Y}}_i(\bar{\mathbf{Z}}_i(\mathbf{D}_{\bar{\mathbf{A}}_i} - \omega\mathbf{L}_{\bar{\mathbf{A}}_i})\bar{\mathbf{Z}}_i^T)^{-1}\bar{\mathbf{B}}_i^T)\mathbf{b} \\
&= \omega\bar{\mathbf{Z}}_j^T(\bar{\mathbf{Z}}_j(\mathbf{D}_{\bar{\mathbf{A}}_j} - \omega\mathbf{L}_{\bar{\mathbf{A}}_j})\bar{\mathbf{Z}}_j^T)^{-1}\bar{\mathbf{B}}_j^T\mathbf{b} \\
&= \omega\mathbf{P}_j \begin{bmatrix} (\bar{\mathbf{Z}}_i(\mathbf{D}_{\bar{\mathbf{A}}_i} - \omega\mathbf{L}_{\bar{\mathbf{A}}_i})\bar{\mathbf{Z}}_i^T)^{-1}\bar{\mathbf{B}}_i^T\mathbf{b} \\ \mathbf{0} \end{bmatrix},
\end{aligned}$$

can be calculated completely on one processor with no communication.

We rewrite (20) to obtain

$$\mathbf{c} = \mathbf{b} - \mathbf{K}\mathbf{K}^T(\sum_{i=1}^{p}(\mathbf{C}_i\tilde{\mathbf{A}}_i\mathbf{C}_i^T))\bar{\mathbf{K}}\bar{\mathbf{K}}^T\mathbf{z}_{\bar{\mathbf{B}}} \tag{30}$$

$$= \mathbf{b} - (\sum_{i=1}^{p}(\mathbf{B}_i\mathbf{P}_i^T\tilde{\mathbf{A}}_i\mathbf{P}_i\bar{\mathbf{B}}_i^T))\mathbf{z}_{\bar{\mathbf{B}}} \tag{31}$$

where (31) follows from (30) via (7), (8),(9), (10), (11), (17) and (18). Equation (31) is a parallel matrix/vector product followed by a distributed *saxpby*. This set of operations has already been shown to require exactly one unification operation.

We now define

$$\mathbf{Y}_i = \mathbf{P}_{\mathbf{B}_{i_L}} \begin{bmatrix} \mathbf{0} \\ \vdots \\ \mathbf{I}_{n_{\mathbf{B}_i}} \\ \vdots \\ \mathbf{0} \end{bmatrix} \tag{32}$$

$$\mathbf{Z}_i = \begin{bmatrix} \mathbf{0} & \mathbf{I}_{n_{\mathbf{B}_i}} \end{bmatrix} \mathbf{P}_{\mathbf{B}_{i_R}}\mathbf{P}_i^T, \tag{33}$$

where $\mathbf{P}_{\mathbf{B}_{i_L}}$ and $\mathbf{P}_{\mathbf{B}_{i_R}}$ permute the rows and nonzero columns, respectively, of $\mathbf{Y}_i\mathbf{Z}_i$, and apply a derivation similar to that used for (28) to rewrite (21) as

$$\mathbf{z}_{\mathbf{B}} = \omega\mathbf{K}(\sum_{i=1}^{p}(\mathbf{Y}_i\mathbf{Z}_i(\mathbf{D}_{\bar{\mathbf{A}}_i} - \omega\mathbf{L}_{\bar{\mathbf{A}}_i})\mathbf{Z}_i^T\mathbf{Y}_i^T))^{-1}\mathbf{K}^T\mathbf{c} \tag{34}$$

Unfortunately, without orthogonality among the matrices $\mathbf{B}_i$ and $\mathbf{B}_j$, $i \neq j$, the translation operation represented by the application of $\mathbf{Y}_i \cdots \mathbf{Y}_i^T$ does not result in a block diagonal matrix and we cannot phrase (21) as the solution of communication free local problems.

To get around this constraint, we introduce a second modification to the serial algorithm. (The first was the implicit reordering of the system of equations.) If we replace the term $(\mathbf{D}_{\bar{\mathbf{A}}_i} - \omega\mathbf{L}_{\bar{\mathbf{A}}_i})$ with $\mathbf{D}_{\bar{\mathbf{A}}_i}$ we obtain

$$\mathbf{z_B} = \omega\mathbf{K}(\sum_{i=1}^{p}(\mathbf{Y}_i\mathbf{Z}_i\mathbf{D}_{\bar{\mathbf{A}}_i}\mathbf{Z}_i^T\mathbf{Y}_i^T))^{-1}\mathbf{K}^T\mathbf{c}. \tag{35}$$

Now assume each processor i has the actual diagonal values for those DOFs which are resident on it and call the resulting local matrix with local indexing $\tilde{\mathbf{D}}_{\bar{\mathbf{A}}_i}$. If we further assume the diagonals of $\tilde{\mathbf{A}}_i$ are stored in a local vector, we can generate $\tilde{\mathbf{D}}_{\bar{\mathbf{A}}_i}$ with a simple vector unification operator. Using this in (35) we obtain

$$\mathbf{z_B} = \omega\mathbf{K}(\bigcup_{i=1}^{p}(\mathbf{Y}_i\mathbf{Z}_i\tilde{\mathbf{D}}_{\bar{\mathbf{A}}_i}\mathbf{Z}_i^T\mathbf{Y}_i^T))^{-1}\mathbf{K}^T\mathbf{c} \tag{36}$$

$$= \omega\mathbf{K}(\bigcup_{i=1}^{p}\mathbf{Y}_i(\mathbf{Z}_i\tilde{\mathbf{D}}_{\bar{\mathbf{A}}_i}\mathbf{Z}_i^T)^{-1}\mathbf{Y}_i^T)\mathbf{K}^T\mathbf{c}, \tag{37}$$

whose local component

$$\tilde{\mathbf{z}}_{\mathbf{B}_j} = C_j^T\mathbf{z_B} \tag{38}$$

$$= \omega\bigcup_{i=1}^{p}(\mathbf{Z}_j^T\mathbf{Y}_j^T\mathbf{Y}_i(\mathbf{Z}_i\tilde{\mathbf{D}}_{\bar{\mathbf{A}}_i}\mathbf{Z}_i^T)^{-1}\mathbf{P}_{\mathbf{B}i_L}^T\mathbf{B}_i^T)\mathbf{c} \tag{39}$$

$$= \omega\mathbf{Z}_j^T(\mathbf{Z}_j\mathbf{D}_{\bar{\mathbf{A}}_j}\mathbf{Z}_j^T)^{-1}\mathbf{P}_{\mathbf{B}j_L}^T\mathbf{B}_j^T\mathbf{c} \tag{40}$$

$$= \omega\mathbf{P}_j\mathbf{P}_{\mathbf{B}i_R}^T\begin{bmatrix}\mathbf{0}\\(\bar{\mathbf{Z}}_j\tilde{\mathbf{D}}_{\bar{\mathbf{A}}_i}\mathbf{Z}_j^T)^{-1}\mathbf{P}_{\mathbf{B}j_L}^T\mathbf{B}_j^T\mathbf{c}\end{bmatrix}, \tag{41}$$

can be generated with no further communication. If we assume that the matrix diagonals $\tilde{\mathbf{D}}_{\bar{\mathbf{A}}_i}$ are calculated once and cached on each processor, then this final phase can be completed with exactly one communication operation regardless of the number of SOR passes actually performed.

Finally, (22) is a *saxpby* operation that can be generated with no communication.

Together, these results show that a parallel version of SOR preconditioning with some modification can be generated using one communication pass per application, assuming the matrix diagonals have been distributed and cached on the processors. Further, if we write other preconditioners, *e.g.* , $SSOR$ and *Gauss-Seidel* in terms of one or more SOR passes, and if we provide both serial and parallel versions of SOR, then these additional preconditioners are transparently parallel.

4 Interface

Based on the analysis presented in Section 3, we define a set of functions sufficient to produce parallel code.

unify(vector x) - Executed on each processor, this function "unifies" shared vector information by summing the values calculated for a particular DOF among all processors which share this DOF. This routine requires successive calls to *send(vector x, processor p)* and *recv(vector x, processor p)* which send information from the current processor to processor p and receive information from processor p onto the current processor, respectively.

send(vector x, processor p) - Executed on processor k, this function selects the set of j DOFs, $\{x[i_1], x[i_2], \ldots, x[i_j]\}$, shared with processor p from vector x, packs them into a transmission buffer along with coding information from *encode(index i_l)*, $l = 1, \ldots, j$, and sends the buffer to processor p.

recv(vector x, processor p) - Executed on processor k, this function receives a transmission buffer from processor p and unpacks the set of j DOFs, $\{x[i_1], x[i_2], \ldots, x[i_j]\}$, into $\mathbf{x}$. The correct local index i_l, $l = 1, \ldots, j$, is obtained from the coding using *decode(coding c_l)*.

coding encode(index i) - Returns a unique encoding for local index i.

index decode(coding j) - Given a unique coding j, returns the local index for a DOF.

When the *encode()* and *decode()* routines are applied successively as part of a $send_i(x, j)/recv_j(y, i)$ pair, they are equivalent to generating $\mathbf{y} = \mathbf{C}_j^T \mathbf{C}_i \mathbf{x}$, the translation operator required as part of the matrix-vector multiplication. A direct implementation of this will use the global DOF index for local index i as the coding value; however, this is not strictly required.

boolean owned(index i) - Given local index i, returns true if the current processor owns $\tilde{x}[i]$, otherwise returns false.

boolean boundary(index i) - Given local index i, returns true if $\tilde{x}[i]$ is resident on more than one processor, otherwise returns false.

value reduce(value y) - Given values y_i, $1 \leq i \leq p$, where x_i is resident on processor i, returns $\sum_{i=1}^{p} y_i$ on every processor.

5 Implementation

Our aim is to provide the greatest flexibility in the vector and matrix storage while still providing the core parallel operations identified in Sections 3 and 4. For example, our current software implements several different matrix and vector storage options including Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and a recursive matrix formulation useful for multi-level preconditioning. Vector options include linear storage, and a sparse vector format. We recognize other options as desirable in specific instances, e.g., skyline or banded matrix formats. Additionally, we recognize a need to derive the connectivity interface of Section 4 from multiple representations of the parallel distribution. Our goal is to allow these multiple data formats and multiple definitions of connectivity without requiring that a different parallel matrix implementation be developed for each combination.

To address flexibility within a single implementation, we employ generic programming techniques [7]. We divide the requirements of the parallel vector and matrix implementations into two sets: (i) those on the serial data representation, and (ii) those on the connectivity representation. Operations on the parallel objects are then phrased as combinations of operations on the serial and connectivity objects and the parallel objects are templated against both specific versions of their serial counterparts and of the connectivity objects.

5.1 Requirements on the Matrix and Vector Classes

While we intend to minimize the requirements imposed on the serial representation, any parallel algorithm can expect that either (i) the serial version of the algorithm exists and can be assumed available,

or (ii) the serial version can be written in terms of a set of base operations required for all matrix or vector classes.

As an example of the first case, we require that a serial matrix-vector multiply be provided for the serial classes over which the parallel classes have been templated. The parallel matrix-vector multiply is then written in terms of the underlying serial matrix-vector multiply with appropriate unification where necessary.

As an example of the second case, we consider the design of a multilevel structure for preconditioning iterative solvers. The algorithm structure, defined and demonstrated in [15], can be defined in terms of underlying operations assumed for the serial vector and matrix. Since we require that all base operations assumed for the serial instances of the matrix and vector also exist for the parallel matrix class, a simple templating of the serial algorithm allows the algorithm to use the parallel classes and gain parallelism without additional work from the algorithm designer. An example of a portion of the complete templated preconditioner is given in Appendix A.

5.1.1 Basic Class Requirements

Table 1 lists the operations assumed for the serial matrix and vector classes. Not all operations are required for solver implementations and, in particular, the submatrix operations $A11()$, $A12()$, $A21()$, $A22()$, and $lastBlock()$ only make sense in the context of a matrix decomposition or a multilevel solution strategy. The goal of this effort is to limit the set of operations required for serial matrices and vectors to only those operations for which the parallelism must be made explicit. However, for completeness we include a row iterator within the matrix class to provide access to the elements of the coefficient matrix in an efficient fashion. This iterator provides access to the non-zero elements of a matrix in an efficient fashion using a Standard Template Library (STL) [10] -like syntax. The set of functions required for the iterator are shown in Table 2.

5.1.2 Debugging Parallel Vectors

In general, we assume that the parallel matrix class will be interchangeable with the serial vector class over a wide range of algorithms. As such, any parallel implementation must meet the requirements imposed on the serial classes. However, since reconciling a parallel vector with its serial counterpart can be difficult, we provide two additional parallel vector functions intended to ease the comparison of serial algorithm output with parallel algorithm output. The parallel vector functions $writeLocal()$ and $writeGlobal$ detailed in Table 3 allow a parallel vector to be written to a $MATLAB$ format file [9]. The vector can then be read into the $MATLAB$ environment via the $load$ command for subsequent validation.

The two parallel vector output routines provide two differing views of the parallel vector data. The member function $parvec.writeLocal()$ writes the values for those global vector components owned by the current processor to the file f when $flag$ is set to true. The vector is assigned the name v within the $MATLAB$ environment. The routine $parvec.writeGlobal()$ behaves similarly except that the complete global vector is written by every processor. For ease of importation into $MATLAB$, all filenames should end with the extension ".mat" and the file and variable names should be made unique across processors (typically by embedding the processor rank within the names).

Assume that each processor i, $i = 0, \ldots, p-1$, in a p processor system calls $x.writeGlobal()$ and $x.writeLocal()$ to generate files $Gi.mat$ and $Li.mat$, respectively. Also assume that each file $Gi.mat$

Component	Applies To	Description
$vec.value(i)$	Vector	Return the i^{th} value in vec (0 relative).
$vec.assign(i, v)$	Vector	Assign v to the i^{th} component of vec.
$vec.len()$	Vector	Return the length of vec.
$vec.init()$	Vector	When passed to a constructor, creates a new vector of the same length.
$vec.subrangeInit(start, end)$	Vector	When passed to a constructor, creates a new vector of the span $start \leq i < end$.
$vec.subrange(start, end)$	Vector	Returns a vector composed of the subrange $vec[start] \ldots vec[end-1]$
$mat.rows()$	Matrix	Return the number of rows in the mat.
$mat.cols()$	Matrix	Return the number of columns in mat.
$mat.diagonals()$	Matrix	Return an array of the diagonals of mat.
$mat.scaleRows(x[])$	Matrix	Scale each row i of mat by $x[i]$.
$mat.scaleColumns(x[])$	Matrix	Scale each column i of mat by $x[i]$.
$mat.begin_row(i)$	Matrix	Return an iterator initialized to the start of row i.
$mat.end_row(i)$	Matrix	Return an iterator initialized to one past the end of row i.
$mat.A11()$	Matrix	Return the $\mathbf{A}_{11}$ block of mat.
$mat.A12()$	Matrix	Return the $\mathbf{A}_{12}$ block of mat.
$mat.A21()$	Matrix	Return the $\mathbf{A}_{21}$ block of mat.
$mat.A22()$	Matrix	Return the $\mathbf{A}_{22}$ block of mat.
$mat.lastBlock()$	Matrix	Return false if a further decomposition for mat exists.
$dot(x, y)$	Vector/Vector	Returns the dot product of x and y.
$saxpby(a, x, b, y, z)$	Vector/Vector	Returns the vector $\mathbf{z} = a * \mathbf{x} + b * \mathbf{y}$.
$multMatVec(A, x, y)$	Matrix/Vector	Calculate $y = \mathbf{A} \times x$.

Table 1: Required components for the serial classes.

contains the *MATLAB* format vector Gi and each file $Li.mat$ contains the *MATLAB* format vector Li. Then $G0 == G1 == G2 == \ldots == G(N-1) = \sum_{i=0}^{n-1} Li$ is the current state of global vector $\mathbf{x}$ in the system.

5.2 Requirements on the Connectivity Class

Table 4 lists the operations required of the connectivity class. Any class capable of supporting these operations in a consistent way over a parallel environment can be used to supply the connectivity information.

Since we currently recognize the need for several different connectivity patterns, we implemented a templated connectivity class capable of handling a wide range of connectivity definitions. On each processor p, the base class defines a series of lists containing first, a mapping from each of the local DOF

Component	Description
$iter.operator++()$	Advance the iterator using pre-increment.
$iter.operator--()$	Retard the iterator using pre-decrement.
$iter.operator++(int)$	Advance the iterator using post-increment.
$iter.operator--(int)$	Retard the iterator using post-decrement.
$iter.operator!=(other)$	Return true if the current iterator and *other* point to different locations.
$iter.operator==(other)$	Return true if the current iterator and *other* point to the same location.
$iter.index()$	Return the column index of the current iterator position.
$iter.index()$	Return the value of the current iterator position.

Table 2: Iterator Components.

Component	Description
$parvec.writeLocal(char*f, char*v, intflag)$	If flag is true, write the locally owned components of the global vector to $MATLAB$ format file f under variable name v.
$parvec.writeGlobal(char*f, char*v, intflag)$	If flag is true, write the complete global vector to $MATLAB$ format file f under variable name v.

Table 3: Optional components for debugging parallel vectors.

indices to the global DOF index and then for every other processor p_i to which processor p must send data, an additional list containing those DOF indices which are shared and the processor to which it must be sent. This information is sufficient for complete functionality of the class.

Initialization is accomplished by templating the connectivity class on one of a number of initializer classes. The initializer classes provide the information necessary to construct the data structures for the connectivity classes. The actual information depends on the distribution assumed for the parallel data and on the mechanism provided by the software environment. The initializers are not referenced once the connectivity class has been instantiated. Table 5 defines the interface provided by the initializer classes. Table 6 contains a list of those initializers currently defined, their constructor arguments, and a brief description of the connectivity they provide.

```
parallelVec<VECTORCLASS, CONNECTIVITYCLASS> vec;
parallelMat<MATRIXCLASS, CONNECTIVITYCLASS> mat;
```

Figure 2: Instantiation of parallel vector and matrix class.

Component	Description
$con.encode(i)$	Return a unique global encoding for local DOF i.
$con.decode(j)$	Return the local DOF number for global encoding j.
$con.isOwned(i)$	Return true if this processor controls local DOF i.
$con.isOnBoundary(i)$	Return true if local DOF i falls on a processor boundary.
$con.numNeighbors()$	Return the number of processor neighbors.
$con.numPoints(i)$	Return the number of points shared with processor i.
$con.neighbor(i)$	Return the identifier for the i^{th} neighbor.
$con.pointIndex(i,j)$	Return the local DOF number for the j^{th} point shared with the i^{th} neighbor.
$con.subrange(start,end)$	Return a new connector valid over the subrange $[start, end)$.

Table 4: Required components for the connectivity class.

```
parallelVec<VECTORCLASS, CONNECTIVITYCLASS> vec;
vec = serVec;
vec = connect;
parallelMat<MATRIXCLASS, CONNECTIVITYCLASS> mat;
mat = serMat;
```

Figure 3: Initialization of parallel vector and matrix classes.

5.3 Examples

Provided the serial and connectivity classes meet the requirements defined above, we can define parallel matrix and vector classes over these serial and connectivity classes as shown in Figure 2. The objects *vec* and *mat* can then be used interchangeably with any other parallel vector and matrix classes over different serial vector and matrix implementations. Further, to aid in initialization, we allow the assignment of a serial matrix class to a parallel matrix class templated on it, and the assignment of both a serial vector class and a connectivity class to a parallel vector templated on them. So given a serial vector, $serVec$, of $VECTORCLASS$, a serial matrix, $serMat$ of $MATRIXCLASS$, and a $CONNECTIVITYCLASS$ object *connect* all of which have been initialized appropriately, the code snippet in Figure 3 instantiates the parallel vector and parallel matrix and initializes them to the values of their serial counterparts and the connectivity class.

Figure 4 gives a short example of the declaration and use of the parallel matrix and vector classes in the context of an iterative solution method. The code assumes that an instance, $serMat$, of a serial matrix class, $serialMatrix$, has been previously declared and initialized to the desired state for the computation. Similarly, *sol* and *rhs* are assumed to be instances of a serial vector class, $serialVector$,

Component	Description
$init.parNumberNeighbors()$	Return the number of neighbors of this processor.
$init.parNumberDOFs()$	Return the number of DOFs resident on this processor.
$init.parMapLocalGlobal()$	Return an array x where $x[i]$ is the global DOF number for local DOF number i.
$init.parOwnership()$	Return an array x where $x[i] = 1$ if local DOF i is owned by this processor or $x[i] = 0$, otherwise.
$init.parOnBoundary()$	Return an array x where $x[i] = 1$ if local DOF i is on the boundary of the processor domain, $x[i] = 0$, otherwise.
$init.parProcId(i)$	Returns the processor ID of the i^{th} neighbor of this processor.
$init.parProcNumShared(i)$	Returns the number of DOFs shared with the i^{th} neighbor.
$init.parProcIndices(i)$	Return an array x, where $x[j]$ is the j^{th} DOF shared with the i^{th} neighbor.

Table 5: Required components for the initializer class.

initialized appropriately to the local initial guess and system right-hand side, respectively.

The vector distribution assumes the complete vector is maintained on every processor and, thus, the full initializer has been chosen to initialize the *genericConnector* class. Further, the class *nullPreconditioner* is a preconditioner by the identity matrix, **I**. Table 7 contains the definition of the preconditioner class interface. The routine $qmr()$ performs the Quasi-minimal residual algorithm [6]. The first four parameters of $qmr()$ are self-explanatory, the fifth is a convergence tolerance and the last is a maximum number of steps.

Appendix B contains the definition of serial vector and matrix classes, Appendix C contains the definition of the templated parallel matrix and parallel vector classes assuming a Message-Passing Interface (MPI) [8] environment, and Appendix D contains the definition of a simple connectivity class sufficient to maintain a global vector (all DOFs shared on every processor).

A sample program demonstrating the use of the parallel classes is shown in Appendix E. The example sets up a serial vector of rank 10 where each entry is the row number of the vector. It also sets up a sparse matrix in CSR format, initializes the matrix to a bi-diagonal, upper triangular matrix and seeds it to some default values. These serial objects are used to construct parallel objects used in a parallel

Class	Constructor Args	Description
fullInitializer	int i	Assumes all DOFs are shared among all processors. The i indicates the global number of DOFs.
vertexInitializer	pMeshPB p	Assumes connectivity occurs at the vertices of shared mesh objects in p. The local DOF number should be attached to the vertices with the tag "LCLN". The global number is obtained from the $pmdb_glob_en_num()$.
pInitializer	pMeshPB, int exp, char* num, char* ldof, char* gdof	Assumes connectivity can occur at any boundary entity of p. The number of DOFs should be attached to the entities with the tag num, a list of num local DOF numbers should be attached with tag $ldof$, and a list of num global DOF numbers should be atttached with tag $gdof$. The parameter exp allows each DOF to be expanded into an $exp \times exp$ block.
redInitializer	connector c int start, int end	This initializer restricts the range over which c is defined to be $[start, end)$. The range is arbitrary (provided $end >= start$) and the user is required to ensure $0 <= start <= end < ndofs$, where $ndofs$ is the number of DOFs managed by c. Failure to do so will result in an exception either from this class or from the connector being restricted. On exit, local DOF numbers have been normalized by subtracting start. Global DOF numbers are left unchanged.

Table 6: Available initializer classes.

matrix-vector multiplication. The serial multiplication and product are

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{bmatrix} = \begin{bmatrix} 3 \\ 10 \\ 21 \\ 36 \\ 55 \\ 78 \\ 105 \\ 136 \\ 171 \\ 100 \end{bmatrix}.$$

The parallel product on p processors will be p times the serial product.

```
// Generate an instance of the full initializer.
fullInitializer initializer(rhs.len());
// Use the full initializer to generate an
// instance of the connector.
genericConnect<fullInitializer> connectorFull(initializer);
// Generate parallel versions of the matrix and vectors.  Since
// it is assumed that the initial guess and RHS for the serial
// vectors was generated using only local information, the
// instantiation of the parallel vector includes a unification
// operation.
parallelMat<serialMatrix, genericConnect<fullInitializer> >
                                mat(serMat, connectorFull);
parallelVec<serialVector, genericConnect<fullInitializer> >
                                pRHS(rhs, connectorFull);
parallelVec<serialVector, genericConnect<fullInitializer> >
                                pSol(sol, connectorFull);
// Define the NULL (identity) preconditioner.
nullPreconditioner<parallelMat<sparseMatrix,
                   genericConnect<fullInitializer> >,
                   parallelVec<serialVector,
                   genericConnect<fullInitializer> > > precon;
// Precondition the RHS vector (not strictly necessary for NULL.)
parallelVec<serialVector, genericConnect<fullInitializer> >
                   preconVec(pRHS.init());
// Solve the system.
ierr = qmrPre(precon, mat, pSol, preconVec, &defTol, &nstep);
```

Figure 4: Use of the parallel matrix and vector classes in an iterative context.

6 Discussion

We addressed the design of interchangeable linear algebraic components with a focus on the design of parallel, iterative, linear algebraic solvers independent of both the underlying data structures and underlying parallel model. We reduced our requirements to a mathematical model and analyzed common iterative solution components in terms of the model to better understand the nature and extent of the possible parallelism. This understanding was used to define an abstract interface defining those operations which must be provided in the parallel implementation. Finally, we provided a concrete parallel matrix class templated on both an underlying data storage format and a data communication pattern. This class provides transparent or nearly transparent parallelism for a variety of iterative algorithms, and is easily adapted to other parallel data distribution models through simple modifications to the connector class. Our code is currently used for several matrix formats and for two different parallel data distributions created within the Scientific Computation Research Center (SCOREC), the Parallel Mesh Database (PMDB) [5], and the Rensselaer Partition Model (RPM) [14].

Component	Description
$precon.applyPrecon(mat, src, res)$	Apply the preconditioner for mat to src giving res.
$precon.applyPreconTranspose(mat, src, res)$	Apply the preconditioner for mat^T to src giving res.
$precon.forward(mat, src, res)$	Compute a preconditioned matrix-vector product of $mat \times src = res$.
$precon.transpose(mat, src, res)$	Compute a preconditioned matrix-vector product of $mat^T \times src = res$.

Table 7: Required components for the connectivity class.

Our parallel design is a departure from other linear algebraic packages including PETSc [2] and LAPACK [1] in that it concentrates on an extensible design applicable over any underlying data structures and parallel data distribution meeting a defined interface. We believe these modifications will greatly improve the flexibility of the package, decreasing the time required to interface the package to other parts of diverse systems, and will also minimize the effort required to port a serial application to a parallel environment. Our flexible approach to attaining parallel execution is compatible with packages such as the MTL [12, 13] and the flexible matrices provided by such packages can be used within our system with minor adjustments to the interface.

References

[1] E. Anderson. *LAPACK users' guide*. SIAM, Philadelphia, 1992.

[2] S. Balay, W. Gropp, L. C. McInnes, and B. Smith. *PETSc 2.0 Users Manual*. Argonne National Laboratory, Argonne, Il., anl-95/11 - revision 2.0.17 edition, 1995.

[3] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1994.

[4] H. C. Edwards. *A Parallel Infrastructure for Scalable Adaptive Finite Element Methods and its Application to Least Squares C^∞ Collocation*. PhD thesis, The University of Texas at Austin, May 1997.

[5] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. *Appl. Numer. Math.*, 26:241–263, 1998.

[6] R. W. Freund and N. M. Nachtigal. QMR: a quasi-minimal residual method for non-hermitian linear systems. *Numer. Math.*, 60:315–339, 1991.

[7] M. Lee and A. Stepanov. The standard template library. Technical report, HP Laboratories, February 1995.

[8] Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee. *MPI: A Message Passing Interface Standard*, first edition, 1994.

[9] C. B. Moler. MATLAB users' guide. Technical Report CS81-1 (Revised), Department of Computer Science, University of New Mexico, 1982.

[10] D. R. Musser, A. Saini, and A. Stepanov. *STL Tutorial and Reference Guide: C++ Programming With the Standard Template Library.* Addison-Wesley, 1996.

[11] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS, Boston, 1996.

[12] J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. *Internatational Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[13] J. G. Siek, A. Lumsdaine, and L. Q. Lie. Generic programming for high performance numerical linear algebra. *SIAM Workshop on Interoperable Object-Oriented Sci. Computing*, 1998.

[14] J. D. Teresco, M. W. Beall, J. E. Flaherty, and M. S. Shephard. A hierarchical partition model for adaptive finite element computation. *Comput. Methods Appl. Mech. Engrg.*, 184:269–285, 2000.

[15] W. D. Turner, J. E. Flaherty, S. Dey, and M. S. Shephard. Multilevel preconditioned qmr methods for unstructured mesh computation. *Comput. Methods Appl. Mech. Engrg.*, 149:339–357, 1997.

[16] W. D. Turner, J. E. Flaherty, and L. Ziantz. The parallel implementation of an octree coarsened, multilevel preconditioned iterative method over unstructured meshes. In process.

A Templated Multilevel Preconditioning

The code below implements a portion of a recursive preconditioner using the interface developed in the previous sections. This code works with both multiple serial and parallel implementations provided the underlying data objects meet the specification from Section 5. In the code, the parameter *nu* represents a preconditioner variable set via a call to a run-time parameter manager, the call to *submatSolve()* represents an approximation to the inversion of the upper left quadrant of a block decomposition of the matrix, and the call to *mgridPreRec()* represents a recursive call to the complete preconditioner on the lower right quadrant. A more thorough definition of the preconditioning algorithm can be found in [15].

```
#ifndef APPMGRIDLREC_H
#define APPMGRIDLREC_H

#include <stdio.h>
#include <math.h>
#include <string.h>

extern "C" {
#include "MSops.h"
#include "structures.h"
#include "utilities.h"
#include "elemental.h"
#include "poisson.h"
#include "multiPre.h"
#include "matlab.h"
#include "qmr.h"
#include "config.h"
}

extern double *q;

/*******************************************
 *
 * Author:  Wesley D. Turner
 *
 * Date: 07/12/99
 *
 * We are trying to perform a multiplication by
 * L^-1 * x.  Set up such that L^-1*x=y, so L*y=x.
 * L decouples into two parts.  First we solve
 * A11 * y0 = x0 to get y0 and then we set up to
 * solve the recursive part based on the relation
 * Mk^-1*(x1 - A21*y0) = y1.
 *
 * This routine is adapted from the original sparse
 * matrix routine precon/appMultigridL.c by modifying
 * it to use the RECURSIVEMAT format.  It has since
 * been templated on matrix/vector classes allowing
 * transparent serial/parallel operation.
 */

template <class M, class V>
void appMgridLRec(
     M &A,  /* Initial A of Ax = b. */
     V &x,  /* Vector to precondition. */
     V &y)  /* resulting vector. */
{
     int cacheFlag=CACHING;  /* Turn on caching? */
     int ctr;                /* Generic loop counter. */
     int degCtr;             /* Generic loop counter. */
     int ierr;               /* Error return code. */
     int nu;                 /* Polynomial degree. */
     int offset;             /* Offset in vector. */
     int endIndex;           /* Ending index. */
```

```
    nu = getMatrixPolyDeg();
/*
 *  First part, set up and solve B11*y0 = x0 to get y0.
 */
    M tempA11(A.A11());
    M tempA21(A.A21());
    M tempA22(A.A22());
    V xSubrange11(x.subrange(0, tempA11.columns()));
    V B11x0(y.subrangeInit(0, tempA11.columns()));

    if ((ierr = submatSolve(getSolutionMethod(), tempA11,
         B11x0, xSubrange11), &cacheFlag))  != 0) {
          fprintf(stderr,"Warning - error %d encountered \
                      during submatrix solve \
                      (from appMgridLRec()).\n", ierr);
    }

/*
 *  B11x0 is exactly what we need for y0, so copy it there.
 */
    for (ctr=0; ctr<tempA11.columns(); ctr++) {
          y.assign(ctr, B11x0.value(ctr));
    }

/*
 *  Now calculate x1-A21B11^(-1)x0 from B11^(-1)x0.
 */
    V A21B11x0(y.subrangeInit(tempA11.columns(),
               A.columns()));
    multMatVec(tempA21, B11x0, A21B11x0);

    offset = tempA11.columns();
    endIndex = A.columns() - tempA11.columns();
    for (ctr=0; ctr<endIndex; ctr++) {
         A21B11x0.assign(ctr, (x.value(ctr+offset)-
                               A21B11x0.value(ctr)));
    }
/*
 *  Implement Horner's method for calculating polynomials
 *  as detailed in Algebraic Multilevel Preconditioning
 *  Methods II, by Axelsson and Vassilevski.  The
 *  cumulative result is maintained in passVar which is
 *  initially set to 0.
 */
    V holder(y.subrangeInit(tempA11.columns(),
                                          A.columns()));
    V passVar(y.subrangeInit(tempA11.columns(),
                                          A.columns()));
    for (ctr=0; ctr<endIndex; ctr++) {
         passVar.assign(ctr, 0.0);
    }

    /* Horner's method. */
    for (degCtr=0; degCtr<nu; degCtr++) {
         multMatVec(tempA22, passVar, holder);
         for (ctr=0; ctr<endIndex; ctr++) {
              holder.assign(ctr, (holder.value(ctr)+
                     (q[degCtr]*A21B11x0.value(ctr))));
         }
         mgridPreRec(tempA22, holder, passVar);
    }

/*
 *  Now copy the correct answer back where we need it.
 */
    for (ctr=0; ctr<endIndex; ctr++) {
         y.assign((ctr+offset), passVar.value(ctr));
    }

}
#endif
```

B Serial Vector and Matrix Class Definitions

The code below implements a portion of serial matrix and vector objects meeting the specifications detailed in Section 5. The matrix class is based on a *C* language implementation of a compressed sparse row format and the vector is based on a *C* language style array of doubles. The *use_count* class provides a smart pointer implementation that tracks the number of pointers referencing a memory block and only frees the block when no more pointers reference it. For efficiency, the matrix class provides for the explicit storage of the transpose when transpose operations are required. Note that this code is intended as an illustration of the design of appropriate serial classes and is not complete. In particular, the implementation files have been omitted for space considerations.

```
#ifndef SPARSEMATRIX_H
#define SPARSEMATRIX_H

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <strings.h>
#include "use_count.h"
#include "MSops.h"
extern "C" {
#include "structures.h"
#include "utilities.h"
#include "elemental.h"
#include "poisson.h"
#include "multiPre.h"
}

//////////////////////////////////////////////////////
// Compressed Sparse Row Storage Format.
//////////////////////////////////////////////////////

class sparseMatrix;
typedef use_count<SPARSEMAT> csr_ptr;
typedef use_count<sparseMatrix> sparseMatrixPtr;

//////////////////////////////
// Sparse matrix wrapper.    //
//////////////////////////////
class sparseMatrix {
public:
// Destructor. (Not needed.)
// Constructors.
     sparseMatrix(SPARSEMAT *mat):transpose_() {
          data_=new SPARSEMAT(*mat);
     }
     sparseMatrix():data_(NULL), transpose_() {}

// Matrix attributes.
     int rows() {return data_->rows;}
     int columns() {return data_->cols;}
     double * diagonals();
     void scaleRows(double *scaleVal);
     void scaleColumns(double *scaleVal);
// Row iterator class.
     class iterator {
     public:
        iterator(int *index=NULL, double *value=NULL)
                                  :current_index_(index),
                                  current_value_(value) {}
        iterator operator++(){
             current_index_++;
             current_value_++;
             return *this;
        }
        iterator operator--(){
```

```
        current_index_--;
        current_value_--;
        return *this;
   }
   iterator operator++(int){
        iterator temp=*this;
        current_index_++;
        current_value_++;
        return temp;
   }
   iterator operator--(int){
        iterator temp=*this;
        current_index_--;
        current_value_--;
        return temp;
   }
   int operator!=(const iterator & other) const {
    return (current_index_ != other.current_index_) ||
           (current_value_ != other.current_value_);
   }
   int operator==(const iterator & other) const {
    return (current_index_ == other.current_index_) &&
           (current_value_ == other.current_value_);
   }
   int index() const {return (*current_index_)-1;}
   double value() const {return *current_value_;}
private:
   int *current_index_;
   double *current_value_;
};
iterator begin_row(int i) {
     int index=data_->rowLocation[i]-1;
     return iterator(&data_->colIndex[index],
                     &data_->element[index]);
}
iterator end_row(int i) {
     int index=data_->rowLocation[i+1]-1;
          return iterator(&data_->colIndex[index],
                          &data_->element[index]);
     }
// Assignment.
     SPARSEMAT * operator=(SPARSEMAT *mat){
          data_=mat;
          return mat;
     }
// Extraction.
     csr_ptr & matrix() {return data_;}
     sparseMatrix & transpose(); // Implementation file.
private:
     csr_ptr data_;
     sparseMatrixPtr transpose_;
};
#endif



#ifndef SERIALVECTOR_H
#define SERIALVECTOR_H

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <strings.h>
extern "C" {
#include "structures.h"
}
////////////////////////////////////////////////////
// The serial vector class.
////////////////////////////////////////////////////
class serialVector {
public:
// Destructor
  ~serialVector();
```

```
// Constructors.
  serialVector();
  serialVector(int i);
  serialVector(int i, double *vec);
  serialVector(VECTOR *vec);
  serialVector(const serialVector& vec);
// Assignment
  inline VECTOR * operator=(VECTOR *c);
  inline const serialVector&
                    operator=(const serialVector& c);
// Vector required interface.
  double value(int i) const {return data[i];}
  double assign(int i, double val) {
                    data[i]=val; return val;
  }
  int len() const {return length;}
  int init() const {return length;}
  int subrangeInit(int start, int end);
  serialVector subrange(int start, int end);
// Extraction.
  double *vector() {return data;}
private:
   int length;          // Length of vector.
   int *usect;          // Multiplicity of reference.
   double *memory_base; // Start of allocated memory.
   double *data;        // Start of data.
};
#endif
```

C Parallel Vector and Matrix Class Definitions

The code below implements a portion of parallel matrix and vector objects meeting the specifications detailed in Section 5. The classes are templated and are compatible with any serial data and connector objects which also meet the requirements of Section 5. This code is not complete and portions of the implementation dealing with the definitions of the various matrix blocks (*A12(), A21(), and A22()*) have been omitted for space considerations. Despite this, the code is significantly longer than that presented in Appendix B. This does not represent an expanded interface, but simply reflects the inclusion of more routine implementations within the interface due to template concerns. Of particular interest in this code is the definition of routines such as *multMatVec()* where the parallel implementation is written as a call to the underlying serial data object, followed by a call to the connector object.

```
#ifndef PARMAT_H
#define PARMAT_H

#include <assert.h>
extern "C" {
#include "comm_lib.h"
}

//////////////////////////////////////////////////////
// The transmission buffer structure used.
//////////////////////////////////////////////////////
class PACKBUFFER
{
public:
     int     index;
     double     val;
};

//////////////////////////////////////////////////////
// The parallel matrix class.
//////////////////////////////////////////////////////
template <class T, class C>
class parallelMat {
public:
// Constructors.
parallelMat():transpose_(NULL) {}
parallelMat(const T& mat, const C& con):matrix(mat),
                         connect(con),transpose_(NULL) {}
parallelMat(const parallelMat & mat):matrix(mat.matrix),
                connect(mat.connect),transpose_(NULL) {}
// Destructors
~parallelMat() {delete transpose_;}
// Assignment
C& operator=(C& c){connect = c; return c;}
T& operator=(T& t){matrix = t; return t;}
const parallelMat<T, C> & operator=
                       (const parallelMat<T, C> & src) {
     if (this != &src) {
          matrix = src.matrix;
          connect = src.connect;
          transpose_ = NULL;
     }
     return src;
}
// Matrix attributes.
int rows() {return matrix.rows();}
int columns() {return matrix.columns();}
double * diagonals();
void scaleRows(double *scaleVal){
     matrix.scaleRows(scaleVal);
}
void scaleColumns(double *scaleVal){
     matrix.scaleColumns(scaleVal);
}
```

```
// Row iterator class.
class iterator {
public:
    iterator():serial_iterator_() {}
    iterator(const typename T::iterator & itr)
                        :serial_iterator_(itr) {}
    iterator operator++(){
              ++serial_iterator_; return *this;}
    iterator operator--(){
              --serial_iterator_; return *this;}
    iterator operator++(int){
         iterator temp = *this;
         serial_iterator_++;
         return temp;
    }
    iterator operator--(int){
         iterator temp = *this;
         serial_iterator_--;
         return temp;
    }
    int operator!=(const iterator & other) {
         return(serial_iterator_ !=
                        other.serial_iterator_);
    }
    int operator==(const iterator & other) {
         return (serial_iterator_ ==
                        other.serial_iterator_);
    }

    int index() {return serial_iterator_.index();}
    double value() const {
              return serial_iterator_.value();}
private:
    typename T::iterator serial_iterator_;
};

iterator begin_row(int i) {
    return iterator(matrix.begin_row(i));
}

iterator end_row(int i) {
    return iterator(matrix.end_row(i));
}

//Parallel routines.
void unifyVector(int cnt, double *vector);

// Recursive matrix utilities.
int lastBlock() {return matrix.lastBlock();}
parallelMat<T, C> A11();
parallelMat<T, C> A12();
parallelMat<T, C> A21();
parallelMat<T, C> A22();

// Extraction.
T& serial() {return matrix;}

C connector() {return connect;}

// Transpose operators
void transpose_clear() {delete transpose_;
                        transpose_=NULL;}
parallelMat<T,C> & transpose()
     {
          if (transpose_ == NULL) {
               transpose_ = new parallelMat<T, C>
                       (matrix.transpose(), connect);
          }
          return *transpose_;
     }

private:
   T matrix;                        // Serial matrix.
   C connect;                       // Parallel model.
```

```
   parallelMat<T, C> * transpose_; // Parallel transpose.
};

//////////////////////////////////////////////////////
// The parallel vector class.
//////////////////////////////////////////////////////

template <class C>
class parallelVecInit {
public:
parallelVecInit(int len, const C& c)
                   :length(len),connect(c){}
const C& connector() const {return connect;}
int len() const {return length;}
private:
C connect;
int length;
};

template <class T, class C>
class parallelVec {
public:
// Constructors.
parallelVec(const parallelVecInit<C>& d){
     vector=new T(d.len());
     connect=new C(d.connector());
}
parallelVec(const parallelVec<T,C>& d){
     vector=new T(*(d.vector));
     connect=new C(*(d.connect));
}
parallelVec(const T& vec, const C& con);
parallelVec(){vector=NULL; connect=NULL;}
parallelVec(int i){vector=new T(i); connect=NULL;}
// Destructors.
~parallelVec(){delete vector; delete connect;}
// Assignment
C& operator=(C& c){connect = &c; return c;}
T& operator=(T& t){vector = &t; return t;}
const parallelVec<T,C>& operator=
                           (const parallelVec<T,C>& d);
// Vector required interface.
parallelVecInit<C> init() {
     parallelVecInit<C> temp(len(), *connect);
     return temp;
}
parallelVecInit<C> subrangeInit
                           (int start, int end) {
     assert((end>start)&&(end<=len()));
     parallelVecInit<C> temp((end-start),
                        connect->subrange(start, end));
     return temp;
}
parallelVec<T,C> subrange(int start, int end);
double value(int i) const {return vector->value(i);}
double assign(int i, double val) {
                         return vector->assign(i, val);}
int len() const {return vector->len();}
// Extraction.
T& serial() {return *vector;}
//Parallel routines.
void unify();
int pack(int neighbor, PACKBUFFER *packed);
int isOwned(int rank) {return connect->isOwned(rank);}
int isOnBoundary(int rank) {
                    return connect->isOnBoundary(rank);}
void unpack(int neighbor, PACKBUFFER *packed);
// Debug routines
void writeGlobal(char *filename, char *varname,
                                        int flag) const;
void writeLocal(char *filename, char *varname,
                                        int flag) const;
private:
   T *vector;
```

```
   C *connect;
};

/////////////////////////////////////////////////////
// Matrix functions.
/////////////////////////////////////////////////////

// Matrix attributes.
template <class T, class C>          // Grab the diagonals.
double * parallelMat<T, C>::diagonals()
{
     double *diag;

     diag = matrix.diagonals();
     unifyVector(columns() , diag);
     return diag;
}

//Parallel routines.
template <class T, class C>          // Unify
void parallelMat<T, C>::unifyVector
                                (int cnt, double *vector)
{
     int     ctr;          // Generic loop counter.
//
// Generate a parallel vector using this data.  This
// results in an implicit unification.
//
     serialVec ser(cnt, vector);
     parallelVec<serialVec, C> par(ser, connect);
//
// Copy the data from the parallel vector back over
// the input data and return.
//
     for (ctr=0; ctr<cnt; ctr++) {
          vector[ctr] = par.value(ctr);
     }
}
// Recursive matrix utilities.
template <class T, class C>
inline parallelMat<T,C> parallelMat<T,C>::A11()
{
     T serial(matrix.A11());
     parallelMat<T,C>(serial,
          connect.subrange(0, serial.columns())) temp;
     return temp;
}

/////////////////////////////////////////////////////
// The remaining submatrix functions omitted for space
// considerations ... (A12, A21, A22).
/////////////////////////////////////////////////////

/////////////////////////////////////////////////////
// Vector functions.
/////////////////////////////////////////////////////
// Constructors.
template <class T, class C>
parallelVec<T, C>::parallelVec(const T& vec, const C& con)
{
     vector = new T(vec);
     connect = new C(con);
     unify();
}

template <class T, class C>
parallelVec<T, C>::parallelVec(const T& vec, C& con)
{
     vector = new T(vec);
     connect = new C(con);
     unify();
}
```

```
// Operators.
template <class T, class C>
inline const parallelVec<T,C>& parallelVec<T, C>::operator=
                              (const parallelVec<T,C>& d) {
     if (this != &d) {
          delete vector;
          vector = new T(*(d.vector));
          delete connect;
          connect=new C(*(d.connect));
     }
     return *this;
}
template <class T, class C>    // Pack transmission buffer.
int parallelVec<T, C>::pack
                           (int neighbor, PACKBUFFER *packed)
{
     int ctr;        // Generic loop counter.
     int size;       // Number of points to this neighbor.
     int index;      // Vector index to be packed.

     //
     // Get the number of values in this transmission.
      //
     size = connect->numPoints(neighbor);

     //
     // Pack the values and indices.
     //
     for (ctr=0; ctr<size; ctr++) {
          index = connect->pointIndex(neighbor, ctr);
          packed[ctr].index = connect->encode(index);
          packed[ctr].val = vector->value(index);
     }
     return (size * sizeof(PACKBUFFER));
}
template <class T, class C>
parallelVec<T,C> parallelVec<T,C>::subrange
                              (int start, int end) {
     parallelVec<T,C> temp(subrangeInit(start, end));
     int ctr;
     int index=0;

     for (ctr=start; ctr<end; ctr++) {
          temp.assign(index, value(ctr));
          index++;
     }
     return temp;
}
template <class T, class C>   // Unpack the buffer.
void parallelVec<T, C>::unpack
               (int neighbor, PACKBUFFER *packed) {
     int ctr;    // Generic loop counter.
     int size;   // Number of points to this neighbor.
     int index;  // Vector index to be packed.
     //
     // Get the number of values in this transmission.
     //
     size = connect->numPoints(neighbor);
     //
     // Unpack and add into the original vector.
     //
     for (ctr=0; ctr<size; ctr++) {
          index = connect->decode(packed[ctr].index);
          vector->assign(index,
             (vector->value(index) + packed[ctr].val));
     }
}
template <class T, class C>          // Unify
void parallelVec<T, C>::unify()
{
// Routine omitted for space considerations.
}
// Debug routines.
template <class T, class C>         // Write vector.
```

```
void parallelVec<T, C>::writeLocal
     (char *filename, char *varname, int flag) const
{
// Routine omitted for space considerations.
}
template <class T, class C>         // Write vector.
void parallelVec<T, C>::writeGlobal
     (char *filename, char *varname, int flag) const
{
// Routine omitted for space considerations.
}

////////////////////////////////////////////////////
// Other functions.
////////////////////////////////////////////////////

template <class T1, class T2, class T3,
          class C1, class C2, class C3>
void multMatVec(parallelMat<T1, C1>& mat,
                    parallelVec<T2, C2>& vec,
                    parallelVec<T3, C3>& out)
{
     multMatVec(mat.serial(), vec.serial(),
                                        out.serial());
     out.unify();
}
template <class T1, class T2, class T3,
          class C1, class C2, class C3>
void multMatTrVec(parallelMat<T1, C1>& mat,
                    parallelVec<T2, C2>& vec,
                    parallelVec<T3, C3>& out)
{
     multMatTrVec(mat.serial(), vec.serial(),
                                        out.serial());
     out.unify();
}
template <class T, class C>
double dot(parallelVec<T, C>& vec1,
                              parallelVec<T, C>& vec2) {
     int ctr;
     double sum=0.0;
     double res;

     assert (vec1.len() == vec2.len());   // Verify sizes.

     for (ctr=0; ctr<vec1.len(); ctr++) { // Accumulate.
          if (vec1.isOwned(ctr)) {
               sum += vec1.value(ctr) * vec2.value(ctr);
          }
     }

     MPI_Allreduce(&sum, &res, 1, MPI_DOUBLE, MPI_SUM,
                                          MPI_COMM_WORLD);
     return res;
}
template <class T, class C>
void saxpby(double a, parallelVec<T, C>& src1, double b,
          parallelVec<T, C>& src2, parallelVec<T, C>& out)
{
     saxpby(a, src1.serial(), b, src2.serial(),
                                          out.serial());
}
#endif
```

D Connectivity Class Definition

The code below implements a trivial connector object meeting the specifications detailed in Section 5. The connector sets $\mathbf{C}_i = \mathbf{B}_i = \mathbf{I}$ for each processor $1 \leq i \leq p$ implying that all DOFs are shared among every processor. While this connector does not provide scalable parallelism, it is useful as an illustration of the connector class and for debug purposes.

```
#ifndef FULLCONNECT_H
#define FULLCONNECT_H
////////////////////////////////////////////////////
// Dummy connectivity function.
////////////////////////////////////////////////////
//
// This connector forces all degrees of freedom to be
// on all processors.
//
class fullConnect {
public:
// Constructors.
fullConnect(int numEls=-1);
fullConnect(const fullConnect& d);
// Destructor.
~fullConnect() {delete [] neighbors;}
// Assignment operator.
fullConnect operator=(const fullConnect& d);
// Encode/decode operators.
int encode(int localNum) {return localNum;}
int decode(int encoding) {return encoding;}
// Connectivity information.
int numNeighbors() {return connections;}
int numPoints(int neighbor) {
     assert(transSize!=-1);
     return transSize;
}
int neighbor(int i) {return neighbors[i];}
int pointIndex(int neighbor, int rank) {return rank;}
int isOwned(int localNum) {
     return ((localNum>=min) && (localNum<max));
}
int isOnBoundary(int localNum) {return 1;}     // PARALLEL
// Debug routine
int globalNum(int ctr) {return encode(ctr);}
private:
int connections;
int transSize;
int *neighbors;
int min, max;
};
// Default constructor.
fullConnect::fullConnect(int numEls)
{
     int ctr;
     int next;
     int rank;
     int owned;
     int remainder;
     MPI_Comm_size(MPI_COMM_WORLD, &connections);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     transSize = numEls;
     min = max = -1;
     if (transSize >= 0) {
          owned = transSize/connections;
          remainder = transSize%connections;
          connections--;
          if (rank<remainder) {
               min = rank * (owned+1);
               max = min + owned + 1;
          } else {
               min = remainder*(owned+1) +
```

```
                    (rank-remainder)*owned;
               max = min+owned;
          }
     }
     neighbors = new int[connections];
     next = 0;
     for (ctr=0; ctr<(connections+1); ctr++) {
          if (ctr != rank) {
               neighbors[next] = ctr;
               next++;
          }
     }
}
// Copy constructor.
fullConnect::fullConnect(const fullConnect& d)
{
     int ctr;
     connections=d.connections;
     min = d.min;
     max = d.max;
     transSize = d.transSize;
     neighbors = new int[connections];
     for (ctr=0; ctr<connections; ctr++) {
          neighbors[ctr] = d.neighbors[ctr];
     }
}
// Assignment operator.
fullConnect fullConnect::operator=(const fullConnect& d)
{
     int ctr;
     //
     // Initialize data elements as in previous connector.
     //
     if (this != &d) {
           connections = d.connections;
          transSize = d.transSize;
          min = d.min;
          max = d.max;
     //
     // Restore neghbors array.
     //
          delete [] neighbors;
          neighbors = new int[connections];
          for (ctr=0; ctr<connections; ctr++) {
               neighbors[ctr] = d.neighbors[ctr];
          }
     }
     return *this;
}
#endif
```

E Sample Parallel Implementation

The code below implements the sample parallel program mentioned in Section 5. The code first declares and builds a serial matrix on each processor using a "C" language compressed sparse row implementation. This serial matrix is used along with the *fullConnector* class from Appendix D to define the parallel matrix components. Parallel vector objects are similarly defined and a parallel matrix vector multiply is performed. The values stored in the matrix and vectors, as well as the results on p processors are detailed in Section 5.

```
#include <stdlib.h>
#include <iostream.h>
#include <assert.h>
#include <strings.h>
extern "C" {
#include "MSops.h"
#include "structures.h"
#include "utilities.h"
#include "elemental.h"
#include "poisson.h"
#include "multiPre.h"
#include "mpi.h"
}

#include "parmat.h"
#include "serialVector.h"
#include "sparseMatrix.h"
#include "fullConnect.h"

main(int argc, char **argv)
{
int ctr;
int myrank;
int numProcs;

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

// Allocate and Initialize a sparse matrix in CSR format.

SPARSEMAT *mat;
mat = initSparse(19, 10, 10);
for (ctr=0; ctr<11; ctr++) {
    mat->rowLocation[ctr] = (2*ctr)+1;
}

for (ctr=0; ctr<9; ctr++) {
    mat->colIndex[2*ctr] = ctr+1;
    mat->colIndex[(2*ctr)+1] = ctr+2;
    mat->element[2*ctr] = (double) ctr+1;
    mat->element[(2*ctr)+1] = (double) ctr+1;
}
mat->colIndex[18] = 10;
mat->element[18] = 10;

// Specify the parallel versions of the matrix and vectors.
// The parallel matrix is templated over the existing
// sparse matrix, mat, and is initialized to the data
// defined above.  The vectors are set up to be of the
// proper rank and vec1 is initialized.

sparseMatrix mat1(mat);
fullConnect conn(10);
parallelMat<sparseMatrix, fullConnect> mat2(mat1, conn);
parallelVec<serialVector, fullConnect> vec1(10, conn);
parallelVec<serialVector, fullConnect> vec2(10, conn);
```

```
// Initialize the source vector.

for (ctr=0; ctr<10; ctr++) {
    vec1.assign(ctr, ctr+1);
}

// Perform a matrix vector multiply in parallel.

multMatVec(mat2, vec1, vec2);

// Print out the results to verify correct operation.

for (ctr=0; ctr<10; ctr++) {
    if (myrank == 0) {
        cout << '[' << myrank << "] " <<
                                        vec2.value(ctr) << endl;
    }
}

// Clean up the sparse matrix.

freeSparse(mat);

MPI_Finalize();

}
```