SAND95-1540J
Unlimited Release
Printed July 1995

Distribution
Category UC-405

Parallel Sparse Matrix-Vector Multiply Software for Matrices with Data Locality[1]

Ray S. Tuminaro[2], John N. Shadid[3], Scott A. Hutchinson[3]

Sandia National Laboratories
Albuquerque, New Mexico 87185-1111

(To be published in Concurrency: Practice and Experience)

Keywords. parallel, software, sparse matrix, performance, Krylov methods

AMS subject classification. 35, 68M20, 68N99, 65Y05

1. This work was partially funded by Department of Energy, Mathematical, Information, and Computational Sciences Division, and was carried out at Sandia National Laboratories operated for the US Department of Energy under contract no. DE-ACO4-94AL85000.

2. Applied and Numerical Mathematics Department

3. Parallel Computational Sciences Department

Parallel Sparse Matrix Vector Multiply Software for Matrices with Data Locality[2]

Ray S. Tuminaro[3], John N. Shadid[4], Scott A. Hutchinson[4]
Sandia National Laboratories
Albuquerque, New Mexico 87185-1111

Keywords. parallel, software, sparse matrix, performance, Krylov methods
AMS subject classification. 35, 68M20, 68N99, 65Y05

Abstract

In this paper we describe general software utilities for performing unstructured sparse matrix-vector multiplications on distributed-memory message-passing computers. The matrix-vector multiply comprises an important kernel in the solution of large sparse linear systems by iterative methods. Our focus is to present the data structures and communication parameters necessary for these utilities for general sparse unstructured matrices with data locality. These type of matrices are commonly produced by finite difference and finite element approximations to systems of partial differential equations. In this discussion we also present representative examples and timings which demonstrate the utility and performance of the software.

1. Introduction

The operation of multiplying a sparse matrix by a vector is one of the most heavily used kernels in scientific computing. Among the computations relying on this operation are applications which numerically solve partial differential equations (PDEs) (e.g. computational fluid mechanics, structural analysis, semiconductor device modeling, etc.) as well as optimization problems. In many cases the sparse matrix-vector operation is performed explicitly on a stored matrix while in other applications, the full matrix is never actually formed. In either case, matrix-vector multiplies are by far the most time consuming aspect of a number of scientific calculations.

Sparse matrix-vector multiply software is not normally available for several reasons. Foremost among these is the fact that the matrix-vector multiply is often not a very complex operation. That

---

2. This work was partially funded by Department of Energy, Mathematical, Information, and Computational Sciences Division, and was carried out at Sandia National Laboratories operated for the US Department of Energy under contract no. DE-ACO4-94AL85000.

3. Applied and Numerical Mathematics Department

4. Parallel Computational Sciences Department

is, users can frequently write their own routine. Another reason is that different users find different sparse matrix storage formats convenient and there are no widely agreed upon formats (although this is beginning to be addressed by the sparse BLAS effort[3]). In fact, not only are several formats in use but many applications require no format at all. These *matrix-free* methods effectively perform matrix-vector products implicitly to avoid storage of the matrix. However, in the parallel computing context, efficiency issues (minimizing communication costs and maximizing computational rates)[4,8,14] and programming issues make such software critical to the development of advanced applications[15]. In this paper, a set of programming tools are described to help transform a global matrix description to a distributed matrix partitioned over multiple processors such that a general unstructured sparse matrix-vector product can be performed. These tools are part of the Aztec [7] parallel iterative solver library which is intended for users who want to avoid cumbersome parallel programming details but who have large sparse linear systems which require an efficiently utilized parallel processing system. Our goal in this paper is to explain these transformation steps, illustrate the significant reduction in programming difficulty associated with these tools, and to demonstrate the overall computational efficiency of the approach. A discussion of other approaches to the implementation of such software can be found in the parallel iterative solver libraries PETSc [1] and PSPARSLIB [12].
In the following discussion it is assumed that the global matrix exhibits data locality in the graph of it's nonzero structure, i.e. it is possible to reorder the matrix into a nearly block diagonal matrix with a limited number of off diagonal block entries. This, data locality, is common for numerical PDE approximations by methods with local support such as finite element (FE) and finite difference (FD) techniques or spectral methods using sparse FE or FD preconditioners. For matrices which do not have this data locality the implementation of similar transformation tools is possible however, the target of a row-wise decomposition for the distributed sparse matrix is no longer appropriate[6].

The paper proceeds in the following manner. Section 2 outlines the data organization and structures needed to perform a low level sparse matrix-vector product. Section 3 defines the user input for the software tools described in following section, Section 4. Section 5 gives a number of high level sample programs using these software tools. Finally, Section 6 gives typical run time data for a parallel application where an iterative method is used in conjunction with a low-level sparse matrix-vector multiply to solve a linear system.

2. Communication and Data Organization

Before discussing the transformation tools, we begin with the data structures and organization needed to perform a low-level parallel matrix-vector product. Although the description follows the data structures needed to use *our* low-level matrix-vector routine, it is intended to be general enough to clarify issues underlying all matrix-vector product software.

In this description we consider the operation $y \leftarrow Ax$ where A is an $n \times n$ sparse matrix and x and y are vectors of length n. It is assumed that a subset of the components in y have been assigned to a particular processor and that this subset is also used to assign the rows of the matrix to processors. This assignment, for a matrix with data locality, can be produced by graph partitioning tools such as Chaco[5] which supports a variety of graph partitioning heuristics. These methods include spectral techniques, geometric methods, multilevel algorithms and the Kernighan-Lin method. All of these approaches may be applied to partition general graphs for mapping onto hypercubes and mesh architectures of arbitrary size. For sparse matrices the graph is implicitly encoded in the nonzero structure of the sparse matrix itself. Using this graph and a utility such as Chaco the assignment of *ownership* for the rows of the matrix, A, and the corresponding components of the vectors, x, y, is obtained after partitioning. In performing a matrix-vector multiply, each processor is only responsible for computing/updating components of y that it owns (termed the owners-compute rule [2, 10, 17]). It is further assumed that all *distributed vectors*, such as x and y, use the same assignment of components to processors. In addition to these assigned components each processor will have copies of additional components (external) which it needs to perform its updates. Thus, each processor contains a subset of the vectors x and y and the components of these vectors are classified into three types:

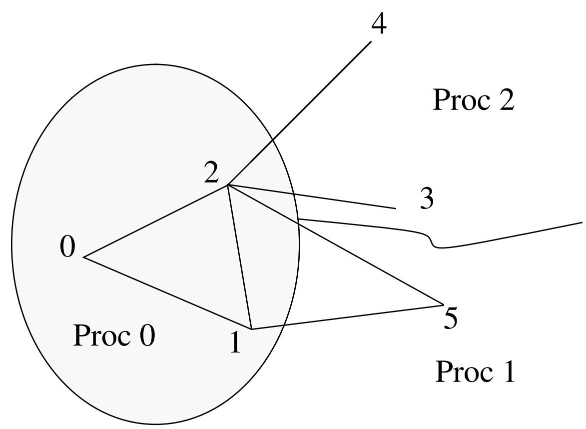
internal - components which are owned (and therefore updated) on this processor without performing any communication. That is, y_i is an *internal* component if it is updated on this processor and if the column index j associated with each nonzero A_{ij} in row i corresponds to a component (x_j) which is also owned by this processor.

border - components which are owned (and therefore updated) on this processor but require some communication before the update can be completed. That is, y_j is a *border* component if it is updated on this processor and if at least one column index j associated with a nonzero A_{ij} found in row i corresponds to a component (x_j) which is not owned by this processor.

external - components which are not owned (and therefore not updated) on this processor but which correspond to column indices associated with nonzeros defined in the rows owned by this processor.

The ordering of the components of a distributed vector on each processor is also of importance. Each of the internal components is local numbered between 0 and N_i-1 where N_i is the number of internal vector elements. Each of the border components is locally labeled between N_i and $N_i + N_b$-1 where N_b is the number of border components. Finally, each of the N_e external vector elements is labeled between N_i+N_b and N_i+N_b+N_e-1. These external components are ordered such that components updated by the same processor are labeled consecutively. It is important to note that on each processor the row and column indices of each matrix nonzero are renumbered using this local ordering scheme. In particular, each row and column index is assigned the same number as the local index assigned to the corresponding vector component. A sample using this ordering scheme is given in Figure 1 for processor 0.

In order to update/receive the external variables on all processors, it is necessary that a few more data structures be specified. First, each processor must know on which processor it's external vector components are updated. Second, each processor must know which vector components that it updates are needed by other processors (and the relative ordering of these components on the neighboring processors). This information is depicted in Figure 2 for the example given in Figure 1. All the required information for the interprocessor communication is encoded in the arrays `send_list`, `neighbors`, `send_length`, `receive_length`. In our example, N_n indicates that processor 0 has two neighbors: *processor 2* and *processor 1*. Processor 0 receives two components from processor 1 and stores them in the lowest numbered external components (i.e. components 3 and 4). Processor 0 receives 1 component from processor 1 which is stored in mem-



matrix:

row	nonzero columns
0	0 1 2
1	1 0 2 5
2	2 0 1 5 3 4

N_i = 1
N_b = 2
N_e = 3

Figure 1: Organization of vector components for a sample problem.

N_n	2
`neighbors:`	2 1
`receive_length:`	2 1
`send_length:`	1 2
`send_list:`	2 1 2

Figure 2: Communication data structures for sample problem.

ory *after* the components received from processor 2 (i.e. component 5). Processor 0 sends 1 component to processor 2 (the first component listed in `send_list`) and 2 components to processor 1 (the components following those sent to processor 2 in `send_list`). Once all this information is available the matrix-vector can be performed [7].

To summarize, a typical matrix-vector product requires the following initialization steps:

1 . Assign a subset of vector components and associated matrix rows to each processor.

2 . Compute and store nonzeros and column indices contained in each of the rows assigned to each processor.

3 . Make a list of external vector components by finding all column indices defined in step 2 above which are not updated by this processor.

4 . Split the update components into a set of internal and border components based on whether the row associated with the element contains any external column indices.

5 . Assign a local ordering to the internal and border components.

6 . Determine which processor updates each external component.

7 . Assign a local ordering to each external component such that components updated by the same processor are numbered contiguously.

8 . Renumber row and column indices in the matrix to use the newly defined local indices.

9 . Determine which vector components need to be sent to other processors (i.e. which update elements are external on other processors).

10 . Initialize `send_list`, `neighbors`, `send_length` and `receive_length`.

From the description above it is evident that there is a large number of initialization tasks required before performing a parallel sparse matrix-vector product. In the remainder of this paper, we discuss software tools which perform tasks 3-10 above using only the information provided in steps 1 and 2.

3. Global Matrix Specification

In order to perform tasks 3-10 itemized above, the user needs to partition the vector components among the processors. Additionally, the user needs to give the sparsity pattern of the global matrix where each processor specifies only the sparsity pattern for its rows. The partitioning of the vector elements can be done in a linear fashion or so-called BLOCK distribution [4](i.e. elements *0* to *N/P-1* are assigned to processor *0*, elements *N/P* to *2N/P-1* are assigned to processor *1*, etc. where *N* is the total number of vector components and *P* is the total number of processors). However, there are much better methods such as those in Chaco[5] which perform this task and reduce communication costs. For the purposes of our software, it is necessary that the user initialize the array `update_pts` on each processor so that it contains the global index of each vector component assigned to that processor. A subroutine read_update_pts is available which reads an input file and assigns vector components to processors (i.e. initializes `N_update_pts` and `update_pts`). The input file contains a list of global indices for each processor preceded by an integer indicating the number of vector components assigned to each processor. Each processor is then responsible for the vector components in `update_pts` as well as the rows of the matrix corresponding to an entry in `update_pts`. The user must further specify the sparsity pattern for each of the rows listed in `update_pts` in terms of the global numbering scheme. The specification of the sparsity pattern is application dependent and is essentially unchanged from its serial counterpart. The sparsity pattern can be encoded in one of two formats: a modified sparse row (MSR) format, or a variable block row (VBR) format. In the MSR format it is assumed that the diagonal is nonzero. The column numbers, `j`, of the other nonzeros in row `update_pts[i]` are given by the user specified array `ija` where

$$j = ija[k],\ ija[i] \leq k < ija[i+1] .$$

In the VBR format the matrix is viewed as having a series of block rows and block columns. Thus, the user partitions blocks of variables over the processors and the array update_pts actually contains the block rows assigned to each processor. The number of rows in block row update_pts[i] is given by rpntr[i+1]-rpntr[i]. The user must now specify which column blocks, j, are nonzero in block row update_pts[i]. This is given by the two arrays bpntr and bindx where

```
j = bindx[k], bpntr[i]  ≤  k < bpntr[i+1] .
```

Figure 3 illustrates the adjacency graph of the matrix corresponding to a simple example. Figure 4 illustrates the initialized data structures corresponding to the adjacency graph and the vector element to processor assignment.

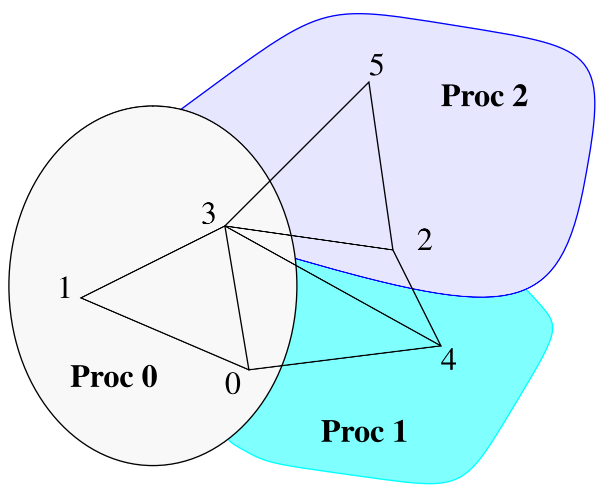


Figure 3: Adjacency graph for a sample problem.

We have omitted the details of how the matrix nonzeros would be specified in MSR or VBR format in order to simplify the description. Detailed descriptions of the MSR and VBR sparse formats can be found in [3]. It should be noted that only the sparsity pattern of the matrix, and not its actual nonzero entries is necessary to completely describe the distributed sparse matrix data structures and their communication pattern. Using only this information, the software that is described in the next section can perform tasks 3-10.

4. Software Tools: Parallel Matrix Specification

The first step in organizing the data for parallel computation is to identify the external vector components needed on each processor. Here, each column number entry must correspond to either a vector component updated by this processor or an external vector component. The subroutine find_local_indices simply checks to see which column entries are not found in the array update_pts. If a column is in update_pts the column number entry is replaced by the index into update_pts corresponding to the entry (i.e. update_pts[new column index] = old column index). If a column is not found in update_pts, it is stored among the list of external variables and the column number entry is replaced by an index into extern_pts (i.e. extern_pts[new column index - $(N_i + N_b)$] = old column index).

```
proc 0:
    update_pts: 0  1  3
    N_update_pts: 3
             matrix:
                     row       nonzero columns
                      0        0  1  3  4
                      1        1  0  3
                      3        3  1  0  4  2  5
proc 1:
    update_pts: 4
    N_update_pts: 1
             matrix:
                     row       nonzero columns
                      4        4  0  3  2

proc 2:
    update_pts: 2  5
    N_update_pts: 2
             matrix:
                     row       nonzero columns
                      2        2  4  3  5
                      5        5  3  2
```

Figure 4: User input to initialize the problem depicted in Figure 3.

Figure 5 shows the new data structures for our sample problem. The nonzero column numbers are in the same order as those given in Figure 4.

On a given processor, each external vector component must also be associated with a processor responsible for updating it. The subroutine find_proc_for_externs queries the other processors to determine which processor update each of its external variables. Given the amount of searching this requires, it is quite important to use an efficient algorithm. Our implementation of find_proc_for_externs assumes that many external variables on a node are updated by the same processor. The basic idea is to identify the updating processor for only a small trial subset of external components and then gather all the trial elements together on all processors using a tree-type gather operation with the number of messages scaling as $O(\log_2(P))$. Once the processors assigned to the trial elements are known, the algorithm checks for other external components on this processor updated by the processors associated with the trial elements. For the remaining unidentified external components the above procedure is repeated. It should also be noted that the assignment of external components to processors is often known. For example, load balancing software will often have this information readily available. In this case, the user can initialize the array `extern_proc` (where `extern_proc[i]` indicates which processor updates `extern_pts[i]`) without calling this routine. Figure 6 gives the `extern_proc` array for our sample problem.

Once the external processors are known, it is possible to reorder the external components such

```
proc 0:
        update_pts: 0  1  3
        extern_pts: 2  4  5
        N_extern_pts: 3
        matrix:
                row      nonzero columns
                 0       0  1  2  4
                 1       1  0  2
                 2       2  1  0  4  3  5
proc 1:
        update_pts: 4
        extern_pts: 0  2  3
        N_extern_pts: 3
        matrix:
                row      nonzero columns
                 0       0  1  3  2
proc 2:
        update_pts: 2  5
        extern_pts: 3  4
        N_extern_pts: 2
        matrix:
                row      nonzero columns
                 0       0  3  2  1
                 1       1  2  0
```

Figure 5: Data organization after calling find_local_indices .

```
proc 0:
         extern_pts:  2  4  5
         extern_proc: 2  1  2
proc 1:
         extern_pts:  0  2  3
         extern_proc: 0  2  0
proc 2:
         extern_pts:  3  4
         extern_proc: 0  1
```

Figure 6: Data organization after calling find_proc_for_externs .

that vector components updated by the same processor are ordered contiguously. The routine order_pts does this reordering as well as the reordering of the update components such that all matrix rows which contain no external components are ordered first. Figure 7 illustrates these orderings for our sample problem.

We are now ready to initialize the communication structures (`send_list`, `neighbors`, `send_length`, `receive_length`) described in Section 2. This is performed by set_message_info where the data structures corresponding to processor 0 are illustrated in Figure 2. Once these arrays are set it is possible to use the local communication routine to update the

```
proc 0:
        update_pts:  0  1  3
        update_indx: 1  0  2
        extern_pts:  2  4  5
        extern_indx: 3  5  4
        extern_proc: 2  1  2
        N_i:1
        N_b:2

proc 1:
        update_pts:  4
        update_indx: 0
        extern_pts:  0  2  3
        extern_indx: 1  3  2
        extern_proc: 0  2  0
        N_i:0
        N_b:1

proc 2:
        update_pts:  2  5
        update_indx: 0  1
        extern_pts:  3  4
        extern_indx: 2  3
        extern_proc: 0  1
        N_i:0
        N_b:2
```

Figure 7: Data organization after calling order_pts .

external vector components whenever necessary. For matrix free applications, the initialization of the local communication routine is essentially the goal of the software. The user can now write their own implicitly defined matrix-vector product using the communication routine and the vector component ordering information. Since one of the motivations for matrix-free products is to minimize storage, a potential user might object to the storage required for the sparsity pattern. However, at this point these sparsity pattern arrays are no longer needed in the computation and so they can be deallocated. Thus, if there is enough space for these arrays in the initialization phase (which can occur before the other arrays are allocated in the computation), it is possible to use the software tools described here.

In situations where the matrix is stored there are two ways in which to use the software described in this section. In one situation, the values of the matrix nonzeros are known without the use of communication. In this case the matrix nonzero values can be specified at the same time that the sparsity pattern is defined. In a second situation communication is needed to compute the values of the matrix nonzeros to be stored (see the finite element example in the next section). At this point the user might proceed with those computations using the communication routine. In either case, the matrix nonzeros must be reordered so that the entries correspond to the reordered local vector components. This is performed by the subroutine reorder_matrix and is illustrated in

Figure 8 for our sample problem.

```
proc 0:
        matrix:
                row      nonzero columns
                 1       1  0  2  5
                 0       0  1  2
                 2       2  0  1  5  3  4

proc 1:
        matrix:
                row      nonzero columns
                 0       0  1  2  3

proc 2:
        matrix:
                row      nonzero columns
                 0       0  3  2  1
                 1       1  2  0
```

Figure 8: Data organization after calling reorder_matrix .

At this point the communication structures and the matrix are in a form suitable for the low-level matrix-vector product (matrix_vector_mult) that we have written [7].

5. Examples

In this section we give a number of sample programs to illustrate the use of the software. The reason for giving these examples is two-fold: 1) to illustrate that the overhead time associated with initializing the communication data structures is minimal when compared to the total simulation time and 2) to illustrate the ease of setting up a number of different problems using the software.

We begin by giving the main program for a 2D Poisson problem in Figure 9. The C-like code fragment in Figure 9 represents the entire main program. The only changes to the working C code are that the declarations and some unimportant parameters in the subroutine calls have been omitted, and we have assumed that subroutines pass parameters by reference (i.e. we do pass pointers to obtain values). We now explain the only subroutines that have not yet been discussed: get_parallel_info , create_matrix , krystart , and krysolve .

The subroutine get_parallel_info returns the number of processors being used and the node id of the processor executing the code. This type of routine is typically provided by the vendor. The subroutines krystart and krysolve initialize and invoke an iterative solver package[7], respectively. The iterative solver package makes extensive use of matrix-vector products using a matrix set up by the software described in this paper. We do not discuss the iterative solver package as it is only used here to give comparisons between the execution time to set up the communication data structures (which represents overhead) and the execution time of an application using the matrix-vector multiply. This leaves only create_matrix that must be supplied by the user along with an input file for read_update_pts .

```
/* Determine processor number and the total number of processors */

   get_parallel_info(proc, nprocs);
   read_update_pts(N_update_pts, update_pts);
   create_matrix(update_pts, a, ija, N_update_pts, N_nonzeros);

   find_local_indices(ija, N_update_pts, update_pts, N_extern_pts,
                        extern_pts);
   find_procs_for_externs(N_update_pts, update_pts, N_extern_pts,
                        extern_pts, extern_proc);
   order_pts(N_internal, N_update_pts, update_indx, N_extern_pts,
                        extern_indx, extern_proc, ija, N_border);
   set_message_info(info, N_update_pts, update_pts, N_extern_pts,
                        extern_pts, extern_proc, update_indx,
                        extern_indx, N_neighbors, send_list);
   reorder_matrix(ija, a, N_update_pts, update_indx, N_extern_pts
                        extern_indx, N_nonzeros);

/* Initialize right hand side and initial guess */

   for (i = 0; i < N_update_pts; i = i + 1) {
      b[update_indx[i]] = (double) update_pts[i];
      x[i] = 0.0;
   }

/* initialize Krylov package */

   krystart(N_update_pts,N_extern_pts,N_internal,N_nonzeros,
                        N_neighbors,send_list,info);

/* Invoke Krylov package */

   krysolve(b, x, ija, a);
```

Figure 9: main program for a Poisson problem.

In Figure 10 we illustrate our create_matrix subroutine. Different matrix problems can be implemented by changing the subroutine add_row which adds the MSR entries corresponding to a new row of the matrix.

The specific subroutine, add_row , for implementing a 5-point 2D Poisson operator on an $n \times n$ grid is shown in Figure 11 (n is a global variable set by the user). With just these few lines of code and the subroutines described earlier, the user implements a distributed matrix corresponding to a parallel 2D Poisson problem. All the communication and variable renumbering is done automatically such that a parallel matrix-vector multiply can be invoked. In fact, there are no references to processors, communication, or any other aspects of parallel computing. Further, it is important to notice that we have not assumed that the assignment of points to processors is structured. The unknown-to-processor mapping could be irregular and require complicated irregular communication. All of these details are hidden from the user.

```
create_matrix(update_pts, a, ija, N_update_pts, N_nonzeros)

   N_nonzeros = N_update_pts+1;
   ija[0] = N_nonzeros;

   for (i = 0; i < N_update_pts; i = i + 1)
      add_row(update_pts[i], i, N_nonzeros, a, ija);
```

Figure 10: create_matrix for a Poisson problem.

To emphasize the ease of use of this software a few more examples are given. We begin with a dis-

```
add_row(row, location, next_nonzero, a, ija)

/* compute (i,j) location of global point 'row' */

   i = mod(row,n);
   j = (row-i)/n;

   old_ptr = next_nonzero;

/* check neighboring points in each direction */
/* and add nonzero entry if the neighbor exits.   */

   for (stride = 1; stride <= n; stride = stride*n) {
      if (i != 0 ) {
         a[next_nonzero]   = -1.0;
         ija[next_nonzero] = row - stride;
         next_nonzero      = next_nonzero + 1;
      }
      if (i != n-1) {
         a[next_nonzero]   = -1.0;
         ija[next_nonzero] = row + stride;
         next_nonzero      = next_nonzero + 1;
      }
      i = j;
   }

    a[location]     = 4.0; /* matrix diagonal */
    ija[location+1] = ija[location] + (next_nonzero - old_ptr);
```

Figure 11: add_row for a 2D Poisson problem.

crete approximation to the 3D Poisson operator on an $n \times n \times n$ cube. Starting from the 2D Poisson program, the only needed changes are: 1) the input file for read_update_pts must now indicate how the n^3 points are to be distributed over the processors and 2) a modified add_row routine must be supplied. In Figure 12, add_row for a 3D Poisson operator is shown where the only lines that are changed from the 2D Poisson operator are highlighted. Thus, by making small changes to add_row it is easy to convert a parallel 2D Poisson matrix to a parallel 3D Poisson matrix. All the communication issues associated with changing from a 2D to a 3D problem are hidden from the user by the software utilities.

```
add_row(row, location, next_nonzero, a, ija)

/* compute (i, j, k) location of global point 'row' */

   i = mod(row, n);
   k = (row-i)/n;
   j = mod(k, n);
   k = (k-j)/n;
   old_ptr = next_nonzero;

/* check neighboring points in each direction */
/* and add nonzero entry if the neighbor exits.   */

   for (stride = 1; stride <= n*n; stride = stride*n) {
      if (i != 0 ) {
         a[next_nonzero]   = -1.0;
         ija[next_nonzero] = row - stride;
         next_nonzero      = next_nonzero + 1;
      }
      if (i != n-1) {
         a[next_nonzero]   = -1.0;
         ija[next_nonzero] = row + stride;
         next_nonzero      = next_nonzero + 1;
      }
      if (stride == 1) i = j;
      else i = k;
   }

   a[location] = 6.0; /* matrix diagonal */
   ija[location+1] = ija[location] + (next_nonzero - old_ptr);
```

Figure 12: add_row for a 3D Poisson problem.

Our next example is again a 2D Poisson operator. This time a higher order 9-point operator is used in the domain interior instead of the 5-point formula. Once again the only code that needs to be changed is add_row which is shown in Figure 13. Although this code is more complicated, none of the complications stem from parallel computing requirements. In this case all of the communication complications arising from the use of different stencils (e.g. points in row *2* need information from row *0* and row *1* while points in row *0* need information from row *1* but not from row *2*) and wider stencils are hidden from the user.

We conclude this section with two applications in which the values of the matrix entries are not defined inside create_matrix . In particular, create_matrix only defines the sparsity pattern (i.e. `ija`). In the first example, a matrix-free 2D Poisson operator, the matrix entries are not defined in order to save storage. The subroutine add_row is identical to that shown in Figure 11 except that the 3 lines referencing `a` are removed. Here it is the user's responsibility to replace the provided matrix-vector multiply. Figure 14 gives an example of a matrix-vector multiply corresponding to the 2D Poisson operator. The subroutines exchange_bdry and find_index are also supplied with the software where exchange_bdry performs the local communication necessary for the matrix-vector multiply and find_index finds the index within an array corresponding to a value. In this sample we have made no effort to improve efficiency. For structured programs the data lay-out is often quite simple and the user can avoid using order_pts and eliminate the references to update_indx and extern_indx in Figure 14. Further, it may be possible to replace

```
add_row(row, location, next_nonzero, a, ija)

/* compute (i, j) location of global point 'row' */

   i = mod(row, n);
   j = (row-i)/n;
   old_ptr = next_nonzero;
   a[location] = 0.0;

   for (stride = 1; stride <= n; stride = stride*n) {

/* If we are on the bottom boundary, create entries for 2nd order
  discretization. Otherwise create high order discretization.*/

      if ((i != 0) && (i != n-1)) {
         if (i != 1) {
            a[next_nonzero]   = 1.0/12.0;
            ija[next_nonzero] = row - stride*2;
            next_nonzero      = next_nonzero + 1;
         }
         a[next_nonzero]   = -16.0/12.0;
         ija[next_nonzero] = row - stride;
         next_nonzero      = next_nonzero + 1;
         if (i != n-2) {
            a[next_nonzero]   =   1.0/12.0;
            ija[next_nonzero] = row + stride*2;
            next_nonzero      = next_nonzero + 1;
         }
         a[next_nonzero]   = -16.0/12.0;
         ija[next_nonzero] = row + stride;
         next_nonzero      = next_nonzero + 1;
      }
      else if (i != n-1) {
         a[next_nonzero]   = -1.0;
         ija[next_nonzero] = row + stride;
         next_nonzero      = next_nonzero + 1;
      }
      else if (i != 0) {
         a[next_nonzero]   = -1.0;
         ija[next_nonzero] = row - stride;
         next_nonzero      = next_nonzero + 1;
      }
      if ((i != 0) && (i != n-1)) a[location] += 30.0/12;
      else a[location] += 2.0;
      i = j;
   }

   ija[location+1] = ija[location] + (next_nonzero - old_ptr);
```

Figure 13: add_row for a 2D high order Poisson problem.

find_index by a simple expression taking advantage of the relationship between the global numbering and the local numbering in structured problems. We do not discuss the details of optimizing the matrix-free code for structured problems. However, we note that in situations where the data organization is simple, the user might quickly write an unoptimized version using all of the software described here and then proceed by making incremental changes to produce a more efficient code by eliminating some of the calls to software described in this paper.

```
matrix_free_mult(N_update_pts, update_pts, update_indx,
                 N_extern_pts, extern_pts, extern_indx, x, y)

      exchange_bdry(x);

      for (k = 0; k < N_update_pts; k = k + 1) {

        /* compute the location */

         point = update_pts[k];
         i     = mod(point,n);
         j     = (point-i)/n;

         y[k] = 4.0*x[k];

         for (stride = 1; stride <= n; stride = stride*n) {
           if (i != 0) {
               new = find_index(point-stride,update_pts,N_update_pts);
               if (new == NOT_FOUND) {
                 new = find_index(pointstride,extern_pts,
                      N_extern_pts);
                 new = extern_indx[new];
              }
              else new = update_indx[new];
              y[k] = y[k] - x[new];
           }

           if (i != n-1) {
              new = find_index(point+stride,update_pts,N_update_pts);
              if (new == NOT_FOUND) {
                 new = find_index(point+stride, extern_pts,
                      N_extern_pts);
                 new = extern_indx[new];
              }
              else new = update_indx[new];
              y[k] = y[k] - x[new];
           }
           i = j;
        }
     }
```

Figure 14: matrix-free Poisson operator.

Our last example is also a 2D Poisson equation. However, this time the computational domain is irregular and the discretization is obtained using triangular finite elements. Additionally, in this implementation the matrix entries cannot be computed without communication. In this case it is necessary to define only the sparsity pattern in create_matrix and to use the sparsity pattern in conjunction with the transformation tools to set up the communication data structures. Once this is done, another subroutine is called to perform the necessary communication and compute the matrix entries. Figure 15 depicts create_matrix while Figure 16 depicts an additional subroutine matrix_fill that must be called before krysolve in Figure 9.

Once again, we have not made any effort to optimize these routines. In both figures the new lines that have been added specifically for a parallel implementation are highlighted. Specifically, create_matrix and matrix_fill have been created by taking a serial program that provides the finite element discretization, splitting this program over two subroutines and then adding a few new lines for the parallel implementation. The only additional change corresponds to replacing a

```
read_triangles();
init_msr(a, ija, N_update_pts);

for (k = 0; k < N_triang; k = k + 1) {
    for (i = 0; i < 3; i = i + 1) {
      row = find_index(T[k][i], update_pts, N_update_pts);
         for (j = 0; j < 3; j = j + 1) {
          if (row  != NOT_FOUND)
             add_to_element(row, T[k][j], 0.0, a, ija, i==j);
          }
      }
    }

compress_matrix(a, ija, N_points);
```

Figure 15: create_matrix for Poisson finite element problem.

```
/* read the x and y coordinates from an input file */

  for (i = 0; i < N_update_pts; i = i + 1) {
     read_from_file(x[update_indx[i]],y[update_indx[i]]);

  exchange_bdry(x);
  exchange_bdry(y);

  for (k = 0; k < N_triang; k = k + 1) {
     for (i = 0; i < 3; i = i + 1) {
        row = find_index(T[k][i], update_pts, N_update_pts);
        if (row == NOT_FOUND)
           row = find_index(T[k][i], extern_pts, N_extern_pts);
           T[k][i] = extern_indx[row];
        }
        else T[k][i]= update_indx[row];
     }
  }

  for (k = 0; k < N_triang; k = k + 1) {
     setup_Ke(Ke, x[T[k][0]], y[T[k][0]], x[T[k][1]],
           y[T[k][1]], x[T[k][2]], y[T[k][2]]);

     for (i = 0; i < 3; i = i + 1) {
        for (j = 0; j < 3; j = j + 1) {
           if (T[k][i] < N_update_pts)
              add_to_element(T[k][i], T[k][j], Ke[i][j],
                          a, ija, i==j)
        }
     }
 }
```

Figure 16: matrix_fill for Poisson finite element problem.

single data file containing the triangle connectivity read using read_triangles by a set of data files containing the triangle connectivity for each processor. We do not discuss the details of this program but only wish to draw attention to the number of lines that need changing to convert the serial application to a parallel application. Most of the main routines such as setup_K e which compute the element contributions and add_to_element which stores the element contributions in the MSR data structures remain the same. In fact, almost all the new lines of code correspond to adding the communication (exchange_bdry) and the conversion of global index values to local index values with the help of find_index .

A set of sample programs have been given to set up a variety of different matrices. In each case the user sets up the matrix using global column indices without concerning themselves with the details of data lay-out and communication. The purpose of showing these examples is to demonstrate how the user needs only to supply a few lines of code in order to implement all of the examples. With the exception of the finite element example, the complete code excluding variable declarations and a few subroutine parameters have been given. In the finite element code all the new code that needed to be added for creating a parallel version has been shown.

6. Performance

In this section we give some performance numbers on a 1840 processor Intel Paragon. Our primary goal is to illustrate that the overhead incurred using the software described in this paper is not significant when compared to the simulation cost of a typical application (taken as the iterative solution of the 3D Poisson equation). Therefore, we compare the run time associated with setting up distributed matrix communication data structures with the actual run time of the iterative solver. The results represent worst case situations due to the low simulation time. In particular, the solution of a Poisson system represents a fairly small supercomputing task. We feel that most application problems involve far more computational work (e.g. many linear system solutions or single linear systems that take many more iterations than the simple Poisson equation). However, while overall run times may be greater, overhead times required to set up the communication software should remain similar. This is because the communication set up phase is only related to the sparsity pattern and the example gives a good estimate of overhead time as a function of the number of matrix nonzeros and the number of processors. In fact, in many applications the sparsity pattern does not change during the computation even though many computing hours are used solving different linear systems. Since the sparsity pattern remains the same, the communication set up phase need only be performed one time.

We first show that the MFlop/s rating and scaled speedups are high for the iterative solver package where scaled speedup on p processors is given as

$$S(p, N) = p * \frac{\textit{(time for 1 iteration of a problem with N points on 1 processor)}}{\textit{(time for 1 iteration of a problem with Np points on p processors)}}$$

Two sets of examples are given. One corresponds to solving a single 3D Poisson equation using the MSR format. The second example corresponds to solving 8 Poisson operators at the same time using the VBR format with 8 x 8 blocks. Our VBR results are representative of actual applications where a number of unknown quantities occur at a given mesh point. Normally, the VBR format should be used to solve multiple PDE equations which are coupled to each other (e.g. Navier-

Stokes equations). To mimic the MFlop/s performance for this type of problem, we treat the 8 x 8 blocks as dense (even though they are really diagonal for our simple test problem). These results are representative since the MFlop/s rating will be equivalent to real applications and the solution time for our test problem will actually be shorter due to the simplicity of the system of equations. All of the examples are solved via the conjugate gradient method[9] where parallelization is accomplished using a box decomposition (i.e. each processor is assigned an $\sqrt[3]{N/p} \times \sqrt[3]{N/p} \times \sqrt[3]{N/p}$ block of unknowns). In these examples a ninth-order least squares polynomial preconditioner [11] is used in conjunction with the conjugate gradient method. To estimate the MFlop/s rating we use a 32 x 32 x 32 grid within each processor for the MSR problem and a 16 x 16 x 16 grid within each processor for the VBR problem. By counting the Flops and dividing by the run time we get

p	MSR			VBR		
	n	scaled speedup	MFlop/s	n	scaled speedup	MFlop/s
1	32	1	4.2	16	1	20.85
8	64	7.8	32.7	32	7.9	165.3
64	128	60.4	254.0	64	62.4	1301.7
512	256	481.6	2024.5	128	505.5	10539.0

Table 1: Scaling of example iterative solver.

an idea of the MFlop/s rate. For example, 15 preconditioned conjugate gradient iterations on 1 processor require 19.98 seconds and approximately 83 million floating point operations. This corresponds to 4.2 MFlop/s on a single processor using the MSR format. In Table 1, we illustrate the MFlop/s and scaled speedups for the MSR and VBR formats. Both the MFlop rating (especially for the VBR example) and the scaled speedups are fairly high and compare well with other sparse applications running on the Intel Paragon[8,14,15]. Thus we can conclude that the iterative solvers perform well on a single processor or on many processors.

In Table 2 and 3 we compare the time of the matrix transformation software with that of the iterative solvers for the same set of test problems. The main result from Table 2 is that the overhead-

p	n	`local_indx`	`find_ext`	`order_pts`	`order_mtrx`	all tools	cg	iters
1	32	.74	0.0	.06	.10	0.92	20.00	15
8	64	1.12	0.22	.06	.30	1.74	37.17	28
64	128	1.36	0.61	.07	.31	2.41	72.48	54
512	256	1.36	0.80	.07	.31	2.62	143.6	103

Table 2: Run time (in seconds) of software tools: Scaled MSR problem sizes.

time is small compared to the conjugate gradient iterations (2% - 5%). In general, the time to order the points and to reorder the matrix is approximately the same as all of the test problems have the same number of points per processor. The time for the local index calculation rises with the number of processors and then levels out once we have reached 64 processors. This initial rise is due to a increase in the number of external points (e.g. on 1 processor there are no external points, on 8 processors there are only 3 external surfaces). Once the number of external points per processor stops increasing, the time for the local index calculation levels out. The only overhead cost that increases is the time required to find the processors associated with each of the external

p	*n*	`local_indx`	`find_ext`	`order_pts`	`order_mtrx`	all tools	cg	iters
1	16	.11	0.0	.008	.04	.15	17.7	10
8	32	.17	.04	.009	.50	.72	27.2	15
64	64	.19	.13	.011	.55	.88	50.1	27
512	128	.21	.20	.011	.55	.96	77.0	42

Table 3: Run time (in seconds) of software tools: Scaled VBR problem sizes.

nodes. As we can see the increase in this time is fairly modest and even on 512 processors is not dominant. Of course the small increase in overhead time is more than offset by an increase in the simulation run time as the problem size grows.

The VBR results shown in Table 3 are similar in that the overhead time associated with the matrix transformation is insignificant compared to the iterative solvers. In the case of block computations it is the matrix reordering calculation which requires the most time for the transformation software. This is because all the other set up routines operate in a block fashion (e.g. order_pts orders blocks) and so these times are insignificant compared to the time required to operate on an 8 x 8 block. By contrast, reorder_matrix moves blocks around and so its run time increases as the block size increases.

In the tables above all of the simulations correspond to reasonably efficient computations from a parallel computing point of view. This is because each processor contains many points. In the next two tables we demonstrate that the overhead costs remain small even when there are only a few points per processor. In Table 4, we illustrate run times when the number of processors is fixed at 512 and the problem size varies. In Table 5, we give run times where the problem size is held fixed

p	MSR				VBR			
	n	all tools	cg	iters	n	all tools	cg	iter
512	32	0.06	0.2	15	16	.04	.35	10
512	64	0.11	1.1	28	32	.06	.76	15

Table 4: Run time (in seconds) of software tools on 512 processors.

p	MSR				VBR			
	n	all tools	cg	iters	n	all tools	cg	iter
512	128	0.46	10.2	54	64	.18	7.16	27
512	160	0.75	23.2	66	80	.29	15.86	33
512	200	1.31	54.0	82	104	.49	41.22	41
512	256	2.62	141.6	103	128	.96	76.93	42

Table 4: Run time (in seconds) of software tools on 512 processors.

and the number of processors varies. It is clear that in each case the overhead time is small compared with simulation time. Thus, we would anticipate that the cost of using the transformation software is negligible for most users. Before concluding we point out that in those cases where the

p	MSR				VBR			
	n	all tools	cg	iters	n	all tools	cg	iter
1	32	0.92	20.0	15	16	.15	17.7	10
8	32	0.25	2.8	15	16	.10	2.8	10
64	32	0.07	0.8	15	16	.03	.66	10
512	32	0.06	0.2	15	16	.04	.35	10

Table 5: Run time (in seconds) of software tools: fixed problem sizes.

overhead time is not negligible, it is often possible to reduce the overhead execution time by changing some of the options. For example, if we do not order the update points into two sets (internal and border) but instead label them all as border points, the run time for both order_pts and reorder_matrix drops drastically. In other situations where information on the neighboring processors is available we need not call the subroutine find_proc_for_externs.

7. Conclusion

We have described software tools that transform serial sparse-matrix descriptions into parallel distributed sparse matrices. The tools enable users to easily use parallel matrix kernels like the sparse matrix-vector multiply. A number of examples have been given which demonstrate a significant reduction in programming effort when using these tools. Additionally, it has been shown that the overhead time is low when compared to the simulation time of most real applications.

8. References

1. Balay, B., Gropp, W., McInnes, L.C., and Smith, B., "PETSc 2.0 Users Manual", Technical Report, ANL-95/11, Argonne National Laboratory, 1996
2. Callahan, D., and Kennedy, K. "Compiling programs for distributed-memory multiprocessors", Journal of Supercomputing, 2:151-169, October 1988
3. Carney, S., Heroux, M. A., and Li, G., "A Proposal for a Sparse BLAS Toolkit- SPARKER Working Note #2", August 1993
4. G. C. Fox, S. Hiranandani, K. Kennedy, C.Koelbel, U. Kremer, C. Tseng, M. Wu, "FORTRAN D Language Specification", CRPC Technical Report CRPC-TR9009, April 1991
5. Hendrickson, B. A., and Leland, R. W., "The Chaco User's Guide - Version 1.0", Technical Report, Sandia National Laboratories, SAND93-2339, November 1993.
6. Hendrickson, B. A., Leland, R. W., and Plimpton, S. J., "An Efficient Parallel Algorithm for Matrix-Vector Multiplication", to appear in Int. J. High Speed Computing, 1995.
7. Hutchinson, S. A., Shadid, J. N. and Tuminaro, R. S., "The Aztec User's Guide - Version 1.0", Technical Report SAND95-1559, Sandia National Laboratories, Albuquerque, NM (1995)
8. Jones, J. T., and Plassmann, P. E., "Computational results for parallel unstructured mesh computations", Technical Report, Argonne National Laboratory, MCS-P466-0894.
9. Golub, G. H. and Van Loan, C. F., "Matrix Computations", Johns Hopkins University Press, Baltimore, 1983.
10. Rogers, A. and Pingali, K. "Process decomposition through locality of reference", in Proc of the SIGPLAN'89 Conference on Programming Language Design and Implementation, OR, June 1989
11. Saad, Y., "Practical use of polynomial preconditioning for the conjugate gradient method", SIAM J. Sci & Stat. Comput., 6(4):865-881, Oct 1985.
12. Saad, Y. and Malevsky, "P-SPARSLIB: a portable library of distributed memory sparse iterative solvers", Technical Report, Dept. of Computer Science, University of Minnesota, 1995
13. Shadid, J. N., Hutchinson, S. A. and Tuminaro, R. S., "Parallel Sparse Matrix-Vector Multiply Software: Performance Issues", in preparation.
14. Shadid, J. N., Hutchinson, S. A., and Moffat, H. K., "Parallel performance of a preconditioned CG solver for unstructured finite element applications" Proceedings of the Colorado Conference on Iterative Methods, Breckenridge, Colorado, April 5-9, 1994
15. Shadid, J. N., Hutchinson S. A., Moffat, H. K., Hennigan, G. L., Hendrickson, B. A., and Leland, R. W., "A 65+ Gflop/s unstructured finite element simulation of chemically reacting flows on the Intel Paragon", Proceedings of Supercomputing '94, Washington, DC, pp 673-679, Nov. 14-18, 1994.
16. Wu, J., Das, R., Saltz, J., Beryryman, H., and Hiranandani, S., "Distributed memory compiler design for sparse problems", IEEE Transactions on Computers, 44, 6, 1995
17. Zima, H. Bast, H.J, and Gernt, M., "SUPERB: A tool for semi-automatic MIMD?SIMD parallelization", Parallel Computing, 6:1-18, 1988