SPOOLES: An Object-Oriented Sparse Matrix Library *

Cleve Ashcraft† Roger Grimes‡

1 Overview

Solving sparse linear systems of equations is a common and important component of a multitude of scientific and engineering applications. The **SPOOLES** software package[1] provides this functionality with a collection of software objects and methods. The package provides a choice of three sparse matrix orderings (minimum degree, nested dissection and multisection), supports pivoting for numerical stability (when required), can compute direct or drop tolerance factorizations, and the computations are based on BLAS3 numerical kernels to take advantage of high performance computing architectures. The factorizations and solves are supported in serial, multithreaded (using POSIX threads) and MPI environments.

The first step to solving a linear system $AX = B$ is to construct "objects" to hold the entries and structure of A, and the entries of X and B. **SPOOLES** provides a flexible set of methods to assemble a sparse matrix. The "input matrix" object allows a choice of coordinate systems (by rows, by columns, and other ways), flexible input (input by single entries, (partial) rows or columns, dense submatrices, or any combination), resizes itself as necessary, and assembles, sorts and permutes its entries. It is also a distributed object for MPI environments. Matrix entries can be created and assembled on different processors, and methods exist to assemble and redistribute the matrix entries as necessary.

There are three methods to order a sparse matrix: minimum degree, generalized nested dissection and multisection. The latter two orderings depend on a domain/separator tree that is constructed using a graph partitioning method. Domain decomposition is used to find an initial separator, and a sequence of network flow problems are solved to smooth the separator. The qualities of our nested dissection and multisection orderings are comparable to other state of the art packages.

Factorizations of square matrices have the form $A = PLDUQ$ and $A = PLDL^TP^T$, where P and Q are permutation matrices. Square systems of the form $A + \sigma B$ may also be factored and solved (as found in shift-and-invert eigensolvers), as well as full rank overdetermined linear systems, where a QR factorization is computed and the solution found by solving the semi-normal equations.

*This research was supported in part by the DARPA Contract DABT63-95-C-0122, the DoD High Performance Computing Modernization Program Common HPC Software Support Initiative and by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Energy Research of the U.S. Department of Energy.

†`cleve.ashcraft@boeing.com`, Boeing Shared Services Group, P. O. Box 24346, Mail Stop 7L-22, Seattle, Washington 98124.

‡`roger.g.grimes@boeing.com`, Boeing Shared Services Group P. O. Box 24346, Mail Stop 7L-22, Seattle, Washington 98124.

[1]**SPOOLES** is an acronym for **SP**arse **O**bject-**O**riented **L**inear **E**quations **S**olver.

When pivoting for stability is enabled, the magnitudes of the off-diagonal entries in L and U are bounded above by a user supplied constant. The "front matrix" is our object that computes a sparse factorization, stores the factor entries, and performs the forward and backsolves. The factorizations are done using a panel-based block general sparse algorithm in order to efficiently support pivoting for stability. The solves are done using a two-dimensional submatrix-based algorithm. The factorization can be direct, where the fronts and submatrices are computed and stored as dense submatrices, or approximate, where small entries are dropped from the factors and submatrices are stored as sparse matrices.

The factorization and solves may be computed in three modes: serial, multithreaded using POSIX threads, or with MPI. There is considerable code overlap between the serial, threaded and MPI versions. In all cases, the important computational kernels are based on BLAS-3 like operations. "What" is done, i.e., what data structures take part in what computations, is the same across all three environments. "Who" does what, i.e., what thread or processor does what computation, is the same for the multithreaded and MPI versions. The MPI code adds little more than explicit message passing of data structures.

The development of this software was funded by DARPA[2] and the DoD[3] with the express purpose that others (academic, government, industrial and commercial) could easily incorporate the data structures and algorithms into application codes. The **SPOOLES** library is totally within the public domain; there are absolutely no licensing restrictions as with other software packages. The web page `http://www.netlib.org/linalg/spooles` contains the latest release of the package — full source code and postscript files of the user and reference manuals.

2 Design Philosophy

The complexity of over 120,000 lines of C code is managed by thirty-eight objects and over 1300 different functions. Encapsulation is the cardinal rule — an object knows and operates on itself, but has very limited knowledge of other objects. Encapsulation is a custom enforced by discipline rather than a language feature. Inheritance is only mildly missed. We have three different tree objects that could benefit from inheritance, but as a whole, the object hierarchy is very flat. What is more important are the informal "protocols" (from Objective-C) or "interfaces" (from Java) that the objects follow. In short, it is possible to follow good object-oriented design principles without explicit language support.

Each object has its own directory and three subdirectories: one holds source code, one holds driver programs to exercise and validate the object's functionality, and one hold LaTeX and HTML files for documentation. The LaTeX files form either a stand-alone document or a chapter of the larger reference manual. Certification programs for the numeric objects frequently generate output files that may be run through MATLAB to check the validity of numeric functions. We are thus able to have a great deal of confidence in the building blocks of the larger computations such as the factors and solves.

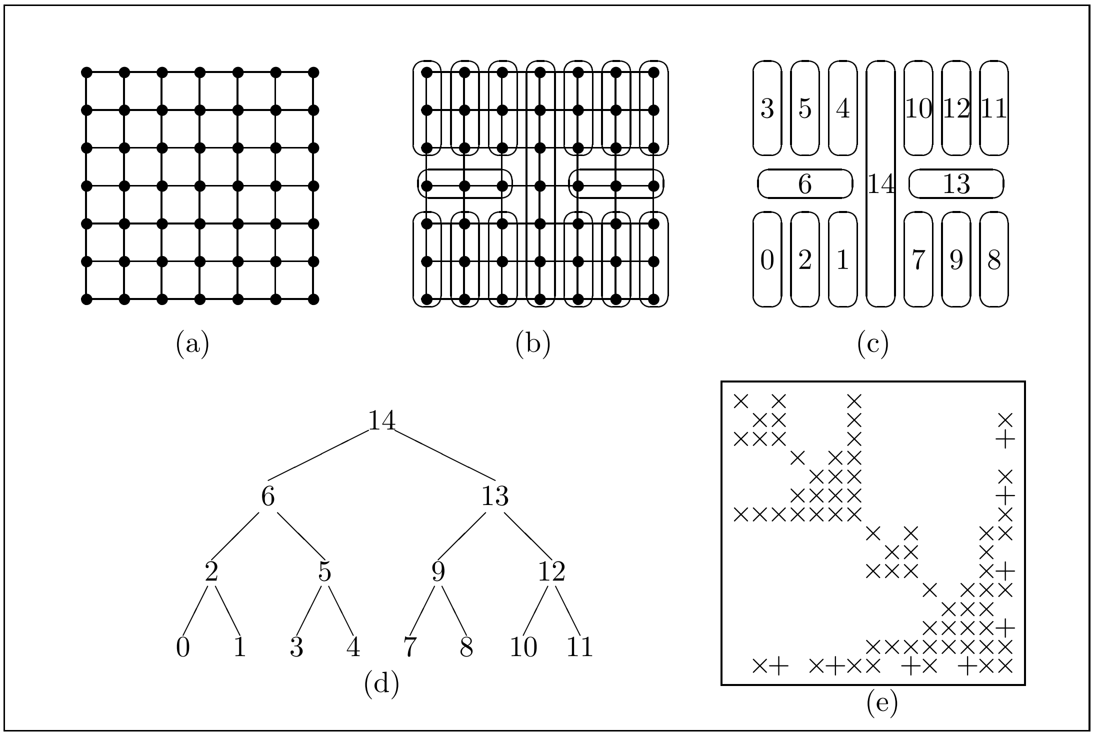
3 Matrix Orderings

The library has three options for ordering sparse matrices: multiple minimum degree, generalized nested dissection, and multisection [6]. Our minimum degree ordering is very similar in quality and ordering time to Liu's `GENMMD` software [15].

[2]DARPA Contract DABT63-95-C-0122.
[3]DoD High Performance Computing Modernization Program Common HPC Software Support Initiative.

To generate both nested dissection and multisection orderings, we perform a graph partitioning step and construct a *domain/separator tree*. The process is recursive in nature. We examine a subgraph, find a separator for the subgraph, split the subgraph into two or more components, link the subgraphs to the separator with a parent-child relation, and repeat the operation on each of the new subgraphs. A parameter dictates how large a subgraph can be before it is not split further into subgraphs, thus the process is really an incomplete recursive dissection. Figure 3 shows the process on a 7×7 grid. Subgraphs with three or fewer nodes were not split, but left as "domains".

FIG. 1. *Recursive Dissection process: (a)* 7×7 *grid, (b) domains and separators overlaid on grid, (c) domains and separators numbered, (d) domain/separator tree, (e) the* 15×15 *domain/separator block matrix,* $\times$ *denotes nonzero block in* A, $+$ *denotes nonzero fill-in block in* L *or* U.



Once the domain/separator tree has been constructed, we evaluate a nested dissection or multisection ordering using our minimum degree code. This second step is really an ordering with constraints, [16], where vertices are assigned to a *stage* when they can be eliminated. For a nested dissection ordering, a vertex in a domain or separator is ordered before any vertices that are in its parent, grandparent, etc, separators, i.e., the relative precedence of the separators is respected. For a multisection ordering, vertices in all domains are in the first stage, while vertices in all separators are in the second stage.

The nested dissection and multisection orderings have been tested against two other state-of-the-art software packages, **METIS** [14] from the University of Minnesota, and the SGI's **EXTREME** software (a descendent of the **CHACO** package [12] with improvements [13].) The qualities of the three orderings are very close over a large selection of structural analysis matrices, some very large.

4 Factors and Solve

We consider the matrix A as a block matrix,

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,N} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,N} \\ \vdots & \vdots & \ddots & \cdots \\ A_{N,1} & A_{N,2} & \vdots & A_{N,N} \end{bmatrix}$$

and its factorization $A = LDU$ is blocked in the same manner. Each block corresponds to a *front* or *supernode*, in the contexts of the multifrontal [7] and supernodal general sparse algorithms. Recall that A is sparse, and that most of the submatrices will be zero, and those not zero will likely be sparse. The triangular factor matrices L and U will also be sparse, (though not as sparse as A) and will suffer *fill-in.* It is important to only store and compute with true nonzero entries of L, D and U, important enough that we define some new notation. The matrix $U_{J,\partial J}$ contains the nonzero columns of $U_{J,J+1:N}$ and $L_{\partial J,J}$ contains the nonzero rows of $L_{J+1:N,J}$. The ∂ sign in ∂J is meant to suggest *boundary* of J.

We use the *Crout* reduction variant of Gaussian elimination, where at step J, the J'th block of D, the J'th row of U, and the J'th column of L are computed using the preceding blocks of D, rows of U and columns of L. The matrix equation for this step is

$$\begin{bmatrix} A_{J,J} & A_{J,\partial J} \\ A_{\partial J,J} & 0 \end{bmatrix} = \sum_{I \le J} \begin{bmatrix} L_{J,I} D_{I,I} U_{I,J} & L_{J,I} D_{I,I} U_{I,\partial J} \\ L_{\partial J,I} D_{I,I} U_{I,J} & 0 \end{bmatrix},$$

where the $L_{*,I}$, $D_{I,I}$ and $U_{I,*}$ matrices have already been computed. The computation of $L_{\partial J,J}$, $L_{J,J}$, $D_{J,J}$, $U_{J,J}$ and $U_{J,\partial J}$ is done in two steps. We first compute the *fully assembled* front matrix

$$T_J = A_J - \sum_{I<J} T_J^I, \tag{1}$$

which is short for

$$\begin{bmatrix} T_{J,J} & T_{J,\partial J} \\ T_{\partial J,J} & 0 \end{bmatrix} = \begin{bmatrix} A_{J,J} & A_{J,\partial J} \\ A_{\partial J,J} & 0 \end{bmatrix} - \sum_{I<J} \begin{bmatrix} L_{J,I} D_{I,I} U_{I,J} & L_{J,I} D_{I,I} U_{I,\partial J} \\ L_{\partial J,I} D_{I,I} U_{I,J} & 0 \end{bmatrix}.$$

T_J^I contains the update from front I to front J. Given the fully assembled front matrix T_J, we then perform the factorization internal to the front, and compute the factor submatrices via the formulae below.

$$T_{J,J} = L_{J,J} D_{J,J} U_{J,J}, \quad T_{J,\partial J} = L_{J,J} D_{J,J} U_{J,\partial J}, \quad \text{and} \quad T_{\partial J,J} = L_{\partial J,J} D_{J,J} U_{J,J}$$

The T_J matrix is contained in the `Chv` object. The name comes from *chevron*, the arrowhead-shaped military insignia. When A is symmetric, only the $T_{J,\partial J}$ matrix and upper triangle of $T_{J,J}$ are stored. A `Chv` object can contain real or complex entries. The storage is aligned by rows of the upper triangle and columns of the lower triangle, to allow easy assembly of original entries and delayed rows and columns, simple pivot searches and tests, and vector based computations during the internal factorization. Changing the particular storage format of the `Chv` object would have little *ripple* effect throughout the rest of the library. Indeed, an earlier version of the `Chv` object stored $T_{J,J}$, $T_{J,\partial J}$ and $T_{\partial J,J}$ as dense submatrices, and changing the storage scheme to the present one required no modification to other objects.

Once $D_{J,J}$, $L_{J,J}$, $L_{\partial J,J}$, $U_{J,J}$ and $U_{J,\partial J}$ are computed, they are stored in `SubMtx` objects. This object is designed to hold small to medium sized dense or sparse matrices, which form the *atomic* data structures of the factors. The `SubMtx` object can hold real or complex entries, and has row and column indices. The format can be dense or sparse by rows or columns (for $L_{\partial J,J}$ and $U_{J,\partial J}$), dense subrows or subcolumns (useful for $L_{J,J}$ and $U_{J,J}$), and diagonal or block diagonal with 1×1 and 2×2 pivots to hold $D_{J,J}$.

The `Chv` objects are temporary objects that exist only during the factorization to hold T_J, the working storage for a front. The `SubMtx` objects that hold $L_{J,J}$, $D_{J,J}$ and $U_{J,J}$ are persistent — they form part of the factor matrix object. The `SubMtx` objects that hold $L_{\partial J,J}$ and $U_{J,\partial J}$ are not persistent, as we now describe.

The factorization as it now stands, where $D_{J,J}$, $L_{J,J}$, $L_{\partial J,J}$, $U_{J,J}$ and $U_{J,\partial J}$ are stored in `SubMtx` objects, has no parallelism for the forward and backsolves aside from that available from the domain/separator tree. This is because the `SubMtx` object is *atomic*, i.e., its entries are not split among processors, and its computations (e.g., $y_J := y_J - U_{J,\partial J} x_{\partial J}$ during the backward solve) are performed in a non-preemptable fashion. To achieve parallelism within a separator, it is necessary to split the $U_{J,\partial J}$ and $L_{\partial J,J}$ matrices into smaller $U_{J,K}$ and $L_{K,J}$ submatrices. When pivoting is enabled, this separation cannot be done during the factorization, because rows and/or columns may move from front to front before they are eliminated, and so the exact decomposition of the rows of $L_{K,J}$ and columns of $U_{J,K}$ is not known until the factorization completes. Thus, a post-processing step converts the the $U_{J,\partial J}$ and $L_{\partial J,J}$ matrices into smaller $U_{J,K}$ and $L_{K,J}$ submatrices.

4.1 A parallel factorization

There are three atomic operations during a serial factorization

- load $A_{J,J}$, $A_{J,\partial J}$ and $A_{\partial J,J}$ into T_J,
- update T_J using $D_{I,I}$, $L_{\partial I,I}$ and $U_{I,\partial I}$ for those I whose $L_{J,I}$ and/or $U_{I,J}$ is not zero,
- and compute $D_{J,J}$, $L_{J,J}$, $L_{\partial J,J}$, $U_{J,J}$ and $U_{J,\partial J}$ from the fully assembled T_J.

The first step is not a candidate to parallelize because there are so few operations involved. The third cannot be parallelized if pivoting for stability is to be supported in an efficient manner. The second step is most rich in floating point operations, and it is this step that we parallelize.

In the multithreaded and parallel environments there is a concept of *ownership* of the fronts. One thread or processor owns a front and is responsible for assembling any original entries and computing the factor submatrices once it is assembled. We use a *map* function to define ownership, $q = m(J)$ means that processor q owns front J. We chose to follow the fan-in paradigm [3], where the owner of front I performs the updates from I to all other fronts. In a distributed environment, the factor submatrices do not travel between processors during the factorization. Instead, processor q computes T_J^q, the *aggregate update matrix* that consists of all updates from fronts owned by q to front J. We can now rewrite equation (1) for a parallel environment.

$$T_J = A_J - \sum_q T_J^q \qquad \text{where} \qquad T_J^q = \sum_{\substack{I<J \\ m(I)=q}} T_J^I \tag{2}$$

The fully assembled front is composed of A_J, the original matrix entries, plus a sum of some number of T_J^q temporary matrices, each a `Chv` object. It is the T_J^q matrices that are computed independently, and in the MPI environment, are communicated between processors.

The parallel factorization can be considered a simultaneous traversal of the front tree by all the threads or processors. When visiting front J, thread q computes as much of T_J^q as possible, and if it owns front J, it assembles T_J. (In the MPI environment, processor q receives the T_J^* aggregate update matrices from the other processors.) When T_J is complete, it is then factored. Inside a thread or processor, one does not wait at a front until it can be completed. Each thread or processor maintains a queue of fronts that are ready to have action taken on them. A front is removed from the queue when all work for it on that thread or processor is complete, and a front is added to the queue when all work on its children is complete. A *lookahead* parameter [4] allows a thread or processor to look further up the tree in order to reduce the idle time.

4.2 Pivoting for stability

To ensure stability of the factorization, we bound the magnitudes of entries in *both* L and U by a user supplied tolerance. For the symmetric case, the algorithm is described and analyzed in [5]. For the nonsymmetric case, it is known as rook pivoting [8, 17]. Partial pivoting, where rows (or columns) are interchanged, bounds entries only in L (or U).

In most cases, pivoting adds little cost to a serial or parallel factorization, as long as the pivot tolerance and/or matrix structure does not induce too many postponed rows and columns. We recommend bounding the magnitude of entries in L and U by 100, or 1000, and for most matrices we have encountered that require pivoting for stability, an additional $5 - 10\%$ of factorization time is necessary.

When T_J is completely assembled, pivot elements can be chosen from $T_{J,J}$. For symmetric and Hermitian matrices, we use the Fast Bunch-Parlett algorithm [5] to find a 1×1 or 2×2 pivot block. For a nonsymmetric matrix, we locate a local maximum element in $T_{J,J}$, one with the largest magnitude in its row and column. It is then tested, where we evaluate a bound on the magnitude of the entries of L and U in its row and column of T_J that would arise if the pivot were to be eliminated. If the pivot passes the test, the T_J matrix is permuted so the pivot lies in the upper left corner. The first row(s) and column(s) are scaled by the inverse of the pivot block, and a rank-1 or rank-2 update is made to the remainder of the T_J matrix. The process then continues on the uneliminated rows and columns in the front.

It is possible that we cannot eliminate all of the rows and columns in $T_{J,J}$. (For a trivial example, consider the case when $T_{J,J}$ is zero.) Permuting the pivot elements into the upper left part of $T_{J,J}$ is equivalent to replacing T_J by $\widehat{T}_J$, where

$$\widehat{T}_J = \begin{bmatrix} P_J & 0 \\ 0 & I \end{bmatrix} T_J \begin{bmatrix} Q_J & 0 \\ 0 & I \end{bmatrix} = \begin{bmatrix} P_J & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} T_{J,J} & T_{J,\partial J} \\ T_{\partial J,J} & 0 \end{bmatrix} \begin{bmatrix} Q_J & 0 \\ 0 & I \end{bmatrix}.$$

If we think of J as representing an index set (a set of rows and columns), then we can partition J into two sets: J_e, a set of *eliminated* rows and columns, and J_d, a set of *delayed* rows and columns. (For a symmetric or Hermitian matrix, the row and column index sets for J_e are identical, for we choose a 1×1 pivot from the diagonal, or a 2×2 pivot with two diagonal elements. For a nonsymmetric matrix, the row and column index sets for J_e will likely be different, but we use J_e to represent both. The same remarks hold for J_d.) The permuted front matrix $\widehat{T}_J$ can be written as a 3×3 block matrix,

$$\widehat{T}_J = \begin{bmatrix} L_{J_e,J_e} D_{J_e,J_e} U_{J_e,J_e} & L_{J_e,J_e} D_{J_e,J_e} U_{J_e,J_d} & L_{J_e,J_e} D_{J_e,J_e} U_{J_e,\partial J} \\ L_{J_d,J_e} D_{J_e,J_e} U_{J_e,J_e} & \widehat{T}_{J_d,J_d} & \widehat{T}_{J_d,\partial J} \\ L_{\partial J,J_e} D_{J_e,J_e} U_{J_e,J_e} & \widehat{T}_{\partial J,J_d} & 0 \end{bmatrix}$$

The first block row and column of $\widehat{T}_J$ contain factor submatrices. The trailing 2×2 block, which contains rows and columns whose elimination is delayed, is also stored in a `Chv` object. It must be merged into the front that is the parent of J in the front tree. (For our example from Figure 3, any postponed rows and columns from front 2 will be merged into front 5, the parent of front 2.

Equation 1 can be modified to account for delayed rows and columns.

$$T_J = A_J - \sum_{I<J} T_J^{I_e} \oplus \sum_{p(I)=J} \widehat{T}_J^{I_d} \tag{3}$$

where I_e are the eliminated rows and columns of I,

$$T_J^{I_e} = \begin{bmatrix} L_{J,I_e} D_{I_e,I_e} U_{I_e,J} & L_{J,I_e} D_{I_e,I_e} U_{I_e,\partial J} \\ L_{\partial J,I_e} D_{I_e,I_e} U_{I_e,J} & 0 \end{bmatrix} \quad \text{and} \quad \widehat{T}_{I_d} = \begin{bmatrix} \widehat{T}_{I_d,I_d} & \widehat{T}_{I_d,\partial I} \\ \widehat{T}_{\partial I,I_d} & \end{bmatrix},$$

and $\oplus$ is a symbol for "concatenate" or "extend". The upper left hand block of T_J will contain rows and columns from J as well as I_d for each child I of J.

4.3 Parallelism and pivoting

A parallel factorization that pivots for stability is very similar to one that does not pivot. The equation for the fully assembled front matrix is

$$T_J = A_J - \sum_{q} T_J^q \oplus \sum_{p(I)=J} \widehat{T}_{I_d}, \qquad \text{where} \qquad T_J^q = \sum_{\substack{I<J \\ m(I)=q}} T_J^{I_e}. \tag{4}$$

The fully assembled front matrix is the sum of A_J (the original entries of A), the T_J^q matrices (updates from different threads or processors), and the $\widehat{T}_{I_d}$ (delayed rows and columns from the children fronts). All that is different is that the $\widehat{T}_{I_d}$ matrices (stored in `Chv` objects, just like the T_J^q matrices) must be communicated between processors.

5 Results

The **SPOOLES** library has been ported and tested to a variety of platforms: SparcStations under Solaris, SGI Origin 2000, IBM SP2, HP V-class multiprocessors, the Cray T3E, and a network of Intel workstations running MPI under Linux. The serial performance is good. Benchmarks show that the factor and solves run at about the same speed as a respected commercial code, **BCSLIB-EXT**, inside the CSAR Nastran Finite Element software. The solver has been incorporated into a second finite element package for commercial release, where it has been ported to use threads on multiple processor WindowsNT workstations.

There are two drawbacks to the **SPOOLES** library.

- Presently, it is an in-core code, there is no capability to compute the factorization or perform solves while keeping part of the factor matrices on a disk file. This limits the effectiveness of the library for very large linear systems.

- Because of the design decision to efficiently support pivoting for numerical stability, the factorization is performed using a *one-dimensional* data decomposition where the front matrices are stored as `Chv` objects, as opposed to a two-dimensional submatrix decomposition (as is done during the solves). It is well known that a 1-D decomposition suffers in a parallel environment due to increased message traffic and a longer critical path through the execution graph [18]. Benchmarking studies confirm this, for there is decent performance for small numbers of processors, but in order to maintain efficiency as the number of processors increases, the problem size must increase at a much faster rate.

To illustrate these two points, Table 1 contains some results for solving linear systems arising from 27-point operators on 3-d $n \times n \times n$ grids. The results were obtained from John Wu of NERSC on the Cray T3E. The lower left corner entries, large n and small numbers of processors, are empty because of memory limitations. The lower right corner entries, large n and large numbers of processors, are empty because of MPI resource limitations.

TABLE 1
Average factorization megaflops per processor for 27-point operators on an $n \times n \times n$ grid using the nested dissection ordering and a subtree-subcube map on the Cray T3E.

	# of processors									
n	1	2	4	8	16	32	64	128	256	512
20	100	84	68	50	25	17	8	4	1	
24	111	91	80	60	40	23	12	6	2	
28	117	101	90	61	49	29	16	8	4	2
34	130	116	103	84	61	36	22	12	6	3
40		128	115	99	74	48	29	16	8	
48			129	113	88	61	38	21	11	
56				122	100	72	47	27		

The matrices were ordered using nested dissection. The maximum size of a front was set to 64 internal rows and columns, and the fronts were mapped to processors using a subtree-subcube mapping. Matrices of this form are an interesting test collection. The ratio of factor operations to factor entries is fairly high, which leads to good computational rates, and mimics many of the finite element matrices we see with several degrees of freedom per grid point. The front tree is very well balanced, except for the lower regions of the tree when using many processors. The factor operations are $O(n^6)$, while the number of bytes communicated is $O(pn^4)$, so the ratio of computation over communication is $O(n^2/p)$. Keeping this ratio constant while doubling the number of processors means that n must increase to $\sqrt{2}n$, and the size of the linear system from n^3 to $2\sqrt{2}n^3$.

The (n, p) pairs in this table were chosen to follow this rule: "down-2, over-1" diagonals have nearly constant n^2/p values. Big $O()$ notation can be deceptive when lower order terms have a significant influence. Table 2 contains the actual ratio of operations to bytes communicated during the factorizations.

Better performance, as measured by a large computation to communication ratio, can be achieved in two ways. One can turn to a submatrix-based algorithm [10] used by the **PSPASES** [9] and **MUPS** [1] software. These methods have $O(\sqrt{p}n^4)$ communication, which leads to a ratio of $O(n^2/\sqrt{p})$. But recall, submatrix-based methods are inefficient when pivoting for stability is required.

TABLE 2

Operations per byte communicated during the factorization of 27-point operators on an $n \times n \times n$ grid using the nested dissection ordering and a subtree-subcube map

	# of processors								
n	2	4	8	16	32	64	128	256	512
20	597	112	37	15	7	4	3	2	1
24	869	162	54	21	10	6	4	2	2
28	1198	222	74	29	13	7	4	3	2
34	1800	323	110	44	19	10	6	4	3
40	2525	466	154	61	27	13	7	5	4
48	3690	679	224	88	39	18	10	6	4
56	5076	932	308	121	53	25	12	7	5

There is another method that maintains the 1-D decomposition for the fronts (which is crucial for pivoting) but requires $O(\sqrt{p}n^4)$ communication. The fan-both method [2] communicates T_J^q aggregate update matrices, but *also* $U_{J,\partial J}$ and $L_{\partial J,J}$ factor submatrices. It is parameterized via the relation $p = p_1 \cdot p_2$, where p is the number of processors. The fan-in method is a special case where $p_1 = 1$. The fan-out method [11] is a special case where $p_2 = 1$. Table 3 contains the operations/byte-communicated results for the $56 \times 56 \times 56$ grid using up to 512 processors. There is some improvement to be gained over the fan-in method for moderate to large numbers of processors. We intend to provide a fan-both implementation in a future release of the library.

TABLE 3

Operations per byte communicated for the fan-both factorizations on a $56 \times 56 \times 56$ grid. The anti-diagonals contain results for a constant number of processors. The boldface number on an anti-diagonal is the best for that number of processors.

	p_2									
p_1	1	2	4	8	16	32	64	128	256	512
1	—	**5076**	**932**	**308**	121	53	25	12	7	5
2	352	279	213	**141**	**82**	42	22	12	8	
4	110	95	83	68	**48**	**31**	19	12		
8	47	42	39	34	28	**21**	**15**			
16	24	14	20	19	16	14				
32	14	12	12	11	10					
64	9	8	8	8						
128	7	6	6							
256	6	6								
512	5									

6 Summary

The **SPOOLES** library is an object oriented C library for solving real or complex sparse linear systems of equations. It contains state-of-the-art ordering algorithms, and computes matrix factorizations and solves in serial, multithreaded (using POSIX threads) and distributed environments using MPI. The package was funded by DARPA and DoD and is completely within the public domain, free of any license issues. More information can be found at the `netlib` website, `http://www.netlib.org/linalg/spooles`.

References

[1] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Tech. Rep. TR/PA/98/22, CERFACS, 1998.

[2] C. Ashcraft, *The fan-both family of column-based distributed Cholesky factorization algorithms*, in Graph Theory and Sparse Matrix Computation, Springer-Verlag, 1993, pp. 159–190.

[3] C. Ashcraft, S. Eisenstat, and J. W. H. Liu, *A fan-in algorithm for distributed sparse numerical factorization*, SIAM J. Sci. Stat. Comput., 11 (1990).

[4] C. Ashcraft, S. C. Eisenstat, J. W. H. Liu, B. W. Peyton, and A. H. Sherman, *A compute-ahead fan-in scheme for parallel sparse matrix factorization*, in Fourth Canadian Supercomputing Symposium (1990), D. Pelletier, ed., June, 1990, pp. 351–361.

[5] C. Ashcraft, R. G. Grimes, and J. G. Lewis, *Accurate symmetric indefinite linear equation solvers*, Tech. Rep. ISSTECH-95-029, Boeing Computer Services, 1995. Accepted for publication in SIAM J. Matrix. Anal.

[6] C. Ashcraft and J. W. H. Liu, *Robust ordering of sparse matrices using multisection*, SIAM J. Matrix Analysis and Applic., 19 (1998), pp. 816–832.

[7] I. Duff and J. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 6 (1983), pp. 302–325.

[8] L. V. Foster, *The growth factor and efficiency of Gaussian elimination with rook pivoting.* Accepted for publication in *J. Comp. and Appl. Math.*

[9] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar, *Design and implementation of a scalable parallel direct solver for sparse symmetric positive definite systems*, in Eight SIAM Conference Conference on Parallel processing, 1997.

[10] A. Gupta, G. Karypis, and V. Kumar, *Highly scalable parallel algorithms for sparse matrix factorization*, IEEE Transactions on Parallel and Distributed Systems, 8 (1997), pp. 502–520.

[11] M. HEATH, ed., *Communication reduction in parallel sparse Cholesky on a hypercube*, SIAM Press, 1987.

[12] B. Hendrickson and R. Leland, *The Chaco user's guide*, Tech. Rep. SAND93-2339, Sandia National Laboratories, Albuquerque, NM, 1993.

[13] B. Hendrickson and E. Rothberg, *Improving the runtime and quality of nested dissection ordering*, SIAM J. Sci. Comput., 20 (1998), pp. 468–489.

[14] G. Karypis and V. Kumar, *Metis 4.0: Unstructured graph partitioning and sparse matrix ordering system*, tech. rep., Department of Computer Science, University of Minnesota, 1998. Available on the WWW at URL *http://www.cs.umn.edu/˜metis*.

[15] J. W. H. Liu, *Modification of the minimum degree algorithm by multiple elimination*, ACM Trans. on Math. Software, 11 (1985), pp. 141–153.

[16] ——, *On the minimum degree ordering with constraints*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 1136–1145.

[17] L. Neal and G. Poole, *A geometric analysis of Gaussian elimination II.*, Linear Alg. and Appl., 173 (1992), pp. 239–264.

[18] R. Schreiber, *Scalability of sparse direct solvers*, in Graph Theory and Sparse Matrix Computation, Springer-Verlag, 1993, pp. 191–211.