TOWARDS A SCALABLE HYBRID SPARSE SOLVER [*]

ESMOND G. NG [†] AND PADMA RAGHAVAN [‡]

Abstract. Consider the solution of very large, sparse linear systems. The most popular techniques can be broadly classified as either "direct" or "iterative." When the sparse matrix is symmetric and positive definite, direct methods use Cholesky factorization while iterative methods rely on Conjugate Gradients. Our goal is to develop a scalable and memory-efficient hybrid of the two methods that can be implemented with high-efficiency on both serial and parallel computers and be suitable for a wide-range of problems. We discuss our overall design with emphasis on performance and scalability issues and report on progress to date.

Key words. sparse matrix factorization, conjugate gradient, incomplete Cholesky, preconditioners

AMS(MOS) subject classifications. 65F05, 65F50.

1. Introduction. The efficient solution of large, sparse linear systems on high-performance multiprocessors continues to be the subject of on-going research. There is no single method that is consistently superior across application domains and computing platforms. For example, for linear systems associated with elliptic partial differential equations, domain decomposition methods provide a natural and efficient parallel formulation [4, 18]. However, domain decomposition relies on close tie-ins to the discretization of partial differential equation and mesh formulation. Consequently, it may not be suitable as a general-purpose "black-box" sparse solver. Both Krylov subspace (KSP) solvers [2, 5, 10, 24, 52] and direct solvers [7, 11] can be used as "black-box" solvers. But, once again, both classes of methods have serious limitations. Direct solvers are not memory scalable; memory requirement typically grows nonlinearly because some of the zero entries will become nonzero. On the other hand, KSP iterative solvers are memory scalable and are easily parallelized, but their convergence can be very slow or fail altogether depending on the spectral properties of the sparse matrix. We contend that robust, scalable sparse solvers suitable for a wide variety of large-scale applications on high-performance computers require a spectrum of methods that range from pure iterative to pure direct methods. We are developing such parallel "flexible," hybrid solvers, which are based on KSP with incomplete matrix factorization as preconditioners. Our goal is to combine the inherent scalability and parallelism of KSP iterative solvers with robust preconditioners obtained using data-structures,

algorithms, and graph-techniques from sparse direct solvers. Our solvers will be implemented using the message-passing model and MPI [9], and will be portable across high-performance multiprocessors and networks of workstations.

The designs and implementations of KSP solvers as well as direct solvers are simpler and better understood for linear systems in which the coefficient matrix is symmetric and positive definite. Our design for a scalable hybrid solver focuses on this class of matrices. For symmetric and positive definite matrices, the conjugate gradient method is typically the most appropriate and the most reliable iterative method. It is well understood and several implementations are available [41, 54]. One of the most popular and broadly-applicable preconditioners is obtained from incomplete Cholesky (IC) factorization [5, 17, 34, 36]. If $A = LL^T$ is the Cholesky factorization, then an incomplete Cholesky factor $\hat{L}$ of A is obtained by discarding some or all of the fill in L. The main issue is that of developing an IC implementation that can be efficient for a wide spectrum of preconditioning requirements, i.e., from an $\hat{L}$ with no more nonzero entries than in A to one with a substantial fraction of the fill in L. On parallel platforms another critical issue is that of efficiently applying the preconditioner. At each iteration, this amounts to using parallel substitution schemes to solve systems of the form $\hat{L}y = b$ and $\hat{L}^T x = y$. Due to the large latency of communication on multiprocessors triangular solution is typically very inefficient. We plan to develop latency tolerant schemes to apply the preconditioner $\hat{L}$ using techniques that have proved successful for repeated triangular solution using L.

The purpose of this paper is to describe our goals, the key performance and scalability issues, and to report on progress to date. In particular, Section 2 provides background and Section 3 concerns efficient IC implementations. Section 4 concerns exploiting parallelism for both constructing and applying the incomplete Cholesky preconditioner. Section 5 contains some concluding remarks.

2. Background. We begin with an overview of sparse Cholesky factorization followed by a description of the two main types of incomplete Cholesky factorizations.

Sparse Cholesky. As mentioned earlier, Cholesky factorization causes some zero entries in A to fill in and become nonzero in L. The amount of fill in L depends on the ordering (i.e., numbering) of rows/columns of A. Consequently, the first step in sparse Cholesky factorization is that of ordering the matrix to reduce fill. The second step is that of symbolic factorization, which determines the zero–nonzero structure of L. Knowing the structure of L prior to numerical factorization will allow an efficient and compact data structure to be set up for storing the nonzero entries in L. A variety of efficient, graph-theoretic algorithms exist for the ordering step [1, 13, 14, 31, 40] and the symbolic factorization steps [11, 15, 32]. The third and the fourth steps are, respectively, numeric factorization and numerical triangular solution using the precomputed data-structure.

Assume that A has been appropriately ordered to reduce fill, and consider the zero–nonzero structure of the corresponding Cholesky factor L. The columns

of L often can be grouped into "supernodes." Each supernode contains a set of consecutive columns that have essentially the same zero–nonzero structure. More specifically, if a supernode contains columns $i, i+1, \ldots, j$, then the lower triangular submatrix of L induced by these columns and the corresponding rows is dense. Moreover, within these columns, the entries in a row numbered larger than j are either all nonzero or all zero. In other words, all the nonzero entries from the columns in a supernode form a dense lower trapezoidal submatrix. Consequently, the zero–nonzero structure of all the columns of a supernode can be represented by that of, for example, the first column of the supernode. Thus, a supernode partition allows a very compact representation of the zero–nonzero structure of L. Another important advantage of a supernode partition is that the dense block structure can be exploited so that sparse Cholesky factorization can be expressed in terms of dense matrix operations; the dense matrices correspond to the dense blocks induced by the supernodes. This allows the use of cache-efficient computational kernels as in BLAS [27], and it reduces the amount of indirect addressing. The reader is referred to [39] for further details on use of dense matrix kernels in the implementation of sparse Cholesky factorization. A supernode partition can be computed very efficiently from the zero–nonzero structure of A during the symbolic factorization step [15, 30]. An example of a "supernode partition" of a matrix A is shown in Figure 1. The matrix corresponds to the model five-point 7×7 finite-difference grid. We use this problem as an illustrative example in later sections as well.

Incomplete Cholesky. Two popular schemes for computing incomplete Cholesky preconditioners are: the *level-of-fill* approach [35] and the *drop-threshold* approach [37, 56].

The level-of-fill method (henceforth, ICL) is best described using the notion of fill-paths in the graph model of sparse Cholesky factorization [42, 47]. The reader is referred to the book by George and Liu for graph terminology and basic concepts in sparse Cholesky factorization [11]. The undirected graph of A, denoted by $G(A)$, has vertices numbered $1, 2, \ldots, n$ corresponding to the rows/columns of A. There is an edge in $G(A)$ joining vertices i and j $(i \neq j)$ if and only if $A_{ij} \neq 0$. A path in $G(A)$ from vertex i to vertex j is called a *fill-path* if all intermediate vertices on the path are numbered less than $\min(i,j)$. Using an inductive argument, it is easy to show that an element L_{ij} $(i > j)$ is nonzero if and only if there is a fill-path between vertices i and j in $G(A)$ [48]. Define the length of a fill-path to be the number of edges on the path. Note that fill-paths are not unique; for each $L_{ij} \neq 0$ there may be more than one fill-path between vertices i and j in $G(A)$.

For ICL using k levels of fill, we permit off-diagonal entry $\hat{L}_{ij} \neq 0$ if and only if there is at least one fill-path in $G(A)$ connecting vertices i and j with length less than or equal to $k + 1$. Note then that the zero–nonzero structure of $\hat{L}$ using level-of-fill is well defined given an integer $k \geq 0$ and an ordering of the columns/rows (hence a numbering of the vertices of $G(A)$). Thus, the zero–nonzero structure of $\hat{L}$ can be determined using "symbolic" techniques independent of the actual
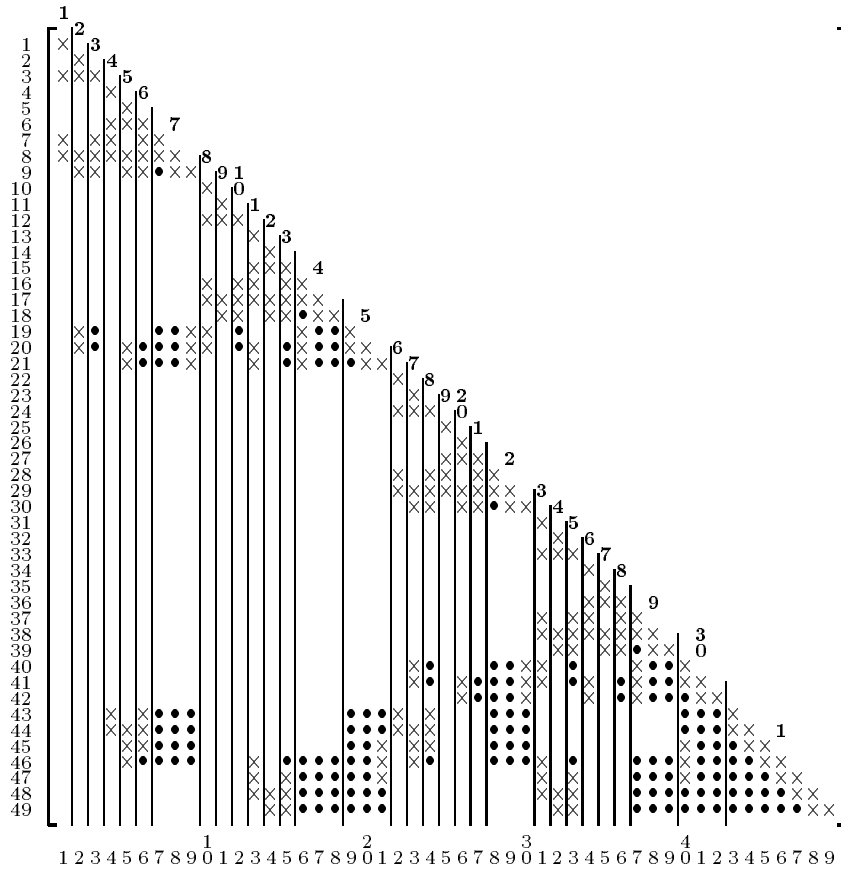
FIG. 1. *Supernodes in a matrix defined on a 7×7 nine-point grid ordered by nested dissection. (Each $\times$ and $\bullet$ represents a nonzero in A and a fill in L, respectively. Numbers over diagonal entries label supernodes.)*

numerical values. With a predetermined zero–nonzero structure of $\hat{L}$ available, one computes, retains, and manipulates only the nonzero entries in the incomplete Cholesky factor. However, as we will see later, the complexity of the "incomplete symbolic factorization" is proportional to the number of floating-point operations required to compute the incomplete Cholesky factorization numerically. Unlike the complete Cholesky factorization, the incomplete Cholesky factor may not possess any supernodal structure.

The drop-threshold approach is a numeric approach; henceforth we refer to it as ICT. Consider a left-looking or inner-product Cholesky factorization. The current column is computed as it is in complete Cholesky factorization, using updates from earlier columns of the factor and finally scaling by the diagonal element. However, nonzero entries in the computed column that have magnitudes smaller than a given threshold are discarded [37, 56]. There are several ways to define and use the threshold. For example, the threshold can be simply a non-negative value or a non-negative factor relative to some measure of the size of A (such as the norm of A or $\max_{i,j} |A_{ij}|$). Other restrictions can also be imposed, such as discarding from each column all but some m nonzero entries satisfying the threshold condition [51], where m is typically a small constant such as five or ten.

3. An Efficient IC Implementation. We begin with some comments on the advantages and disadvantages of the two types of incomplete Cholesky factorization in terms of (1) efficiency of the implementation, and (2) effectiveness of the preconditioning. We then provide an overview of our cache-efficient implementation and discuss its performance.

Computing an incomplete Cholesky factor using the drop-tolerance approach (ICT) is similar to performing the numerical factorization step in complete Cholesky. The only difference is that nonzero entries with small magnitude are discarded by applying a threshold condition. It therefore seems natural that the ordering step for limiting fill is essential for ICT, as it is for complete factorization. However, unlike complete sparse Cholesky factorization, the zero–nonzero structure of $\hat{L}$ is revealed *only* during numeric factorization. Hence, there is no direct counterpart to the symbolic factorization step for ICT.

At first glance, computing an incomplete Cholesky factorization using level-of-fill (ICL) and computing the sparse complete Cholesky factorization appear to be remarkably similar as well. In both cases, after the ordering step, the zero–nonzero structure of the factor can be determined using symbolic techniques so that a compact data structure can be set up for storing the nonzero entries. Next we can perform numerical factorization using the predetermined data structure. However, a major difference between ICL and sparse complete Cholesky factorization lies in the cost of symbolic factorization. For sparse complete Cholesky factorization, the symbolic factorization step is typically very efficient and takes relatively little time. This is because it can exploit the supernodal structure [32], which can be computed very efficiently from the structure of A. On the other hand, the columns of an ICL generally do not share common sparsity structure

and hence do not possess any supernodal structure. That is, the technology for identifying and exploiting supernodes in L will not applied directly to $\hat{L}$. Thus a symbolic scheme for computing the zero–nonzero structure of $\hat{L}$ in ICL must maintain the lengths of the shortest fill-paths, and must essentially simulate numerical incomplete factorization. Consequently the symbolic phase has the same time complexity as numeric incomplete factorization.

When comparing the quality of preconditioning provided by ICL with that provided by ICT, much depends on the nature of the sparse matrices and in particular on the applications where the matrices come from. In our earlier work we evaluated the performance of ICL and ICT on the set of sparse matrices described in Section 5 [16]. We concluded that ICT preconditioners generally are more robust and have a wider range of applicability than ICL preconditioners. As a result our recent work has focused on efficient implementations of ICT. In future, we plan to revisit ICL with the goal of developing a suitable approximation which can be used in a preprocessing step followed by our ICT scheme.

Based on our experience with sparse complete Cholesky factorization, implementing ICT in a column-by-column manner will not be cache efficient on modern computers; a blocked scheme is essential for reducing indirect indexing and making good use of the memory hierarchy. We have therefore designed and implemented a blocked version of ICT, which exploits technology developed for sparse direct solvers. In particular, it attempts to take advantages of dense matrix kernels. Like complete sparse Cholesky factorization, we perform the ordering step to limit fill. However, as discussed earlier, there is no direct counterpart to the symbolic factorization step. Since our goal is to develop a blocked ICT algorithm that can be implemented using dense matrix kernels, we decided to use the supernodal structure in a complete sparse Cholesky factor as a means to produce the blocks that are needed in ICT. Thus, prior to computing the ICT we compute the supernodal partitioning, as well as perform the complete symbolic factorization (for L).

The numerical phase of our ICT scheme is left-looking, and the computation is organized in terms of blocks of columns that have essentially identical zero–nonzero structure. The block structure in $\hat{L}$ is derived from the supernodal structure in the complete Cholesky factor L. The strategy for dropping nonzero entries is designed so that a block-column structure is preserved in $\hat{L}$. We provide further details below. First, note that dropping nonzero elements in individual columns (as described in the previous paragraph) will not in general result in blocks of columns with essentially the same sparsity structure (as in a supernode). Few columns, if any, could be combined to form nontrivial supernodes. To deal with this problem, we modify the drop-threshold strategy to use not only the magnitude of the nonzero entries but also the supernodal structure of the complete Cholesky factor L. Consider a set of columns of $\hat{L}$ corresponding to a supernode of L. In our blocked ICT, we either drop or retain an entire row of nonzero entries (within the set of columns). Then the entire row in the set of columns is retained even if only one element does not satisfy the drop-threshold condition. This blocked

6

version of the drop-threshold criterion is obviously more restrictive than the non-blocked version. In particular, it is highly undesirable to apply the drop-threshold criterion to a large supernode (which can be as large as the square-root of the matrix dimension for many 2-dimensional finite element matrices) since it will be more unlikely for all the nonzero entries to satisfy the dropping criterion. To alleviate this problem, we subdivide each supernode into blocks of either 1, 2, 4, or 8 columns and organize the computation around these blocks. (Thus, a block-size of 1 is equivalent to a non-blocked column-oriented ICT.) We have found that block sizes greater than or equal to 16 typically lead to more fill, and hence more operations, offsetting the reduction in run-time from cache-efficiency.

Our implementation of blocked ICT requires storage for $\hat{L}$, storage for the zero–nonzero structure of $\hat{L}$ (that exploits the block structure), and storage for the temporary array in which new columns of $\hat{L}$ are computed. Our implementation can use a variety of drop threshold criteria as well as allow other features such as diagonal-modification, etc.

The detailed performance of our blocked ICT code is described in [38]. We summarize some of the highlights in this report. Our test suite consisted of 18 matrices from three different application areas. The first six are bcsstk matrices that arise from finite-element methods in structural mechanics (see the Harwell-Boeing collection [6]). The next six spa-dis matrices are from visible-surface interpolation in a regularization framework using thin-plate and membrane splines [19]. The last six xerox matrices are from a materials science simulation used at Xerox Corporation. We discuss performance relative to a direct solver using Cholesky factorization. Thus, the performance of the direct solver (either time or number of nonzero entries) is set at 1; the hybrid is better if the time for ICTCG or the size of $\hat{L}$ is smaller than 1. We consider best cases over a range of threshold values; the best cases are either (1) least total time instances (ICTCG) or (2) least fill instances. Table 1 contains geometric means for the best-case instances. On average, with block-size 8 (block size 1), our least-storage ICTCG requires 51% (137%) more execution time than the direct solver with 26% (15%) the storage requirement. The largest block-size gives the best ICTCG time performance, requiring on average approximately the same time as direct solution but with 50% fewer nonzero entries. The unblocked code requires on average 87% more time than direct solution with 72% fewer nonzero entries.

On average our hybrid results in performance improvement, but in some instances a significant amount of the fill in L is required for convergence. In these instances, a direct method (given sufficient memory) is typically faster because it can make better use of the cache and does not have the overheads of thresholding. Now unblocked ICTCG is noticeably slower than blocked ICTCG; for such problems it may be useful to consider even larger block-sizes in combination with adaptations of the block-threshold condition. Figure 2 shows in detail the relative performance of ICT and ICTCG for two hard-to-precondition problems from the test-suite.

TABLE 1

Average (relative) values for best instances with respect to $|\hat{L}|$ and ICTCG time (MMD orderings).

blocksizes →	1	2	4	8	1	2	4	8
Geometric Means of Relative Performance Measures								
	min. relative nonz.				relative ICTCG time			
bcsstk	0.24	0.29	0.32	0.36	3.17	2.58	2.76	2.60
spadis	0.72	0.77	0.80	0.82	3.64	2.48	2.05	1.82
xerox	0.02	0.04	0.05	0.06	1.16	0.70	0.73	0.73
overall	0.15	0.20	0.23	0.26	2.37	1.65	1.60	1.51
	min. relative ICTCG time				relative nonzeroes			
bcsstk	2.68	2.08	1.76	1.59	0.40	0.49	0.63	0.75
spadis	3.63	2.37	1.79	1.55	0.73	0.83	0.90	0.94
xerox	0.68	0.54	0.47	0.46	0.08	0.10	0.14	0.18
overall	1.87	1.38	1.14	1.04	0.28	0.34	0.43	0.50

4. Parallel Implementation Issues for Incomplete Cholesky Factorization.

On multiprocessor systems, an implementation of incomplete Cholesky factorization (ICT) should be cache-efficient on each processor as well as utilize the large-grain parallelism from sparsity. Perhaps, a more important issue is that of effectively applying the incomplete factor as a preconditioner.

To design an efficient parallel ICT algorithm we again exploit technology that has been developed for parallel complete sparse Cholesky solvers. Recent work by us and other researchers shows that scalable, parallel implementations of sparse Cholesky factorization are possible via the development of new schemes that use sophisticated distributed dense matrix kernels [3, 20, 21, 22, 25, 26, 33, 44, 46, 45, 49, 50, 53, 55, 57]. The resulting codes are typically more complicated than a parallel implementation of the conjugate gradient algorithm.

We describe our strategy for parallel implementation in terms of a "compute-tree." Without loss of generality, assume the compute-tree is a complete binary tree with as many leaves as the total number of processors. Columns of the matrix A are assigned to the nodes in the compute-tree such that those associated with a given node will depend only on the columns associated with the descendants of this node. In terms of computing ICT, each leaf node now represents computations on a subset of columns of A that are independent of the subsets assigned to the other leaf nodes. More generally, if two nodes in the compute-tree are at the same level, then the two subsets of columns assigned to these two nodes are independent of each other. Such a design will reduce the communication requirements. By the same token, the columns in a supernode (or a block within a supernode, if the supernode is further divided) should be assigned to a single node in the compute-tree. At levels other than the leaves, each node of the compute-tree represents processors cooperating to compute the columns associated with this node. The
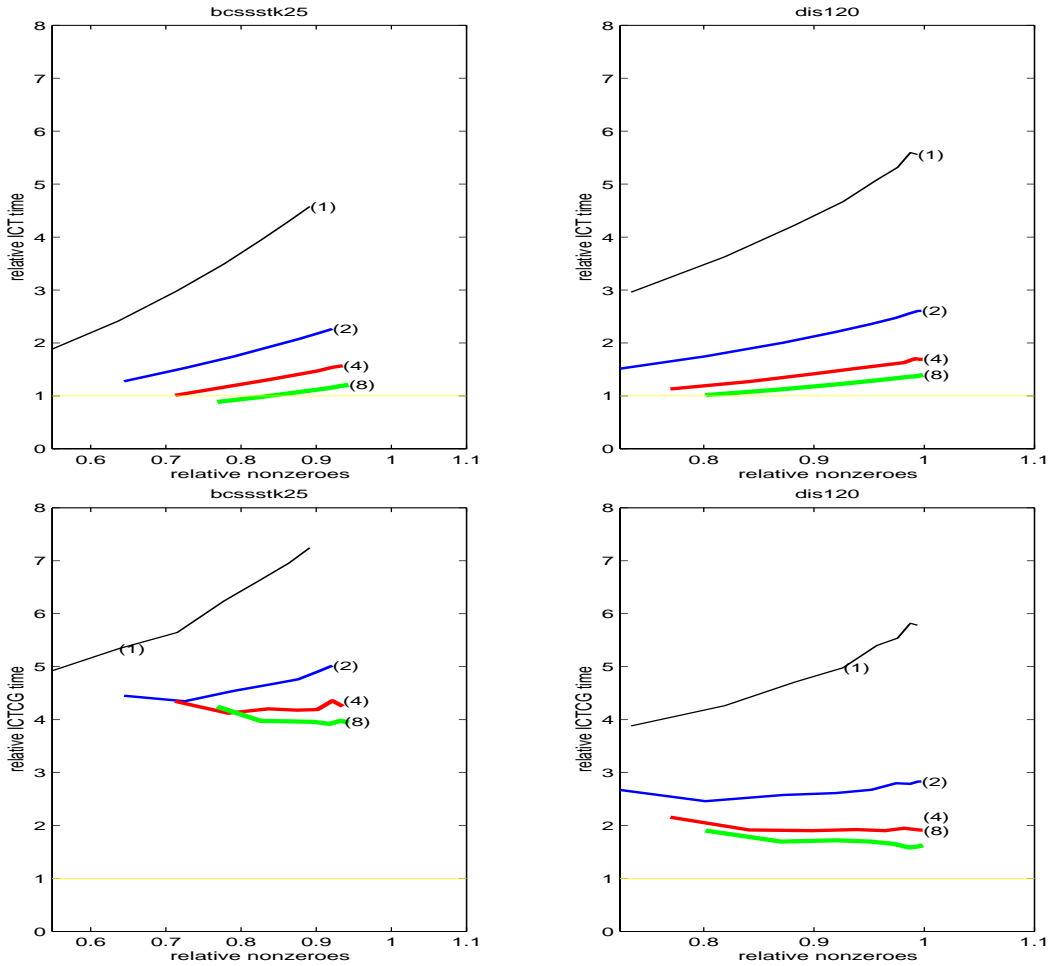
FIG. 2. *Performance of ICT and ICTCG for block-sizes 1,2,4 and 8.*

processors associated with the node are exactly those processors associated with the leaves of the subtree rooted at this node. Thus, all processors participate at the root node of the compute-tree, while at one level below, two disjoint processor groups of half the total size work on each node and so on. This approach is similar to the "subtree-to-subcube" idea in [12].

In broad terms this compute-tree can represent two different parallel sparse Cholesky schemes. As explained earlier, columns in a supernode form a dense lower trapezoidal submatrix. Consider a column k associated with the lower trapezoidal submatrix at a compute-tree node a. Such a column typically needs to be updated by suitable multiples of columns associated with several nodes in the subtree rooted at a. The manner in which this data-dependency is handled results in either a column-oriented approach or a multifrontal scheme.

In a column-oriented approach the computation of column k reflects the data-

dependency on several columns in the subtree rooted at a. First, columns in the trapezoidal matrix are block-wrap mapped to the processors assigned to node a. For a block-size of β, the first β columns are assigned to the first processor, the next β columns to the second processor and so on. Updates to column k from earlier columns in the subtree rooted at a (and hence on several processors) are collected in a "fan-in" manner. Each processor computes and accumulates updates to column k from all columns it owns; it then sends this (in the form of a vector) to the processor assigned column k. Hence updates to column k fan-in from processors in the subtree at a. This is considered the best form of implementing a parallel column-oriented sparse Cholesky factorization [3]. Figure 3 relates the compute-tree to the column-oriented factorization scheme for a matrix corresponding to the 7×7 model finite-difference grid. The same approach can be applied to a left-looking implementation of ICT.
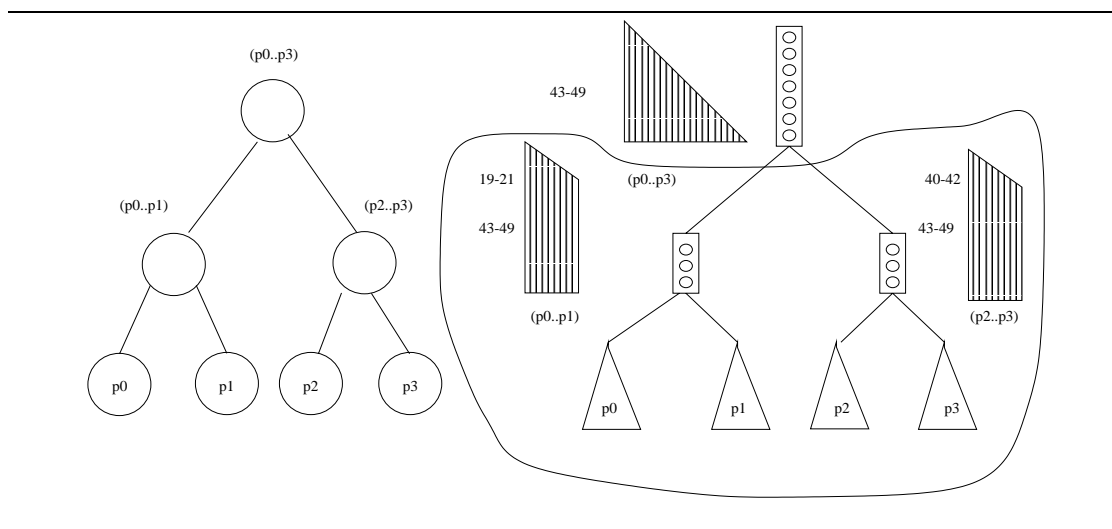

FIG. 3. *The compute-tree for 4 processors for a sparse matrix associated with a 7X7 grid is shown on the left; in a column oriented fan-in scheme this compute-tree represents operations on blocks of columns as shown on the right. Computing a column at the root, typically involves updates using columns associated several nodes in the subtree below the root.*

In a multifrontal scheme the data-dependency is modified at the expense of extra storage. Each compute-node is now associated with (i) a dense lower trapezoidal submatrix corresponding to columns in the supernode and (ii) a triangular matrix to hold updates from the columns in the supernode (lower trapezoidal submatrix) to columns in supernodes belonging to the ancestors. Now the two submatrices can be combined to give one larger lower triangular matrix at each compute node. Figure 4 relates the compute-tree to a multifrontal scheme for a matrix corresponding to the 7×7 grid. Now consider computations for columns

in the supernode associated with compute-node a. First the update portion of the matrices at the children nodes of a are assembled into the matrix associated with a. Next the columns in the supernode at a are processed by performing dense-distributed parallel Cholesky. Finally, updates from the columns formed are accumulated into the final triangular portion to be used at later ancestor nodes. Hence in a multifrontal scheme, updates from earlier columns are propagated through a stack of update matrices. This allows computation at a node to more closely match dense-distributed Cholesky. Now each dense triangular matrix associated with a compute-node could be distributed to the processors involved in several ways. The matrix could be wrap-mapped using small blocks of columns or distributed in a block-cyclic manner; the latter reduces the volume of communication and typically leads to better performance [20].
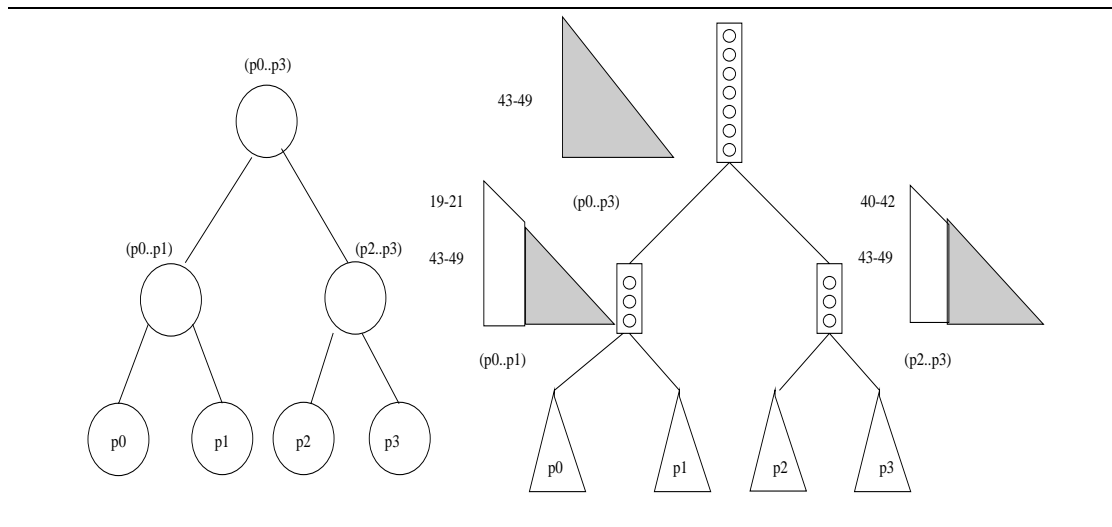

FIG. 4. *The compute-tree for 4 processors for a sparse matrix associated with a 7X7 grid is shown on the left; in a multifrontal scheme this compute-tree represents operations on triangular dense matrices as shown on the right. Computations at the root involve assembling the shaded submatrices at the child nodes and then performing a dense- distributed Cholesky decomposition.*

It is somewhat tricky to implement ICT using the multifrontal approach. The main issue is managing the stack; i.e., what updates should be accumulated for later use at ancestor supernodes. Dropping elements affects the matrix shape; the matrix will no longer be dense. Hence one of the advantages of the multifrontal scheme (dealing with dense matrices) will be lost. Furthermore the stack storage can be substantial, especially for sparse matrices associated with three-dimensional grids. At this stage we are experimenting with a serial implementation of a limited-stack multifrontal ICT to see if this approach could be made feasible.

A parallelization of a column-oriented ICT seems more natural. We plan

11

to implement a blocked fan-in ICT. If the supernode involves n columns on p processors, then we wrap-map the columns in blocks of sizes of 1, 2, 4, or 8. The processor owning a block would receive and apply updates from other processors before applying the block-threshold condition.

Let us focus now on the issue of applying an ICT as a preconditioner in parallel for conjugate gradient iterations. For direct methods, only two triangular systems have to be solved to obtain the solution for each right-hand side vector. It is well known, however, that sparse triangular systems are difficult to parallelize well because of data dependency and because of the relatively small number of operations. The effect is more pronounced in the case of parallel ICCG since repeated parallel solutions of sparse triangular systems will typically be required to apply the incomplete Cholesky factor as a preconditioner. This may be true even for a single right-hand side vector. Thus, triangular solution can become a serious bottleneck in parallel ICCG. The triangular solutions in the local-phase can be performed as efficiently as in the serial case. But triangular solutions in the distributed-phase may cause significant slowdown.

On message-passing multiprocessor systems, multiplying a vector by a matrix can perform more than ten times faster than solving a triangular system of the same order that uses the most sophisticated substitution scheme [23]. We thus propose to improve the performance of triangular solutions in the distributed-phase by new innovative methods that replace a triangular solution by a matrix-vector multiplication.
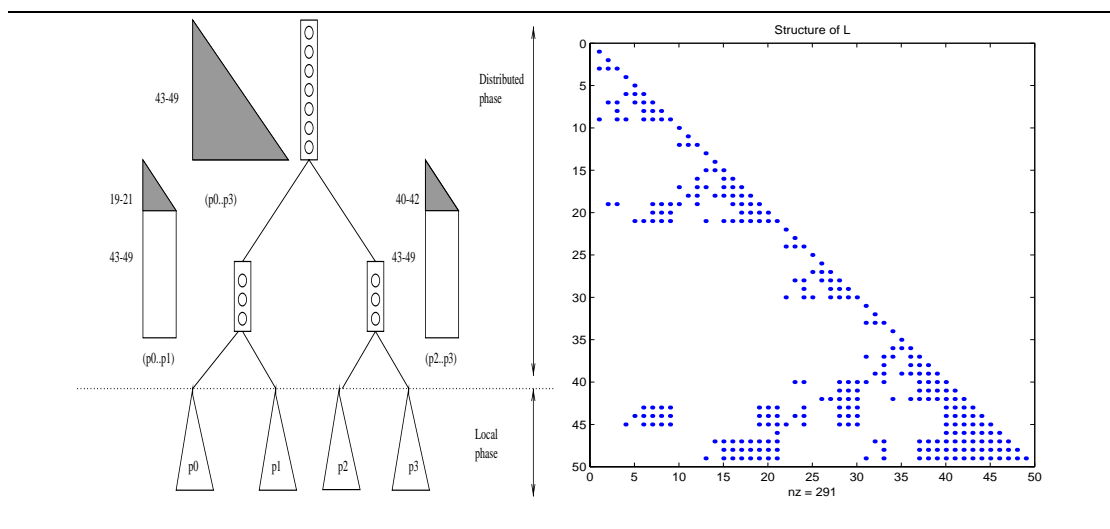

FIG. 5. *The structure of L for a 7X7 2-dimensional grid and its associated compute-tree for 4 processors; shaded triangular portions of supernodal submatrices in the distributed phase are affected by Selective Inversion for speeding up solves.*

Figure 5 shows the 4-processor implementation of the solution step for a sparse

matrix corresponding a 7×7 model finite-difference grid. The triangular solutions involve the shaded parts of supernodal matrices. When the supernodal submatrix at a node in the distributed phase is relatively dense, we can use our "selective inversion" (SI) scheme [46]. Selective inversion inverts the *dense diagonal* block in each dense submatrix (corresponding to a supernode), which is then used to replace substitution by distributed matrix-vector multiplication. As shown in [46], the scheme leads to ideal scalability and efficiency at a slight overhead of computing the inversion. The inversion overhead for complete sparse Cholesky factor is limited to under 6% of the cost of factorization for representative sparse matrices corresponding to 2- and 3-dimensional grids. We expect the overhead for inversion with ICT to behave similarly.

Recall than in parallel fan-in ICT, we will drop rows within column blocks based on a threshold condition. Consequently, the resulting supernode submatrix could be relatively sparse. Now applying SI to the top triangular part would make it dense. Hence, we plan to adapt SI by limiting inversion to smaller blocks within the supernode. Consider a node with n columns to be computed using a subset of processors of size p. A sequence of diagonal blocks of size βp are inverted (β is the block-size of 1,2,4 or 8.) This may cause a small amount of extra fill than in the incomplete factor. A triangular solution is performed in $n/(\beta p)$ steps; each step involves matrix-vector multiplication with the inverse to compute values of the solution. These values are in turn used to update the solution vector for the the next block. With a suitable data-assignment to processors, the matrix-vector multiplication is done with no communication and then a vector of the update is accumulated for the next step. This accumulation can be done using clever pipelining techniques such as those in [8, 28, 29]. We believe this will lead to greater data parallelism compared to distributed substitution while decreasing the latency related cost significantly. This scheme is illustrated in Figure 6.

Figure 7 shows the performance SI for a sequence of finite element matrices defined on regular meshes. The matrix dimensions have been chosen so that the amount of work in computing the triangular solution per processor is roughly constant. Here, we are showing the performance of solving a linear system using the complete sparse Cholesky factorization. The data shown are the ratios t_{SI}/t_{std}. Here t_{std} is the time to compute the factorization and to perform triangular solution(s) using the standard approach, while t_{SI} is the time to compute the factorization, the selective inverse, and to perform triangular solution(s) using the selective inverse. The top curve corresponds to the case when no right-hand side was provided. The remaining curves correspond to the cases when the number of right-hand sides were 1, 2, 4, and 8, respectively. The figure confirms that indeed there can be significant savings in time when selective inverse is employed in computing the solutions. The savings become more pronounced as the number of right-hand sides increase.

Based on the results from complete sparse Cholesky factorization, we conjecture that triangular solutions with ideal parallelism in the local-phase followed by
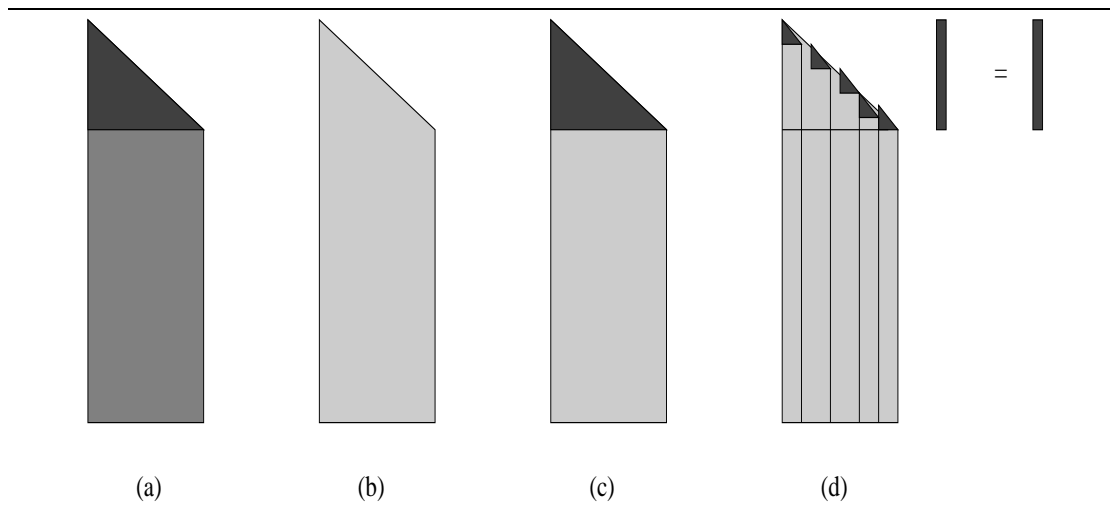
Fig. 6. *The modified SI scheme for a supernodal matrix; (a) shows the scheme with complete factorization, the entire matrix is dense and the inverted portion is shown in a darrker shade, (b) shows the sparser supernodal matrix in ICT, (c) shows that SI applied as is would cause the triangular part to become dense, and (d) shows our adaptation using the inversion of a sequence of smaller triangular blocks.*

a suitable low-overhead substitution replacement scheme for the distributed-phase will have the desired result, namely, a scalable, high-performance method to apply ICT factors over a wide range of fill-in to accelerate parallel CG. Significant implementation and experimentation is required before we can test our conjecture.

5. Conclusions. We have outlined in this paper our design of a scalable sparse solver, which can be considered as a hybrid of a direct method (using Cholesky factorization) and an iterative method (using preconditioned conjugate gradients). The goal is to develop a robust and high-performance solver that can be used in a wide variety of applications requiring the range from pure iterative to pure direct methods. The key is to leverage technology that has been developed for sparse direct methods.

Our results on the serial implementation of blocked ICTCG and the parallel implementation of selective inversion scheme for sparse triangular solution are encouraging. We are currently in the process of completing the implementation of our parallel ICTCG using the strategies discussed in this paper. We will report the results elsewhere when they become available.

REFERENCES

[1] P. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905.
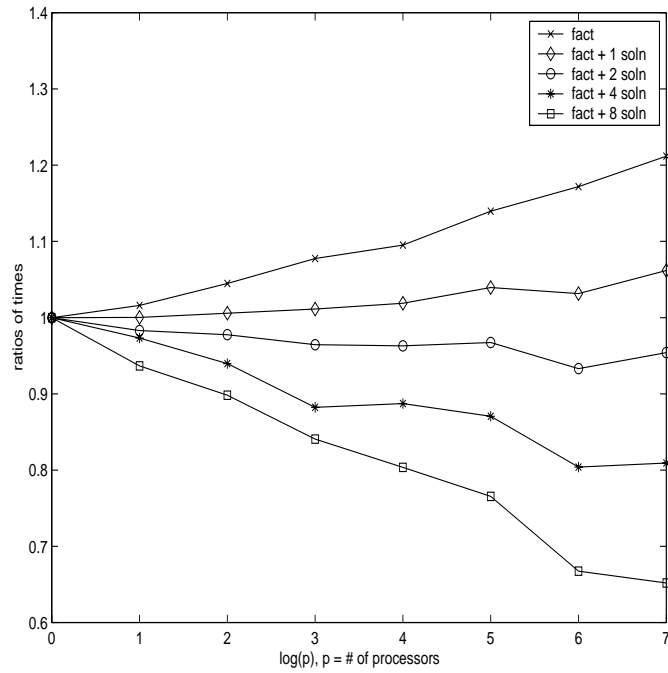
FIG. 7. *Performance of selective inversion for different number of right-hand sides. The matrices are finite element matrices defined on regular meshes. The dimensions of the matrices were chosen so that the amount of work per processor for each triangular solution is roughly constant. The ratios are t_{SI}/t_{std}. Here t_{std} is the time to compute the factorization and to perform triangular solution(s) using the standard approach, while t_{SI} is the time to compute the factorization, the selective inverse, and to perform triangular solution(s) using the selective inverse.*

[2] S. F. ASHBY AND M. K. SEAGER, *A proposed standard for iterative solvers*, Tech. Rep. UCRL-102860, Numerical Mathematics Group, Computing and Mathematics Division, Lawrence Livermore Lab., January 1990.

[3] C. ASHCRAFT, S. EISENSTAT, J. W.-H. LIU, B. PEYTON, AND A. SHERMAN, *A compute-ahead implementation of the fan-in sparse distributed factorization scheme*, Tech. Rep. ORNL/TM-11496, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.

[4] T. F. CHAN AND B. SMITH, *Domain decomposition and multigrid algorithms for elliptic problems on unstructured meshes*, Tech. Rep. CAM 93–42, University of California at Los Angeles, 1993.

[5] P. CONCUS, G. GOLUB, AND D. O'LEARY, *A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations*, in Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, 1976, pp. 309–332.

[6] I. DUFF, R. GRIMES, AND J. LEWIS, *Sparse matrix test problems*, ACM Trans. Math. Software, 15 (1989), pp. 1–14.

[7] I. S. DUFF, A. M. ERISMAN, AND J. K. REID, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, 1986.

[8] S. C. EISENSTAT, M. T. HEATH, C. S. HENKEL, AND C. H. ROMINE, *Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 589–600.

[9] M. P. I. FORUM, *MPI: A message -passing interface standard*, Tech. Rep. CS-94-230, Computer Science Dept., Univ. of Tennessee, Knoxville, April 1994.

[10] R. FREUND AND N. NACHTIGAL, *QMR: A quasi-minimum residual method for non-Hermitian linear systems*, Numerische Mathematik, 60 (1991), pp. 315–339.

[11] A. GEORGE AND J. W.-H. LIU, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[12] A. GEORGE, J. W.-H. LIU, AND E. G.-Y. NG, *Communication results for parallel sparse Cholesky factorization on a hypercube*, Parallel Computing, 10 (1989), pp. 287–298.

[13] J. A. GEORGE, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[14] J. A. GEORGE AND J. W.-H. LIU, *An automatic nested dissection algorithm for irregular finite element problems*, SIAM J. Numer. Anal., 15 (1978), pp. 1053–1069.

[15] J. GILBERT, E. NG, AND B. PEYTON, *An efficient algorithm to compute row and column counts for sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 1075–1091.

[16] J. GILBERT, E. NG, B. PEYTON, AND P. RAGHAVAN, *Performance of ICCG*, Tech. Rep. UTK-CS-97-381, Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, 1997.

[17] G. GOLUB AND C. V. LOAN, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 1996.

[18] W. GROPP AND D. KEYES, *Complexity of parallel implementation of domain decomposition techniques for elliptic partial differential equations*, SIAM J. Sci. Comput., 9 (1988), pp. 312–326.

[19] B. GUO AND J. LIU, *Direct visible surface interpolation*, tech. rep., York University, Department of Computer Science, North York, Ontario, Canada, M3J 1P3, 1997.

[20] A. GUPTA AND V. KUMAR, *A scalable parallel algorithm for sparse matrix factorization*, Tech. Rep. 94-19, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. A short version submitted for Supercomputing '94.

[21] A. GUPTA, E. ROTHBERG, E. NG, AND B. W. PEYTON, *Parallel sparse Cholesky factorization algorithms for shared-memory mutliprocessor systems*, in Advances in Computer Methods for Partial Differential Equations—VII, R. Vichnevetsky, D. Knight, and G. Richter, eds., International Association for Mathematics and Computers in Simulation (IMACS), 1992, pp. 622–628.

[22] T. R. HARROLD, *A portable scalable implementation of multifrontal cholesky on distributed memory machines*, tech. rep., Dept. of Computer Science, Univ. of Tennessee, July 1997.

[23] M. HEATH AND P. RAGHAVAN, *Parallel sparse triangular solution*, in IMA Volumes in Mathematics and its Applications, R. Gulliver, M. Heath, R. Schreiber, and P. Bjorstad, eds., vol. 105, Springer-Verlag, 1998, pp. 289–306.

[24] M. R. HESTENES AND E. STIEFEL, *Methods of conjugate gradients for solving linear systems*, J. Res. Nat. Bur. Stand., 49 (1952), pp. 409–436.

[25] G. KARYPIS AND V. KUMAR, *Parallel multilevel graph partitioning*, Tech. Rep. 95-036, Department of Computer Science, University of Minnesota, Minneapolis, MN, May 1995.

[26] S. KRATZER, *Massively parallel sparse matrix computations*, Tech. Rep. SRC-TR-90-008, Supercomputing Research Center, Bowie, MD 20715, 1990.

[27] C. L. LAWSON, R. HANSON, D. KINCAID, AND F. KROGH, *Basic linear algebra subprograms for fortran usage*, ACM Trans. Math. Software, 5 (1979), pp. 308–323.

[28] G. LI AND T. COLEMAN, *A parallel triangular solver for a hypercube multiprocessor*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 458–502.

[29] ——, *A new method for solving triangular systems on distributed memory message passing multiprocessors*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 382–396.

[30] J. LIU, E. NG, AND B. PEYTON, *On finding supernodes for sparse matrix computations*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 242–252.

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

[32] J. W. H. LIU, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[33] R. LUCAS, *Solving planar systems of equations on distributed-memory multiprocessors*, PhD thesis, Department of Electrical Engineering, Stanford University, 1987.

[34] T. A. MANTEUFFEL, *An incomplete factorization technique for positive definite linear systems*, Math. Comput., 34 (1980), pp. 473–497.

[35] J. A. MEIJERINK AND H. A. VAN DER VORST, *An iterative solution methods for which the coefficient matrix is a symmetric M-matrix*, Math. Comput., 31 (1977), pp. 148–162.

[36] N. MUNKSGAARD, *Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients*, ACM Trans. Math. Software, 6 (1980), pp. 206–219.

[37] ——, *Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients*, ACM Trans. Math. Software, 6 (1980), pp. 206–219.

[38] E. NG, B. PEYTON, AND P. RAGHAVAN, *A blocked incomplete cholesky preconditioner for hierarchical-memory computers*, in Proceedings of the Fourth IMACS International Symposium on Iterative Methods, 1999. To appear.

[39] E. NG AND B. W. PEYTON, *Block sparse Cholesky algorithms on advanced uniprocessor computers*, SIAM J. Sci. Comput., 14 (1993), pp. 1034–1056.

[40] E. G. NG AND P. RAGHAVAN, *The performance of greedy ordering heuristics for sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., (1998). To appear.

[41] T. C. OPPE, W. D. JOUBERT, AND D. KINCAID, *ITPACKV 2D User's Guide*, Tech. Rep. CNA-232, Center for Numerical Analysis, The University of Texas at Austin, May 1989.

[42] S. PARTER, *The use of linear graphs in Gaussian elimination*, SIAM Review, 3 (1961), pp. 364–369.

[43] P. RAGHAVAN, *Distributed sparse matrix factorization: QR and Cholesky decompositions*, PhD thesis, Department of Computer Science, Pennsylvania State University, University Park, PA, 1991.

[44] ——, *Distributed sparse Gaussian elimination and orthogonal factorization*, SIAM J. Sci. Comput., 16 (1995), pp. 1462–1477.

[45] ——, *Parallel ordering using edge contraction*, Parallel Computing, 23 (1997), pp. 1045–1067.

[46] ——, *Efficient parallel triangular solution with selective inversion*, Parallel Processing Letters, 8 (1998), pp. 29–40.

[47] D. ROSE, *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*, in Graph Theory and Computing, R. C. Read, ed., Academic

Press, 1972, pp. 183–217.

[48] D. Rose, R. Tarjan, and G. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.

[49] E. Rothberg, *A parallel implementation of the multiple minimum degree heuristic*. Presentation, Fifth Siam Conference on Applied Linear Algebra 1994, Snowbird, Utah.

[50] E. Rothberg, *Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multiprocessors*, tech. rep., Intel Supercomputer Systems Division, 14924 N. W. Greenbrier Parkway, Beaverton, OR 97006, September 1993.

[51] Y. Saad, *ILUT: A dual threshold incomplete ILU factorization*, Num. Lin. Alg. Appl., 1 (1994), pp. 387–402.

[52] Y. Saad and M. Schultz, *GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 856–869.

[53] R. Schreiber, *Scalabilty of sparse direct solvers*, in Proc. of the IMA Workshop on Graph Theory and Sparse Matrices, Springer-Verlag, 1992.

[54] B. Smith, L. McInnes, and W. Gropp, *PETSc 2.0 user's manual*, Tech. Rep. ANL-95/11 - Revision 2.0.22, Mathematics and Computer Science Division Argonne National Laboratory, Argonne IL 60439, 1997.

[55] C. Sun, *Efficient parallel solutions of large sparse SPD systems on distributed memory multiprocessors*, Tech. Rep. CTC-92-TR-102, Advanced Computing Research Institute, Center For Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY 14853, August 1992.

[56] Z. Zlatev, *Use of iterative refinement in the solution of sparse linear systems*, SIAM J. Numer. Anal., 19 (1982), pp. 381–399.

[57] E. Zmijewski, *Sparse Cholesky Factorization on a Multiprocessor*, PhD thesis, Department of Computer Science, Cornell University, August 1987.