

RC 22038 (98932) April 20, 2001  
Computer Science/Mathematics

# IBM Research Report

## WSMP: A High-Performance Shared- and Distributed-Memory Parallel Sparse Linear Equation Solver

**Anshul Gupta and Mahesh Joshi**


IBM Research Division  
T. J. Watson Research Center  
P. O. Box 218  
Yorktown Heights, NY 10598  
{*anshul,joshim*}@*watson.ibm.com*

**Vipin Kumar**

Department of Computer Science  
University of Minnesota  
Minneapolis, MN 55455  
*kumar@cs.umn.edu*

### LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

 Research Division  
Almaden · Austin · China · Delhi · Haifa · Tokyo · Watson · Zurich

# WSMP: A High-Performance Shared- and Distributed-Memory Parallel Sparse Linear Equation Solver

Anshul Gupta and Mahesh Joshi  
IBM T.J. Watson Research Center  
P.O. Box 218, Yorktown Heights, NY 10598, U.S.A.

Vipin Kumar  
Department of Computer Science  
University of Minnesota, Minneapolis, MN 55455, U.S.A.

**Abstract.** The Watson Sparse Matrix Package, *WSMP*, is a high-performance, robust, and easy to use software package for solving large sparse systems of linear equations. It can be used as a serial package, or in a shared-memory multiprocessor environment, or as a scalable parallel solver in a message-passing environment, where each node can either be a uniprocessor or a shared-memory multiprocessor. A unique aspect of *WSMP* is that it exploits both SMP and MPP parallelism using Pthreads and MPI, respectively, while mostly shielding the user from the details of the architecture. Sparse symmetric factorization in *WSMP* has been clocked at up to 1.2 Gigaflops on RS6000 workstations with two 200 MHz Power3 CPUs and in excess of 90 Gigaflops on 128-node (256-processor) SP with two-way SMP 200 MHz Power3 nodes. This paper gives an overview of the algorithms, implementation aspects, performance results, and the user interface of *WSMP* for solving symmetric sparse systems of linear equations.

**Keywords:** Parallel software, Scientific computing, Sparse linear systems, Sparse matrix factorization, High-performance computing.

## 1 Introduction

Solving large sparse systems of linear equations is at the core of many problems in science, engineering, and optimization. A direct method for solving a sparse linear system of the form  $Ax = b$  involves explicit factorization of the sparse coefficient matrix  $A$  into the product of lower and upper triangular matrices  $L$  and  $U$ . This is a highly time and memory consuming step; nevertheless, direct methods are important because of their generality and robustness and are used extensively in many application areas. Direct methods also provide an effective means for solving multiple systems with the same coefficient matrix and different right-hand side vectors because the factorizations needs to be performed only once. A wide class of sparse linear systems arising in practical

applications have symmetric coefficient matrices, whose factorization is numerically stable with any symmetric permutation. This paper gives an overview of the algorithms, implementation aspects, performance results, and the user interface of *WSMP* [14, 15] for solving symmetric sparse systems of linear equations. *WSMP* can be used as a serial package, or in a shared-memory multiprocessor environment with threads, or as a scalable parallel solver in a message-passing environment, where each node can either be a uniprocessor or a shared-memory multiprocessor. It uses a modified multifrontal algorithm and scalable parallel algorithms for sparse symmetric factorization and triangular solves. The ordering/permutation of matrices is also parallelized. Sparse symmetric factorization in *WSMP* has been clocked at up to 1.2 Gigaflops on RS6000 workstations with two 200 MHz 630 (Power3) CPUs and in excess of 90 Gigaflops on a 128-node (256-processor) SP with two-way SMP 200 MHz 630 nodes. This paper gives an overview of the algorithms, implementation aspects, performance results, and the user interface of WSMP.

### 1.1 Related Work

Parallelization of sparse Cholesky factorization has been an area of active research over the past decade [1, 3, 4, 9, 10, 11, 12, 17, 18, 19, 21, 28, 29, 30]. A few parallel software packages for solving large sparse symmetric systems have emerged recently as a result of this research: MUMPS [2], SPOOLES [5], MP\_SOLVE [8], PSPASES [23] and WSMP [16]. Of these, WSMP and PSPASES employ the theoretically most scalable algorithms for Cholesky factorization [17]. These are also the only two packages that perform parallel ordering. The main differences between the two are that WSMP supports both  $LL^T$  and  $LDL^T$  factorizations and is available in object-code for IBM RS6000 and SP hardware platforms only. In addition to Cholesky factorization, MUMPS, MP\_SOLVE, and SPOOLES support  $LU$  factorization, which the current versions of WSMP and PSPASES don't. From results published until the time of writing this paper, WSMP holds the record for achieving the highest performance and speedup in sparse Cholesky factorization.

As discussed in [17], both the subtree-to-subcube mapping of the elimination tree among processors and a two-dimensional distribution of frontal and update matrices among subgroups of processors are crucial to obtaining the highest scalability. MP\_SOLVE employs none of these techniques. SPOOLES employs subtree-to-subcube mapping but uses a one-dimensional distribution. MUMPS uses a two-dimensional distribution of data at only the topmost supernode of the elimination tree.

### 1.2 Overview

The organization of this paper is as follows. Section 2 describes the distributed-memory parallel algorithms used by *WSMP* for the four major phases of computation of the direct solution of a sparse symmetric system. Section 3 discusses the shared-memory variations of these parallel algorithms. Section 4 gives an

overview of the organization and the user-interface of the software. Performance results are presented in Section 5. Section 6 contains concluding remarks.

In this paper, the term *node* refers to a uniprocessor or a multiprocessor computing unit with shared memory. A node may consist of one or more *processors* or CPUs. Nodes communicate with other nodes only via message-passing over the SP high-speed switch. In *WSMP*, parallelism within a multiprocessor node is exploited by threads. In order to keep the definition of a node unique, we would refer to a meeting point of edges of a graph or a tree as a *vertex*. Accordingly, the term *supervertex* is used to describe a set consecutive columns of the sparse matrix with identical sparsity pattern, which is more commonly referred to as a *supernode* in literature.

## 2 Distributed-Memory Algorithms

The process of obtaining a direct solution of a sparse symmetric system of linear equations of the form  $Ax = b$  consists of the following four phases: *Ordering*, which determines a symmetric permutation of the coefficient matrix  $A$  such that the factorization incurs low fill-in; *Symbolic Factorization*, which determines the structure of the triangular matrices that would result from factoring the coefficient matrix; *Numerical Factorization*, which is the actual factorization step that performs arithmetic operations on the coefficient matrix  $A$  to produce a lower triangular matrix  $L$  such that  $A = LL^T$  or  $A = LDL^T$ ; and *Solution of Triangular Systems*, which produces the solution vector  $x$  by performing forward and backward eliminations on the triangular matrices resulting from numerical factorization. In this section, we give an overview of the algorithms used in these four phases.

Recall that *WSMP* parallelizes the tasks among processes that communicate using MPI and each process itself uses multiple threads to utilize multiple CPU's on an SMP node. The algorithms described in this section are used in the distributed memory mode and an MPI process is assumed to be the smallest unit of computation. The SMP algorithms are described in Section 3.

### 2.1 Ordering

Finding an optimal ordering is an NP-hard problem and heuristics must be used to obtain an acceptable non-optimal solution. Two main classes of successful heuristics have evolved over the years: (1) Local greedy heuristics, and (2) Global graph-partitioning based heuristics. In [17], we presented an optimally scalable parallel algorithm for factoring a large class of sparse symmetric matrices. This algorithm works efficiently only with graph-partitioning based ordering. Although, traditionally, local ordering heuristics have been preferred, recent research [24, 6, 20, 13] has shown the graph-partitioning based ordering heuristics can match and often exceed the fill-reduction of local heuristics.

The serial ordering heuristics that *WSMP* uses have been described in detail in [17]. Basically, the sparse matrix is regarded as the adjacency matrix of an

undirected graph and a divide-and-conquer strategy (also known as nested dissection) is applied recursively to label the vertices of the graph by partitioning it into smaller subgraphs. At each stage, the algorithm attempts to find a small vertex-separator that partitions the graph into two disconnected subgraphs satisfying some balance criterion. The vertices of the separator are numbered after the vertices in the subgraphs are numbered by following the same strategy recursively.

There are two main approaches to parallelizing this algorithm. In the first approach [25, 26], the process of finding the separators is performed in parallel. The advantages of this approach are that a reasonably good speedup on ordering can be obtained and the graphs (both original and the subsequent subgraphs) are stored on multiple nodes, thereby avoiding excessive memory use on any single node in a distributed-memory environment. A disadvantage of this approach, as the results in [23] show, is that the refinement of partitions becomes less effective as the number of nodes in the parallel computer increases. As a result, the quality of ordering for a given sparse matrix gradually degrades as the number of nodes increases.

The second approach is more conservative and exploits only the natural parallelism of nested-dissection ordering. A single node finds the first separator, then two different nodes independently work on the two subgraphs, and so on. An advantage of this approach is that the quality of ordering does not deteriorate with as the number of nodes in the parallel computer increases. In fact, in *WSMP*, the quality of ordering actually improves as the number of nodes increases. The reason is that we employ the nodes that would otherwise be idle to compute redundant instances of separators of the same subgraphs at practically no extra cost. Among the multiple separators, we then chose the best one. For example, all the nodes of an SP would compute a separator of the original graph and the best one would be chosen. In the next step, half of the nodes compute a separator of each of the two subgraphs resulting from the first partition. The disadvantage of this approach is that, at least theoretically, it is not scalable with respect to memory use and speedup.

After performing some analysis and studying practical problems, we decided to implement the second approach in parallel *WSMP*. In a majority of applications, ordering is performed only once for several factorizations of matrices with identical structure, but different numerical values. Therefore, the speed of ordering is often unimportant relative to the quality of ordering, which determines the speed of factorization. The inability of this approach to scale in terms of speedup is not a serious handicap given the fact that it can yield good quality orderings even for a large number of nodes. The memory requirement of ordering is also not a major practical concern because usually the amount of memory required for ordering does not exceed the amount of memory required for subsequent factorization. For instance, we were able to order a linear system of a million equations derived from a  $100 \times 100 \times 100$  finite-difference discretization on a single RS6000/397 node with 1 Gigabytes of RAM, but it took a 32-node SP with 1 Gigabytes of RAM on each node to actually solve the system. In addition,

in many finite-element applications, each vertex of the discretized domain has multiple degrees of freedom and the graph corresponding to the original system can be compressed into a much smaller graph with identical properties. The ordering algorithm works on the smaller compressed graph and uses much less time and memory than ordering the original graph.

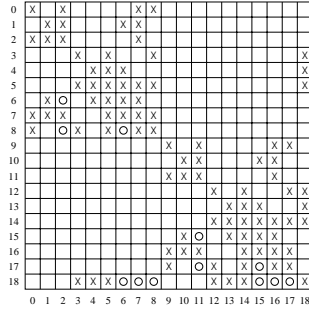
## 2.2 Symbolic Factorization

The phase of processing between ordering and numerical factorization in *WSMP* does much more than the traditional symbolic factorization task of computing the nonzero pattern of the triangular factor matrix and allocating data-structures for it. Since this phase takes a very small amount of time and, like ordering, is usually performed only once for several numerical factorization steps, it is performed serially in *WSMP*. In this phase, we first compute the elimination tree, then perform a quick symbolic step to predict the amount of potential numerical computation associated with each part of the elimination tree, and then adjust the elimination tree (as described in [17]) in order to balance the distribution of numerical factorization work among nodes. The elimination tree is also manipulated to reduce amount of stack memory required for factorization [27]. If the parallel machine has heterogeneous nodes in terms of processing power or the amount of memory, the symbolic phase takes that into account while assigning tasks to nodes. Finally, the vertices of the modified elimination tree are renumbered in a post-ordered sequence, the actual symbolic factorization is performed, respective information on the structure of the portion of the factor matrix to reside on each node is conveyed to all nodes, which, upon receipt of this information, allocate the data-structures for storing their portion of the factor.

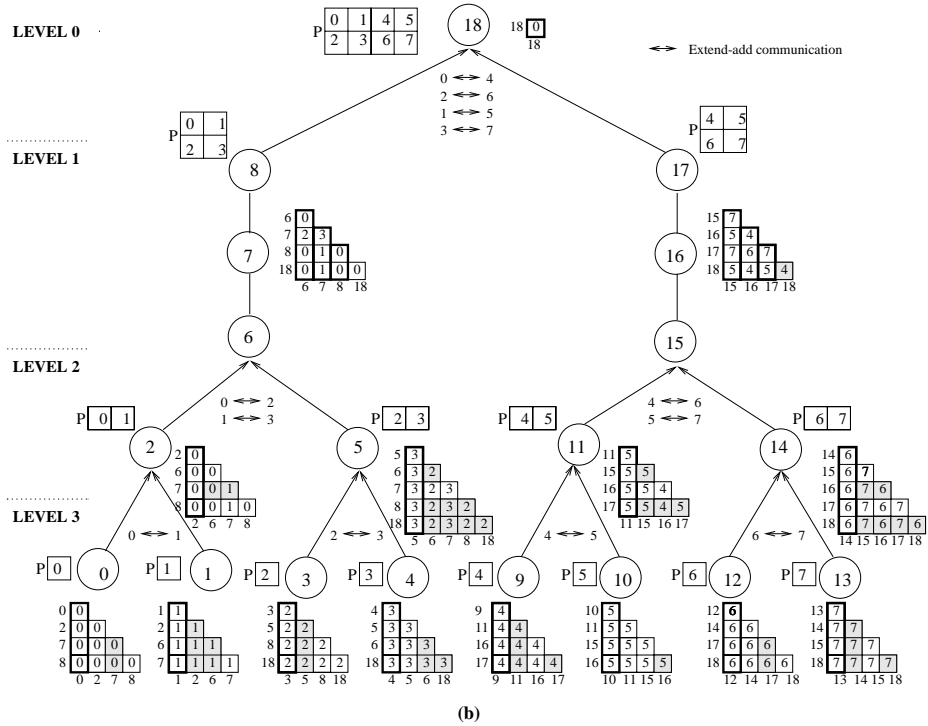
## 2.3 Numerical Factorization

*WSMP* can perform either Cholesky ( $LL^T$ ) or  $LDL^T$  factorization on symmetric sparse matrices. We use a highly scalable parallel algorithm for this step, the detailed description and analysis of which can be found in [17].

The parallel symmetric factorization in *WSMP* is based on the multifrontal algorithm [27]. Given a sparse matrix and the associated elimination tree, the multifrontal algorithm can be recursively formulated as follows. Consider an  $N \times N$  matrix  $A$ . The algorithm performs a postorder traversal of the elimination tree associated with  $A$ . There is a frontal matrix  $F^k$  and an update matrix  $U^k$  associated with any vertex  $k$ . The row and column indices of  $F^k$  correspond to the indices of row and column  $k$  of  $L$ , the lower triangular Cholesky factor, in increasing order. In the beginning,  $F^k$  is initialized to an  $(s+1) \times (s+1)$  matrix, where  $s+1$  is the number of non-zeros in the lower triangular part of column  $k$  of  $A$ . The first row and column of this initial  $F^k$  is simply the upper triangular part of row  $k$  and the lower triangular part of column  $k$  of  $A$ . The remainder of  $F^k$  is initialized to all zeros. After the algorithm has traversed all the subtrees rooted at a vertex  $k$ , it ends up with a  $(t+1) \times (t+1)$  frontal matrix  $F^k$ , where  $t$  is the number of non-zeros in the strictly lower triangular part of column  $k$  in



(a)



(b)

**Fig. 1. (a).** An example symmetric sparse matrix. The non-zeros of  $A$  are shown with symbol “ $\times$ ” in the upper triangular part and non-zeros of  $L$  are shown in the lower triangular part with fill-ins denoted by the symbol “ $o$ ”. **(b).** The process of parallel multifrontal factorization using 8 nodes. At each supervertex, the factored frontal matrix, consisting of columns of  $L$  (thick columns) and update matrix (remaining columns), is shown.

$L$ . The row and column indices of the final assembled  $F^k$  correspond to  $t + 1$  (possibly) noncontiguous indices of row and column  $k$  of  $L$  in increasing order. If  $k$  is a leaf in the elimination tree of  $A$ , then the final  $F^k$  is the same as the initial  $F^k$ . Otherwise, the final  $F^k$  for eliminating vertex  $k$  is obtained by merging the initial  $F^k$  with the update matrices obtained from all the subtrees rooted at  $k$  via an extend-add operation. The extend-add is an associative and commutative operator on two update matrices such the index set of the result is the union of the index sets of the original update matrices. After  $F^k$  has been assembled, a single step of the standard dense Cholesky factorization is performed with vertex  $k$  as the pivot. At the end of the elimination step, the column with index  $k$  is removed from  $F^k$  and forms the column  $k$  of  $L$ . The remaining  $t \times t$  matrix is called the update matrix  $U^k$  and is passed on to the parent of  $k$  in the elimination tree.

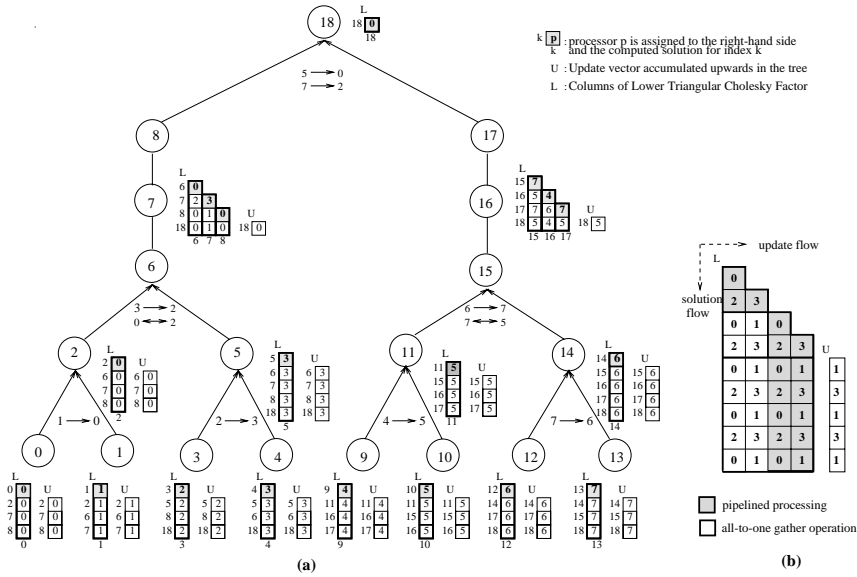
We assume that the supernodal tree is binary in the top  $\log p$  levels. The portions of this binary supernodal tree are assigned to the nodes using a subtree-to-subcube strategy illustrated in Figure 1(b), where eight nodes are used to factor the example matrix of Figure 1(a). The subgroup of nodes working on various subtrees are shown in the form of a logical mesh labeled with P. The frontal matrix of each supervertex is distributed among this logical mesh using a bit-mask based block-cyclic scheme [17]. Figure 1(b) shows such a distribution for unit block size. This distribution ensures that the extend-add operations required by the multifrontal algorithm can be performed in parallel with each node exchanging roughly half of its data *only* with its partner from the other subcube. Figure 1(b) shows the parallel extend-add process by showing the pairs of nodes that communicate with each other. Each node sends out the shaded portions of the update matrix to its partner. The parallel factor operation at each supervertex is a pipelined implementation of the dense block Cholesky factorization algorithm.

## 2.4 Triangular Solves

The solution of triangular systems involves a forward elimination  $y = L^{-1}b$  followed by a backward substitution  $x = (L^T)^{-1}y$  to determine the solution  $x = A^{-1}b$ . Our parallel algorithms for this phase are guided by the supernodal elimination tree. They use the same subtree-to-subcube mapping and the same two-dimensional distribution of the factor matrix  $L$  as used in the numerical factorization.

Figure 2(a) illustrates the parallel formulation of the forward elimination process. The right hand side vector  $b$ , is distributed to the nodes that own the corresponding diagonal blocks of the  $L$  matrix as shown in the shaded blocks in Figure 2(a). The computation proceeds in a bottom-up fashion. Initially, for each leaf vertex  $k$ , the solution  $y_k$  is computed and is used to form the update vector  $\{l_{ik}y_k\}$  (denoted by “U” in Figure 2(a)). The elements of this update vector need to be subtracted from the corresponding elements of  $b$ , in particular  $l_{ik}y_k$  will need to be subtracted from  $b_i$ . However, our algorithm uses the structure of the supernodal tree to accumulate these updates upwards in the tree and subtract





**Fig. 2.** Parallel triangular solve. **(a).** Entire process of parallel forward elimination for the example matrix. **(b).** Processing within a hypothetical supernodal matrix for forward elimination.

them only when the appropriate vertex is being processed. For example consider the computation involved while processing the supervertex  $\{6,7,8\}$ . First the algorithm merges the update vectors from the children supervertex to obtain the combined update vector for indices  $\{6,7,8,18\}$ . Note that the updates to the same  $b$  entries are added up. Then it performs forward elimination to compute  $y_6, y_7$  and  $y_8$ . This computation is performed using a two dimensional pipelined dense forward elimination algorithm. At the end of the computation, the update vector on node 0 contains the updates for for  $b'_{18}$  due to  $y_6, y_7$  and  $y_8$  as well as the updates received from supervertex  $\{5\}$ . In general, at the end of the computation at each supervertex, the accumulated update vector resides on the column of nodes that store the last column of the  $L$  matrix of that supervertex. This update vector needs to be sent to the nodes that store the first column of the  $L$  matrix of the parent supervertex. Because of the bit-mask based block-cyclic distribution, this can be done by using at most two communication steps [22].

The details of the two-dimensional pipelined dense forward elimination algorithm are illustrated in Figure 2(b) for a hypothetical supervertex. The solutions are computed by the nodes owning diagonal elements of  $L$  matrix and flow down along a column. The accumulated updates flow along the row starting from the first column and ending at the last column of the supervertex. The processing is pipelined in the shaded regions and in other regions the updates are accumulated using a reduction operation along the direction of the flow of update vector.

The algorithm for parallel backward substitution is similar except for two differences. First, the computation proceeds from the top supernode of the tree down to the leaf. Second, the computed solution that gets communicated across the levels of the supernodal tree instead of accumulated updates and this is achieved with at most one communication per processor. The reader is referred to [22] for further details.

### 3 Shared-Memory Algorithms

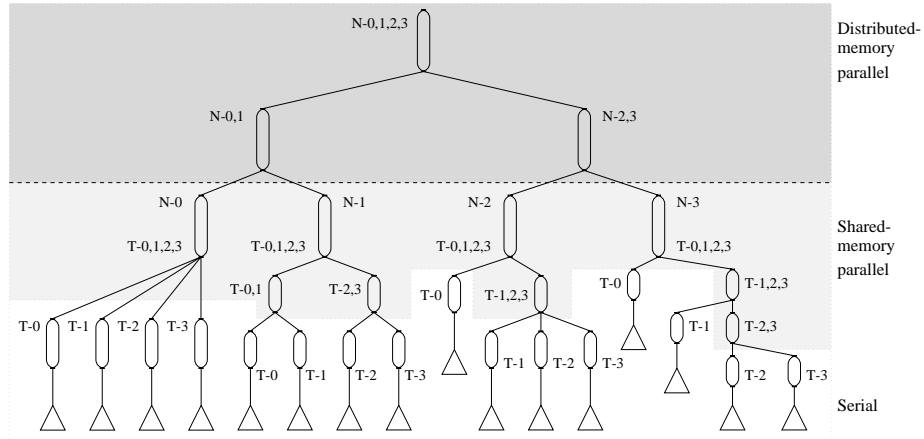
The key components of the *WSMP* library are multithreaded and automatically exploit SMP parallelism while running on a workstation or SP nodes with multiple CPU's. This section describes the SMP algorithms used in *WSMP*.

#### 3.1 The Symbolic Steps

Among the two symbolic steps, namely, ordering and symbolic factorization, only the former is parallelized. The SMP parallelization scheme for ordering closely follows that for message-passing parallelization. After a graph or a subgraph is assigned to a node for independent processing, a single thread finds the first separator, then two different threads independently work on the two subgraphs, and so on until as many threads as the number of CPU's are working on independent subgraphs.

#### 3.2 The Numerical Steps

The multithreaded algorithms for numerical factorization and triangular solves are similar to their message-passing counterparts described in Section 2 in overall strategy. However, there are significant differences in implementation. Just like the message-passing algorithms, tasks at each subroot of the elimination tree are assigned to independent groups of processors until each processor ends up with its own subtree. The most important difference is that a mapping of rows and columns to processors based on the binary representation of the indices is not used in the SMP implementation because all processors can access all rows and columns with the same overhead. This lifts the restriction that the subtree assigned to a group of  $P$  processors be binary in its top  $\log_2 P$  levels. Therefore, in the portion of the elimination tree that is executed upon in the SMP mode, the number of threads assigned to work on a subtree at a branching point is roughly proportional to the amount of work associated with that subtree. This ensures a high degree of load-balance. Moreover, since the relative ratio of work in a subtree with respect to its siblings in the factorization and solve phases is different, the mapping of subtrees to subgroups of processors in these two phases can be different—a flexibility that is not available in the message-passing portion of the code where the same subtree-to-subcube mapping is used in both factorization and triangular solves.



**Fig. 3.** A mapping of a hypothetical elimination tree on a 4-node SP with 4-way SMP nodes.

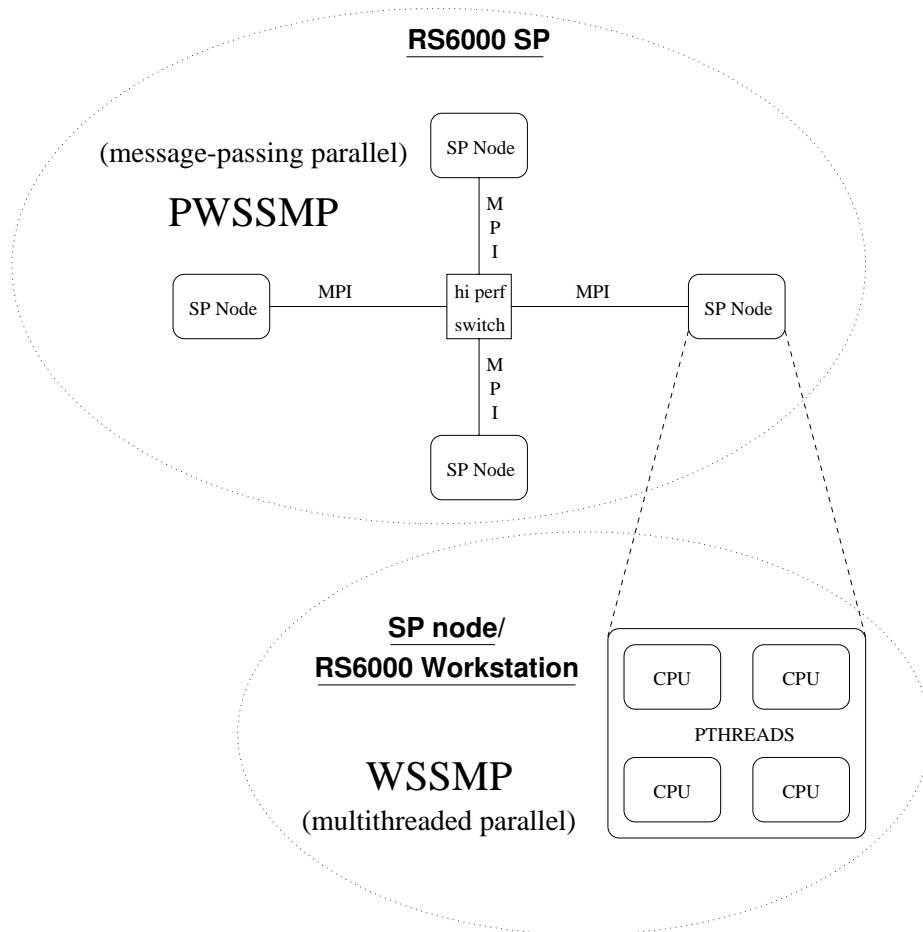
Figure 3 shows one hypothetical mapping of an elimination tree among the 16 CPU's of a 4-node SP with 4-way SMP nodes. The symbolic computation preceding the numerical phase ensures that the tree is binary at the top  $\log_2 P$  levels, where  $P$  is the number of distributed-memory nodes. At each of these levels multithreaded BLAS are used to utilize all the CPU's on each node. Below the top  $\log_2 P$  levels, subtrees are assigned to groups of threads until they are mapped on to single threads. Multithreaded BLAS is used in any portion of the tree in this region that is assigned to multiple threads.

## 4 The Software

The *WSMP* software is organized into two libraries (see Figure 4). The user's program needs to be linked with the *WSMP* library on a serial or an SMP workstation. This is a multi-threaded library and uses *Pthreads* to exploit parallelism on a multiprocessor workstation. On an SP, the user's program needs to be linked with both *WSMP* and *PWSMP* libraries. The latter contains all the message-passing code and exploits parallelism across the nodes via the MPI library. If the SP has multiprocessor nodes, the *WSMP* library exploits SMP parallelism on each node.

In this section, we give a brief description of the main routines of the package. The details of the user interface can be found in [16]. *WSMP* accepts as input a triangular portion of the sparse coefficient matrix. *WSMP* supports two popular formats for the coefficient matrix: Compressed Sparse Row/Column (CSR/CSC) and Modified Sparse Rows/Column (MSR/MSR), which are described in detail in [7, 16]. Figure 5(a) illustrates the serial CSC input format.

All major functions of the package for solving symmetric sparse systems can be performed by calling a single subroutine *WSSMP*. Its calling sequence is as



**Fig. 4.** The two-tier organization of the *WSMP* software for solving symmetric sparse systems. The message-passing layer works on top of the multithreaded layer.

follows:

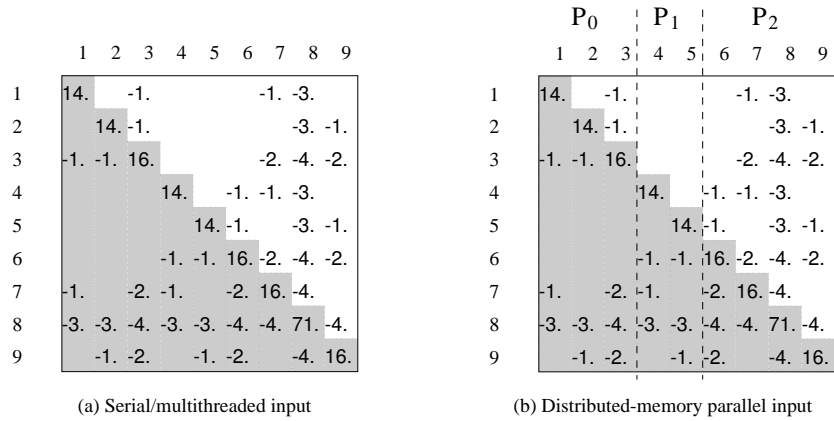
```

SUBROUTINE WSSMP (N, IA, JA, AVALS, DIAG, PERM, INVP, B,
+               LDB, NRHS, AUX, NAUX, MRP, IPARM, DPARM)

INTEGER*4  N, IA(N+1), JA(*), PERM(N), INVP(N), LDB, NRHS,
+         NAUX, MRP(N), IPARM(64)
REAL*8    AVALS(*), DIAG(N), B(LDB,NRHS), AUX(NAUX),
+         DPARM(64)

```

$N$  is the number of equations in the system. In CSR/CSC format,  $IA$ ,  $JA$ , and  $AVALS$  contain the coefficient matrix; in MSR/MSF format, the coefficient matrix is contained in  $IA$ ,  $JA$ ,  $AVALS$ , and  $DIAG$ .  $IA$  contains pointers



Serial CSC-LT				Parallel CSC-LT				
K	IA(K)	JA(K)	AVALS(K)	Proc#	K	IA(K)	JA(K)	AVALS(K)
1	1	1	14.0	P <sub>0</sub>	1	1	1	14.0
2	5	3	-1.0		2	5	3	-1.0
3	9	7	-1.0		3	9	7	-1.0
4	13	8	-3.0		4	13	8	-3.0
5	17	2	14.0		5		2	14.0
6	21	3	-1.0		6		3	-1.0
7	25	8	-3.0		7		8	-3.0
8	27	9	-1.0		8		9	-1.0
9	29	3	16.0		9		3	16.0
10	30	7	-2.0		10		7	-2.0
11		8	-4.0		11		8	-4.0
12		9	-2.0		12		9	-2.0
13		4	14.0	P <sub>1</sub>	1	1	4	14.0
14		6	-1.0		2	5	6	-1.0
15		7	-1.0		3	9	7	-1.0
16		8	-3.0		4		8	-3.0
17		5	14.0		5		5	14.0
18		6	-1.0		6		6	-1.0
19		8	-3.0		7		8	-3.0
20		9	-1.0		8		9	-1.0
21		6	16.0	P <sub>2</sub>	1	1	6	16.0
22		7	-2.0		2	5	7	-2.0
23		8	-4.0		3	7	8	-4.0
24		9	-2.0		4	9	9	-2.0
25		7	16.0		5	10	7	16.0
26		8	-4.0		6		8	-4.0
27		8	71.0		7		8	71.0
28		9	-4.0		8		9	-4.0
29		9	16.0		9		9	16.0

**Fig. 5.** Illustration of the serial/multithreaded and distributed-memory parallel CSC-LT input formats for the *WSSMP* routine.

to row/column indices in *JA*, which contains the actual indices, and *AVALS* contains the values corresponding to the indices in *JA*. In MSR/MSF format, the diagonal entries are stored separately in *DIAG*. *PERM* and *INVP* contain the permutation and inverse permutation vectors, respectively, to reorder the matrix for reducing fill-in. This permutation can either be supplied by the user, or generated by *WSMP*. *B* is the  $LDB \times NRHS$  right-hand side matrix (vector, if  $NRHS = 1$ ), which is overwritten by the solution matrix/vector by *WSMP*. *AUX* is optional working storage that the user may supply to *WSMP* and *NAUX* is the size of *AUX*. *MRP* is an integer vector in which the user can request output information about the pivots. *IPARM* and *DPARM* are integer and double precision arrays that contain the various input-output numerical parameters that direct program flow (input parameters) and convey useful information to the user (output parameters). The most important of these parameters control which task(s) *WSMP* will be performing in a given call. *WSMP* can perform any set of consecutive tasks from the following list:

- Task # 1: Ordering
- Task # 2: Symbolic Factorization
- Task # 3: Cholesky or  $LDL^T$  Factorization
- Task # 4: Forward and Backward Elimination
- Task # 5: Iterative Refinement

The user specifies a starting task and an ending task and *WSMP* performs these tasks and all the tasks between them. The user can obtain the entire solution in one call or perform multiple factorizations for different matrices with identical structure by performing ordering and symbolic factorization only once, or solve for multiple RHS vectors with a given factorization, or perform multiple steps of iterative refinement for a given solution.

The input format and the calling sequence of the distribute memory parallel subroutine *PWSMP* are almost identical to those of *WSMP*. The distributed input format is illustrated in Figure 5(b) and the calling sequence is as follows.

```

SUBROUTINE PWSMP (N_i, IA_i, JA_i, AVALS_i, DIAG_i, PERM,
+                INVP, B_i, LDB_i, NRHS, AUX_i, NAUX_i,
+                MRP_i, IPARM, DPARM)

INTEGER*4  N_i, IA_i(N_i+1), JA_i(*), PERM(*), INVP(*),
+         LDB_i, NRHS, NAUX_i, MRP_i(N_i), IPARM(64)
REAL*8    AVALS_i(*), DIAG_i(N_i), B_i(LDB_i,NRHS),
+         AUX_i(NAUX_i), DPARM(64)

```

In the distributed version, an argument can be either *local* or *global*. A global array or variable must have the same size and contents on all nodes. The size and contents of a local variable or array vary among the nodes. In the context of *PWSMP*, global does not mean globally shared, but refers to data that is replicated on all nodes. In the above calling sequence, all arguments with a subscript are local.

$N_i$  is the number of columns/rows of the matrix  $A$  and the number of rows of the right-hand side  $B$  residing on node  $P_i$ . The total size  $N$  of system of equations is  $\sum_{i=0}^{p-1} N_i$ , where  $p$  is the number of nodes being used. The columns of the coefficient matrix can be distributed among the nodes in any fashion (see [16] for details).  $N_0 = N$  and  $N_i = 0$  for  $i > 0$  is also an acceptable distribution. Thus, it is rather easy to migrate from using *WSMP* on a single node to using it in the distributed-memory environment of an SP.

## 5 Performance Results

The performance of *WSMP* on a given number of processors for a problem depends on several factors, such as fill-in, load-imbalance, size of supernodes, etc. Many of these factors depend on ordering and the structure of the sparse matrix being factored. Therefore, the performance on the same number of processors can vary widely for different problems of the same size (in terms of the total amount of computation required). In Tables 1 and 2, we present some performance results of *WSMP* on two practical classes of symmetric sparse matrices on up to 128 nodes of an SP (each node of the SP is a 2-way SMP with 200 MHz Power3 processors with one/two Gigabytes of RAM). We feel that a reader may find these results more interpretive than those for a large number of unrelated problems. The first column of the tables gives some relevant information about the matrices, including the number of nonzeros, nnz\_L, in the triangular factor (in millions), the number of floating point operations, F\_opc, required for factorizations (in billions), and the number of equations, N, in the system. The number of factor nonzeros and the factorization op-count values are only averages; the actual values vary depending on the number of nodes used because the ordering is different on different number of nodes.

Table 1 shows the performance of factorization and triangular solves for sparse matrices arising from three-dimensional finite-difference grids discretized using a 7-point stencil. The grid dimension  $n$  is increased in such steps as to increase the factorization work by approximately a factor of two. For this, we have used the fact that with an optimal ordering (finding which is an NP-hard problem), the number of floating point operations required to factor a sparse matrix derived from an  $n \times n \times n$  grid is  $\Theta(n^6)$ .

Table 2 shows the performance of factorization and triangular solves for some sparse matrices arising from finite-element meshes in a sheet metal forming application, some matrices from CFD applications, and one from a linear programming model. In the first five problems, the same surface is discretized with different degrees of fineness (and thus, with different number of elements) to control the problem size.

## 6 Concluding Remarks

In this paper, we have described a scalable parallel solver, *WSMP*, for sparse symmetric systems of linear equations, which works on serial, message-passing

Problem Description	Number of 2-way SMP Nodes								
	1	2	4	8	16	32	64	128	
<b>Cube-49</b>	60.4	33.9	19.2	12.2	7.57	4.16	2.64	1.86	F_time
F_opc = 71 B	1156	2152	3920	5969	9818	16700	26666	37476	F_Mflops
nnz_L = 42 M	.778	.443	.256	.164	.120	.098	.092	.098	S_time
N = 117,649	219	394	694	1048	1455	1728	1829	1714	S_Mflops
<b>Cube-55</b>	126.	64.8	38.2	22.2	13.2	7.93	4.83	3.15	F_time
F_opc = 149 B	1184	2276	4032	6682	11380	18806	30243	45477	F_Mflops
nnz_L = 71 M	1.27	.695	.403	.240	.170	.132	.119	.122	S_time
N = 166,375	227	410	723	1175	1691	2158	2345	2255	S_Mflops
<b>Cube-62</b>	269.	131.	74.6	45.4	26.1	12.4	7.14	4.76	F_time
F_opc = 255 B	1203	2348	4222	6977	12136	19332	31652	47021	F_Mflops
nnz_L = 99 M	2.04	1.10	.858	.665	.264	.168	.145	.149	S_time
N = 238,328	240	430	773	1278	1819	2348	2616	2529	S_Mflops
<b>Cube-70</b>		291.	151.	88.2	52.1	29.3	16.5	10.5	F_time
F_opc = 649 B		2313	4480	7548	12888	22190	39056	61582	F_Mflops
nnz_L = 198 M		1.80	.982	.556	.361	.258	.215	.215	S_time
N = 343,000		451	832	1452	2229	3049	3678	3678	S_Mflops
<b>Cube-79</b>			313.	178.	93.0	54.3	31.2	18.4	F_time
F_opc = 1342 B			4525	8041	14092	24390	42273	70510	F_Mflops
nnz_L = 329 M			1.59	.897	.521	.366	.280	.268	S_time
N = 493,039			851	1521	2481	3551	4619	4769	S_Mflops
<b>Cube-89</b>				357.	215.	97.7	38.7	24.2	F_time
F_opc = 2254 B				8197	11510	23071	41574	66460	F_Mflops
nnz_L = 466 M				1.36	.955	.501	.344	.310	S_time
N = 704,969				1646	2182	3721	4562	5061	S_Mflops
<b>Cube-100</b>					410.	143.	82.4	44.4	F_time
F_opc = 4333 B					14534	24836	43390	78663	F_Mflops
nnz_L = 742 M					1.23	.812	.498	.425	S_time
N = 1,000,000					2985	4022	5397	6184	S_Mflops
<b>Cube-112</b>						435.	235.	128.	F_time
F_opc = 11507 B						27227	50028	87131	F_Mflops
nnz_L = 1401 M						1.74	.801	.640	S_time
N = 1,404,928						3397	7173	8913	S_Mflops
<b>Cube-120</b>							335.	188.	F_time
F_opc = 17135 B							52336	91928	F_Mflops
nnz_L = 1832 M							1.06	.701	S_time
N = 1,728,000							7314	10012	S_Mflops

**Table 1.** *WSMP* performance on for sparse matrices arising from finite-difference discretization of 3-D domains. A matrix *Cube-n* is obtained from an  $n \times n \times n$  3-D mesh. *F\_time* is factorization time in seconds, *F\_Mflops* is factorization Megaflops, *S\_time* is solution time, *S\_Mflops* is solution Megaflops, *F\_opc* is number of factorization floating point operations in billions, and *nnz\_L* is the number of nonzeros in the triangular factor in millions.



Problem Description	Number of 2-way SMP Nodes								
	1	2	4	8	16	32	64	128	
<b>Sheet-20000</b>	13.6	6.79	4.53	2.35	1.68	1.13	.773	.622	F_time
F_opc = 12 B	1014	1744	2827	5228	7447	10947	16052	19898	F_Mflops
nnz_L = 23 M	.489	.237	.148	.090	.067	.058	.056	.067	S_time
N = 91,854	192	386	631	1028	1388	1595	1678	1399	S_Mflops
<b>Sheet-40000</b>	42.0	18.8	12.0	6.95	4.14	2.65	1.67	1.24	F_time
F_opc = 36 B	941	1971	3029	5298	8806	13609	21328	28695	F_Mflops
nnz_L = 52 M	1.05	.509	.287	.178	.123	.097	.090	.094	S_time
N = 182,574	200	411	721	1173	1693	2150	2309	2204	S_Mflops
<b>Sheet-60000</b>	65.6	37.6	22.3	12.5	7.62	4.26	2.90	1.89	F_time
F_opc = 69 B	1054	1859	3185	5466	9194	15984	23314	35130	F_Mflops
nnz_L = 84 M	1.67	.802	.455	.278	.174	.136	.127	.119	S_time
N = 273,159	202	421	748	1211	1953	2484	2657	2820	S_Mflops
<b>Sheet-80000</b>	95.8	54.2	28.4	17.7	10.6	6.47	4.06	2.65	F_time
F_opc = 117 B	1074	1988	3703	5844	10224	16329	26340	39835	F_Mflops
nnz_L = 105 M	2.17	1.09	.577	.354	.225	.172	.143	.142	S_time
N = 363,699	216	432	811	1316	2095	2717	3306	3325	S_Mflops
<b>Sheet-100000</b>	161.	83.4	42.7	29.5	13.8	8.25	5.21	3.63	F_time
F_opc = 151 B	1037	1911	3620	5301	10809	18061	28787	42039	F_Mflops
nnz_L = 152 M	3.05	1.42	.739	.471	.279	.199	.182	.160	S_time
N = 454,059	211	434	828	1304	2181	3051	3483	3847	S_Mflops
<b>Gismondi</b>	104.	58.3	33.5	21.4	9.87	6.42	3.82	2.56	F_time
F_opc = 110 B	1113	1958	3331	5214	11190	17215	27130	40607	F_Mflops
nnz_L = 30 M	.492	.322	.198	.134	.072	.081	.078	.090	S_time
N = 17,447	259	392	629	933	1727	1544	1537	1382	S_Mflops
<b>Torso</b>	130.	63.2	41.0	26.3	16.2	8.47	5.33	4.03	F_time
F_opc = 133 B	1036	2193	3303	5115	8208	15612	24112	32395	F_Mflops
nnz_L = 87 M	1.72	.800	.564	.315	.200	.156	.140	.136	S_time
N = 454,059	202	436	617	1102	1727	2221	2434	2523	S_Mflops
<b>Aorta</b>	81.5	55.0	35.4	20.8	13.1	8.41	5.83	3.77	F_time
F_opc = 87 B	1015	1595	2547	4258	6808	10414	15106	22812	F_Mflops
nnz_L = 87 M	2.42	1.97	1.42	.979	.595	.412	.267	.213	S_time
N = 454,059	139	177	247	356	586	844	1298	1620	S_Mflops
<b>Mdual2</b>	224.	143.	84.9	45.8	27.5	17.3	9.49	6.25	F_time
F_opc = 267 B	967	2063	3366	6215	9417	14800	26604	39202	F_Mflops
nnz_L = 179 M	4.46	2.38	1.32	.746	.485	.323	.248	.239	S_time
N = 454,059	152	308	551	975	1460	2182	2832	2904	S_Mflops

**Table 2.** *WSMP* performance on some sparse matrices generated by a 2-D finite-element model (Sheet-n matrices), CFD applications (Torso, Aorta, and Mdual2), and linear programming (Gismondi). F\_time is factorization time in seconds, F\_Mflops is factorization Megaflops, S\_time is solution time, S\_Mflops is solution Megaflops, F\_opc is number of factorization floating point operations in billions, and nnz\_L is the number of nonzeros in the triangular factor in millions.

parallel, and shared-memory parallel machines, or any combination of these. Our experimental performance results show that the solver scales well on up to hundreds of processors. On large enough problems, the factorization performance exceeds 90 Gigafllops on 128 nodes (256 processors), thereby obtaining about 45% of the peak theoretical performance of each processor and about 60% efficiency with respect to the serial performance of the solver on similar problems. Even on a relatively small problem like Sheet-100000 that can fit in the memory of a single node and requires only 3.6 seconds for factorization on 128 nodes, the 256-processor ensemble operates at more than one-fifth of the peak theoretical performance of each processor and at more than one-third the efficiency of a single node (a speedup of 44 on 128 nodes). As a result, *WSMP* is being used in several academic, research, and commercial applications, in many of which, it is solving problems that were too large to be solved with a direct solver until recently.

## References

1. Patrick R. Amestoy, Iain S. Duff, and J. Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. Technical Report RAL-98/51, Rutherford Appleton Laboratory, Oxon, England, 1998.
2. Patrick R. Amestoy, Iain S. Duff, and J. Y. L'Excellent. MUMPS: Multifrontal massively parallel solver version 2.0. Technical Report PA-98/02, CERFACS, Toulouse Cedex, France, 1998.
3. Cleve Ashcraft. The domain/segment partition for the factorization of sparse symmetric positive definite matrices. Technical Report ECA-TR-148, Boeing Computer Services, Seattle, WA, 1990.
4. Cleve Ashcraft. The fan-both family of column-based distributed Cholesky factorization algorithms. In Alan George, John R. Gilbert, and Joseph W.-H. Liu, editors, *Graph Theory and Sparse Matrix Computations*. Springer-Verlag, New York, NY, 1993.
5. Cleve Ashcraft and Roger G. Grimes. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.
6. Cleve Ashcraft and Joseph W.-H. Liu. Robust ordering of sparse matrices using multisection. Technical Report CS 96-01, Department of Computer Science, York University, Ontario, Canada, 1996.
7. E. Chu, Alan George, Joseph W.-H. Liu, and Esmond G.-Y. Ng. Users guide for SPARSPAK-A: Waterloo sparse linear equations package. Technical Report CS-84-36, University of Waterloo, Waterloo, IA, 1984.
8. William Dearholt, Steven Castillo, and Gary Hennigan. Direct solution of large, sparse systems of equations on a distributed memory computer. Technical report, New Mexico State University, 1999.
9. G. A. Geist and Esmond G.-Y. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291-314, 1989.
10. Alan George, M. T. Heath, Joseph W.-H. Liu, and Esmond G.-Y. Ng. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327-340, 1988.

11. John R. Gilbert and Robert Schreiber. Highly parallel sparse Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 13:1151–1172, 1992.
12. Anoop Gupta and Edward Rothberg. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Supercomputing '93 Proceedings*, 1993.
13. Anshul Gupta. Fast and effective algorithms for graph partitioning and sparse matrix ordering. *IBM Journal of Research and Development*, 41(1/2):171–183, January/March, 1997.
14. Anshul Gupta. WSMP: Watson sparse matrix package (Part-I: direct solution of symmetric sparse systems). Technical Report RC 21886 (98462), IBM T. J. Watson Research Center, Yorktown Heights, NY, November 16, 2000. <http://www.cs.umn.edu/~agupta/wsmmp.html>.
15. Anshul Gupta. WSMP: Watson sparse matrix package (Part-II: direct solution of general sparse systems). Technical Report RC 21888 (98472), IBM T. J. Watson Research Center, Yorktown Heights, NY, November 20, 2000. <http://www.cs.umn.edu/~agupta/wsmmp.html>.
16. Anshul Gupta, Mahesh Joshi, and Vipin Kumar. WSSMP: Watson symmetric sparse matrix package: The user interface. Technical Report RC 20923 (92669), IBM T. J. Watson Research Center, Yorktown Heights, NY, July 17, 1997.
17. Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, May 1997.
18. M. T. Heath, Esmond G.-Y. Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.
19. M. T. Heath and Padma Raghavan. Distributed solution of sparse linear systems. Technical Report 93-1793, Department of Computer Science, University of Illinois, Urbana, IL, 1993.
20. Bruce A. Hendrickson and Edward Rothberg. Improving the runtime and quality of nested dissection ordering. Technical Report SAND96-0868J, Sandia National Laboratories, Albuquerque, NM, 1996.
21. Laurie Hulbert and Earl Zmijewski. Limiting communication in parallel sparse Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1184–1197, September 1991.
22. Mahesh Joshi, Anshul Gupta, George Karypis, and Vipin Kumar. Two-dimensional scalable parallel algorithms for solution of triangular systems. Technical Report TR 97-024, Department of Computer Science, University of Minnesota, Minneapolis, MN, July 1997. A short version appears in *Proceedings of the 1997 International Conference on High Performance Computing (HiPC'97)*, Bangalore, India.
23. Mahesh Joshi, George Karypis, Vipin Kumar, Anshul Gupta, and Fred G. Gustavson. PSPASES: An efficient and scalable parallel sparse direct solver. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.
24. George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995.
25. George Karypis and Vipin Kumar. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Technical Report TR 97-060, Department of Computer Science, University of Minnesota, 1997.

26. George Karypis and Vipin Kumar. Parallel algorithms for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–95, 1998.
27. Joseph W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34:82–109, 1992.
28. Robert F. Lucas, Tom Blank, and Jerome J. Tiemann. A parallel solution method for large sparse systems of equations. *IEEE Transactions on Computer Aided Design*, CAD-6(6):981–991, November 1987.
29. Alex Pothén and Chunguang Sun. Distributed multifrontal factorization using clique trees. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 34–40, 1991.
30. Edward Rothberg and Robert Schreiber. Improved load distribution in parallel sparse Cholesky factorization. In *Supercomputing '94 Proceedings*, 1994.