An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination

James W. Demmel*　　John R. Gilbert†　　Xiaoye S. Li‡

February 27, 1997

Abstract

Although Gaussian elimination with partial pivoting is a robust algorithm to solve unsymmetric sparse linear systems of equations, it is difficult to implement efficiently on parallel machines, because of its dynamic and somewhat unpredictable way of generating work and intermediate results at run time. In this paper, we present an efficient parallel algorithm that overcomes this difficulty. The high performance of our algorithm is achieved through (1) using a graph reduction technique and a supernode-panel computational kernel for high single processor utilization, and (2) scheduling two types of parallel tasks for a high level of concurrency. One such task is factoring the independent panels on the disjoint subtrees in the column elimination tree of A. Another task is updating a panel by previously computed supernodes. A scheduler assigns tasks to free processors dynamically and facilitates the smooth transition between the two types of parallel tasks. No global synchronization is used in the algorithm. The algorithm is well suited for shared memory machines (SMP) with a modest number of processors. We demonstrate 4–7 fold speedups on a range of 8 processor SMPs, and more on larger SMPs. One realistic problem arising from a 3-D flow calculation achieves factorization rates of 1.0, 2.5, 0.8 and 0.8 Gigaflops, on the 12 processor Power Challenge, 8 processor Cray C90, 16 processor Cray J90, and 8 processor AlphaServer 8400, respectively.

1 Introduction

In earlier work with Eisenstat and Liu, we described a publically released sequential software library, SuperLU, to solve unsymmetric sparse linear systems using Gaussian elimination with partial pivoting [5]. This left-looking, blocked algorithm includes symmetric structural reduction for fast symbolic factorization, and supernode-panel updates to achieve better data reuse in cache and floating-point registers.

*Computer Science Division, University of California, Berkeley, CA 94720 (demmel@cs.berkeley.edu). The research of Demmel and Li was supported in part by NSF grant ASC–9313958, DOE grant DE–FG03–94ER25219, UT Subcontract No. ORA4466 from ARPA Contract No. DAAL03–91–C0047, DOE grant DE–FG03–94ER25206, and NSF Infrastructure grants CDA–8722788 and CDA–9401156.

†Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304 (gilbert@parc.xerox.com). The research of this author was supported in part by the Institute for Mathematics and Its Applications at the University of Minnesota and in part by DARPA Contract No. DABT63-95-C0087. Copyright © 1994-1997 by Xerox Corporation. All rights reserved.

‡National Energy Research Scientific Computing (NERSC) Center, Lawrence Berkeley National Lab, 1 Cyclotron Rd, Berkeley, CA 94720 (xiaoye@nersc.gov). This work was supported in part by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number 76SF00098.

In this paper we study an efficient parallel algorithm based on SuperLU. The primary objective of this work is to achieve good efficiency on shared memory systems with a modest number of processors (for example, between 10 and 20). In addition to measuring the efficiency of our parallel algorithm on these machines, we also study a theoretical upper bound on performance of this algorithm. The efficiency of the algorithm has been demonstrated on several shared memory parallel machines. When compared to the best sequential runtime of SuperLU, the parallel algorithm typically achieved 4 to 7 speedups on 8 processors platforms, for large sparse matrices.

The rest of the paper is organized as follows. In Section 2 we review the sequential SuperLU algorithm. Section 3 presents the sources and the characteristics of the test matrices. Section 4 presents the parallel machines used in our study. In Section 5 we describe several design choices we have made in parallelization, including how to find parallelism, how to define individual tasks and memory management for supernodes. Section 6 sketches the high-level parallel scheduling algorithm. In Section 7, we present the parallel performance achieved with the test matrices on a number of platforms. Both time and space efficiency will be illustrated. We also quantify the sources of the overhead in parallelization, and give a thorough analysis of their impact on performance. In the end of this section we establish a PRAM (Parallel Random-Access Machine) model to predict an upper bound on speedups attainable by the proposed algorithm. Finally, Section 8 draws conclusions and suggests future research.

2 Overview of sequential algorithm in SuperLU

Figure 1 sketches the supernode-panel factorization algorithm used in SuperLU. A *supernode* is defined to be a range $(r\colon s)$ of columns of L with the triangular block just below the diagonal being full, and with the same row structure below this block. We store a supernode as a rectangular block, including the triangle of U in rows and columns r through s, see Figure 2. This allows us to address each supernode as a two-dimensional array in calls to BLAS routines [6, 7], and so get high performance. To increase the average size of supernodes (and hence performance), we merge groups of consecutive columns (usually no more than 4 columns) at the fringe of the column elimination tree (Section 5.1) into *relaxed* supernode regardless of their row structures. A *panel* is a block of w consecutive columns in the matrix which are updated simultaneously by a supernode using calls to the BLAS. The row structures of the columns in a panel may not be correlated in any fashion, and the boundaries between panels may be different from those between supernodes. Each panel factorization, outer loop i n Figure 1, consists of three distinct steps: (1) the symbolic factorization to determine the nonzero structure, (2) the numerical updates by supernodes, and (3) the factorization of each column in the panel. The pivot selection, detection of the supernode boundary, and symmetric structure reduction (to reduce the cost of later symbolic factorization steps) are all performed in the inner factorization step. Both panel and column symbolic steps use depth-first search (DFS). A further refinement, a two-dimensional supernode partitioning (defined by the blocking parameters t and b in Figure 2), enhances performance for large matrices and machines with small caches. A more detailed description of SuperLU is in the paper [5].

We conducted extensive performance evaluation for SuperLU on several recent superscalar architectures. For large sparse matrices, SuperLU achieves up to 40% of the peak floating-point performance on both IBM RS/6000-590 and MIPS R8000. It achieves nearly 25% peak on the DEC Alpha 21164. More details can be found in [24].

```
**for** column $j = 1$ **to** $n$ **step** $w$ **do**
    $F(:, j : j + w - 1) = A(:, j : j + w - 1)$;
    (1) *Predict the nonzero structure of panel* $F(:, j : j + w - 1)$*:*
        Determine which supernodes will update any of $F(:, j{:}\, j + w - 1)$;
    (2) *Update panel* $F(:, j : j + w - 1)$ *using previous supernodes:*
        **for** each updating supernode $(r{:}\, s) < j$ in topological order **do**
            • Triangular solve:
              $U(r : s, j : j + w - 1) = L(r : s, r : s) \backslash F(r : s, j : j + w - 1)$;
            • Matrix update:
              $F(s + 1 : n, j : j + w - 1) = F(s + 1 : n, j : j + w - 1) -$
                                  $L(s + 1 : n, r : s) \cdot U(r : s, j : j + w - 1)$;
        **end for** $(r : s)$;
    (3) *Inner factorization for each column in the panel:*
        **for** column $jj = j$ **to** $jj + w - 1$ **do**
            • Supernode-column update for column $F(j : n, jj)$;
            • Row pivoting for column $F(jj : n, jj)$;
            • Determine whether $jj$ belongs to the same supernode as $jj - 1$;
            • Symmetric structure pruning;
        **end for** $jj$;
**end for** $j$;
```

Figure 1: The supernode-panel factorization algorithm.

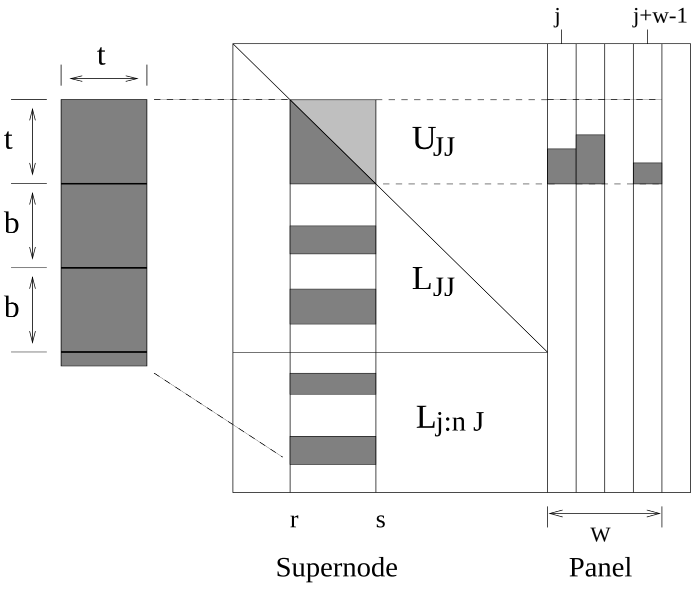


Figure 2: Illustration of a supernode-panel update. $J = 1{:}\, j - 1$.

3 Test matrices

To evaluate our algorithms, we have collected matrices from various sources, with their characteristics summarized in Table 1.

Some of the matrices are from the Harwell-Boeing collection [8]. Many of the larger matrices are from the ftp site maintained by Tim Davis of the University of Florida.[1] Those matrices are as follows. Memplus is a circuit simulation matrix from Steve Hamm of Motorola. Rdist1 is a reactive distillation problem in chemical process separation calculations, provided by Stephen Zitney at Cray Research, Inc. Shyy161 is derived from a direct, fully-coupled method for solving the Navier-Stokes equations for viscous flow calculations, provided by Wei Shyy of the University of Florida. Goodwin is a finite element matrix in a nonlinear solver for a fluid mechanics problem, provided by Ralph Goodwin of the University of Illinois at Urbana-Champaign. Venkat01, Inaccura and Raefsky3/4 were provided by Horst Simon then of NASA and currently at NERSC. Venkat01 comes from an implicit 2-D Euler solver for an unstructured grid in a flow simulation. Raefsky3 is from a fluid structure interaction turbulence problem. Raefsky4 is from a buckling problem for a container model. Af23560 is from solving an unsymmetric eigenvalue problem, provided by Zhaojun Bai of the University of Kentucky. Ex11 is from a 3-D steady flow calculation in the SPARSKIT collection maintained by Youcef Saad at the University of Minnesota. Wang3 is from solving a coupled nonlinear PDE system in a 3-D ($30 \times 30 \times 30$ uniform mesh) semiconductor device simulation, as provided by Song Wang of the University of New South Wales, Sydney. Vavasis3 is an unsymmetric augmented matrix for a 2-D PDE with highly varying coefficients [31]. Dense1000 is a dense 1000×1000 random matrix.

This paper does not address the performance of column preordering for sparsity. We simply use the existing ordering algorithms provided by Matlab [17]. For all matrices except 1, 15 and 21, the columns were permuted by Matlab's minimum degree ordering of $A^T A$, also known as "column minimum degree" ordering. However, this ordering produces a tremendous amount of fill for matrices 1, 15 and 21, because it only attempts to minimize the upper bound on the actual fill and the upper bounds are too loose in these cases. We found that when these three matrices were symmetrically permuted by Matlab's symmetric minimum degree ordering on $A + A^T$, the amount of fill is much smaller than using column minimum degree ordering. The last column in Table 1 shows the number of nonzeros in matrix F when using these column preorderings.

The matrices are sorted in increasing order of $flops/nnz(F)$, the ratio of the number of floating-point operations to the number of nonzeros $nnz(F)$. This "figure of merit" gives the maximum potential data reuse, as described in [5]. Thus, we expect our performance to increase with increasing $flops/nnz(F)$.

4 Shared memory multiprocessor systems used for testing

We evaluated the parallel algorithm on several commercially popular machines, including the Sun SPARCcenter 2000 [30], SGI Power Challenge [29], DEC AlphaServer 8400 [11], and Cray C90/J90 [32, 33]. Table 2 summarizes the configurations and several key parameters of the five parallel systems. In the column "Bus Bandwidth" we report the effective or sustainable bandwidth to main memory. In "Read Latency" we report the minimum amount of time it takes a processor to fetch a piece of data from main memory into a register in response to a load instruction.

The last column in the table shows the programming model used to enable multiprocessing. All the systems provide lightweight multithreading or multitasking libraries. Synchronization and

[1]URL: http://www.cis.ufl.edu/~davis.

	Matrix	s	n	$nnz(A)$	$\frac{nnz(A)}{n}$	$nnz(F)$	#flops/$nnz(F)$
1	MEMPLUS	.983	17758	99147	5.6	140388	12.5
2	GEMAT11	.002	4929	33185	6.7	93370	16.3
3	RDIST1	.062	4134	9408	2.3	338624	38.1
4	ORANI678	.073	2529	90158	35.6	280788	53.3
5	MCFE	.709	765	24382	31.8	69053	59.9
6	LNSP3937	.869	3937	25407	6.5	427600	91.1
7	LNS_3937	.869	3937	25407	6.5	449346	99.7
8	SHERMAN5	.780	3312	20793	6.3	249199	101.3
9	JPWH_991	.947	991	6027	6.1	140746	127.7
10	SHERMAN3	1.000	5005	20033	4.0	433376	139.8
11	ORSREG_1	1.000	2205	14133	6.4	402478	148.6
12	SAYLR4	1.000	3564	22316	6.3	654908	160.0
13	SHYY161	.769	76480	329762	4.3	7634810	205.8
14	GOODWIN	.642	7320	324772	44.4	3109585	213.9
15	VENKAT01	1.000	62424	1717792	27.5	12987004	247.9
16	INACCURA	1.000	16146	1015156	62.9	9941478	414.3
17	AF23560	.947	23560	460598	19.6	13986992	454.9
18	DENSE1000	1.000	1000	1000000	1000	1000000	666.2
19	RAEFSKY3	1.000	21200	1488768	70.2	17544134	690.7
20	EX11	1.000	16614	1096948	66.0	26207974	1023.1
21	WANG3	1.000	26064	177168	6.8	13287108	1095.5
22	RAEFSKY4	1.000	19779	1316789	66.6	26678597	1172.6
23	VAVASIS3	.001	41092	1683902	41.0	49192880	1813.5

Table 1: Characteristics of the test matrices. Structural symmetry s is defined to be the fraction of the nonzeros matched by nonzeros in symmetric locations. None of the matrices are numerically symmetric. $nnz(A)$ is the number of nonzeros in A. $F = L + U - I$ is the filled matrix, and I is an identity matrix.

Machine	Processor	CPUs	Bus Bandwidth	Read Latency	Memory Size	Programming Model
Sun SPARCcenter 2000	SuperSPARC	4	500 MB/s	1200 ns	196 MB	Solaris thread
SGI Power Challenge	MIPS R8000	16	1.2 GB/s	252 ns	2 GB	Parallel C
DEC AlphaServer 8400	Alpha 21164	8	1.6 GB/s	260 ns	4 GB	pthread
Cray PVP	C90	8	245.8 GB/s	96 ns	640 MB	microtasking
Cray PVP	J90	16	51.2 GB/s	330 ns	640 MB	microtasking

Table 2: Characteristics of the parallel machines used in our study.

	Clock MHz	On-chip Cache	External Cache	#Flops/ 1 cycle	Peak Mflops	DGEMM Mflops	DGEMV Mflops
MIPS R8000	90	16 KB	4 MB	4	360	340	210
Alpha 21164	300	8 KB-L1 96 KB-L2	4 MB	2	600	350	135
SuperSPARC	50	16 KB	1 MB	1	50	45*	–
C90	240	–	–	4	960	900	890
J90	100	–	–	2	200	190	167

Table 3: Some characteristics of the processors used in the parallel systems.

context switching of the threads are accomplished rapidly at the user level, without entering the OS kernel. For P processors, we usually create P (logical) threads for the scheduling loop `Slave_worker()` (Figure 10). Scheduling these threads on available physical processors is done by the operating system or runtime library. Thread migration between processors is usually invisible to us. The program is easily portable to multiple platforms. The source codes on different machines differ only in thread spawning and locking primitives.

Table 3 summarizes the characteristics of the individual processors in the parallel machines, including the clock speed, the cache size, the peak Mflop rate, and the DGEMM and DGEMV peak Mflop rate. Most DGEMM and DGEMV Mflop rates were measured using vendor-supplied BLAS libraries. When the vendors do not provide a BLAS library, we report the results from PHiPAC [4], with an asterisk (*) beside such a number. For some machines, PHiPAC is often faster than the vendor-supplied DGEMM.

5 Parallel strategies

In this section, we present crucial design choices we have made to parallelize SuperLU, such as, how we shall exploit both coarse and fine levels of parallelism, how we shall define the individual tasks, and how we shall deal with the issue of dynamic memory growth.

In order to make the parallel algorithm efficient, we need to make non-trivial modifications to serial SuperLU. All these changes are summarized in Table 4 and discussed in the subsections below. These show that the parallel algorithm is not a straightforward parallelization of the serial one, and illustrate the program complications arising from parallelization. In the performance evaluation, we will time various parts of the algorithms. The time notation to be used is listed in Table 5.

Construct	Parallel algorithm
panel	restricted so it does not contain branchings in the etree (Section 5.2)
supernode	restricted to be a fundamental supernode in the etree (Section 5.3)
supernode storage	use either static or dynamic upper bound (Section 5.3)
pruning & DFS	use both $G(L^T)$ and pruned $G(L^T)$ to avoid locking (Section 5.4)

Table 4: The differences of the parallel algorithm from serial SuperLU.

Notation	Meaning
T_s	SuperLU best serial time
T'_s	SuperLU serial time with smaller blocking tuned for parallel code
T_1	execution time of the parallel code on one processor
T_P	parallel execution time on P processors
T_I	total idle time of all processors

Table 5: The differences of the parallel algorithm from serial SuperLU.

5.1 Parallelism

We exploit two sources of parallelism in the sparse LU factorization. The coarse level parallelism comes from the sparsity of the matrix, and is exposed to us by the *column elimination tree* (or column etree for short) of A. The vertices of this tree are the integers 1 through n, representing the columns of A. The column etree of A is the (symmetric) elimination tree of A^TA provided there is no cancellation in computing A^TA. More specifically, if L_c denotes the Cholesky factor of A^TA, then the parent of vertex j is the row index i of the first nonzero entry below the diagonal of column $L_c(:, j)$. The column etree can be computed from A in time almost linear in the number of nonzeros of A by a variation of an algorithm of Liu [25].

Theorem 1 (Column Elimination Tree) [18] *Let A be a square, nonsingular, possibly unsymmetric matrix, and let $PA = LU$ be any factorization of A with pivoting by row interchanges. Let T be the column elimination tree of A.*

1. *If vertex i is an ancestor of vertex j in T, then $i \geq j$.*

2. *If $l_{ij} \neq 0$, then vertex i is an ancestor of vertex j in T.*

3. *If $u_{ij} \neq 0$, then vertex j is an ancestor of vertex i in T.*

4. *Suppose in addition that A is strong Hall (that is, it cannot be permuted to a nontrivial block triangular form). If vertex j is the parent of vertex i in T, then there is some choice of values for the nonzeros of A that makes $u_{ij} \neq 0$ when the factorization $PA = LU$ is computed with partial pivoting.*

Since column i updates column j in LU factorization if and only if $u_{ij} \neq 0$, part 3 of Theorem 1 implies that the columns in different subtrees do not update one another. Furthermore, the columns in independent subtrees can be computed without referring to any common memory, because the columns they depend on have completely disjoint row indices [19, Theorem 3.2]. It has been shown in a series of studies [13, 14, 18, 19] that the column etree gives the information about all potential dependencies.

In general we cannot predict the nonzero structure of U precisely before the factorization, because the pivoting choice and hence the exact nonzero structure depend on numerical values. The column etree can overestimate the true column dependencies. An example is

$$\mathrm{A} = \begin{pmatrix} 1 & \bullet & & \\ \bullet & 2 & \bullet & \\ \bullet & & 3 & \bullet \\ \bullet & & & 4 \end{pmatrix},$$

in which the Cholesky factor L_c of $A^T A$ is symbolically full, so the column etree is a single chain. But if the numerical values are such that row 4 is selected as the pivot row at the first step of elimination, column 1 will update neither column 2 nor column 3. Despite the possible overestimate, part 4 of Theorem 1 says that if A is strong Hall, this dependency is the strongest information obtainable from the structure of A alone.

Having studied the parallelism arising from different subtrees, we now turn our attention to the dependent columns, that is, the columns having ancestor-descendant relations. When the elimination process proceeds to a stage where there are more processors than independent subtrees, we need to make sure all processors work cooperatively on dependent columns. Thus the second level of parallelism comes from *pipelining* the computations of the dependent columns.

Consider a simple situation with only two processors. Processor 1 gets a task *Task 1* containing column j, processor 2 gets another task *Task 2* containing column k, and node j is a descendant of node k in the etree. The (potential) dependency says only that *Task 2* cannot finish its execution before *Task 1* finishes. However, processor 2 can start *Task 2* right away with the computations not involving column j; this includes performing the symbolic structure prediction and accumulating the numeric updates using the finished columns that are descendants in the etree. After processor 2 has finished the other part of the computation, it has to wait for *Task 1* to finish. (If *Task 1* is already finished at this moment, processor 2 does not waste any time waiting.) Then processor 2 will predict the new fills and perform numeric updates that may result from the finished columns in *Task 1*. In this way, both processors do useful work concurrently while still preserving the precedence constraint. Note that we assume the updates can be done in any order. This could give different (indeed, nondeterministic) numerical results from run to run.[2]

Although this pipelining mechanism is complicated to implement, it is essential to achieve higher concurrency. This is because, in most problems, a large percentage of the computation occurs at upper levels of the etree, where there are fewer branches than processors. An extreme example is a dense matrix, the etree of which is a single chain. In this case, the parallel SuperLU "reduces to" a pipelined column-oriented dense LU algorithm.

5.2 Panel tasks

As studied in [5], the introduction of supernodes and panels makes the computational kernels highly efficient. To retain the serial algorithm's ability to reuse data in cache and registers, we treat the factorization of one panel as a unit task to be scheduled; it computes the part of U and the part of L for all columns within this panel. Choosing a panel as scheduling unit affords the best granularity on the SMPs we targeted, and requires only modest changes to the serial code [5]. The alternative,

[2]In order to guarantee determinism, we must statically assign the tasks to processors. The performance cost we pay for determinism may be load imbalance and reduced parallelism. We are considering adding a debugging option to the software that guarantees determinism.

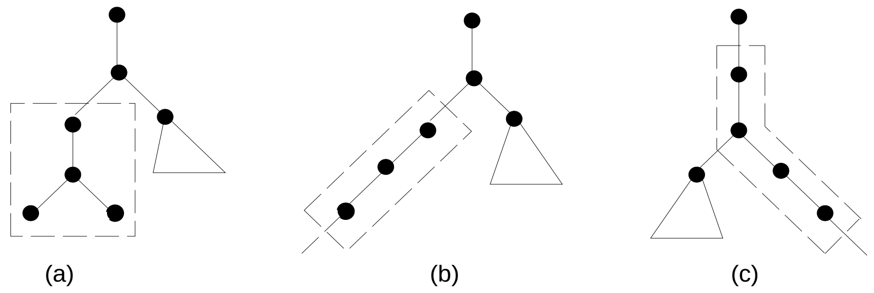


Figure 3: Panel definition. (a) a relaxed supernode at the bottom of the column etree; (b) consecutive columns from part of one branch of the etree; (c) consecutive columns from more than one branch of the etree.

blocking the matrix by rows and columns [22, 28], introduces too much synchronization overhead to make it worthwhile on SMPs with modest parallelism.

A panel task consists of two distinct subtasks. The first corresponds to the outer factorization, which accumulates the updates from the descendant supernodes. The second subtask is to perform the panel's inner factorization. We exploit parallelism within the first subtask, but not the second.

Since the parallel algorithm uses the column etree as the main scheduling tool, it is worth studying the relationship between the panels and the structure of the column etree. We assume that the columns of the matrix are ordered according to a postorder on the column etree. We expect a postorder on the column etree to bring together unsymmetric supernodes, just as a postorder on the symmetric etree brings together symmetric supernodes. Pictorially, panels can be classified into three types, depending on where they are located in the etree, as illustrated in Figure 3.

In the pipelining algorithm, panels of type (c) complicates the record-keeping if a processor owning this panel needs to wait for, and later perform, the updates from the busy panels down the etree. To simplify this, we imposed two restrictions. We first restricted the definition of panels so that type (c) panels do not occur. We will let a panel stop before a node (column) that has more then one child in the etree. That is, every branching node necessarily starts a new panel. Secondly, we make sure that all busy descendant panels always form one path in the etree. So the processor waiting for these busy panels can simply walk up the path in the etree starting from the most distant busy descendant.

By this restricted definition of panels, there will be more panels of smaller sizes. The question arises whether this will hurt performance. We studied the distribution of floating-point operations on different panel sizes for all of our test matrices, and observed that usually more than 95% of the floating-point operations are performed in the panels of largest size, and these panels tend to occur at a few topmost levels of the etree. Thus, panels of small sizes normally do not represent much computation. On uniprocessors, we see almost identical performance using the earlier and the new definitions of panels. Therefore, we believe that this restriction on panels simplifies and accelerates the parallel scheduling algorithm with little performance loss on individual processors.

5.3 Supernode storage using nonzero column counts in QR factorization

It is important to store the columns of a supernode *consecutively* in memory, so that we can call BLAS routines directly in-place without paying the cost of copying the columns into contiguous memory. Although this contiguity is easy to achieve in a sequential code, it poses problems in the parallel algorithm.

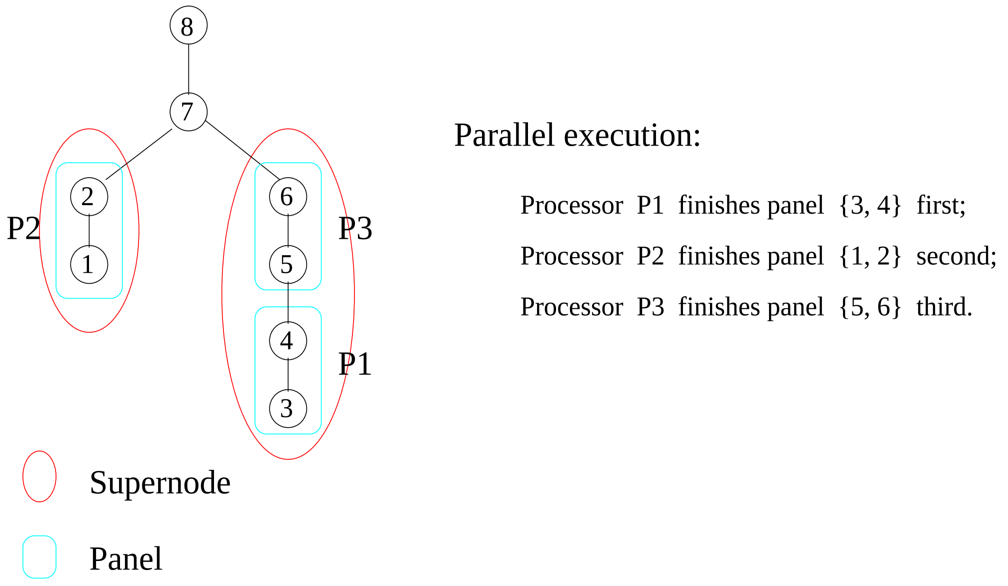

8
7
2
1
6
5
4
3
P2
P3
P1
Supernode
Panel
Parallel execution:
Processor P1 finishes panel {3, 4} first;
Processor P2 finishes panel {1, 2} second;
Processor P3 finishes panel {5, 6} third.

Figure 4: A snapshot of parallel execution.

Consider the scenario of parallel execution depicted in Figure 4. According to the order of the finishing times specified in the figure, the panel consisting of columns {3,4} will be stored in memory first, followed by panel {1,2}, and then followed by panel {5,6}. The supernode {3,4,5,6} is thus separated by the panel {1,2} in memory. The major difficulty comes from the fact that the supernodal structure emerges dynamically as the factorization proceeds, so we cannot statically calculate the amount of storage required by each supernode. Another difficulty is that panels and supernodes can overlap in several different ways.

One immediate solution would be not to allow any supernode to cross the boundary of a panel. In other words, the leading column of a panel would always be treated as the beginning of a new supernode. Thus a panel could possibly be subdivided into more than one supernode, but not vice versa. In such circumstances, the columns of a supernode would always be contiguous in memory because they would be assigned to a single processor by the scheduler. Each processor would simply store a (partial) ongoing supernode in its local temporary store, and copy the whole supernode into the global data structure as soon as it was finished.

This restriction on supernodes would mean that the maximum size of supernodes would be bounded by the panel size. As discussed in Section 7.5 (also [24]), for best per-processor efficiency and parallelism, we would like to have large supernodes but relatively small panels. These conflicting demands make it hard to find one good size for both supernodes and panels. We conducted an experiment with this scheme for the sequential algorithm. Figure 5 shows the uniprocessor performance loss for various panel sizes (i.e., maximum sizes of supernodes). For large matrices, say matrices 12 − 21, the smaller panels and supernodes result in more performance loss. For example, when $w = 16$, the slowdown can be as large as 20% to 68%. Even for large panel sizes, such as $w = 48$, the slowdown is still between 5% and 20%. However, in the parallel algorithm, such large panels give rise to too large a task granularity and severely limit the level of concurrency. We therefore feel that this simple solution is not satisfactory. Instead, we seek a solution that does not impose any restriction on the relation between panels and supernodes, and that allows us to vary the size of panels and supernodes independently in order to better trade off concurrency and single-processor efficiency.

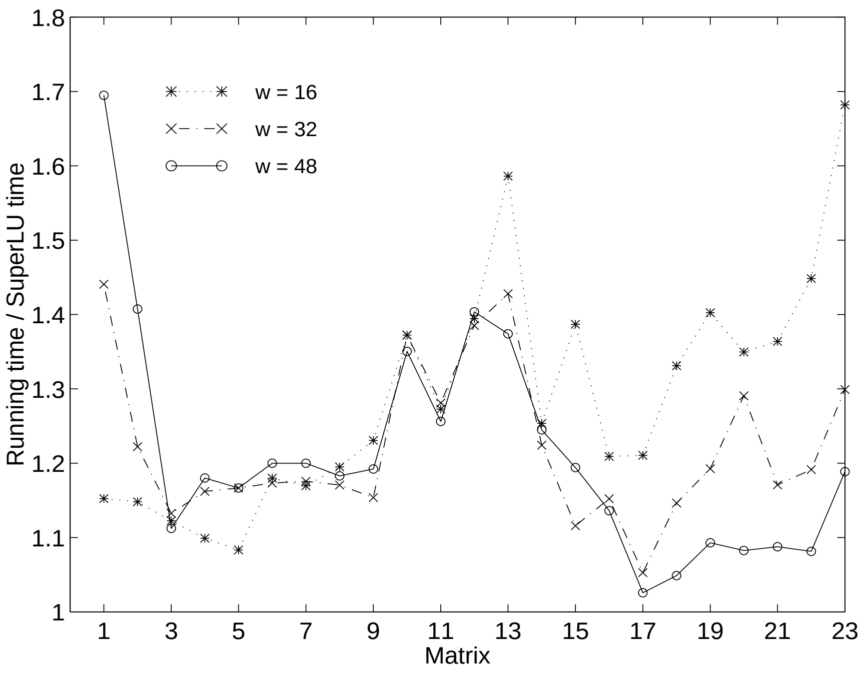


Figure 5: The sequential runtime penalty for requiring that a leading column of a panel also starts a new supernode. The times are measured on the RS/6000-590.

Our second and preferred solution is to preallocate space that is an upper bound on the actual storage needed by each supernode in the L factor, *irrespective of the numerical pivoting choice*. Then there will always be space to store supernode columns as they are computed. We now describe how we preallocate enough (but not too much) space.

After Gaussian elimination with partial pivoting, we can write $A = P_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1}\, U$, where P_i is an elementary permutation matrix representing the row interchange at step i, and L_i is a unit lower triangular matrix with its i-th column containing the multipliers at step i. We now define L as the unit lower triangular matrix whose i-th column is the i-th column of L_i, such that $L - I = \sum_i (L_i - I)$.[3] We shall make use of the following structure containment property in our storage scheme. Here we only quote the result without proof.

Theorem 2 [13, 15] *Consider the QR factorization $A = QR$ using Householder transformations. Let H be the symbolic Householder matrix consisting of the sequence of Householder vectors used to represent the factored form of Q. In other words, we assume no entries of H or R are zero because of numerical cancellation. If A is a nonsingular matrix with nonzero diagonal, and L and U are the triangular factors of A represented as above, then $Struct(L) \subseteq Struct(H)$, and $Struct(U) \subseteq Struct(R)$.*

In what follows, we describe how this upper bound can facilitate our storage management for the L supernodes. First, we need a notion of *fundamental* supernode, which was introduced by Ashcraft and Grimes [3] for symmetric matrices. In a fundamental supernode, every column except the last (numbered highest) is an only child in the elimination tree. Liu *et al.* [26] gave several

[3]This L is different from the $\hat{L}$ in $PA = \hat{L}U$. Both L and $\hat{L}$ contain the same nonzero values, but in different positions. In this section, L is used as a data structure for storing $\hat{L}$.

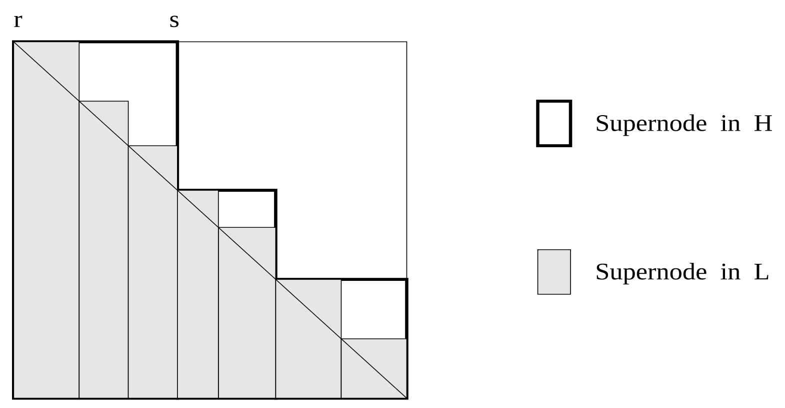


Figure 6: Bound the L supernode storage using the supernodes in H.

reasons why fundamental supernodes are appropriate, one of which is that the set of fundamental supernodes is the same regardless of the particular etree postordering. For consistency, we now also impose this restriction on the supernodes in L and H, respectively. For convenience, let S_L denote the fundamental supernodes in the L factor, and S_H denote the fundamental supernodes in the symbolic Householder matrix H. We shall omit the word "fundamental" when it is clear.

Our code breaks the L supernode at the boundary of an H supernode, forcing the L supernode to be contained in the H supernode. In fact, if we use fundamental L supernodes and ignore numerical cancellation (which we must do anyway for symmetric pruning), we can show that an L supernode is always contained in an H supernode [20].

Our objective is to allocate storage based on number of nonzeros in S_H, so that this storage is sufficiently large to hold S_L. Figure 6 illustrates the idea of using S_H as a bound. Two supernodes in S_L from different branches of the etree will go to their corresponding memory locations of the enclosing supernodes in S_H. Even if an H supernode breaks into multiple L supernodes, those L supernodes will all lie on one path in the column etree. Thus an L supernode from a different subtree cannot interrupt the storage for a supernode as in Figure 4. Since the panels (and hence the supernodes) within an H supernode are finished in order of increasing column numbers, the columns of each S_L supernode are contiguous in the storage of the S_H supernode.

To determine the storage for S_H, we need an efficient algorithm to compute the column counts $nnz(H_{*j})$ for H. We also need to identify the first vertex of each supernode in S_H. Then the number of nonzeros in each supernode is simply the product of the column count of the first vertex and the number of columns in the supernode.

Finding the first vertex and computing the column count can be done using a variant of the QR-column-count algorithm by Gilbert *et al.* [20]. The modified QR-column-count algorithm takes $Struct(A)$ and the postordered T as inputs, and computes $nnz(H_{*j})$ and S_H. The complexity of the algorithm is $O(m\ \alpha(m,n))$, where $m = nnz(A)$ and $\alpha(m,n)$ is the slowly-growing inverse of Ackermann's function coming from disjoint set union operations. In practice, it is as fast as computing the column etree T [24, Table 5.2]. In both the etree and QR-column-count algorithms, the disjoint set union operations are implemented using path halving and no union by rank (see [21] for details.)

One remaining issue yet to be addressed is what we should do if the static storage given by an upper bound structure is much too generous than actually needed. We developed a dynamic prediction scheme as a fallback for this situation. In this scheme, we still use the supernode partition S_H. Unlike the static scheme, which uses the column counts $nnz(H_{*j})$, we dynamically

	Matrix	Static	Dynamic
1	MEMPLUS	.04	.68
2	GEMAT11	.85	.90
3	RDIST1	.72	.73
4	ORANI678	.56	.90
5	MCFE	.73	.89
6	LNSP3937	.84	.92
7	LNS_3937	.86	.94
8	SHERMAN5	.92	.96
9	JPWH_991	.88	.94
10	SHERMAN3	.89	.91
11	ORSREG_1	.90	.92
12	SAYLR4	.89	.92
13	SHYY161	.91	.92
14	GOODWIN	.95	.98
15	VENKAT01	.11	.74
16	INACCURA	.96	.99
17	AF23560	.95	.97
18	DENSE1000	1.00	1.00
19	RAEFSKY3	.99	.99
20	EX11	.99	1.00
21	WANG3	.14	.89
22	RAEFSKY4	.99	.99
23	VAVASIS3	.95	.98

Table 6: Supernode storage utilization, using static and dynamic upper bounds. The number tabulated is the ratio of the number of nonzeros in supernodes of L to that in the prediction H.

compute the column count for the first column of each supernode in S_H as follows. When a processor obtains a panel that includes the first column of some supernode $H(:, r:s)$ in S_H, the processor invokes a search procedure on the directed graph $G(L(:, 1:r-1)^T)$, using the nonzeros in $A(:, r:s)$, to determine the union of the row structures in the submatrix $(r:n, r:s)$. We use the notation $D(r:n, r:s)$ to denote this structure. It is true that

$$Struct((\hat{L}+U)(r:n, r:s)) \subseteq Struct(D(r:n, r:s)) \subseteq Struct(H(r:n, r:s)) \; . \qquad (1)$$

The search procedure is analogous to (but simpler than) the panel symbolic step (Figure 1, step (1)); now we only want to determine the count for the column $D(r:n, r)$, without the nonzero structure or the topological order of the updates. Then we use the product of $nnz(D(r:n, r))$ and $s-r+1$ to allocate storage for the L supernodes within columns r through s. Since $nnz(L(r:n, r)) \leq nnz(D(r:n, r)) \leq nnz(H(r:n, r))$, the dynamic storage bound so obtained is usually tighter than the static bound.

The storage utilizations for the supernodes in S_L are tabulated in Table 6. The utilization is calculated as the ratio of the actual number of nonzeros in the supernodes of L to the number of nonzeros in the supernodes of H. When collecting this data, the maximum supernode size t was set to 64. For most matrices, the storage utilizations using the static bound by H are quite high; they are often greater than 70% and are over 85% for 14 out of the 21 problems. However, in the static scheme, the storage utilizations for matrices 1, 15 and 21 are only 4%, 11% and 14%, respectively. The dynamic scheme overcomes those low utilizations. For the three matrices above, the utilizations in the dynamic scheme are 68%, 74% and 89%. These percentage utilizations are quite satisfactory. For other problems, the dynamic approaches also result in higher utilizations.

The runtime overhead associated with the dynamic scheme is usually between 2% and 15% on the single processor RS/6000-590. From these experiments, we conclude that the static scheme using H often gives a tight enough storage bound for S_L. For some problems, such as matrices 15 and 21, the dynamic scheme must be employed to achieve better storage utilization. Then the program will suffer from a certain amount of slowdown. Our code tries the static scheme first and switches to the dynamic scheme only if the static scheme requests more space than is available.

5.4 Nonblocking pruning and depth-first search

The idea of symmetric pruning [9, 10] is to use a graph G' with fewer edges than the graph G of L^T to represent the structure of L. Traversing G' gives the same reachable set as traversing G, but is less expensive. As shown in [10], this technique significantly reduces the symbolic factorization time.

In the sequential algorithm, in addition to the adjacency structure for G, there is another adjacency structure to represent the reduced graph G'. For each supernode, since the row indices are the same among the columns, we only store the row indices of the first column of G and the row indices of the last column of G'. (If we use only one adjacency list for each supernode, since pivoting may have reordered the rows so that the pruned and unpruned rows are intermingled in the original row order, it is then necessary to reorder all of L and A to account for it.)

Figure 7 illustrates the storage layout for the adjacency lists of G and G' of a sample matrix. Array `Lsub[*]` stores the row subscripts. `G_ptr[*]` points to the beginning of each supernode in array `Lsub[*]`. `G'_ptr[*]` points to the pruned location of each supernode in array `Lsub[*]`. Using `G_ptr` and `G'_ptr` together can locate the adjacency list for each supernode in G'. This matrix has four supernodes: {1,2}, {3}, {4,5,6}, and {7,8,9,10}. The adjacency lists for G and G' are interleaved by supernodes in the global memory `Lsub[*]`. The storage for the adjacency structure of G' is reclaimed at the end of the factorization.

The pruning procedure works on the adjacency lists for G'. Each adjacency list of a supernode (actually only the last column in the supernode) is pruned at the position of the first symmetric nonzero pair in the factored matrix F, as indicated by the small "·" in the figure. Both panel DFS and column DFS traverse the adjacency structure of G', as given by `G'_ptr[*]` in Figure 7.

In the parallel algorithm, contention occurs when one processor is performing DFS using G''s adjacency list of column j (a READ operation), while another processor is pruning the structure of column j, because pruning will reorder the row indices in the list (a MODIFY operation). There are two possible solutions to avoid this contention. The first solution is to associate one mutually exclusive (mutex) lock with each adjacency list of G'. A processor acquires the lock before it prunes the list and releases the lock thereafter. Similarly, a processor uses the lock when performing DFS on the list. Although the critical section for pruning can be very short, the critical section for DFS may be very long, because the list must be locked until the entire depth-first search starting from all nodes in the list is completed. During this period, all the other processors attempting to prune the list or to traverse the list will be blocked. Therefore this approach may incur too much overhead, and the benefit of pruning may be completely offset by the cost of locking.

We now describe a better algorithm that is free from locking. We will use both graphs G' and G to facilitate the depth-first search. Recall that each adjacency list is pruned only *once* throughout

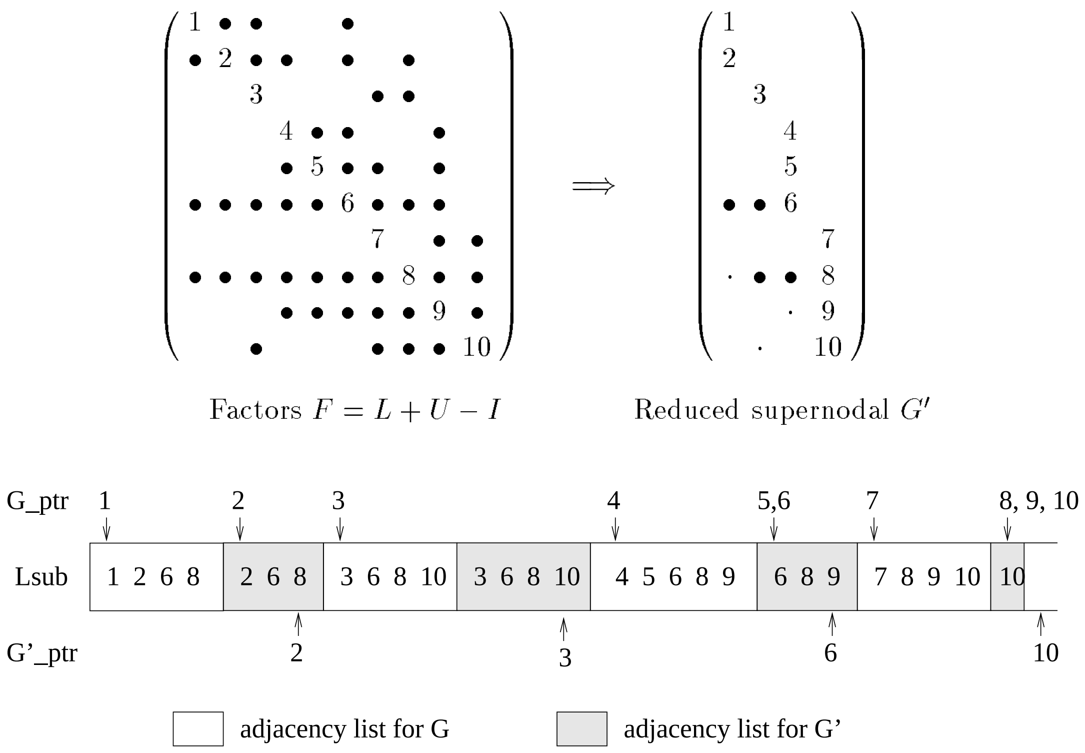


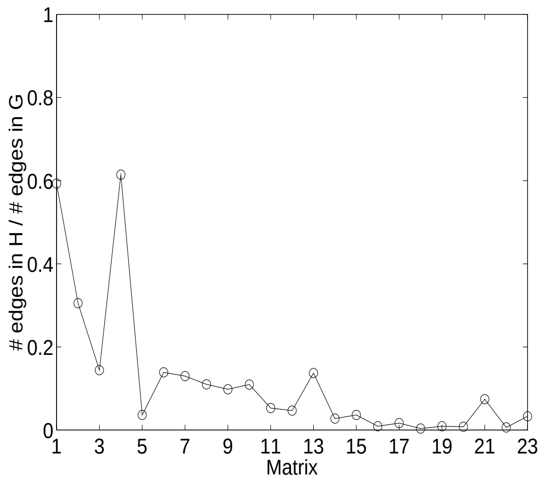
Figure 7: Storage layout for the adjacency structures of G and G'.

the factorization. We will associate with each list a status bit indicating whether it is pruned or not. Once a list is pruned, all the subsequent traversals on the list involve only READ operations, and hence do not require locking. If the search procedure reaches a list of G' that has not yet been pruned, we will direct the search procedure to traverse the list of the corresponding column in G, rather than G'. So, when the search algorithm reaches column j, it does the following:

```
if column $j$ has been pruned then
    continue search from nodes in the $G'$-list of column $j$;
else
    continue search from nodes in the $G$-list of column $j$;
endif
```

This scheme prevents us from using one minor working-storage optimization from the sequential algorithm: sequential SuperLU uses separate G and G' lists for supernodes with two or more columns, but overlaps the lists for singleton supernodes. The parallel code must use both lists for every supernode.

Since G' is generally a subgraph of G, the depth-first searches in the parallel code may traverse more edges than those in the sequential code. This is because in the parallel algorithm, a supernode may be pruned later than in the sequential algorithm. However, because of the effectiveness of symmetric reduction, very often the search still uses the pruned list in G'. So it is likely that the time spent in the occasional extra search in the G-lists is much less than that when using the locking mechanism. Figure 8 shows the relative size of the reduced supernodal graph H, and Figure 9 shows the fraction of searches that use the G'-lists. The numbers in both figures are collected on a single processor Alpha 21164.



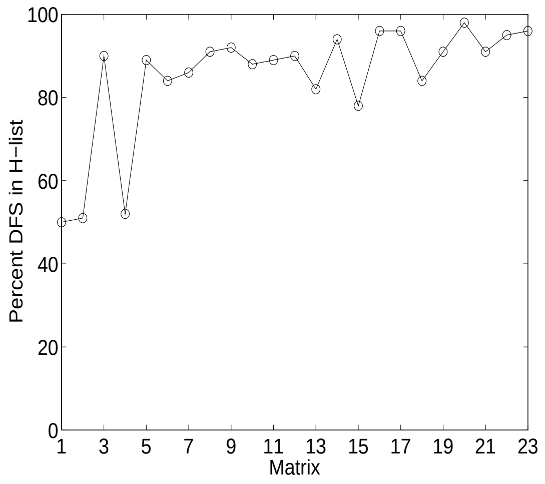


Figure 8: Number of edges in G' versus number of edges in G.

Figure 9: Percent of the depth-first search in adjacency lists in G'.

6 The asynchronous scheduling algorithm

Having described the parallel strategies, we are now in a position to describe the parallel factorization algorithm. Several methods have been proposed to perform sparse Cholesky factorization [12, 23, 27] and sparse *LU* factorization [2, 16, 19] on shared memory machines. A common practice is to organize the program as a self-scheduling loop, interacting with a global pool of tasks that are ready to be executed. Each processor repeatedly takes a task from the pool, executes it, and puts new ready task(s) in the pool. This pool-of-tasks approach has the merit of balancing work load automatically even for tasks with large variance in granularity. There is no notion of *ownership* of tasks or submatrices by processors – the assignment of tasks to processors is completely dynamic, depending on the execution speed of the individual processors. Our scheduling algorithm employs this model as well. This is in contrast to some implementations of sparse Cholesky, which can schedule work to processors carefully and cheaply ahead of time [22]. The dynamic nature of partial pivoting prevents us from doing this.

Our scheduling approach used some techniques from the parallel column-oriented algorithm developed by Gilbert [19]. Figure 10 sketches the top level scheduling loop. Each processor executes this loop until its termination criterion is met, that is, until all panels have been factorized.

The parallel algorithm maintains a central priority queue of tasks (panels), that are ready to be executed by any free processor. The content of this task queue can be accessed and altered by any processor. At any moment during the elimination, a panel is tagged with a certain state, such as READY, BUSY, or DONE. Every processor repeatedly asks the scheduler (at line 4) for a panel task in the queue. The `Scheduler()` routine implements a priority-based scheduling policy described below. The input argument *oldpanel* denotes the panel that was just finished by this processor. The output argument *newpanel* is a newly selected panel to be factorized by this processor. The selection preference is as follows:

(1) The scheduler first checks whether all the children of *oldpanel*'s parent panel, say *parent*, are DONE. If so, *parent* now becomes a new leaf and is immediately assigned to *newpanel* on the same processor.

(2) If *parent* still has unfinished children, the scheduler next attempts to take from the queue a

```
Slave_worker()
1.  newpanel = NULL;
2.  while ( there are more panels ) do
3.      oldpanel := newpanel;
4.      Scheduler( oldpanel, newpanel, queue );
5.      if ( newpanel is a relaxed supernode ) then
6.          relaxed_supernode_factor( newpanel );
7.      else
8.          panel_symbolic_factor( newpanel );
9.              • Determine which supernodes will update panel newpanel;
10.             • Skip all BUSY panels/supernodes;
11.         panel_numeric_factor( newpanel );
12.             • Accumulate updates from the DONE supernodes, updating newpanel;
13.             • Wait for the BUSY supernodes to become DONE, then predict
                  new fills and accumulate more updates to newpanel;
14.         inner_factorization( newpanel ); /* independent from other processors */
15.             • Supernode-column update within the panel;
16.             • Row pivoting;
17.             • Detect supernode boundary;
18.             • Symmetric structure pruning;
19.     end if;
20. end while;
```

Figure 10: The parallel scheduling loop to be executed on each processor.

panel which can be computed without pipelining, that is, a leaf panel.

(3) If no more leaf panels exist, the scheduler will take a panel that has some BUSY descendant panels currently being worked by other processors. Then the new panel must be computed by this processor in a pipelined fashion.

One might argue that (1) and (2) should be reversed in priority. Choosing to eliminate the immediately available parent first is primarily concerned with locality of reference. Since a just-finished panel is likely to update its parent or other ancestors in the etree, it is advantageous to schedule its parent and other ancestors on the same processor.

To implement the above priority scheme, the task queue is initialized with the leaf panels, that is, the relaxed supernodes, which are marked as READY. Later on, `Scheduler()` may add more panels at the tail of the queue. This happens when all the children of *newpanel*'s parent, *parent*, are BUSY; *parent* is then enqueued and is marked as eligible for pipelining. By rule (1), some panel in the middle of the queue may be taken when all its children are DONE. This may happen even before all the initial leaf panels are finished. All the intermediate leaf panels are taken in this way. By rules (2) and (3), `Scheduler()` removes tasks from the head of the queue.

It is worth noting that the executions of different processors are completely asynchronous. There is no global barrier; the only synchronization occurs at line 13 in Figure 10, where a processor stalls when it waits for some BUSY updating supernode to finish. As soon as this BUSY supernode is finished, all the processors waiting on this supernode are awakened to proceed. This type of synchronization is commonly referred to as *event notification*. Since the newly finished supernode may produce new fills to the waiting panels, the symbolic mechanism is needed to discover and accommodate these new fills.

7 Parallel performance

We now evaluate the performance of the algorithm. The organization of this section is as follows. Section 7.1 summarizes the observed speedups on various platforms. The speedup is compared to that of serial SuperLU. Section 7.2 quantifies parallel overhead and their impact on performance. Section 7.3 gives the statistics of load balance. Section 7.4 studies the space efficiency of the algorithm.

7.1 Speedup summary

Figures 11 through 15 report the speedups of the parallel algorithm on the five platforms, with number of threads "P" varied. Because of memory limits we could not test all problems on the SPARCcenter 2000. The speedup is measured against the best sequential runtime achieved by SuperLU on a single processor of each parallel machine.

In each figure, the bottom curve labeled "$P = 1$" illustrates the overhead in the parallel code when compared to the serial SuperLU, using the same blocking parameters. The structure of the parallel code, when run on a single processor, does not differ much from sequential SuperLU, except that a global task queue and various locks are involved. The extra work in the parallel code is purely integer arithmetic. In order to achieve a higher degree of concurrency, the panel size (w) and maximum size of a supernode ($maxsup$) for "$P > 1$" are set smaller than those used for "$P = 1$".[4]

[4]Both w and $maxsup$ denote the size in number of columns.

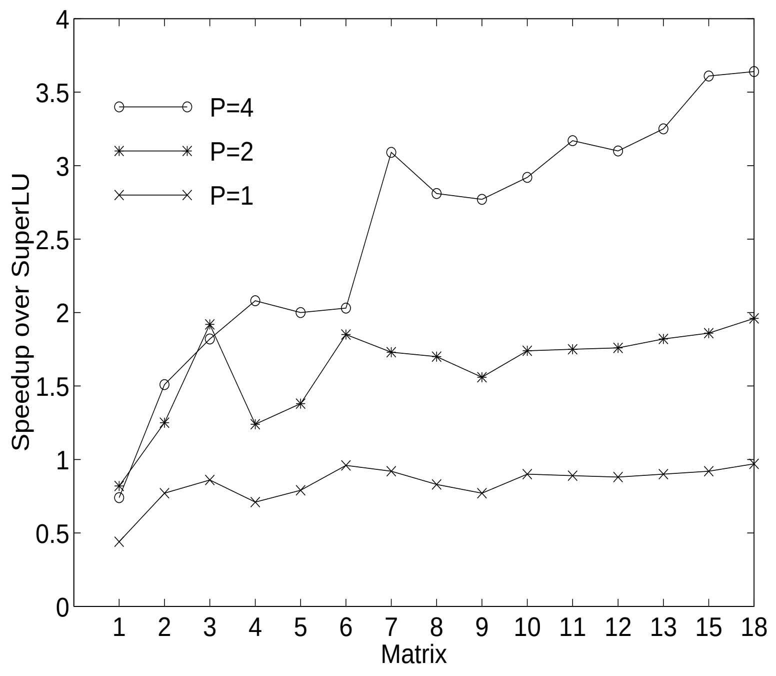


Figure 11: Speedup on a 4-CPU Sun SPARCcenter 2000.

We also tabulate these speedup figures in the Appendix (Tables 13 through 17), where the last two columns in each table show the factorization time and Megaflop rate, respectively, corresponding to the largest number of processors used.

7.2 Impact of overhead on parallel efficiency

The parallel algorithm experiences some overhead, which mainly comes from three sources: the reduced per-processor efficiency due to smaller granularity of unit tasks, accessing critical sections via locks, and orchestrating the dependent tasks via event notification. The purpose of this section is to understand how much time is spent in each part of the algorithm and explain the speedups we saw in Section 7.1.

7.2.1 Decreased per-processor performance due to smaller blocking

The first overhead is due to the necessity to reduce the blocking parameters in order to achieve more concurrency. Recall that two blocking parameters affect performance: panel size (w) and maximum size of a supernode ($maxsup$). For better per-processor performance, we prefer larger values. On the other hand, the large granularity of unit tasks limits the degree of concurrency.

On the Cray J90, this trade-off is not so important, because $w = 1$ is good for the sequential algorithm. We therefore also use $w = 1$ in the parallel algorithm. When varying the value of $maxsup$, we find that performance is quite robust in the range between 16 and 64.

On the Power Challenge and AlphaServer 8400, we observe more dramatic differences with varied blockings. Figure 16 and 17 illustrate this loss of efficiency for several large problems on single processors of the two machines. In this experiment, the parallel code is run on single processors with two different settings of w and $maxsup$. Figure 16 shows, on a single processor Power Challenge, the ratio of the runtime with the best blocking for 1 CPU ($w = 24, maxsup = 64$)

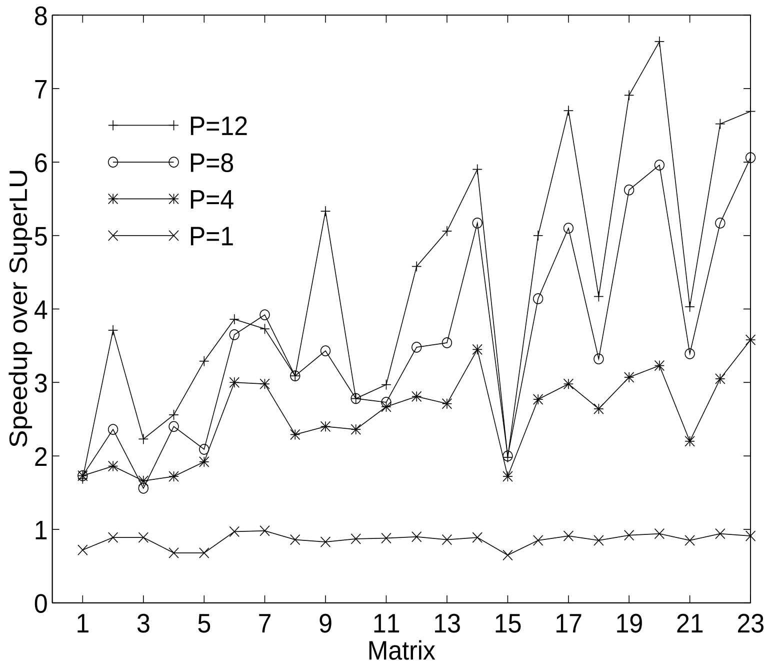


Figure 12: Speedup on a 12-CPU SGI Power Challenge.

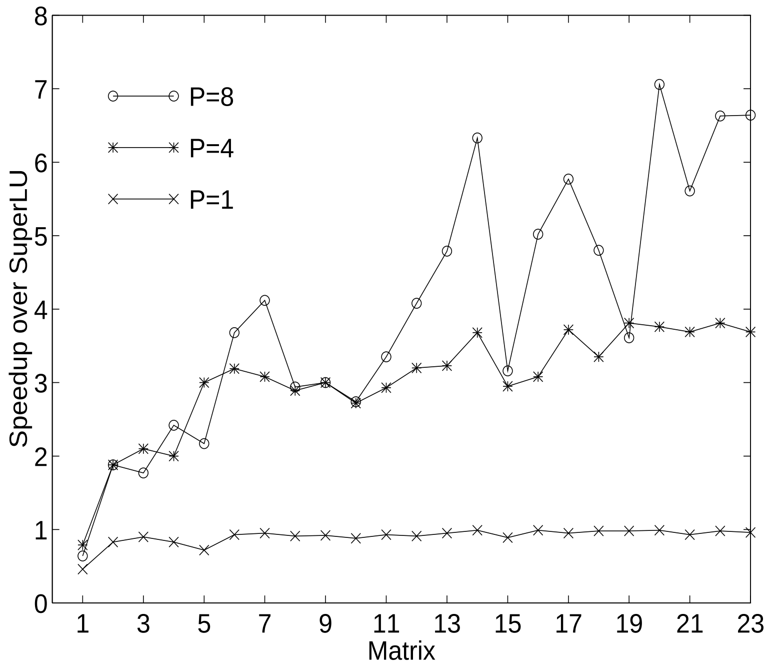


Figure 13: Speedup on a 8-CPU DEC AlphaServer 8400.

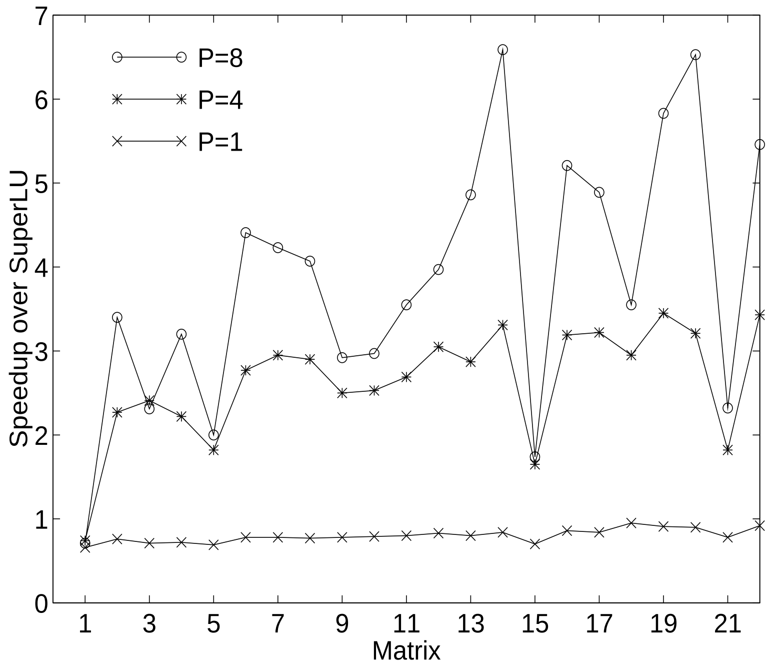


Figure 14: Speedup on a 8-CPU Cray C90.

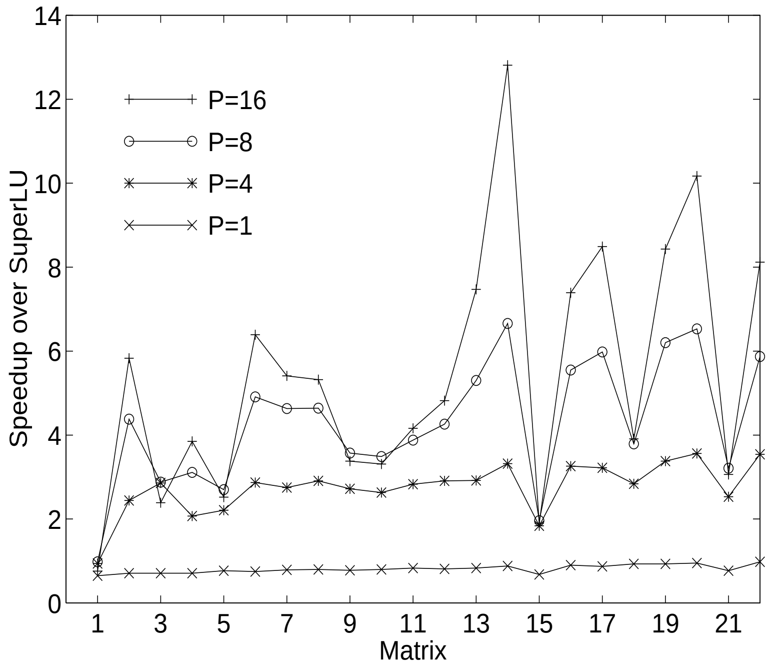


Figure 15: Speedup on a 16-CPU Cray J90.

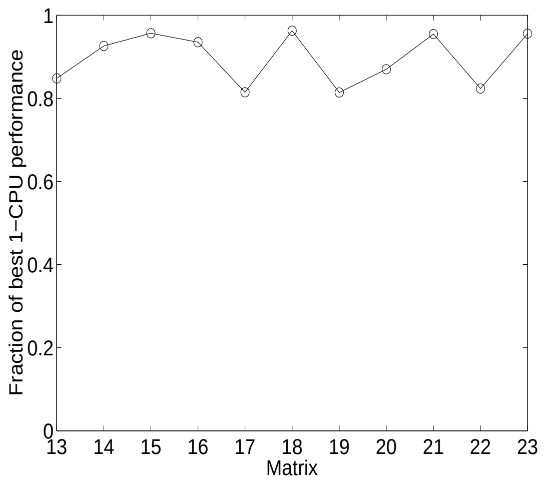


Figure 16: $\frac{T_s}{T_s'}$ for serial SuperLU on 1-CPU Power Challenge.

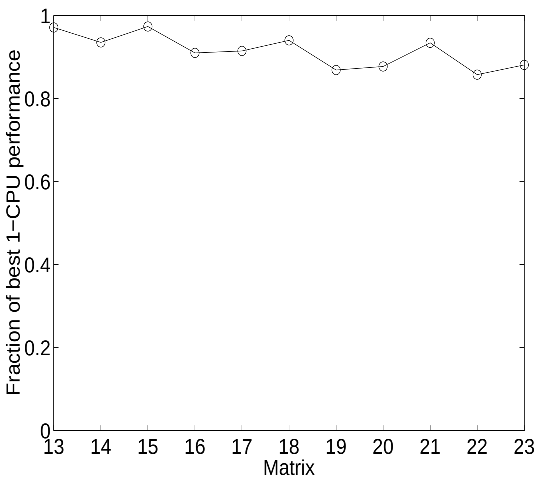


Figure 17: $\frac{T_s}{T_s'}$ for serial SuperLU on 1-CPU AlphaServer 8400.

to the runtime with the best blocking for 12 CPUs ($w = 12, maxsup = 48$). Figure 17 shows the analogous ratio for the 8-CPU AlphaServer 8400. On the Power Challenge, the blocking used for best parallel performance achieves only 81% uniprocessor efficiency for matrices 17 and 19. The corresponding lowest number on the AlphaServer 8400 is 86% for matrix 22.

7.2.2 Accessing critical sections

Several places in the program must be protected by mutual exclusion. In Table 7, we roughly count the number of times the program acquires and relinquishes various locks. Note that the total number of lockings performed is independent of the number of processors. Since we want to allow different processors to enter different critical sections simultaneously, we use five mutex variables to guard the five critical regions.

To see how much cost is associated with locking, in Table 8 we measured the time it takes to acquire and relinquish a lock on several platforms, with different numbers of threads P. The figure in the parenthesis is the number of clock cycles. In this small benchmark code, the critical section is simply one statement, to increment a counter. The locking and unlocking are placed around this statement. The measurement is done in a tight loop with many iterations. When there is more than one thread, the time increases slightly, but not linearly in the number of threads.

The uniprocessor slowdown is partly due to the overhead incurred by using these locks, when there are no other processors competing for the locks. By multiplying the time for a single lock/unlock in Table 8 by the number of the lockings performed in Table 7, we can estimate the locking overhead. As a concrete example, let us consider a medium size matrix 13, on a single processor Cray J90. Since the sequential code performance is 26 Mflops, each lock/unlock is equivalent to roughly 69 floating-point operations. When the factorization is performed with panel size $w = 1$, the total number of lock acquisitions is 237004, which, when multiplied by 2.67 microseconds, results in about 0.64 seconds. This is less than 3% of the entire factorization time (24.85 seconds). We observe that this percentage is typical for large matrices (also the bottom curve in Figure 18). The locking overhead also varies with machines. For example, it is higher on the Cray J90 than on the Power Challenge or the AlphaServer 8400.

Critical section	Counts
call `Scheduler()`	number of panels (approx.)*
allocate storage for row indices of L (`Lsub`)	number of supernodes
allocate storage for L supernodes (S_L)	number of supernodes
allocate storage for a column of U (`Usub/Uval`)	number of columns
increment supernode number `nsuper`	number of supernodes

Table 7: Number of lockings performed.
* Here we assume that `Scheduler()` returns a new panel upon each call.

Machine	$P = 1$	$P = 4$	P=8
SPARCcenter 2000	1.63 (82)	4.34 (217)	4.36 (218)
Power Challenge	1.13 (102)	1.98 (179)	2.02 (182)
AlphaServer 8400	0.98 (294)	2.26 (678)	2.71 (814)
Cray C90	1.34 (323)	1.09 (261)	1.40 (336)
Cray J90	2.67 (267)	4.17 (417)	4.42 (442)

Table 8: Time in microseconds (cycles) to perform a single lock and unlock.

7.2.3 Coordinating dependent tasks

The third source of overhead is due to insufficient parallelism in the pipelined executions of the dependent panels. Dependent panels are those that have an ancestor-descendant relation in the column etree. When a processor factoring a panel needs an update from a BUSY descendant panel, this processor simply spins, waiting for that panel to finish, as shown at line 13 in the scheduling loop of Figure 10. During the spin wait the processor does nothing useful. The total amount of spin wait time observed is significant in some cases, especially with larger numbers of processors. For example, for matrix 16, on the 12-CPU Power Challenge, about 40% of the parallel runtime is spent spinning. The corresponding number for the dense matrix is about 58%. The dense matrix is the worst one, because the factorization of all panels must be carried out in pipelined fashion.

Figure 18 depicts the locking overhead (Section 7.2.2) and the spinning due to dependencies on the 8-CPU Cray J90. The locking overhead also includes the possible contention from the 8 processors. In this figure, we also plot the inefficiency (i.e., $1-$ *efficiency*) of the parallel algorithm. For most matrices, the spinning overhead due to dependencies is much higher than the overhead from lock acquisition.

7.2.4 Putting all overheads together

In this subsection we evaluate the effect of the combined overheads on the parallel efficiency. In summary, the overheads include

Overhead (1): reduced uniprocessor performance due to smaller blocking

Overhead (2): accessing critical sections

Overhead (3): idle time (from spin wait in the panel pipeline and in the top-level scheduling loop)

Overhead (1) only affects uniprocessor performance. Overhead (2) decreases both uniprocessor performance of the parallel code and parallel performance. Compared with the serial execution, the parallel execution experiences more contention for locks. But Table 8 and Figure 18 indicate that runtime does not increase significantly because of contention. Therefore, we may model (2)

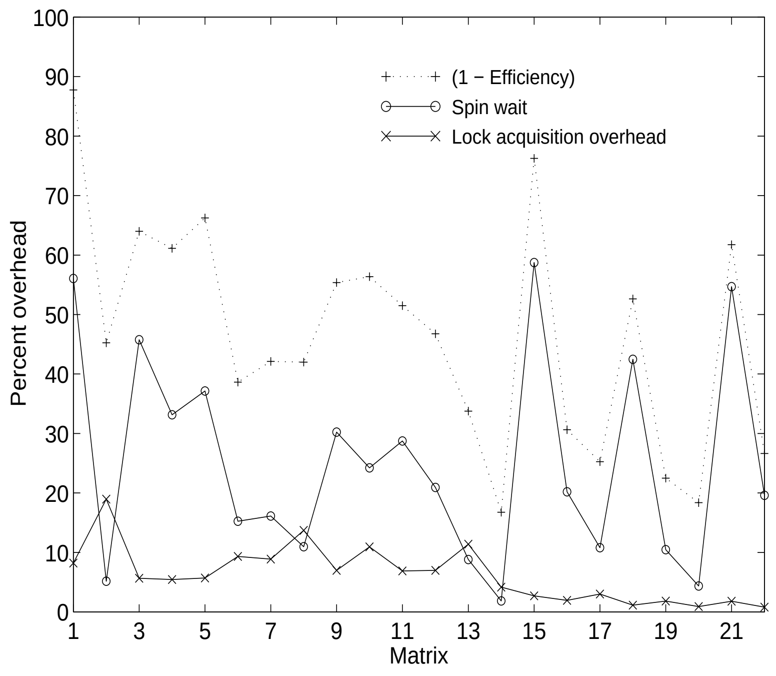


Figure 18: Parallel overhead in percent on an 8-CPU Cray J90.

as only adding overhead to the uniprocessor execution. Overhead (3) exists only in the parallel computations.

We now analyze the relations of the various times defined in Table 5. All the times are measured independently. In particular, T_I is obtained by timing two kinds of idle periods on each processor and summing over all processors: one is the spin wait in the panel update pipeline, and the other is when a processor calls `Scheduler()` (line 4 in Figure 10) and fails to get a panel from the scheduler. We found that, for the test matrices and the numbers of processors being considered, failure from the scheduler rarely occurs. So most of the idle time is due to pipeline waiting. The following relation holds for the parallel runtime:[5]

$$P\,T_P \approx T_1 + T_I\ . \qquad (2)$$

We now compute the observed efficiency (E_{actual}) as follows:

$$E_{actual} = \frac{T_s}{P\,T_P}\ . \qquad (3)$$

Since T_P, T_1, and T_I are obtained from different runs of the program, the left-hand side and the right-hand side of Equation (2) may not match well. For the purpose of checking, we also compute the following quantity:

$$E_{check} = \frac{T_s}{T_1 + T_I}\ . \qquad (4)$$

The closeness of E_{check} to E_{actual} indicates the accuracy of the timings, see Tables 9 and 10.

In order to understand the impact of the overheads discussed in previous subsections on the parallel efficiency, we introduce two parameters α_1 and α_p, which are calculated based on T_s, T_1,

[5]In the absence of errors in the individual time measurement, equality should hold.

	Matrix	Efficiency		Overhead			
		E_{actual}	E_{est}	α_1	α_p	E_{check}	B
13	SHYY161	.47	.63	.17	.23	.59	.66
14	GOODWIN	.80	.79	.12	.10	.79	.97
15	VENKAT01	.12	.17	.32	.74	.13	.99
16	INACCURA	.46	.48	.10	.46	.47	.97
17	AF23560	.53	.57	.13	.34	.55	.93
18	DENSE1000	.25	.30	.07	.67	.26	.99
19	RAEFSKY3	.53	.58	.07	.37	.56	.96
20	EX11	.64	.73	.05	.23	.70	.98
21	WANG3	.19	.22	.23	.71	.19	.99
22	RAEFSKY4	.51	.55	.02	.43	.53	.97

Table 9: Efficiencies and overheads on a 16-CPU Cray J90.

T_P, and T_I as follows:

$$\alpha_1 = \frac{T_1 - T_s}{T_1} = 1 - \frac{T_s}{T_1} . \tag{5}$$

$$\alpha_p = \frac{T_I}{P\,T_P} . \tag{6}$$

Both α_1 and α_p are in the range $[0, 1)$; α_1 measures the overhead that degrades the uniprocessor performance, while α_p measures the overhead in the parallel execution. The smaller α_1 and α_2 are, the more efficient is the parallel algorithm. Since

$$(1 - \alpha_1) \cdot (1 - \alpha_p) = \frac{T_s}{T_1} \cdot \frac{P\,T_P - T_I}{P\,T_P} \approx \frac{T_s}{T_1} \cdot \frac{T_1}{P\,T_P} = E_{actual} ,$$

we can use

$$E_{est} = (1 - \alpha_1) \cdot (1 - \alpha_p) . \tag{7}$$

as an estimate for the actual efficiency.

In Tables 9 and 10, we report E_{actual}, E_{est}, α_1, α_p and E_{check} obtained on the two parallel machines.

Cray J90

In the first two columns of Table 9, we compare the estimated efficiency E_{est} in Equation (7) with the actually observed efficiency E_{actual} in Equation (3). The estimated and observed efficiencies are very close. Their differences are mostly within 5%, except for matrices 13 and 20 which have 15% and 9% difference, respectively. For these two matrices, E_{actual} and E_{check} differ significantly, indicating that some overhead is not reflected in T_1 or T_I.

As mentioned in Section 7.2.1, the uniprocessor performance on the J90 does not degrade much with smaller *maxsup*, that is, Overhead (1) does not exist ($T_s' = T_s$). Therefore, $\frac{T_s}{T_1}$ can be from the bottom curve in Figure 15. We gathered the statistics for α_p and B on 16 processors, as shown in Table 9. For most problems, the pipeline spin waiting, as measured by α_p, is the primary cause of inefficiency. This is particularly evident for matrices 15, 18 and 21, for which 74%, 67% and 71% of the time processors are idle, respectively. This explains the low speedups achieved for these matrices.

	Matrix	Efficiency		Overhead			
		E_{actual}	E_{est}	α_1	α_p	E_{check}	B
13	SHYY161	.42	.58	.27	.20	.54	.70
14	GOODWIN	.49	.61	.18	.25	.57	.87
15	VENKAT01	.17	.27	.38	.56	.20	.91
16	INACCURA	.42	.47	.21	.40	.45	.88
17	AF23560	.56	.59	.26	.20	.58	.93
18	DENSE1000	.35	.34	.18	.58	.35	.92
19	RAEFSKY3	.58	.63	.25	.16	.62	.95
20	EX11	.64	.74	.18	.09	.73	.98
21	WANG3	.34	.38	.19	.52	.36	.93
22	RAEFSKY4	.54	.65	.23	.15	.63	.95
23	VAVASIS3	.56	.71	.14	.17	.68	.97

Table 10: Efficiencies and overheads on a 12-CPU Power Challenge.

Power Challenge

On a cache-based machine, the uniprocessor performance loss of the parallel code is a combination of performing lockings and less efficient cache utilization. Therefore, $\frac{T_s}{T_1}$ equals the product of the numbers from the bottom curve in Figure 12 ($\frac{T_s'}{T_1}$) and the numbers from Figure 16 ($\frac{T_s}{T_s'}$). Compared to the J90, we observe that α_1 is much larger, because the cache plays an important role on the Power Challenge. In fact, for matrices 13, 17, 19, 20 and 22, uniprocessor performance loss is more severe than the parallel overhead, α_p.

Again, for matrices 15, 18 and 21, the spin wait time is the major bottleneck; the processors are idle more than 50% of the time. We found that E_{est} and E_{actual} did not match as well as they did on the J90. For matrices 13, 14, and 23, the gaps are 16%, 12%, and 15%, respectively. The corresponding gaps between E_{check} and E_{actual} are large as well. This again indicates some overhead not accounted for in T_1 or T_I. We need further study to fully understand why this is.

7.3 Load balance

As mentioned earlier, our dynamic scheduling approach can automatically balance the workload. One way to measure the load balance is as follows. Let f_i denote the number of floating-point operations performed on processor i, and P denote the number of processors. We define the load balance B as

$$B = \frac{\sum_i (f_i)}{P \ \max_i (f_i)} \ . \tag{8}$$

In words, B equals the average work load divided by the maximum work load. It is readily seen that $0 < B \leq 1$, and higher B indicates better load balance. If load imbalance is the sole overhead in a parallel program, the parallel execution time is simply the execution time of the slowest processor, whose work load is highest.

We should note that the load balance measured by Equation (8) is an accurate measure of work distribution only under the condition that each floating-point operation takes the same amount of time. This is not the case in practice, but the large values of B shown in the last columns of Tables 9 and 10 still show that good load balance was achieved in terms of flop counts. Matrix 13 is an exception.

	Matrix	LU storage (MB)	Fraction of $P = 1$	LU storage $P = 8$
1	MEMPLUS	16.27	.23	1.51
2	GEMAT11	1.15	.89	5.92
3	RDIST1	3.70	.23	1.54
4	ORANI678	4.77	.11	.73
5	MCFE	0.88	.18	1.26
6	LNSP3937	4.93	.16	1.10
7	LNS_3937	7.04	.12	.77
8	SHERMAN5	2.75	.25	1.66
9	JPWH_991	1.58	.13	.88
10	SHERMAN3	4.68	.22	1.47
11	ORSREG_1	4.23	.11	.72
12	SAYLR4	6.98	.10	.70
13	SHYY161	80.01	.19	1.31
14	GOODWIN	34.25	.04	.30
15	VENKAT01	566.09	.02	.15
16	INACCURA	106.06	.03	.21
17	AF23560	145.02	.03	.22
18	DENSE1000	9.90	.02	.14
19	RAEFSKY3	183.65	.02	.16
20	EX11	277.59	.01	.08
21	WANG3	459.14	.01	.07
22	RAEFSKY4	271.28	.02	.10
23	VAVASIS3	521.75	.02	.11

Table 11: Working storage requirement as compared with the storage needed for L and U. The blocking parameter settings are: $w = 8$, $t = 100$, and $b = 200$.

7.4 Working storage requirement

The parallel algorithm may require more working storage than the sequential one. Multiple threads share heap storage, static storage, and code, all residing in main memory. Each thread, upon execution, is allocated a private stack and has its own register set. Our program does not use many stack variables, so the stack size for each thread need not be very large. All working storage is allocated via `malloc()` from the heap. The working storage consists of two parts, where one part is shared among all threads, and another part is local to each thread. The shared working storage is mainly used to facilitate the central scheduling activity and memory management. It includes:

- one integer array of size p used as the task queue, where p is the total number of panels;
- one bit vector of size n to mark whether a column is busy;
- four integer arrays of size n to record the status of each panel;
- one integer array of size n to record a column's most distant busy column down the etree during pipelining;
- three integer arrays of size n to implement storage layout for supernodes (Section 5.3).

The local working storage used by each thread is very similar to that used by sequential SuperLU, that is, all that is necessary to factorize one single panel. It includes:

- eight integer arrays of size n to perform the panel and column depth-first search;
- one n-by-w integer array to keep track of the position of the first nonzero of each supernodal segment in U;
- one n-by-w integer array to temporarily store the row subscripts of the nonzeros filled in the panel;
- one n-by-w real array used as the SPA.
- one scratch space of size $(t + b) \times w$ to help BLAS calls. See Figure 1 for the definition of t, b and w.

This amount of local storage should be multiplied by P, where P is the number of threads created. Thus the working storage grows affinely with respect to P, and this algorithm, albeit efficient, is hard to scale up from a memory point of view.

To put this in perspective, Table 11 compares the working storage requirement with the actual LU storage. The last two columns report the amount of working storage as a fraction of the total LU storage in Megabytes, for 1 and 8 threads, respectively. It is clear that for $P = 8$, the working storage requirement can be comparable to the LU storage for small problems. For large problems, working storage is typically 10% to 20% of the LU storage. Matrix 13 is exceptionally bad: it is a matrix of medium size for which the required working storage is more than LU storage. Since we would not use multiple processors on the small problems anyway, the overall working storage requirement is quite small.

7.5 A PRAM model to predict optimal speedup

Given a matrix with a fixed column ordering, we want to establish a performance model to estimate the maximum speedup attainable by our algorithm, and indeed to determine the limitations of algorithms based on partitioning a matrix by columns, and using a column as a scheduling unit.

Because of various precedence constraints in the algorithm, some parts of the work must be finished before other parts can start. Thus, the completion time of the parallel algorithm is constrained by the amount of work that must be done *serially*, i.e., the critical path. Our objective here is to give a lower bound on parallel completion time.

In the model we make the following simplifying assumptions: (1) The work only includes floating-point operations, and each floating-point operation takes one unit of time. (2) There is an infinite number of processors. Whenever a task is ready, there will be a free processor to execute this task immediately. (3) Accessing memory and communication are free. (4) We ignore various overheads associated with the actual implementation of the scheduling algorithm and the synchronizations. This model gives an optimistic estimate; therefore, we can use it to prove upper bounds on the performance of the parallel algorithm on a real machine.

The left-looking LU factorization algorithm can be modeled by a data structure called a directed acyclic graph (DAG), in which edges are directed from groups of the etree vertices representing supernodes to groups of the etree vertices representing panels. (Panels and supernodes can overlap in arbitrary fashion.) Each node in the DAG corresponds to the computation of a panel. An edge directed from s to p corresponds to an update of panel p by supernode s. The edges also represent precedence relations between the updating supernodes and the destination panels. Figure 19 illustrates such a DAG for a 10-by-10 matrix.

In presenting our model, we employ the following notation:

$$\begin{pmatrix}
1 & \bullet & & & & & & & & \\
\bullet & 2 & \bullet & & & & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & & & & \bullet & \\
& & & \bullet & & \bullet & \bullet & & & \\
& & & \bullet & \bullet & 6 & & & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & & & & & & & & & \\
& & & \bullet & & \bullet & & \bullet & 9 & \\
& & \bullet & & & & & & & 10
\end{pmatrix}
\Longrightarrow
\begin{pmatrix}
1 & \bullet & & & & & & & & \\
\bullet & 2 & \bullet & & & & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & & & & \bullet & \\
& & & \bullet & 5 & \bullet & & & \bullet & \\
& & & \bullet & \bullet & 6 & \bullet & & & \\
& & & & & & 7 & & \bullet & \bullet \\
& & & \bullet & \bullet & \bullet & \bullet & 8 & \bullet & \bullet \\
\bullet & \bullet & \bullet & & & & \bullet & \bullet & 9 & \bullet \\
& & \bullet & & & & \bullet & \bullet & \bullet & 10
\end{pmatrix}$$

Original A Factors $F = L + U - I$

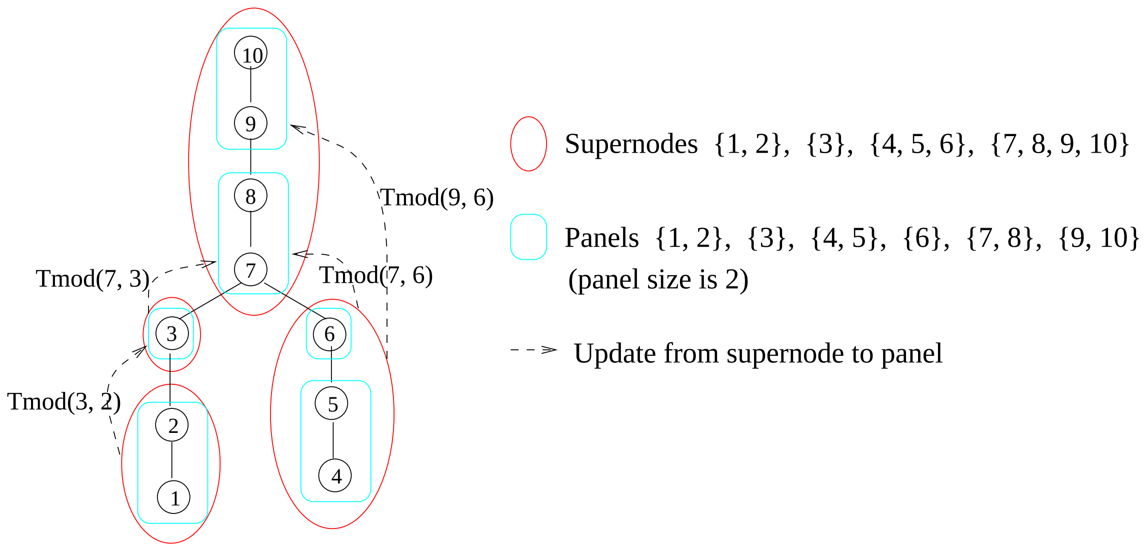

10
9
8
7
3
6
5
4
2
1
Tmod(9, 6)
Tmod(7, 3)
Tmod(7, 6)
Tmod(3, 2)
Supernodes {1, 2}, {3}, {4, 5, 6}, {7, 8, 9, 10}
Panels {1, 2}, {3}, {4, 5}, {6}, {7, 8}, {9, 10}
(panel size is 2)
Update from supernode to panel

Figure 19: An example of computational DAG to model the factorization.

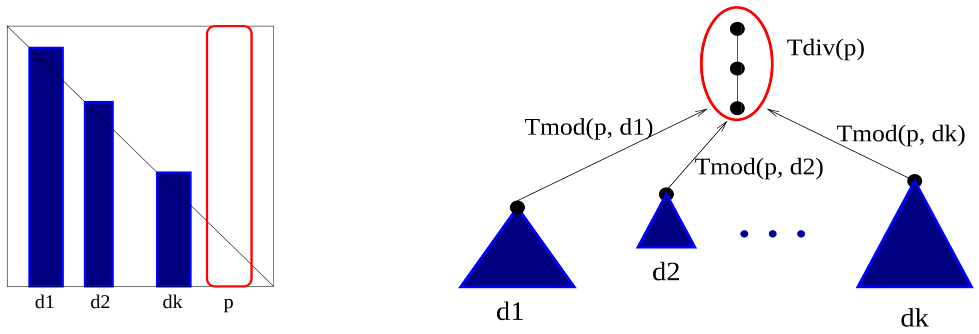


Figure 20: Tasks associated with panel p.

- $T_{mod}(p,d)$:= the task of updating panel p by a descendant supernode d
- $T_{div}(p)$:= the task of performing the inner factorization of panel p
- $tmod(p,d)$:= time taken by task $T_{mod}(p,d)$
- $tdiv(p)$:= time taken by task $T_{div}(p)$
- $EST(p)$:= earliest possible starting time of $T_{div}(p)$
- $EFT(p)$:= earliest possible finishing time of $T_{div}(p)$

All times are expressed in units of floating-point operations. It is clear that for any panel p the following relation holds: $EFT(p) = EST(p) + tdiv(p)$.

According to our scheduling algorithm, each panel task is assigned to a single processor. A panel task for panel p consists of the following two types of subtasks:

$$T_{panel}(p) := \{T_{mod}(p,d) \mid d \in \mathcal{D}\} \cup \{T_{div}(p)\} ,$$

where $\mathcal{D}$ is the set of descendant supernodes that update the destination panel p. Figure 20 shows the part of the DAG associated with a particular panel p.

Each T_{mod} and T_{div} is the indivisible task, and is carried out sequentially on one processor. Clearly, T_{div} cannot start until all the T_{mod}'s have finished. By looking at the precedence relations of these two types of tasks, we can determine the runtime of $T_{panel}(p)$ on processor P. We will try to schedule these tasks as early as possible, in order to derive **the** minimum parallel execution time.

We first look at the tasks associated with one particular panel p, as shown in Figure 20. Suppose there are k descendant supernodes to update panel p, and that all the times $\{EFT(d), d \in \mathcal{D}\}$ have been computed. We schedule the tasks $\{T_{mod}(p,d), d \in \mathcal{D}\}$ to processor P in the order of $T_{mod}(p,1), \ldots, T_{mod}(p,k)$, such that:

$$EFT(1) \leq EFT(2) \leq \ldots \leq EFT(k) .$$

Here, $EFT(i)$ is the finishing time of the last column of supernode i, because a supernode i cannot update any ancestor panel before its last column is completed. We call this scheduling policy Sched-A. Then we can compute $EST(p)$ and $EFT(p)$ as follows.

1. Run the following recurrence to get the completion times of the T_{mod}'s:

```
    $t = 0$;
    for $i = 1$ to $k$
        $t = \max \{ t,\ EFT(i) \} + tmod(i)$;
    endfor;
```

2. Set $EST(p) = t$ and $EFT(p) = t + tdiv(p)$.

Now we will give an informal argument for the optimality of the parallel runtime resulting from Sched-A.

Theorem 3 *For panel p, scheduling the T_{mod}'s by Sched-A gives the shortest completion time.*

Proof: Processor P requires at least $\sum_{i=1}^{k} tmod(p, i)$ units of time to finish all the updates to panel p. Now suppose another scheduling strategy Sched-B starts with a task $T_{mod}(p, i), i \neq 1$. Due to the precedence constraint, $T_{mod}(p, i)$ cannot start until after time $EFT(i)$ ($\geq EFT(1)$). That means processor P will be idle during the period of $LAG := EFT(i) - EFT(1)$. Thus the amount of time to finish all the T_{mod} s will be at least $LAG + \sum_{i=1}^{k} tmod(p, i)$.

On the other hand, in Sched-A, at least some $T_{mod}(p, j), j < i$ have been scheduled in the time period LAG. Hence the amount of work left after time $EFT(i)$ is less than the work left when using Sched-B. Sched-A will give shorter finishing time than Sched-B. □

We are now ready to simulate parallel computation for the whole factorization. To begin with, the ESTs of the leaf panels in the column etree are initialized to zero. Various times can be computed successively from the bottom of the etree to the top. By applying the argument above inductively to all the panels in the DAG, with leaf panels as the basis, we can show that EFT(root panel) gives the minimum execution time. The (predicted) optimal speedup can then be computed by

$$\text{Predicted speedup} = \frac{\text{Total flops}}{EFT(\text{root panel})} .$$

There are several points worth noting in this model. First, because of numerical pivoting, we do not know the computational DAG in advance of the factorization; rather, the DAG is built incrementally as the factorization proceeds. Also, the floating-point operations associated with all the tasks are calculated on the fly. So this model gives an *a posteriori* estimate. Secondly, for each panel computation, the scheduling method of Sched-A requires sorting the EFT's of all the descendant supernodes that will update this panel. The cost associated with this sorting is prohibitively high, and so this method cannot be used to schedule panel updates in practice. Nevertheless, this gives us an upper bound on the theoretically attainable speedup.

In our algorithm, two parameters control task granularity: The panel size w determines the amount of work in a T_{div} task, and both w and the maximum supernode size $maxsup$ determine the amount of work in a T_{mod} task. Any large supernode of size exceeding $maxsup$ (such as in a dense matrix) is divided into smaller ones so that they fit in cache.

Table 12 reports the predicted speedups when varying w and $maxsup$. For a fixed value of $maxsup$, the simulated speedups decrease with increasing w. For sequential SuperLU we find empirically that the best choice for w is between 8 and 16, depending on matrices and architectures. In the parallel setting, a smaller w, say between 4 and 8, seems to give the best overall performance. This embodies an interesting trade-off between available concurrency and per-processor efficiency.

We now compare the results when fixing w but varying $maxsup$. In relatively sparser matrices, such as matrices $1 - 10$, the actual sizes of supernodes may be much smaller than $maxsup$. The

		$maxsup = 32$			$maxsup = 64$			
	Matrix	$w = 4$	$w = 8$	$w = 16$	$w = 4$	$w = 8$	$w = 16$	$height/n$
1	MEMPLUS	4.8	3.6	2.8	2.9	2.5	2.1	0.95
2	GEMAT11	7.3	5.3	4.1	6.4	4.9	3.6	0.06
3	RDIST1	4.6	3.2	2.1	4.6	3.2	2.1	0.99
4	ORANI678	42.2	28.4	16.6	42.2	28.4	16.6	0.64
5	MCFE	6.6	4.3	2.6	6.6	4.3	2.6	0.67
6	LNSP3937	23.2	15.4	9.7	23.2	15.4	9.7	0.25
7	LNS_3937	24.1	15.8	9.6	22.9	15.3	9.6	0.27
8	SHERMAN5	15.8	11.4	7.5	14.0	10.7	7.2	0.20
9	JPWH_991	13.4	9.7	6.4	11.3	8.3	6.0	0.46
10	SHERMAN3	12.7	9.7	7.0	8.2	6.9	5.5	0.20
11	ORSREG_1	14.4	11.0	7.5	9.2	7.8	5.9	0.34
12	SAYLR4	19.8	16.1	11.0	13.1	11.4	8.6	0.29
13	SHYY161	47.9	36.2	24.1	28.1	23.8	18.1	0.04
14	GOODWIN	97.4	71.3	43.6	83.4	63.4	40.1	0.19
15	VENKAT01	22.0	20.2	17.0	14.3	14.2	13.1	0.73
16	INACCURA	62.6	43.5	26.0	44.5	33.6	22.2	0.45
17	AF23560	70.9	55.3	37.2	41.4	35.7	27.4	0.20
18	DENSE1000	33.1	23.7	18.4	18.2	14.9	12.7	1.00
19	RAEFSKY3	140.2	110.6	80.8	80.4	69.6	56.5	0.21
20	EX11	106.7	83.5	58.2	61.6	53.2	41.7	0.35
21	WANG3	57.6	43.4	29.4	34.3	28.9	22.1	0.94
22	RAEFSKY4	99.1	77.1	52.0	56.3	48.5	37.3	0.33
23	VAVASIS3	176.5	133.9	90.7	106.2	89.5	68.2	0.18

Table 12: Optimal speedup predicted by the model, and the column etree height.

performance for such matrices are not so sensitive to $maxsup$. However, for larger and denser matrices, larger value of $maxsup$ results in poorer speedup.

Finally we note that the speedups for small matrices are very low, even with small values of w and $maxsup$. Fortunately, for large matrices such as $13 - 21$, the predicted speedups are greater than 20 when $w = 8$ and $maxsup = 32$. These matrices perform more than one billion floating-point operations in the factorization. It is these matrices that require parallel processing power. The current column-oriented algorithm is well suited for most of the commercially popular SMPs, because the number of processors on these systems is usually below 20.

The height of the column etree can also be used as a crude prediction of the parallel performance. The height of a node i is defined as

$$height(i) = \begin{cases} 0, & \text{if } i \text{ is a leaf node} \\ 1 + \max\{\, height(j) \mid j \in child(i)\} & \text{otherwise} \end{cases}$$

The height of the etree is the height of the root, which represents the longes path in the etree. The computation of all the nodes along this path must be performed in succession. Therefore, the length of the critical path constrains performance. The last column of Table 12 shows the height of the etree over total numbers of nodes n in the etree. The larger $height/n$ is, the larger the fraction of panels will be factorized in pipelined manner, resulting in poor parallelism and more synchronizations. For example, $height/n$ for matrices 1, 3, 15 and 21 is rather large. This is consistent with the relatively lower predicted speedups. However, we must note that the etree height alone is not an accurate measure of parallelism. For example, both dense matrix (18) and a tridiagonal matrix have $height/n = 1.00$, but the former possesses much more concurrency than the later.

The actual speedups achieved are much lower than the upper bounds predicted by the PRAM model (Figures 11 through 15). This is because the model does not capture the details of the machines and the implementation, such as cache behavior, synchronization, etc. However, we do see a similar shape of speedup curves. For example, the model predicts that matrices 15 and 21 have lower speedups compared with the other large matrices. In reality, these two matrices perform worse than the others. The poor performance is primarily due to two factors: (1) The column etree is tall, and contains substantial false dependencies. (2) The dynamic algorithm is needed to allocate memory for the supernodes (Section 5.3), because the static upper bound on the supernodes storage is too large for these two problems (Table 6).

8 Conclusions

We have designed and implemented a parallel algorithm for shared memory multiprocessors of modest size. The efficiency of the algorithm has been demonstrated on several parallel machines. Figure 21 shows the speedups on 8 processors of three parallel machines. Figures 22 through 25 summarize the factorization rate in Megaflops for six large matrices, with increasing number of processors. We believe these large problems are the primary candidates to be solved on parallel machines. In fact, the largest one in our test suite takes a little more than 0.5 GBytes memory, far less than most parallel machines offer. Our algorithm is expected to work well for even larger problems.

For a realistic problem arising from a 3-D flow calculation (matrix 20), on the 12-CPU Power Challenge, the 8-CPU Cray C90, the 16-CPU J90, and the 8-CPU AlphaServer 8400, our parallel algorithm achieves 23%, 33%, 25%, and 17% peak floating-point performance. The respective Mflop rates are 1002, 2583, 831 and 781. These are the fastest results for the unsymmetric LU factorization on these powerful high-performance machines. Previous results showed much lower factorization rates because the machines used were relatively slow and the computational kernel in the earlier parallel algorithms was based on Level 1 BLAS. The closest work is the parallel symmetric pattern multifrontal factorization by Amestoy and Duff [1], also on shared memory machines. However, that approach may result in too many nonzeros and so be inefficient for unsymmetric pattern sparse matrices.

Another contribution was to provide detailed performance analysis and modeling for the underlying algorithm. In particular, we identified the three main factors limiting parallel performance: (1) contention for accessing critical sections, (2) processors sitting idle due to pipeline waiting, and (3) the need to sacrifice some per-processor efficiency in order to gain more concurrency. Which factor plays the most significant role depends on the relative performance of integer and floating-point arithmetic in the underlying architecture.

We have developed a theoretical model to analyze our parallel algorithm and predict the optimally attainable speedup. When comparing the theoretical prediction (Table 12) with the actual speedups (Figure 21), we find that there exists a discrepancy between the two. This is because our hypothetical machine and the optimal scheduling used in the model do not capture all the details of a real machine with real scheduling. Nevertheless, we do see a similar behavior in the predicted and actual speedups. That is, for the matrices predicted lower speedups, such as 11, 15, 18 and 21, the actual speedups are also lower. The model is a useful tool to help identify the inherently sequential problems with bad column orderings. The model also suggests that the panel-wise

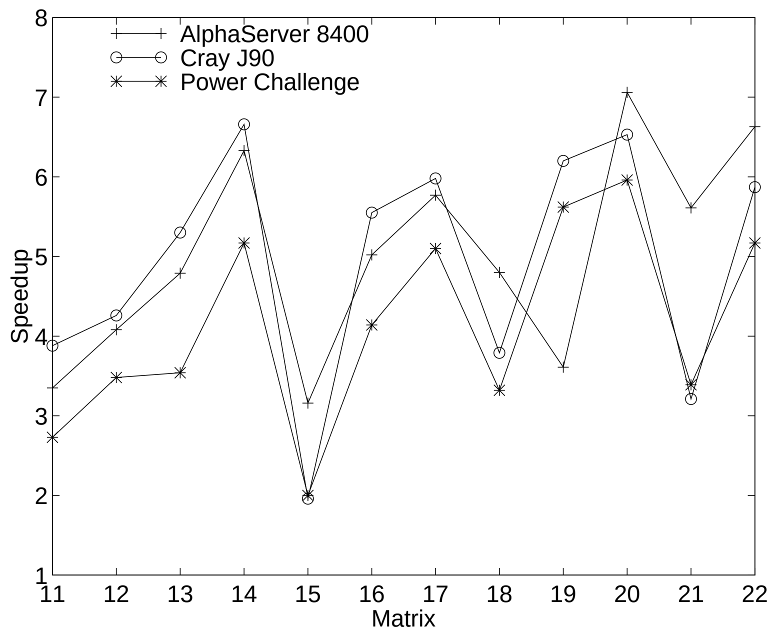


Figure 21: Speedups on 8 processors of the Power Challenge, the AlphaServer 8400 and the Cray J90.

parallel algorithm, although efficient on small scale SMPs, cannot effectively utilize more than 50 processors.

We plan to expand this research in several directions. We will study a more scalable algorithm for larger parallel machines. This algorithm is likely to partition the matrix by both rows and columns, and schedule blocks of submatrices onto processors. This will potentially increase parallelism, and reduce the panel update pipeline waiting time. In the framework of SuperLU, both serial and parallel, we will investigate incomplete *LU* factorizations, which can be used as a class of preconditioners for unsymmetric sparse iterative solvers.

9 Acknowledgements

Ed Rothberg of Silicon Graphics not only provided us access to the SGI Power Challenge, but also helped improve performance of our algorithm. We thank Esmond Ng of Oak Ridge National Lab for correspondences on the issues of nonzero structure prediction, which helped design the memory management scheme discussed in Section 5.3. We thank Kathy Yelick for suggestions on the presentation of the performance analysis section. We thank Osni Marques and Peter Tang for suggestions on improving the presentation of the material.

References

[1] P. R. Amestoy and I.S. Duff. MUPS: a parallel package for solving sparse unsymmetric sets of linear equations. Technical report, CERFACS, Toulouse, France, 1994.

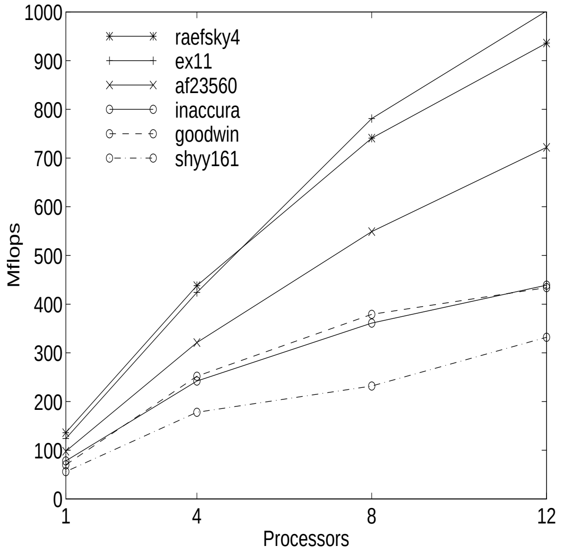


Figure 22: Mflop rate on a SGI Power Challenge.

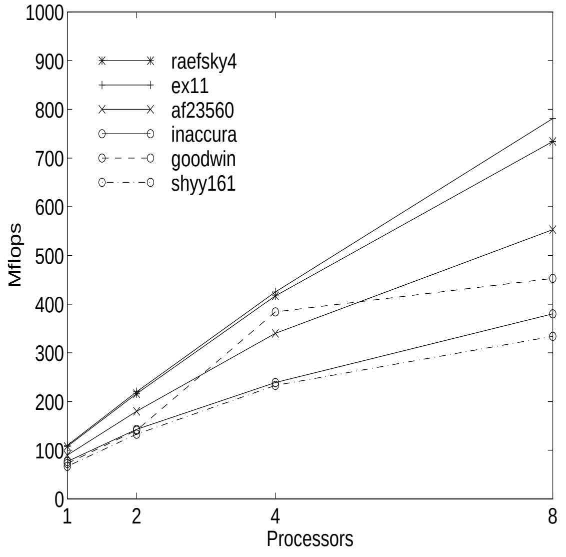


Figure 23: Mflop rate on a DEC AlphaServer 8400.

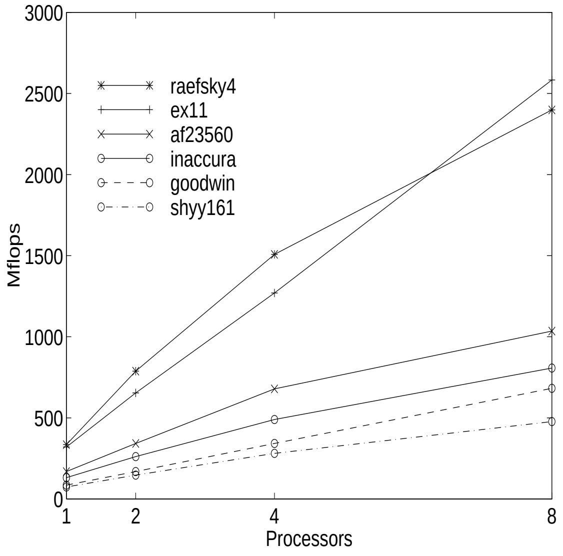


Figure 24: Mflop rate on a Cray C90.

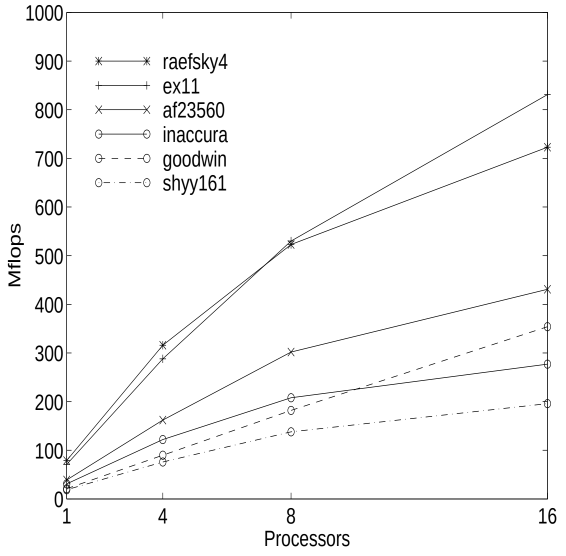


Figure 25: Mflop rate on a Cray J90.

[2] Patrick R. Amestoy. Factorization of large unsymmetric sparse matrices based on a multifrontal approach in a multiprocessor environment. Technical Report TH/PA/91/2, CERFACS, Toulouse, France, February 1991. Ph.D thesis.

[3] C. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Mathematical Software*, 15:291–309, 1989.

[4] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C.-W. Chin. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. Computer Science Dept. Technical Report CS-96-326, University of Tennessee, Knoxville, May 1996. (LAPACK Working Note #111).

[5] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W.H. Liu. A supernodal approach to sparse partial pivoting. Technical Report UCB//CSD-95-883, Computer Science Division, U.C. Berkeley, July 1995. (Xerox PARC report CSL-95-03, LAPACK Working Note #103).

[6] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[7] J. Dongarra, J. Du Croz, Duff I., and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[8] I. S. Duff, R. Grimes, and J. Lewis. Sparse matrix test problems. *ACM Trans. Mathematical Software*, 15:1–14, 1989.

[9] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in sparse unsymmetric symbolic factorization. *SIAM J. Matrix Analysis and Applications*, 13:202–211, 1992.

[10] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in a sparse partial pivoting code. *SIAM J. Scientific and Statistical Computing*, 14:253–257, 1993.

[11] David M. Fenwick, Denis J. Foley, William B. Gist, Stephen R. VanDoren, and Daniel Wissel. The AlphaServer 8000 series: High-end server platform development. *Digital Technical Journal*, 7(1):43–65, 1995.

[12] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. Solution of sparse positive definitive systems on a shared-memory multiprocessor. *International Journal of Parallel Programming*, 15(4):309–325, 1986.

[13] Alan George, Joseph Liu, and Esmond Ng. A data structure for sparse QR and LU factorizations. *SIAM J. Sci. Stat. Comput.*, 9:100–121, 1988.

[14] Alan George and Esmond Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM J. Sci. Stat. Comput.*, 6(2):390–409, 1985.

[15] Alan George and Esmond Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM J. Sci. Stat. Comput.*, 8(6):877–898, 1987.

[16] Alan George and Esmond Ng. Parallel sparse Gaussian elimination with partial pivoting. *Annals of Operation Research*, 22:219–240, 1990.

[17] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in Matlab: Design and implementation. *SIAM J. Matrix Analysis and Applications*, 13:333–356, 1992.

[18] J. R. Gilbert and E. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, 1993.

[19] John R. Gilbert. An efficient parallel sparse partial pivoting algorithm. Technical Report CMI No. 88/45052-1, Computer Science Department, University of Bergen, Norway, 8 1988.

[20] John R. Gilbert, Esmond G. Ng, and Barry W. Peyton. Computing row and column counts for sparse QR factorization. Talk presented at SIAM Symposium on Applied Linear Algebra, June 1994. Journal version in preparation.

[21] John R. Gilbert, Esmond G. Ng, and Barry W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.*, 15:1075–1091, 1994.

[22] A. Gupta and V. Kumar. Optimally scalable parallel sparse Cholesky factorization. In *Proceedings of the Seventh SIAM Conference on Parallel Proceesing for Scientific Computing*, pages 442–447. SIAM, 1995.

[23] A. Gupta, E. Rothberg, E. Ng, and B. W. Peyton. Parallel sparse Cholesky factorization algorithms for shared-memory multiprocessor systems. In R. Vichnevetsky, D. Knight, and G. Richter, editors, *Advances in Computer Methods for Partial Differential Equations–VII*. IMACS, 1992.

[24] Xiaoye S. Li. Sparse Gaussian elimination on high performance computers. Technical Report UCB//CSD-96-919, Computer Science Division, U.C. Berkeley, September 1996. Ph.D dissertation.

[25] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Analysis and Applications*, 11:134–172, 1990.

[26] Joseph W.H. Liu, Esmond G. Ng, and Barry W. Peyton. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 14(1):242–252, January 1993.

[27] Esmond G. Ng and Barry W. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM J. Sci. Comput.*, 14(4):761–769, July 1993.

[28] Edward Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *SIAM J. Scientific Computing*, 17(3):699–713, May 1996.

[29] SGI Power Challenge. Silicon Graphics, 1995. Technical Report.

[30] SPARCcenter 2000 architecture and implementation. Sun Microsystems, Inc., November 1993. Technical White Paper.

[31] S. A. Vavasis. Stable finite elements for problems with wild coefficients. Technical Report 93–1364, Department of Computer Science, Cornell University, Ithaca, NY, 1993. To appear in *SIAM J. Numerical Analysis*.

[32] The Cray C90 series. http://www.cray.com/PUBLIC/product-info/C90/. Cray Research, Inc.

[33] The Cray J90 series. http://www.cray.com/PUBLIC/product-info/J90/. Cray Research, Inc.

A Performance of the parallel algorithm

A.1 On the Sun SPARCcenter 2000

	Matrix	$P = 1$	$P = 2$	$P = 4$	Seconds	Mflops
1	Memplus	0.44	0.82	0.74	2.35	1
2	Gemat11	0.77	1.25	1.51	0.47	3
3	Rdist1	0.86	1.92	1.82	1.71	8
4	Orani678	0.71	1.24	2.08	1.98	8
5	Mcfe	0.79	1.38	2.00	0.45	9
6	Lnsp3937	0.96	1.85	2.03	2.26	18
7	Lns3937	0.92	1.73	3.09	2.41	19
8	Sherman5	0.83	1.70	2.81	1.26	20
9	Jpwh991	0.77	1.56	2.77	0.84	22
10	Sherman3	0.90	1.74	2.92	2.77	22
11	Orsreg1	0.89	1.75	3.17	2.27	27
12	Saylr4	0.88	1.76	3.10	4.17	25
13	Shyy161	0.90	1.82	3.25	59.55	26
15	Goodwin	0.92	1.86	3.61	20.50	33
18	Dense1000	0.97	1.96	3.64	16.39	41
	Mean speedup	0.83	1.62	2.64		
	Std deviation	0.13	0.32	0.83		

Table 13: Speedup, factorization time and Mflop rate on a 4-CPU SPARCcenter 2000.

A.2 On the SGI Power Challenge

	Matrix	$P = 1$	$P = 4$	$P = 8$	$P = 12$	Seconds	Mflops
1	Memplus	0.72	1.73	1.73	1.69	0.42	4
2	Gemat11	0.89	1.86	2.36	3.71	0.07	22
3	Rdist1	0.89	1.66	1.56	2.23	0.44	32
4	Orani678	0.68	1.72	2.40	2.56	0.45	33
5	Mcfe	0.68	1.92	2.09	3.29	0.07	59
6	Lnsp3937	0.97	3.00	3.65	3.86	0.35	122
7	Lns3937	0.98	2.98	3.92	3.73	0.40	117
8	Sherman5	0.86	2.29	3.09	3.09	0.23	111
9	Jpwh991	0.83	2.40	3.43	5.33	0.09	205
10	Sherman3	0.87	2.36	2.78	2.78	0.40	157
11	Orsreg1	0.88	2.67	2.73	2.97	0.34	180
12	Saylr4	0.90	2.81	3.48	4.58	0.38	284
13	Shyy161	0.86	2.71	3.54	5.06	4.64	332
14	Goodwin	0.89	3.45	5.17	5.90	1.56	433
15	Venkat01	0.65	1.72	2.00	1.98	15.37	209
16	Inaccura	0.85	2.77	4.14	5.00	9.53	438
17	Af23560	0.91	2.98	5.10	6.70	8.87	722
18	Dense1000	0.85	2.64	3.32	4.17	0.90	740
19	Raefsky3	0.92	3.07	5.62	6.91	11.35	1070
20	Ex11	0.94	3.23	5.96	7.64	26.95	1046
21	Wang3	0.85	2.20	3.39	4.03	21.37	681
22	Raefsky4	0.94	3.05	5.17	6.52	33.57	936
23	Vavasis3	0.91	3.58	6.06	6.69	105.06	862
	Mean speedup	0.86	2.56	3.59	4.37		
	Std deviation	0.09	0.59	1.36	1.73		

Table 14: Speedup, factorization time and Mflop rate on a 12-CPU SGI Power Challenge.

A.3 On the DEC AlphaServer 8400

	Matrix	$P = 1$	$P = 2$	$P = 4$	$P = 6$	$P = 8$	Seconds	Mflops
1	MEMPLUS	0.46	0.79	0.79	0.78	0.64	0.59	3
2	GEMAT11	0.83	1.63	1.88	1.88	1.88	0.08	20
3	RDIST1	0.90	1.98	2.10	1.77	1.77	0.31	40
4	ORANI678	0.83	1.29	2.00	2.33	2.42	0.26	57
5	MCFE	0.72	1.80	3.00	2.17	2.17	0.06	66
6	LNSP3937	0.93	1.94	3.19	3.68	3.68	0.25	159
7	LNS3937	0.95	1.83	3.08	3.81	4.12	0.25	187
8	SHERMAN5	0.91	1.89	2.89	2.94	2.94	0.17	151
9	JPWH991	0.92	1.89	3.00	3.30	3.00	0.11	178
10	SHERMAN3	0.88	1.83	2.72	2.74	2.74	0.34	180
11	ORSREG1	0.93	1.88	2.93	3.35	3.35	0.26	231
12	SAYLR4	0.91	1.98	3.20	3.78	4.08	0.38	276
13	SHYY161	0.95	1.93	3.23	4.21	4.79	4.66	334
14	GOODWIN	0.99	1.98	3.68	5.39	6.33	1.49	453
15	VENKAT01	0.89	1.92	2.95	3.04	3.16	10.62	303
16	INACCURA	0.99	1.83	3.08	4.15	5.02	10.94	380
17	AF23560	0.95	1.98	3.72	5.03	5.77	11.58	553
18	DENSE1000	0.98	1.86	3.35	4.32	4.80	0.99	675
19	RAEFSKY3	0.98	1.98	3.81	3.16	3.61	28.65	422
20	EX11	0.99	1.98	3.76	5.56	7.06	34.23	781
21	WANG3	0.93	1.98	3.69	4.75	5.61	21.36	682
22	RAEFSKY4	0.98	1.98	3.81	5.44	6.63	42.79	734
23	VAVASIS3	0.96	1.97	3.69	5.28	6.64	124.24	724
	Mean speedup	0.92	1.74	2.89	3.59	4.01		
	Std deviation	0.13	0.28	0.81	1.31	1.77		

Table 15: Speedup, factorization time and Mflop rate on an 8-CPU DEC AlphaServer 8400.

A.4 On the Cray C90

	Matrix	$P = 1$	$P = 2$	$P = 4$	$P = 6$	$P = 8$	Seconds	Mflops
1	MEMPLUS	0.66	0.75	0.74	0.72	0.71	1.24	2
2	GEMAT11	0.76	1.36	2.27	3.09	3.40	0.10	15
3	RDIST1	0.71	1.98	2.41	2.41	2.31	0.48	34
4	ORANI678	0.72	1.24	2.22	2.91	3.20	0.41	37
5	MCFE	0.69	1.25	1.82	2.00	2.00	0.10	43
6	LNSP3937	0.78	1.51	2.77	2.84	4.41	0.27	151
7	LNS3937	0.78	1.51	2.95	3.97	4.23	0.30	156
8	SHERMAN5	0.77	1.49	2.90	3.59	4.07	0.15	170
9	JPWH991	0.78	1.52	2.50	3.18	2.92	0.12	164
10	SHERMAN3	0.79	1.48	2.53	2.97	2.97	0.29	214
11	ORSREG1	0.80	1.53	2.69	3.25	3.55	0.22	278
12	SAYLR4	0.83	1.58	3.05	3.85	3.97	0.33	318
13	SHYY161	0.80	1.50	2.87	3.87	4.86	3.29	477
14	GOODWIN	0.84	1.65	3.31	4.83	6.59	0.99	682
15	VENKAT01	0.70	1.28	1.65	1.73	1.74	14.04	229
16	INACCURA	0.86	1.70	3.19	4.38	5.21	5.18	807
17	AF23560	0.84	1.63	3.22	4.56	4.89	6.24	1035
18	DENSE1000	0.95	1.86	2.95	3.30	3.55	0.71	943
19	RAEFSKY3	0.91	1.74	3.45	4.77	5.83	6.17	1977
20	EX11	0.90	1.65	3.21	5.02	6.53	10.37	2583
21	WANG3	0.78	1.48	1.82	2.31	2.32	14.62	996
22	RAEFSKY4	0.92	1.80	3.43	4.60	5.46	13.13	2399
	Mean speedup	0.80	1.53	2.63	3.42	3.85		
	Std deviation	0.08	0.27	0.67	1.11	1.55		

Table 16: Speedup, factorization time and Mflop rate on an 8-CPU Cray C90.

A.5 On the Cray J90

	Matrix	$P = 1$	$P = 4$	$P = 8$	$P = 12$	$P = 16$	Seconds	Mflops
1	MEMPLUS	0.65	0.94	0.98	0.97	0.76	3.67	1
2	GEMAT11	0.71	2.44	4.38	5.25	5.83	0.18	8
3	RDIST1	0.71	2.86	2.88	2.71	2.39	1.53	10
4	ORANI678	0.71	2.07	3.11	3.82	3.85	1.13	13
5	MCFE	0.77	2.21	2.70	2.70	2.52	0.29	15
6	LNSP3937	0.75	2.87	4.91	6.21	6.39	0.66	62
7	LNS3937	0.79	2.75	4.63	5.41	5.41	0.83	58
8	SHERMAN5	0.80	2.91	4.64	5.07	5.32	0.41	63
9	JPWH991	0.78	2.72	3.57	3.68	3.38	0.37	49
10	SHERMAN3	0.80	2.63	3.49	3.42	3.31	0.96	66
11	ORSREG1	0.83	2.83	3.88	4.22	4.16	0.70	89
12	SAYLR4	0.81	2.91	4.26	4.82	4.82	0.99	108
13	SHYY161	0.83	2.92	5.30	6.94	7.47	8.06	196
14	GOODWIN	0.88	3.32	6.66	10.02	12.81	1.94	354
15	VENKAT01	0.68	1.84	1.96	1.98	1.90	47.34	68
16	INACCURA	0.90	3.26	5.55	6.64	7.39	15.09	277
17	AF23560	0.87	3.22	5.98	7.55	8.49	15.05	431
18	DENSE1000	0.93	2.84	3.79	3.92	3.91	2.61	256
19	RAEFSKY3	0.93	3.38	6.20	7.69	8.43	19.03	641
20	EX11	0.95	3.56	6.53	9.47	10.17	32.48	831
21	WANG3	0.77	2.53	3.21	3.14	3.06	50.42	288
22	RAEFSKY4	0.98	3.54	5.87	7.36	8.12	43.54	723
	Mean speedup	0.81	2.75	4.29	5.13	5.45		
	Std deviation	0.09	0.60	1.51	2.38	2.97		

Table 17: Speedup, factorization time and Mflop rate on a 16-CPU Cray J90.