RC 22039 (98933) April 20, 2001
Computer Science/Mathematics

IBM Research Report

Recent Advances in Direct Methods for Solving Unsymmetric Sparse Systems of Linear Equations

Anshul Gupta

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

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

Recent Advances in Direct Methods for Solving Unsymmetric Sparse Systems of Linear Equations

Anshul Gupta
IBM T.J. Watson Research Center

During the past few years, algorithmic improvements alone have reduced the time required for the direct solution of unsymmetric sparse systems of linear equations by almost an order of magnitude. This paper compares the performance of some well-known software packages for solving general sparse systems. In particular, it demonstrates the consistently high level of performance achieved by WSMP—the most recent of such solvers. It compares the various algorithmic components of these solvers and discusses their impact on solver performance. Our experiments show that the algorithmic choices made in WSMP enable it to run more than twice as fast as the best among similar solvers and that WSMP can factor some of the largest sparse matrices available from real applications in only a few seconds on a 4-CPU workstation. Thus, the combination of advances in hardware and algorithms makes it possible to solve those general sparse linear systems quickly and easily that might have been considered too large until recently.

Categories and Subject Descriptors: G.1.3 [**Mathematics of Computing**]: Numerical Linear Algebra

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Sparse Matrix Factorization, Sparse LU Decomposition, Multifrontal Method, Parallel Sparse Solvers

1. INTRODUCTION

Developing an efficient parallel, or even serial, direct solver for general unsymmetric sparse systems of linear equations is a challenging task that has been a subject of research for the past four decades. Several breakthroughs have been made during this time. As a result, a number of serial and parallel software packages for solving such systems are available [Amestoy et al. 2000; Ashcraft and Grimes 1999; Davis and Duff 1997b; Grund 1998; Gupta 2000; Li and Demmel 1999; Shen et al. 2001; Schenk et al. 2000].

In this paper, we compare the performance and the main algorithmic features of some prominent software packages for solving general sparse systems and show that the algorithmic improvements of the past few years have reduced the time required to factor general sparse matrices by almost an order of magnitude. Combined with significant advances in the performance to cost ratio of parallel computing hard-

Author's address: Anshul Gupta, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598.

ware during this period, current sparse solver technology makes it possible to solve those problems quickly and easily that might have been considered impractically large until recently. We demonstrate the consistently high level of performance achieved by the Watson Sparse Matrix Package (WSMP) and show that it can factor some of the largest sparse matrices available from real applications in only a few seconds on 4-CPU workstation. The key features of WSMP that contribute to its performance include a prepermutation of rows to place large entries on the diagonal, a symmetric fill-reducing permutation based on the nested dissection ordering algorithm, and an unsymmetric-pattern multifrontal factorization that is guided by near-minimal static task- and data-dependency graphs and uses symmetric inter-supernode threshold pivoting [Gupta 2001b]. Some of these techniques, such as an unsymmetric pattern multifrontal algorithm based on static near-minimal DAGs, are new, while others have been used in the past, though not as a combination in a single sparse solver. In this paper, we will discuss the impact of various algorithmic components of the sparse general solvers on their performance with particular emphasis on the contribution of the components of the analysis and numerical factorization phases of WSMP to its performance.

The paper is organized as follows. In Section 2, we compare the serial time that some of the prominent general sparse solvers spend in factoring a sparse matrix A into triangular factors L and U—the most important phase in the solution of a sparse system of linear equations—and discuss the relative robustness and speed of these solvers under uniform and realistic test conditions. In Section 3, we list the main algorithms and strategies that these packages use in their symbolic and numerical phases, and discuss the effect of these strategies on their respective performance. In Section 4, by means of experimental comparisons, we highlight the role that various algorithms used in WSMP play in the performance of its LU factorization. In Section 5, we present a detailed performance comparison in a practical setting between WSMP and MUMPS—the general purpose sparse solver that we show in Section 2 to be the best available at the time of WSMP's release. Section 6 contains concluding remarks.

2. SERIAL PERFORMANCE OF SOME GENERAL SPARSE SOLVERS

In this section, we compare the performance of some of the well-known software packages for solving sparse systems of linear equations on a single CPU of an IBM RS6000 model S80. This is a 600 Mhz processor with a 64 KB 2-way set-associative level-1 cache and a peak theoretical speed of of 1200 Megaflops, which is representative of the performance of a typical high-end workstation available before 1999. Table 2 lists the test matrices used in this paper, which are some of the largest publicly available unsymmetric sparse matrices from real applications. The table also includes the dimension, the number of nonzeros, and the application area of the origin of each of these matrices.

The sparse solvers compared in this section are UMFPACK Version 2.2 [Davis and Duff 1997b; 1997a], SuperLU$_{MT}$ [Demmel et al. 1999], SPOOLES [Ashcraft and Grimes 1999], SuperLU$_{dist}$ [Li and Demmel 1998; 1999], MUMPS 4.1.6 [Amestoy et al. 2001; Amestoy et al. 2000], WSMP [Gupta 2000] and UMFPACK Version 3.2 [Davis 2002]. The solver suite contains two versions each of SuperLU and UMF-

Table I. Test matrices with their order (N), number of nonzeros (NNZ), and the application area of origin.

Matrix	*N*	*NNZ*	*Application*
af23560	23560	484256	Fluid dynamics
av41092	41092	1683902	Finite element analysis
bayer01	57735	277774	Chemistry
bbmat	38744	1771722	Fluid dynamics
comp2c	16783	578665	Linear programming
e40r0000	17281	553956	Fluid dynamics
e40r5000	17281	553956	Fluid dynamics
ecl32	51993	380415	Circuit simulation
epb3	84617	463625	Thermodynamics
fidap011	16614	1091362	Fluid dynamics
fidapm11	22294	623554	Fluid dynamics
invextr1	30412	1793881	Fluid dynamics
lhr34c	35152	764014	Chemical engineering
mil053	530238	3715330	Structural engineering
mixtank	29957	1995041	Fluid dynamics
nasasrb	54870	2677324	Structural engineering
onetone1	36057	341088	Circuit simulation
onetone2	36057	227628	Circuit simulation
pre2	659033	5959282	Circuit simulation
raefsky3	21200	1488768	Fluid dynamics
raefsky4	19779	1316789	Fluid dynamics
rma10	46835	2374001	Fluid dynamics
tib	18510	145149	Circuit simulation
twotone	120750	1224224	Circuit simulation
venkat50	62424	1717792	Fluid dynamics
wang3old	26064	177168	Circuit simulation
wang4	26068	177196	Circuit simulation

PACK because these versions employ very different algorithms in some or all of the important phases of the solution process (see Section 3 for details). Some other well-known packages are not featured in this section; however, their comparisons with one or more of the packages included here are easily available in literature and would not alter the inferences that can be drawn from the results in this section. Davis and Duff compare UMFPACK with MUPS [Amestoy and Duff 1989] and MA48 [Duff and Reid 1993]. MUPS is a classical multifrontal code and the predecessor of MUMPS. MA48 is a sparse unsymmetric factorization code in the HSL package [HSL 2000] and is based on conventional sparse data structures. Grund [Grund 1998] presents an experimental comparison of GSPAR [Grund 1998] with a few other solvers, including SuperLU and UMFPACK; however, this comparison is limited to sparse matrices arising in two very specific applications. MA41 [Amestoy and Duff 1989; 1993] is a commercial shared-memory parallel version of MUPS that has been available in HSL since 1990. Amestoy and Puglisi [Amestoy and Puglisi 2000] have since introduced the unsymmetrization of frontal matrices in MA41. Since MUMPS is more robust in parallel than MA41, we have chosen to use MUMPS instead of MA41. A comparison of the S+ package [Shen et al. 2001] from University of California at Santa Barbara with some others can be found in [Cosnard and Grigori 2000; Gupta and Muliadi 2001; Shen et al. 2001]. We have

Table II. LU factorization times on a single CPU (in seconds) for UMFPACK Version 2.2, SuperLU$_{MT}$, SPOOLES, SuperLU$_{dist}$, MUMPS, WSMP, and UMFPACK Version 3.2, respectively. The best pre-2000 time is underlined and the overall best time is shown in boldface. The last row shows the approximate smallest pivoting threshold that yielded a residual norm close to machine precision after iterative refinement for each package. F_M indicates that a solver ran out of memory, F_C indicates an abnormal or no termination, and F_N indicates that the numerical results were inaccurate.

Year →	1994	1997	1998	1999	1999	2000	2001
Matrices ↓	UMFP 2	SLU$_{MT}$	SPLS	SLU$_{dist}$	MUMPS	WSMP	UMFP 3
af23560	45.5	27.5	10.5	14.7	**<u>6.31</u>**	6.34	10.5
av41092	186.	87.3	F_M	F_N	<u>18.7</u>	**7.09**	220.
bayer01	1.76	1.61	F_M	3.23	<u>1.17</u>	**1.10**	1.26
bbmat	682.	373.	97.7	256.	<u>72.3</u>	**37.1**	123.
comp2c	120.	<u>11.1</u>	287.	42.0	23.5	**4.45**	802.
e40r5000	29.9	17.8	395.	F_N	<u>1.18</u>	**1.10**	7.10
ecl32	F_M	961.	562.	201.	<u>116.</u>	**37.7**	320.
epb3	29.7	6.75	5.00	5.67	<u>3.02</u>	**2.09**	6.59
fidap011	168.	F_M	<u>12.2</u>	F_N	12.7	**6.53**	22.6
fidapm11	944.	145.	<u>15.1</u>	F_N	16.3	**10.4**	63.2
invextr1	1110	F_C	1351	141.	<u>69.1</u>	**16.3**	294.
lhr34c	<u>3.46</u>	F_M	F_M	11.5	3.53	**1.32**	5.81
mixtank	F_M	F_M	346.	198.	<u>86.7</u>	**36.5**	695.
nasasrb	81.8	F_M	25.0	26.7	<u>22.0</u>	**11.0**	76.1
onetone1	12.2	27.2	113.	10.7	<u>5.79</u>	**3.52**	7.33
onetone2	<u>1.79</u>	2.01	20.7	3.55	1.95	**0.94**	0.97
pre2	F_M	F_C	F_M	F_M	F_M	**223.**	F_M
raefsky3	39.0	34.2	10.0	6.86	<u>6.09</u>	**5.04**	20.4
raefsky4	109.	F_M	157.	28.6	<u>28.5</u>	**7.93**	36.2
rma10	15.7	F_M	10.7	<u>5.78</u>	5.92	**3.92**	13.5
tib	<u>0.52</u>	132.	1.75	1.47	0.53	**0.27**	28.8
twotone	<u>30.0</u>	F_M	724.	637.	79.4	**26.6**	46.5
venkat50	16.2	33.6	11.6	<u>8.11</u>	8.70	**4.39**	14.3
wang3old	106.	281.	62.7	36.9	<u>25.3</u>	**10.7**	63.7
wang4	97.3	223.	<u>16.2</u>	23.7	18.6	**10.9**	85.2
Pthresh →	0.25	0.01	0.01	0.00	0.01	0.01	0.10

excluded another recent software, PARDISO [Schenk et al. 2000], because it is designed for unsymmetric matrices with a symmetric structure, and therefore, would have failed on many of our test matrices unless they were padded with zero valued entries to structurally symmetrize them.

Each package is compiled in 32-bit mode with the -O3 optimization option of the AIX Fortran or C compilers and is linked with IBM's Engineering and Scientific Subroutine Library (ESSL) for the basic linear algebra subprograms (BLAS) that are optimized for RS6000 processors. Almost all the floating point operations in each solver are performed inside the BLAS routines. Using the same compiler and BLAS library affords each code an equal access to the hardware specific optimizations. A maximum of 2 GB of memory was available to each code.

Table 2 shows the LU factorization time taken by each code for the matrices in our test suite. In addition to each solver's factorization time, the table also lists the year in which the latest version of each package became available. F_M,

F_C, or F_N entries indicate the failure of a solver to factor a matrix satisfactorily. Subscripts $_M$, $_C$, and $_N$ indicate failure due to running out of memory, abnormal or no termination, and numerically inaccurate results, respectively.

One of the ground rules for the experiments reported in Table 2 was that all input parameters that may influence the behavior of a program were fixed and were not modified to accommodate the demands of individual matrices. However, through a series of pre-experiments, we attempted to fix these parameters to values that yielded the best results on an average on the target machine. For example, we tested all the packages on all matrices in our test suite for various values of the pivoting threshold (*Pthresh*), and for each, chose the smallest value for which all completed runs yielded a backward error that was close to machine precision after iterative refinement. The message-passing version of SuperLU (SuperLU_{dist}) does not have a provision for partial pivoting; hence the threshold is 0. Other parameters easily accessible in the software, such as the various block sizes in SuperLU, were also fixed to values that appeared to be the best on average.

Note that some of the failures in the first four columns of Table 2 can be fixed by changing some of the options or parameters in the code. However, as noted above, the options chosen to run the experiments reported in Table 2 are such that they are best for the test suite as a whole. Changing these options may avoid some failures, but cause many more. We have chosen to report results with a consistent set of options because we believe that that is representative of the real-world environment in which these solvers are expected to be used. Moreover, in most cases, even if an alternative set of options exists that can solve the failed cases reported in Table 2, those options are not known a-priori and can only be determined by trial and error. Some memory allocation failures, however, are the result of static memory allocation for data structures with variable and unknown sizes. Therefore, such failures are artifacts of the implementation and neither reflect the actual amount of memory needed (if allocated properly) nor that the underlying algorithms are not robust.

The best factorization time for each matrix using any solver released before year 2000 is underlined in Table 2 and the overall best factorization time is shown in boldface. Several interesting observations can be made in this Table 2. Perhaps the most striking observation in the table pertains to the range of times that different packages available before 1999 would take to factor the same matrix. It is not uncommon to notice the fastest solver being faster than the slowest by one to two orders of magnitude. Additionally, none of the pre-1999 solvers yielded a consistent level of performance. For example, UMFPACK 2.2 is 13 times faster than SPOOLES on *e40r5000* but 14 times slower on *fidap011*. Also noticeable is the marked increase in the reliability and ease of use of the softwares released in 1999 or later. There are 21 failures in the first four columns of Table 2 and only two in the last three columns. MUMPS is clearly the fastest and the most robust amongst the solvers released before 2000. However, WSMP is more than twice as fast as MUMPS on this machine based on the average ratio of the factorization time of MUMPS to that of WSMP. WSMP also has the most consistent performance. It has the smallest factorization time for all but two matrices and is the only solver that did not fail on any of the test matrices.

3. KEY ALGORITHMIC FEATURES OF THE SOLVERS

In this section, we list the key algorithms and strategies that solvers listed in Table 2 use in the symbolic and numerical phases of the computation of the LU factors of a general sparse matrix. We then briefly discuss the effect of these choices on the performance of the solvers. Detailed descriptions of all the algorithms are beyond the scope of this paper, but are readily available in the citations provided. Many of the solvers, whose single-CPU performance is compared in Table 2, are designed for shared- or distributed-memory parallel computers. The target architecture of each of the solvers is also listed.

(1) UMFPACK 2.2 [Davis and Duff 1997b; 1997a]
- *Fill reducing ordering:* Approximate minimum degree [Amestoy et al. 1996] on unsymmetric structure, combined with suitable numerical pivot search during LU factorization.
- *Task dependency graph:* Directed acyclic graph.
- *Numerical factorization:* Unsymmetric-pattern multifrontal.
- *Pivoting strategy:* Threshold pivoting implemented by row-exchanges.
- *Target architecture:* Serial.

(2) SuperLU$_{MT}$ [Demmel et al. 1999]
- *Fill reducing ordering:* Multiple minimum degree (MMD) [George and Liu 1981] computed on the symmetric structure of A^TA or $A + A^T$ and applied to the columns of A. (The structure of A^TA was used in the experiments for Table 2.)
- *Task dependency graph:* Directed acyclic graph.
- *Numerical factorization:* Supernodal left-looking.
- *Pivoting strategy:* Threshold pivoting implemented by row-exchanges.
- *Target architecture:* Shared-memory parallel.

(3) SPOOLES [Ashcraft and Grimes 1999]
- *Fill reducing ordering:* Generalized nested dissection/multisection [Ashcraft and Liu 1996] computed on the symmetric structure of $A + A^T$ and applied symmetrically to rows and columns of A.
- *Task dependency graph:* Tree based on the structure of $A + A^T$.
- *Numerical factorization:* Supernodal Crout.
- *Pivoting strategy:* Threshold rook pivoting that performs row and column exchanges to control growth in both L and U.
- *Target architecture:* Serial, shared-memory parallel, and distributed-memory parallel.

(4) SuperLU$_{dist}$ [Li and Demmel 1998; 1999]
- *Fill reducing ordering:* Multiple minimum degree [George and Liu 1981] computed on the symmetric structure of $A + A^T$ and applied symmetrically to the rows and columns of A.
- *Task dependency graph:* Directed acyclic graph.
- *Numerical factorization:* Supernodal right-looking.
- *Pivoting strategy:* No numerical pivoting during factorization. Rows are preordered to maximize the magnitude of the product of the diagonal entries [Duff and Koster 1999; 2001].

—*Target architecture:* Distributed-memory parallel.

(5) MUMPS [Amestoy et al. 2001; Amestoy et al. 2000]
   —*Fill reducing ordering:* Approximate minimum degree [Amestoy et al. 1996] computed on the symmetric structure of $A + A^T$ and applied symmetrically to rows and columns of A.
   —*Task dependency graph:* Tree based on the structure of $A + A^T$.
   —*Numerical factorization:* Symmetric-pattern multifrontal.
   —*Pivoting strategy:* Preordering rows to maximize the magnitude of the product of the diagonal entries [Duff and Koster 1999; 2001], followed by unsymmetric row exchanges within supernodes and symmetric row and column exchanges between supernodes.
   —*Target architecture:* Distributed-memory parallel.

(6) WSMP [Gupta 2001b; 2001a; 2000]
   —*Fill reducing ordering:* Nested dissection [Gupta 1997] computed on the symmetric structure of $A + A^T$ and applied symmetrically to rows and columns of A.
   —*Task dependency graph:* Minimal directed acyclic graph [Gupta 2001b].
   —*Numerical factorization:* Unsymmetric-pattern multifrontal.
   —*Pivoting strategy:* Preordering rows to maximize the magnitude of the product of the diagonal entries [Gupta and Ying 1999], followed by unsymmetric partial pivoting within supernodes and symmetric pivoting between supernodes. Rook pivoting (which attempts to contain growth in both L and U) is an option.
   —*Target architecture:* Shared-memory parallel.

(7) UMFPACK 3.2 [Davis 2002]
   —*Fill reducing ordering:* Column approximate minimum degree algorithm [Davis et al. 2000] to compute fill-reducing column preordering,
   —*Task dependency graph:* Tree based on the structure of $A^T A$.
   —*Numerical factorization:* Unsymmetric-pattern multifrontal.
   —*Pivoting strategy:* Threshold pivoting implemented by row-exchanges.
   —*Target architecture:* Serial.

The performance of the solvers calibrated in Table 2 is greatly affected by the algorithmic features outlined above. We now briefly describe, in order of importance, the relationship between some of these algorithms and the performance characteristics of the solvers that employ these algorithms.

3.1 Pivoting Strategy and Application of Fill-Reducing Ordering

The application of the fill-reducing permutation and the pivoting strategy used in different solvers seem to be the most important factors that distinguish MUMPS and WSMP from the others and allows these two to deliver consistently good performance.

3.1.1 *The Conventional Strategy.* In [George and Ng 1985], George and Ng showed that the fill-in as a result of LU factorization of an irreducible square unsymmetric sparse matrix A, irrespective of its row permutation, is a subset of the fill-in that a symmetric factorization of $A^T A$ would generate. Guided by this result, many unsymmetric sparse solvers developed in 1990's adopted variations of the following ordering and pivoting strategy. An ordering algorithm would seek to compute a fill-reducing permutation of the columns of A based on their sparsity pattern, because a column permutation of A is equivalent to a symmetric permutation of $A^T A$. The numerical factorization phase of these solvers would then seek to limit pivot growth via threshold pivoting involving row interchanges.

A problem with the above strategy is that the upper-bound on the fill in the LU factorization of A predicted by Gilbert and Ng's result can be very loose, especially in the presence of even one relatively dense row in A. As a result, the initial column ordering could be very ineffective. Moreover, two different column orderings, both equally effective in reducing fill in the symmetric factorization of $A^T A$, could enjoy very different degrees of success in reducing the fill in the LU factorization of A. There is some evidence of this being a factor in the extreme variations in the factorization times of different solvers for the same matrices in Table 2. The matrices that have a symmetric structure and require very little pivoting, such as *nasasrb, raefsky3, rma10, venkat50,* and *wang4* exhibit relatively less variation in the factorization times of different solvers. On the other hand, consider the performance of WSMP and UMFPACK 3.2 on matrices *comp2c* and *tib*, which contain a few rows that are much denser than the rest. Both WSMP and UMFPACK 3.2 use very similar unsymmetric-pattern multifrontal factorization algorithms. However, since the column ordering in UMFPACK 3.2 seeks to minimize the fill in a symmetric factorization of $A^T A$ rather than directly in the LU factorization of A, it is more than two orders of magnitude slower than WSMP on these matrices. Our experiments (Section 5) have verified that WSMP did not enjoy such a dramatic advantage over UMFPACK 3.2 for these matrices due to other differences such as the use of a nested-dissection ordering or a pre-permutation of matrix rows.

3.1.2 *The Strategy Used in MUMPS and WSMP.* We now briefly describe the ordering and pivoting strategy of MUMPS and WSMP in the context of a structurally symmetric matrix. Note that pivoting in MUMPS would be similar even in the case of an unsymmetric matrix because it uses a symmetric-pattern multifrontal algorithm guided by the elimination tree [Liu 1990] corresponding to the symmetric structure of $A + A^T$. On the other hand, WSMP uses an unsymmetric-pattern multifrontal algorithm and an elimination DAG (directed acyclic graph) to guide the factorization. Therefore, the pivoting is somewhat more complex if the matrix A to be factored has an unsymmetric structure. However, the basic pivoting idea and the reason why it is effective remain the same.

Both MUMPS and WSMP start with a symmetric fill-reducing permutation computed on the structure of $A+A^T$. Just like most modern sparse factorization codes, MUMPS and WSMP work with supernodes—adjacent groups of rows and columns with the same or nearly same structures in the factors L and U. An interchange amongst the rows and columns of a supernode has no effect on the overall fill-in, and is the preferred mechanism for finding a suitable pivot. However, there is no guarantee that the algorithm would always succeed in finding a suitable pivot within the pivot block; that is, an element whose row as well as column index lies within the indices of the supernode being currently factored. When the algorithm reaches a point where it cannot factor an entire supernode based on the prescribed threshold, it merges the remaining rows and columns of the supernode with its parent supernode in the elimination tree. This is equivalent to a symmetric permutation of the failed rows and columns to a location with higher indices within the matrix. By virtue of the properties of the elimination tree [Liu 1990], the new location of these failed rows and columns also happens to be their "next best" location from the perspective of the potential fill-in that these rows and columns would produce. For example, in the context of a fill-reducing ordering based on the nested dissection [George and Liu 1978; Lipton et al. 1979] of the graph of the coefficient matrix, this pivoting strategy is equivalent to moving the vertex corresponding to a failed pivot from its partition to the immediate separator that created that partition. Merged with a parent supernode, the unsuccessful portion of the child supernode has more rows and columns available for potential interchange. However, should a part of the new supernode remain unfactored due to a lack of suitable intra-supernode pivots, it can again be merged with its parent supernode, and so on.

The key point is that, with this strategy, pivot failures increase the fill-in gracefully rather than arbitrarily. Moreover, the fewer the inter-supernode pivoting steps, the closer the final fill-in stays to that of the original fill-reducing ordering. Although, unlike the conventional strategy, there is no proven upper-bound on the amount of fill-in that can potentially be generated, the empirical evidence clearly suggests that the extra fill-in due to pivoting stays reasonably well-contained. To further aid this strategy, it has been shown recently [Amestoy et al. 2001; Duff and Koster 1999; Li and Demmel 1998] that permuting the rows or the columns of the matrix prior to factorization so as to maximize the magnitude of its diagonal entries can often be very effective in reducing the amount of pivoting during factorization. Both MUMPS and WSMP use this technique to reduce inter-supernode pivoting and the resulting extra fill-in.

Like MUMPS and WSMP, SPOOLES too uses a symmetric fill-reducing permutation followed by symmetric inter-supernode pivoting. However, SPOOLES employs a pivoting algorithm known as rook pivoting that seeks to limit pivot growth in both L and U. Other than SPOOLES, in all solvers discussed in this paper, a pivot is considered suitable as long as it is not smaller in magnitude than pivot threshold times the entry with the largest magnitude in that column. The pivoting algorithm thus seeks to control pivot growth only in L. The more stringent pivot suitability criterion of SPOOLES causes a large number of pivot failures and the resulting fill-in overshadows a good initial ordering. Simple threshold partial pivoting yields a sufficiently accurate factorization for most matrices, including all our test cases. Therefore, rook pivoting is an option in WSMP, but the default is the standard threshold pivoting.

3.2 Ordering Algorithms

In addition to the decision whether to compute a fill-reducing symmetric ordering or column ordering, the actual ordering algorithm itself affects the performance of the solvers. WSMP uses a nested dissection based on a multilevel partitioning scheme, which is very similar to Metis [Karypis and Kumar 1999]. As explained in [Gupta 1997], WSMP's ordering scheme adds some heuristics to the basic multilevel approach to make it more robust for some very unstructured problems. SPOOLES uses a similar ordering that generates multisections of graphs instead of bisections [Ashcraft and Liu 1996]. Section 4 of this paper and [Amestoy et al. 2001] present empirical evidence that graph-partioning based orderings used in SPOOLES and WSMP are generally more effective in reducing the fill-in and operation count in factorization than local heuristics, such as multiple minimum degree (MMD) [Liu 1985] used in SuperLU$_{MT}$ and SuperLU$_{dist}$, approximate minimum degree (AMD) [Amestoy et al. 1996] used in MUMPS, a variation of AMD used in UMFPACK 2.2, and the column approximate minimum degree (COLAMD) [Davis et al. 2000] algorithm used in UMFPACK 3.2. In most solvers in which ordering is a separate phase, the users can override the default ordering and can provide their own permutation vectors.

In their ordering phase, UMFPACK 2.2 and WSMP perform another manipulation of the sparse coefficient matrix prior to performing any other symbolic or numerical processing on it. They seek to reduce it into a block triangular form [Duff et al. 1990], which can be achieved very efficiently [Tarjan 1972]. Solving the original system then requires analyzing and factoring only the diagonal block matrices. Some of the symbolic algorithms employed in WSMP [Gupta 2001b] are valid only for irreducible sparse matrices. The cost of reduction to a block triangular form is insignificant, but it can offer potentially large savings when it is effective. In our test suite, however, *lhr34c* is the only matrix that benefits significantly from reduction to block triangular form. The others either have only one block or the size of the largest block is fairly close to the dimension of the overall coefficient matrix.

3.3 Symbolic Factorization Algorithms

The process of factoring a sparse matrix can be expressed by a directed acyclic task-dependency graph, or task-DAG in short. The vertices of this DAG correspond to the tasks of factoring row-column pairs or groups of row-column pairs of the sparse matrix and the edges correspond to the dependencies between the tasks. A task is ready for execution if and only if all tasks with incoming edges to it have completed. In addition to a task-DAG, there is a data-dependency graph or a data-DAG associated with sparse matrix factorization. The vertex set of the data-DAG is the same as that of the task-DAG for a given sparse matrix. An edge from a vertex i to a vertex j in the data-DAG denotes that at least some of the data produced as the result of the output of task i is required as input by task j. While the task-DAG is unique to a given sparse matrix, the data-DAG can be a function of the sparse factorization algorithm. Multifrontal algorithms [Duff and Reid 1984; Liu 1992; Davis and Duff 1997b] for sparse factorization can work with a minimal data-DAG (i.e., a data-DAG with the smallest possible number of edges) for a given matrix.

The task- and data-dependency graph involved in the factorization of a symmetric matrix is a tree, known as the elimination tree [Liu 1990]. However, for unsymmetric matrices, the task- and data-DAGs are general directed acyclic graphs. Moreover, the edge-set of the minimal data-DAG for unsymmetric sparse factorization can be a superset of the edge-set of a task-DAG. In [Gilbert and Liu 1993], Gilbert and Liu describe elimination structures for unsymmetric sparse LU factors and give an algorithm for sparse unsymmetric symbolic factorization. These elimination structures are two directed acyclic graphs (DAGs) that are transitive reductions of the graphs of the factor matrices L and U, respectively. The union of these two directed acyclic graphs is the minimal task-dependency graph of sparse LU factorization; that is, it is a task-dependency graph in which all edges are necessary. Using a minimal elimination structure to guide factorization is useful because it avoids overheads due to redundancy and exposes maximum parallelism. However, some researchers have argued that computing an exact transitive reduction can be too expensive [Davis and Duff 1997b; Eisenstat and Liu 1993] and have proposed using sub-minimal DAGs with more edges than necessary. Traversing or pruning the redundant edges in the elimination structure during numerical factorizations, as is done in UMFPACK and SuperLU$_{MT}$, can be a source of overhead. Alternatively, many unsymmetric factorization codes, such as SPOOLES and MUMPS adopt the elimination tree corresponding to the symmetric structure of $A + A^T$ as the task- and data-dependency graph to the guide the factorization. This adds artificial dependencies to the elimination structure and can lead to diminished parallelism and extra fill-in and operations during factorization.

WSMP uses a modified version of the classical unsymmetric symbolic factorization algorithm [Gilbert and Liu 1993] that detects the supernodes as it processes the rows and columns of the sparse matrix and enables a fast computation of the exact transitive reductions of the structures of L and U to yield a minimal task-dependency graph [Gupta 2001b]. In addition, WSMP uses a fast algorithm for the derivation of a near-minimal data-dependency DAG from the minimal task-dependency DAG. The data-dependency graph of WSMP is such that it is valid in the presence of any amount of inter-supernode pivoting and yet has been empirically shown to contain only between 0 and 14% (4% on an average) more edges than the minimal task-dependency graph on a suite of large unsymmetric sparse matrices. The edge-set of this static data-DAG is sufficient to capture all possible dependencies that may result from row and column permutations due to numerical pivoting during factorization. The powerful symbolic algorithms used in WSMP enable its numerical factorization phase to proceed very efficiently spending minimal time on non-floating-point operations.

3.4 Numerical Factorization Algorithms

The multifrontal method [Duff and Reid 1984; Liu 1992] for solving sparse systems of linear equations usually offers a significant performance advantage over more conventional factorization schemes by permitting efficient utilization of parallelism and memory hierarchy. Our detailed experiments in [Gupta and Muliadi 2001] show that all three multifrontal solvers—UMFPACK, MUMPS, and WSMP—run at a much higher Megaflops rate than their non-multifrontal counterparts. The original multifrontal algorithm proposed by Duff and Reid [Duff and Reid 1984] uses the symmetric-pattern of $A + A^T$ to generate an elimination tree to guide the numerical factorization, which works on symmetric frontal matrices. This symmetric-pattern multifrontal algorithm used in MUMPS can incur a substantial overhead for very unsymmetric matrices due to unnecessary dependencies in the elimination tree and extra zeros in the artificially symmetrized frontal matrices. Davis and Duff [Davis and Duff 1997b] and Hadfield [Hadfield 1994] introduced an unsymmetric-pattern multifrontal algorithm, which is used in UMFPACK and overcomes the shortcomings of the symmetric-pattern multifrontal algorithm. However, UMFPACK does not reveal the full potential of the unsymmetric-pattern multifrontal algorithm. UMFPACK 2.2 used a degree approximation algorithm similar to AMD [Amestoy et al. 1996] fill-reducing ordering algorithm, which has now been shown to be less effective than nested dissection [Amestoy et al. 2001]. Moreover, the merging of the ordering and symbolic factorization within numerical factorization in UMFPACK 2.2 slows down the latter and excludes the possibility of using a better ordering while retaining the factorization code. UMFPACK 3.2 separates the analysis (ordering and symbolic factorization) from numerical factorization. However, as discussed in Section 3.1, it suffers from the pitfalls of permuting only the columns based on a fill-reducing ordering rather than using a symmetric fill-reducing permutation.

The unsymmetric-pattern multifrontal LU factorization in WSMP, described in detail in [Gupta 2001b], is aided by powerful algorithms in the analysis phase and uses efficient dynamic data structures to perform potentially multiple steps of numerical factorization with minimum overhead and maximum parallelism. It uses a technique similar to the one described in [Hadfield 1994] to efficiently handle any amount of pivoting and different pivot sequences without repeating the symbolic phase for each factorization. In [Gupta 2001b], the author defines near-minimal static task- and data-dependency DAGs that can be computed during symbolic factorization and describes an unsymmetric-pattern multifrontal factorization algorithm based on these DAGs. As shown in [Gupta 2001b], an important property of these DAGs is that even though they are static, they can handle an arbitrary amount of dynamic pivoting to guarantee numerical stability. In other unsymmetric multifrontal solvers [Hadfield 1994; Davis and Duff 1997a; 1997b], the task- and data-DAGs are computed on the fly during numerical factorization. The use of static pre-computed DAGs contributes significantly to the simplification and efficiency of WSMP's numerical factorization. Furthermore, the resulting unsymmetric-pattern multifrontal algorithm is also more amenable to efficient parallelization. A description of WSMP's shared-memory parallel LU factorization algorithm can be found in [Gupta 2001a].

4. ROLE OF WSMP ALGORITHMS IN ITS LU FACTORIZATION PERFORMANCE

The speed and the robustness of WSMP's sparse LU factorization stems from (1) its overall ordering and pivoting strategy, (2) an unsymmetric-pattern multifrontal numerical factorization algorithm guided by static near-minimal task- and data-dependency DAGs, (3) the use of nested-dissection ordering, and (4) permutation of high-magnitude coefficients to the diagonal. In Section 3, we presented arguments based on empirical data that a symmetric fill-reducing ordering followed by a symmetric inter-supernode pivoting is a major feature distinguishing MUMPS and WSMP from most other sparse unsymmetric solvers. The role that this strategy plays in the performance of sparse LU factorization is evident from the results in Table 2. In this section, we present the results of some targeted experiments on MUMPS and WSMP to highlight the role of each one of the other three key algorithmic features of WSMP in its LU factorization performance.

The experiments described in this section were conducted on one and four pro-

Table III. Number of factor nonzeros (nnz_f), operation count (Ops), LU factorization time, and speedup (S) of MUMPS and WSMP run on one (T_1) and four (T_4) 375 Mhz Power 3 processors with default options.

Matrices	MUMPS					WSMP				
	nnz_f $\times 10^6$	Ops $\times 10^9$	T_1 (s)	T_4 (s)	S ($\frac{T_1}{T_4}$)	nnz_f $\times 10^6$	Ops $\times 10^9$	T_1 (s)	T_4 (s)	S ($\frac{T_1}{T_4}$)
af23560	8.34	2.56	**4.05**	**2.27**	1.8	9.58	3.27	**3.96**	**1.83**	2.2
av41092	14.1	8.42	**12.0**	**6.98**	1.7	9.10	2.14	**4.59**	**2.65**	1.8
bayer01	2.82	.125	**1.10**	**0.63**	1.7	1.57	.040	**0.95**	**0.95**	1.0
bbmat	46.0	41.4	**48.0**	**20.7**	2.3	31.9	20.1	**22.9**	**8.26**	2.8
comp2c	7.05	4.22	**10.2**	**7.33**	1.3	2.98	0.78	**1.64**	**0.67**	2.4
e40r0000	1.72	.172	**0.83**	**0.61**	1.3	2.06	.250	**0.56**	**0.28**	2.0
ecl32	42.9	64.6	**64.7**	**31.5**	2.0	25.8	21.0	**23.1**	**7.41**	3.1
epb3	6.90	1.17	**2.70**	**1.39**	1.9	4.99	.452	**1.66**	**1.23**	1.4
fidap011	12.5	7.01	**8.73**	**7.64**	1.1	8.69	3.20	**3.93**	**1.78**	2.2
fidapm11	14.0	9.67	**11.6**	**7.38**	1.6	12.8	5.21	**6.50**	**2.60**	2.5
invextr1	30.3	35.6	**38.9**	**23.6**	1.6	15.1	6.90	**9.93**	**4.67**	2.1
lhr34c	5.58	.641	**2.21**	**1.15**	1.9	2.91	.163	**0.92**	**0.93**	1.0
mil053	75.9	31.8	**42.8**	**16.6**	2.5	58.9	14.4	**23.0**	**10.6**	2.2
mixtank	38.5	64.4	**64.8**	**31.0**	2.1	23.2	19.5	**21.9**	**8.32**	2.6
nasasrb	24.2	9.45	**13.1**	**10.2**	1.3	18.9	5.41	**6.98**	**3.37**	2.1
onetone1	4.72	2.29	**3.66**	**2.67**	1.4	3.54	1.25	**2.25**	**1.52**	1.5
onetone2	2.26	.510	**1.17**	**0.82**	1.4	1.41	.191	**0.72**	**0.70**	1.0
pre2	358.	Fail	**Fail**	**Fail**	-	79.2	96.3	**127.**	**55.3**	2.3
raefsky3	8.44	2.90	**4.56**	**3.45**	1.3	8.09	2.57	**3.16**	**1.40**	2.3
raefsky4	15.7	10.9	**13.0**	**8.97**	1.4	10.3	4.11	**4.91**	**2.34**	2.1
rma10	8.87	1.40	**4.13**	**2.98**	1.4	9.58	1.48	**2.47**	**0.99**	2.5
twotone	22.1	29.3	**43.5**	**26.1**	1.6	10.8	9.46	**13.5**	**9.05**	1.5
venkat50	12.0	2.31	**4.87**	**2.74**	1.8	11.4	1.75	**2.83**	**1.13**	2.5
wang3old	13.8	13.8	**15.1**	**6.48**	2.3	9.66	5.91	**6.65**	**3.50**	1.9
wang4	11.6	10.5	**11.8**	**5.84**	2.0	9.93	6.09	**6.84**	**3.08**	2.2

cessors of an IBM RS6000 WH-2. Each of its four 375 Mhz Power 3 processors have a peak theoretical speed of 1.5 Gigaflops. The peak theoretical speed of the workstation is therefore 6 Gigaflops, which is representative of the performance of a high-end workstation available in 2001. The four CPUs of this workstation share an 8 MB level-2 cache and have a 64 KB level-1 cache each. 2 GB of memory was available to each single CPU run and the 4-CPU runs of WSMP. MUMPS, when run on 4 processors, had a total of 4 GB of memory available to it. MUMPS uses the message-passing paradigm and MPI processes for parallelism. The distributed-memory parallel environment for which MUMPS is originally designed, may add some constraints and overheads to it. For example, numerical pivoting is generally easier to handle in a shared-memory parallel environment that in a distributed-memory one. However, MUMPS was run in mode in which MPI was aware of and took advantage of the fact that the multiple processes were running on the same machine (by setting the MP_SHARED_MEMORY environment variable to '*yes*'). The current version of WSMP is designed for the shared-address-space paradigm and uses the Pthreads library.

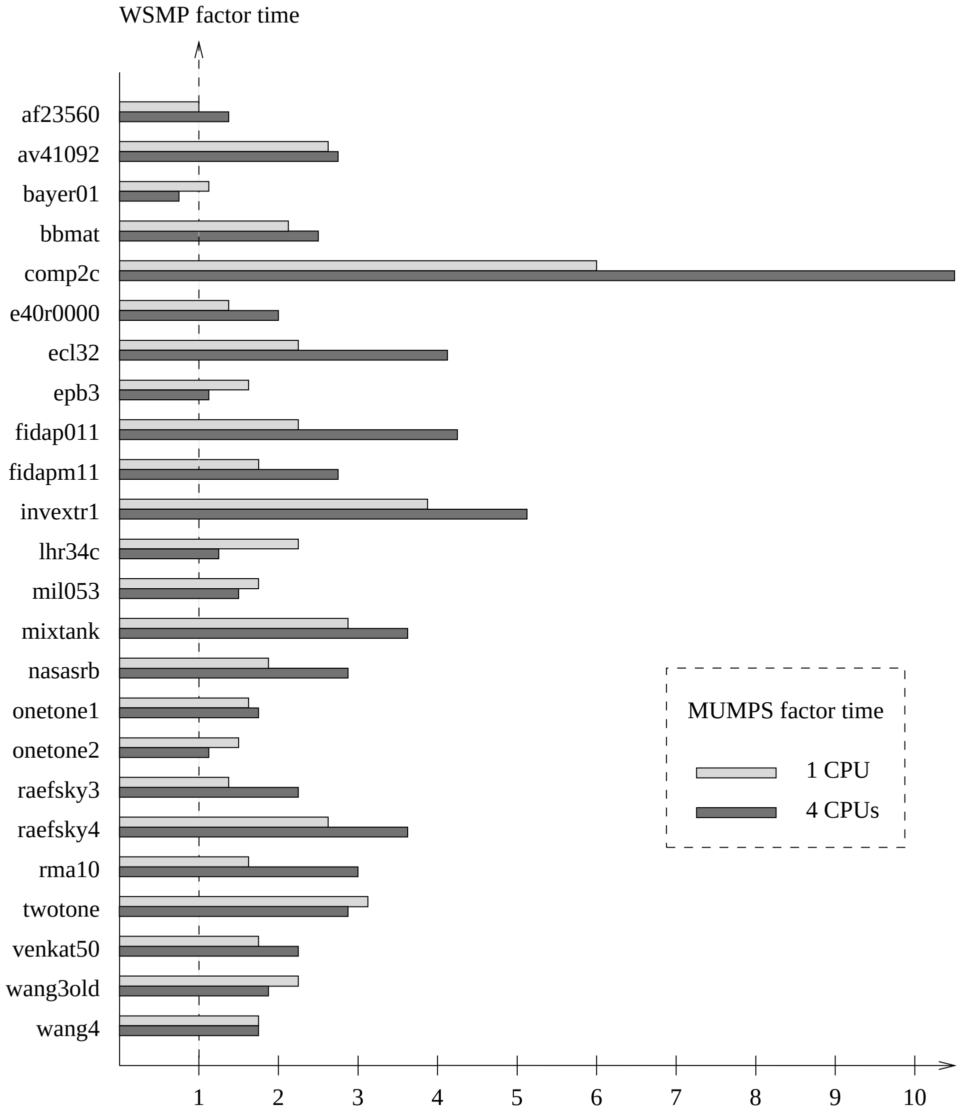

Fig. 1. Ratios of the factorization time of MUMPS to that of WSMP with default options in both. This graph reflects the relative factorization performance of the two softwares that users are likely to observe in their applications.

We give the serial and parallel sparse LU factorization performance of MUMPS and WSMP, including fill-in and operation count statistics, in Table 4. Figure 1 shows bar graphs corresponding to the factorization time of MUMPS normalized with respect to the factorization time WSMP for each matrix. The relative performance of WSMP improves on the Power 3 machine as it is able to extract a higher Megaflops rate from it. WSMP factorization is, on an average, about 2.3 times faster than MUMPS on a single CPU and 2.8 times faster on four CPUs.

The relative performance of sparse LU factorization in MUMPS and WSMP shown in Table 4 and Figure 1 corresponds to what the users are likely to observe in their applications. However, the factorization times are affected by the preprocessing of the matrices, which is different in MUMPS and WSMP. WSMP always uses a row permutation to maximize the product of the magnitudes of the diagonal entries of the matrix. MUMPS does not use such a row permutation for matrices whose nonzero pattern has a significant symmetry to avoid destroying the structural symmetry. Additionaly, WSMP uses a nested dissection based fill-reducing ordering, whereas MUMPS uses the approximate minimum degree (AMD) [Amestoy et al. 1996] algorithm. In order to eliminate the impact of these differences on LU factorization performance, we ran WSMP with AMD ordering and with a selective row permutation logic similar to that in MUMPS.

Figure 2 compares the relative serial and parallel factorization performance of MUMPS with that of the modified version of WSMP. Although, for most matrices, the ratio of MUMPS factor time to WSMP factor time decreases, the overall averages remain more or less the same due to a significant increase in this ratio for the matrix *twotone*. Since both codes are run with similar preprocessing and use the same BLAS libraries for the floating point operations of the factorization process, it would be fair to say that Figure 2 captures the advantage of WSMP's unsymmetric-pattern multifrontal algorithm over MUMPS' symmetric-pattern multifrontal algorithm. Other than the sparse factorization algorithm, there is only one minor difference between the way MUMPS and WSMP are used to collect the performance data for Figure 2. WSMP attempts a decomposition into a block-triangular form, while MUMPS doesn't. However, other than *lhr34c*, this does not play a significant role in determining the factorization performance on the matrices in our test suite. Amestoy and Puglisi [Amestoy and Puglisi 2000] present a mechanism to reduce the symmetrization overhead of the conventional tree-guided multifrontal algorithm used in MUMPS. If incorporated into MUMPS, this mechanism may reduce the performance gap between MUMPS and WSMP on some matrices.

Next, we look at the role of the algorithm used to compute a fill-reducing ordering on the structure of $A + A^T$. Figure 3 compares the LU factorization times of WSMP with AMD and nested dissection ordering. The bars show the factorization time with AMD ordering normalized with respect to a unit factorization time with nested-dissection ordering for each matrix. On the whole, factorization with AMD ordering is roughly one and a half times slower than factorization with WSMP's nested-dissection ordering.

Finally, we observe the impact of a row permutation to maximize the product of the diagonal magnitudes [Duff and Koster 2001; Gupta and Ying 1999] on the factorization performance of WSMP. Figure 4 shows the factorization times of WSMP with this row permutation switched off normalized with respect to the factorization times in the default mode when the row permutation is active. This figure shows that factorization of about 40% of the matrices is unaffected by the row-permutation option. These matrices are probably already diagonally-dominant or close to being diagonally dominant. So the row order does not change and the same matrix is factored, whether the row-permutation option is switched on or not. In

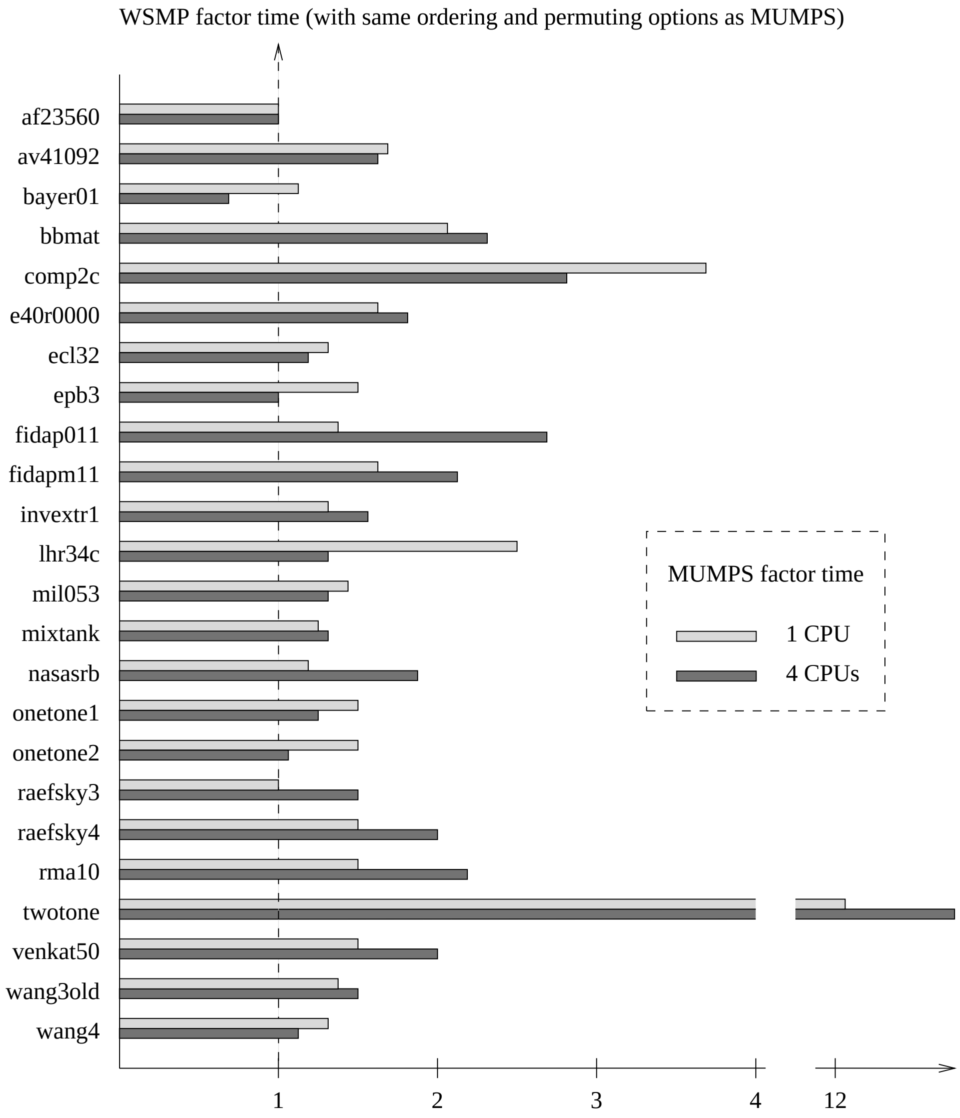

Fig. 2. Ratios of the factorization time of MUMPS to that of WSMP run with the same ordering and row-prepermutation option as MUMPS. This graph enables a fair comparison of the numerical factorization components of the two packages.

a few cases, there is a moderate decline, and in two cases, there is a significant decline in performance as the row permutation option is switched off. On the other hand there are a few cases in which there is a moderate advantage, and there is one matrix, *twotone*, for which there is a significant advantage in switching off the row permutation. This can happen when the original structure of the matrix is symmetric or nearly symmetric and permuting the rows destroys the structural symmetry (although, *twotone* is an exception and is quite unsymmetric). The extra fill-in and

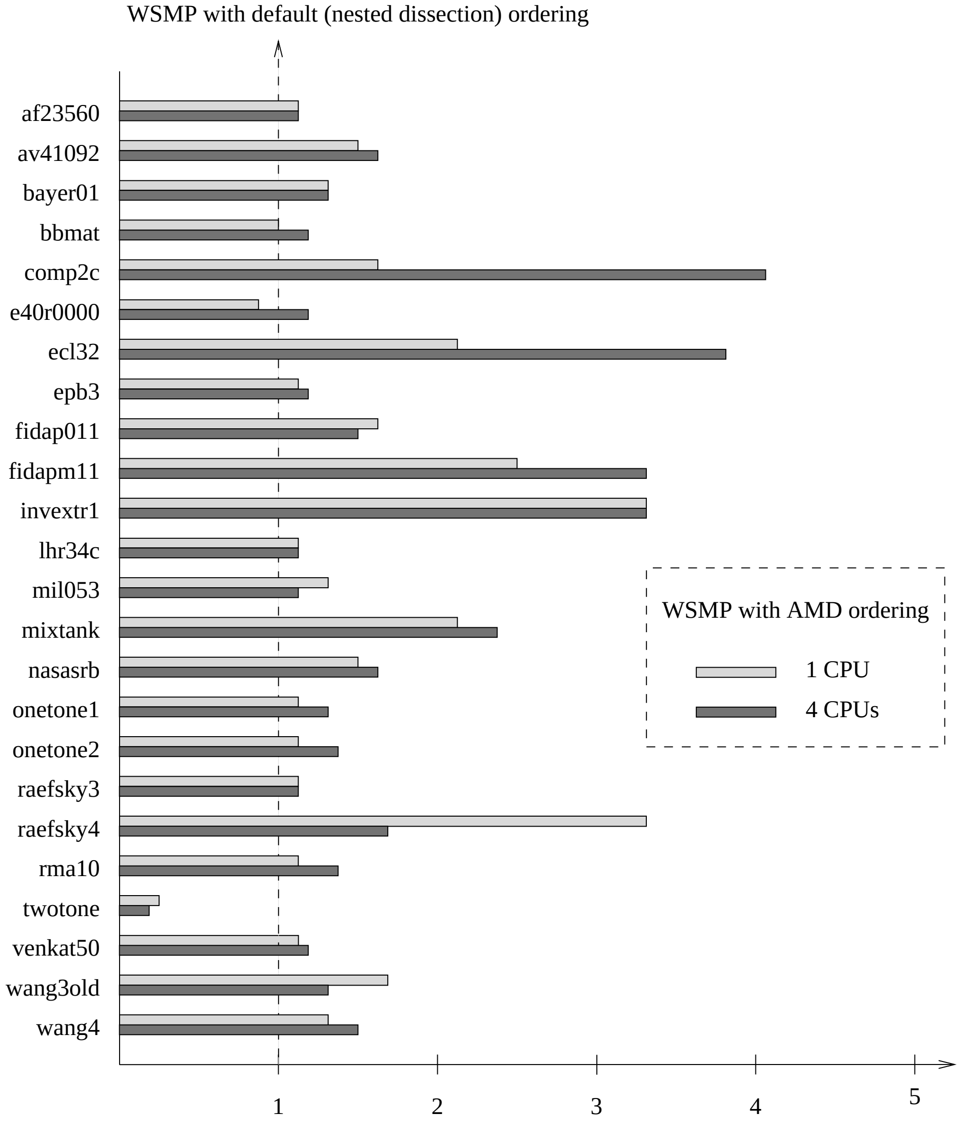

Fig. 3. A comparison of the factorization time of WSMP run with AMD ordering and with its default nested-dissection ordering. The bars correspond to the relative factorization time with AMD ordering compared to the unit time with nested-dissection ordering. This graph shows the role of ordering in WSMP.

computation resulting from the disruption of the original symmetric pattern may more than offset the pivoting advantage, if any, gained by moving large entries to the diagonal. On the whole, it appears that permuting the rows of the matrix to maximize the magnitude of the product is a useful safeguard against excessive fill-in due to pivoting, such as in the case of *raefsky4* and *wang3old*.

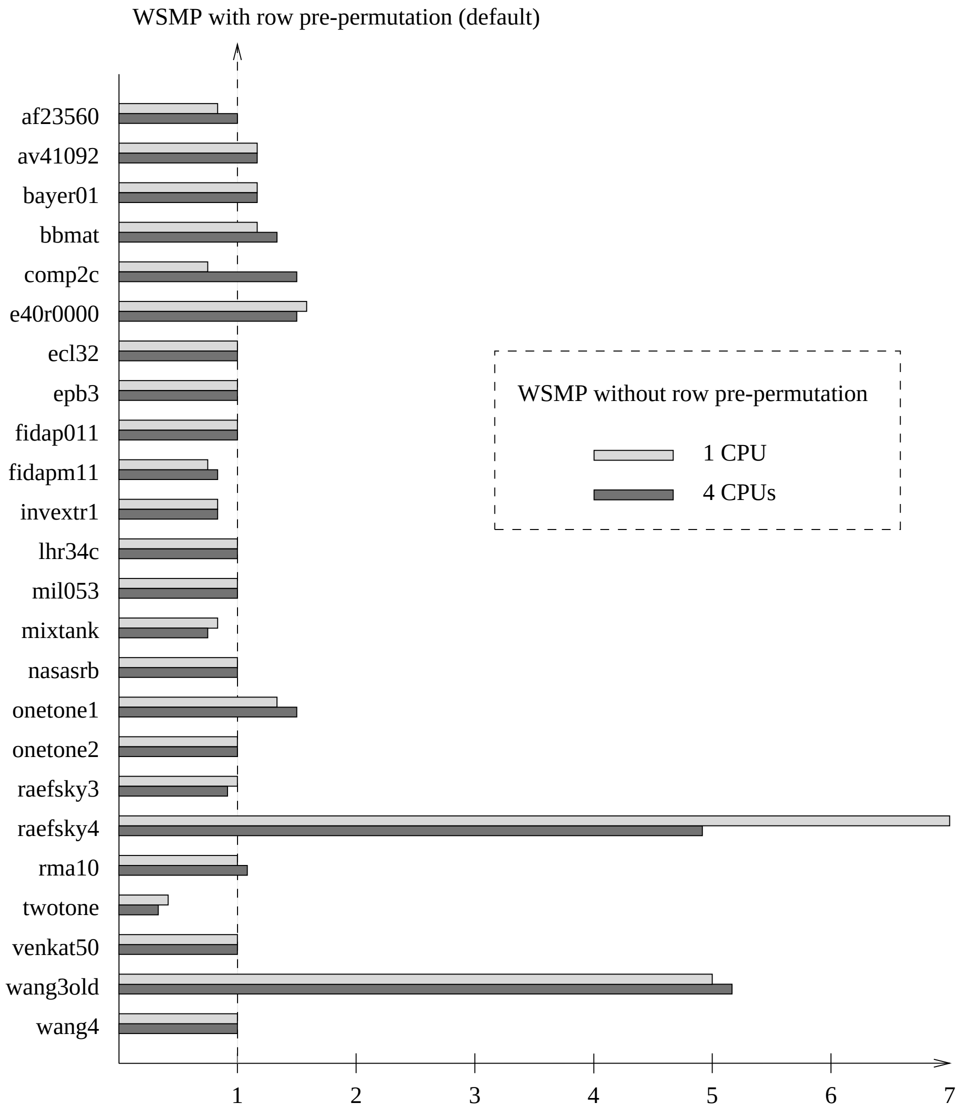

Fig. 4. A comparison of the factorization time of WSMP run without pre-permuting the rows to move matrix entries with relatively large magnitudes to the diagonal. The bars correspond to the relative factorization time without row pre-permutation compared to the unit time for the default option.

5. A PRACTICAL COMPARISON OF MUMPS AND WSMP

In Section 2, we empirically demonstrated that MUMPS and WSMP contain the fastest and the most robust sparse LU factorization codes among the currently available general sparse solvers. In this section, we review the relative performance of these two packages in further detail from the perspective of their use in a real application.

In Table 4, we compared the factorization times of MUMPS and WSMP on one and four CPU's of an RS6000 WH-2 node. A noteworthy observation from this table (the T_4 column of WSMP) is that out of the 25 test cases, only six require more than 5 seconds on a mere workstation and all but one of the matrices can be factored in less than 11 seconds.

In real applications, although factorization time is usually of primary importance, users are concerned about the total completion time, which includes analysis, factorization, triangular solves, and iterative refinement. In Figure 5, we compare the total time that MUMPS and WSMP take to solve our test systems of equations from beginning to end on four CPUs of an RS6000 WH-2 node. For each matrix, the WSMP completion time is considered to be one unit and all other times of both the packages are measured relative to it. The analysis, factorization, and solve times are denoted by bars of different shades. The solve time includes iterative refinement steps necessary to bring the relative backward error down to the order of magnitude of the machine precision.

Two new observations can be made from Figure 5. First, the analysis phase of MUMPS is usually much shorter than that of WSMP. This is not surprising because the AMD ordering algorithm used in MUMPS is much faster than the nested-dissection algorithm used in WSMP. In addition, AMD yields a significant amount of symbolic information about the factors that is available to MUMPS as a byproduct of ordering. On the other hand, WSMP must perform a full separate symbolic factorization step to compute the structures of the factors and the task and data dependency DAGs. Secondly, the solve phase of MUMPS is significantly slower than that of WSMP. This is mostly because of a slower triangular solve in MUMPS, but also partly because MUMPS almost always requires two steps of iterative refinement to reach the desired degree of accuracy, whereas a single iterative refinement step suffices for WSMP for roughly half the problems in our test suite.

A majority of applications of sparse solvers require repeated solutions of systems with gradually changing values of the nonzero coefficients, but the same sparsity pattern. In Figure 6, we compare the performance of MUMPS and WSMP for this important practical scenario. We call the analysis routine of each package once, and then solve 100 systems with the same sparsity pattern. We attempt to emulate a real application situation as follows. After each iteration, 20% randomly chosen coefficients are changed by a random amount between 1 and 20% of their value from the previous iteration, 4% of the coefficients are similarly altered by at most 200% and 1.6% of the coefficients are altered by at most 2000%. The total time that each package spends in the analysis, factor, and solve phases is then used to construct the bar chart in Figure 6. Since the speed of the factorization and solve phases is relatively more important than that of the analysis phase, WSMP performs significantly better, as expected.

Recall that both MUMPS (for very unsymmetric matrices only) and WSMP (for all matrices) permute the rows of the coefficient matrix to maximize the product of the diagonal entries. This permutation is based on the values of the coefficients, which are evolving. Therefore, the row-permutation slowly loses its effectiveness as the iterations proceed. For some matrices, for which the row permutation is not

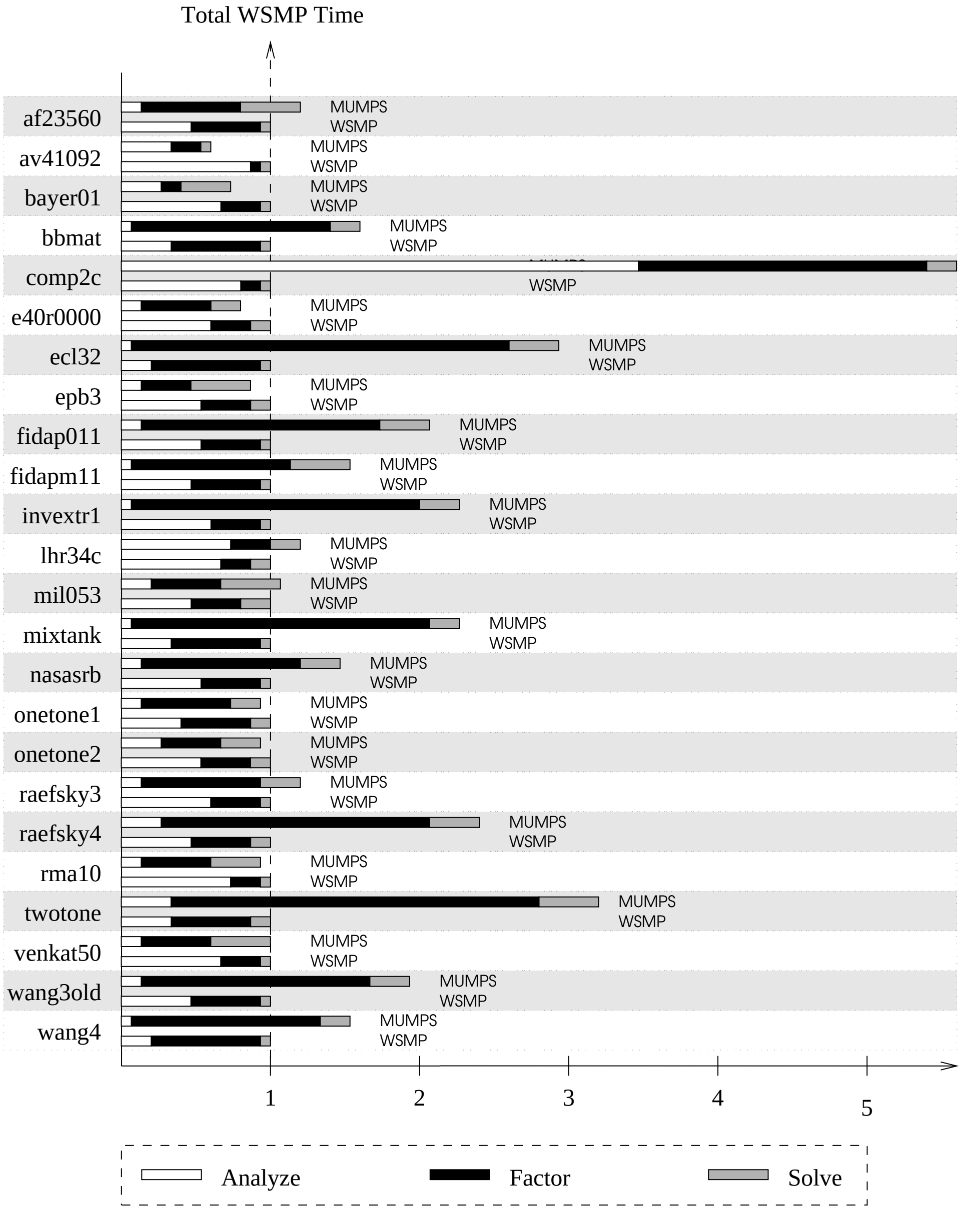

Fig. 5. A comparison of the total time taken by WSMP and MUMPS to solve a system of equations once on four CPUs of an RS6000 WH-2 node. All times are normalized with respect to the time taken by WSMP. Furthermore, the time spent by both packages in the Analysis, Factorization, and Solve (including iterative refinement) phases is denoted by regions with different shades.

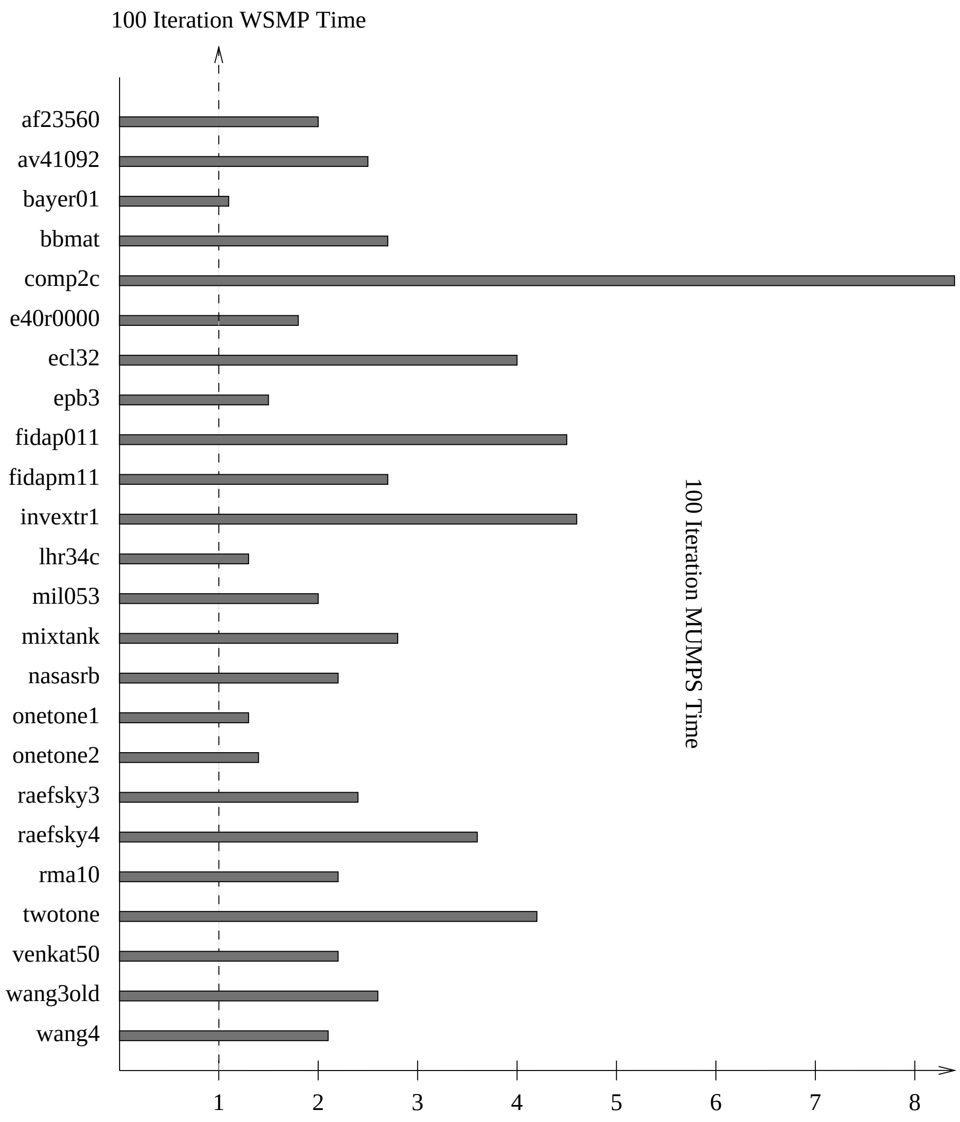

Fig. 6. A comparison of the total time taken by WSMP and MUMPS to solve 100 sparse linear systems with the same nonzero pattern and evolving coefficient values on a 4-CPU RS6000 WH-2 node.

useful anyway (see Figure 4), this does not affect the factorization time. However, for others that rely on row permutation to reduce pivoting, the factorization time may start increasing as the iterations proceed. WSMP internally keeps track of growth in the time of the numerical phases over the iterations and may automatically trigger a re-analysis when it is called to factor a coefficient matrix with a new set of values. The frequency of the re-analysis is determined based on the analysis time relative to the increase in the time of the numerical phases as the iterations proceed. The re-analysis is completely transparent to the user. So although the analysis phase was explicitly called only once in the experiment yielding the data for Figure 6, the actual time reported includes multiple analyses for many matrices. As Figure 6 shows, this does not have a detrimental effect on the overall performance of WSMP because the re-analysis frequency is chosen to optimize the total execution time. Note that a similar reanalysis strategy can also be implemented in MUMPS and due to its small analysis time, may reduce the total time for matrices with unsymmetric nonzero patterns.

6. CONCLUDING REMARKS

In this paper, we show that recent sparse solvers have significantly improved the state of the art of the direct solution of general sparse systems. For instance, compare the first four columns of Table 2 with the second last column of Table 4. This comparison would readily reveal that a state-of-the-art solver running on today's single-user workstation is easily an order of magnitude faster than the best solver-workstation combination available prior to 1999 for solving sparse unsymmetric linear systems. Moreover, the new solvers offer significant scalability of performance that can be utilized to solve these problems even faster on parallel supercomputers [Amestoy et al. 2001]. Therefore, it would be fair to conclude that recent years have seen some remarkable advances in the general sparse direct solver algorithms and software. As discussed in the paper, improvements in all phases of the sparse direct solution process have contributed to these performance gains. These include the use of a maximum bipartite matching to pre-permute large magnitude elements to the matrix diagonal, nested-dissection based fill-reducing permutation applied symmetrically to rows and columns, an unsymmetric pattern multifrontal algorithm using minimal static task- and data-dependency DAGs, and implementing partial pivoting using symmetric row and column interchanges.

ACKNOWLEDGMENTS

The author would like thank Yanto Muliadi for help with conducting the experiments to gather data for Table 2 and the anonymous referees for their detailed comments that helped improve the presentation in the paper.

REFERENCES

Amestoy, P. R., Davis, T. A., and Duff, I. S. 1996. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications 17,* 4, 886–905.

Amestoy, P. R. and Duff, I. S. 1989. Vectorization of a multiprocessor multifrontal code. *International Journal of Supercomputer Applications 3,* 41–59.

Amestoy, P. R. and Duff, I. S. 1993. Memory management issues in sparse multifrontal methods on multiprocessors. *International Journal of Supercomputer Applications 7,* 64–82.

Amestoy, P. R., Duff, I. S., Koster, J., and L'Excellent, J. Y. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications 23,* 1, 15–41.

Amestoy, P. R., Duff, I. S., and L'Excellent, J. Y. 2000. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computational Methods in Applied Mechanical Engineering 184,* 501–520.

Amestoy, P. R., Duff, I. S., L'Excellent, J. Y., and Li, X. S. 2001. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software 27,* 4, 388–421.

AMESTOY, P. R. AND PUGLISI, C. 2000. An unsymmetrized multifrontal LU factorization. Tech. Rep. RT/APO/00/3, ENSEEIHT-IRIT, Toulouse, France. Also available as Technical Report 46474 from Lawrence Berkeley National Laboratory.

ASHCRAFT, C. AND GRIMES, R. G. March 1999. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing.*

ASHCRAFT, C. AND LIU, J. W.-H. 1996. Robust ordering of sparse matrices using multisection. Tech. Rep. CS 96-01, Department of Computer Science, York University, Ontario, Canada.

COSNARD, M. AND GRIGORI, L. 2000. Using postordering and static symbolic factorization for parallel sparse LU. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS).*

DAVIS, T. A. 2002. UMFPACK V3.2: an unsymmetric-pattern multifrontal method with a column pre-ordering strategy. Tech. Rep. TR-02-2002, Computer and Information Sciences Department, University of Florida, Gainesville, FL.

DAVIS, T. A. AND DUFF, I. S. 1997a. A combined unifrontal/multifrontal method or unsymmetric sparse matrices. *ACM Transactions on Mathematical Software 25,* 1, 1–19.

DAVIS, T. A. AND DUFF, I. S. 1997b. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications 18,* 1, 140–158.

DAVIS, T. A., GILBERT, J. R., LARIMORE, S. I., AND NG, E. G.-Y. 2000. A column approximate minimum degree ordering algorithm. Tech. Rep. TR-00-005, Computer and Information Sciences Department, University of Florida, Gainesville, FL.

DEMMEL, J. W., GILBERT, J. R., AND LI, X. S. 1999. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications 20,* 4, 915–952.

DUFF, I. S., ERISMAN, A. M., AND REID, J. K. 1990. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, UK.

DUFF, I. S. AND KOSTER, J. 1999. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications 20,* 4, 889–901.

DUFF, I. S. AND KOSTER, J. 2001. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications 22,* 4, 973–996.

DUFF, I. S. AND REID, J. K. 1984. The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing 5,* 3, 633–641.

DUFF, I. S. AND REID, J. K. 1993. MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations. Tech. Rep. RAL-93-072, Rutherford Appleton Laboratory.

EISENSTAT, S. C. AND LIU, J. W.-H. 1993. Exploiting structural symmetry in a sparse partial pivoting code. *SIAM Journal on Scientific Computing 14,* 1, 253–257.

GEORGE, A. AND LIU, J. W.-H. 1978. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis 15*, 1053–1069.

GEORGE, A. AND LIU, J. W.-H. 1981. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, NJ.

GEORGE, A. AND NG, E. 1985. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM Journal Scientific and Statistical Computing 6,* 2, 390–409.

GILBERT, J. R. AND LIU, J. W.-H. 1993. Elimination structures for unsymmetric sparse LU factors. *SIAM Journal on Matrix Analysis and Applications 14,* 2, 334–352.

GRUND, F. 1998. Direct linear solver for vector and parallel computers. Tech. Rep. Preprint No./ 415, Weierstrass Institute for Applied Analysis and Stochastics.

GUPTA, A. 2001a. A high-performance GEPP-based sparse solver. In *Proceedings of PARCO. http://www.cs.umn.edu/˜agupta/doc/parco-01.ps.*

GUPTA, A. August 1, 2001b. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. Tech. Rep. RC 22137 (99131), IBM T. J. Watson Research Center, Yorktown Heights, NY. Available at *http://www.cs.umn.edu/~agupta/wsmp*; to appear in *SIAM Journal on Matrix Analysis and Applications.*

Gupta, A. January/March, 1997. Fast and effective algorithms for graph partitioning and sparse matrix ordering. *IBM Journal of Research and Development 41,* 1/2, 171–183.

Gupta, A. November 20, 2000. WSMP: Watson sparse matrix package (Part-II: direct solution of general sparse systems). Tech. Rep. RC 21888 (98472), IBM T. J. Watson Research Center, Yorktown Heights, NY. *http://www.cs.umn.edu/~agupta/wsmp.*

Gupta, A. and Muliadi, Y. 2001. An experimental comparison of some direct sparse solver packages. In *Proceedings of International Parallel and Distributed Processing Symposium.*

Gupta, A. and Ying, L. October 19, 1999. On algorithms for finding maximum matchings in bipartite graphs. Tech. Rep. RC 21576 (97320), IBM T. J. Watson Research Center, Yorktown Heights, NY.

Hadfield, S. M. 1994. On the LU factorization of sequences of identically structured sparse matrices within a distributed memory environment. Ph.D. thesis, University of Florida, Gainsville, FL.

HSL. 2000. A collection of Fortran codes for scientific computation. Tech. rep., AEA Technology Engineering Software, Oxfordshire, England. *http://www.cse.clrc.ac.uk/Activity/HSL.*

Karypis, G. and Kumar, V. 1999. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20,* 1.

Li, X. S. and Demmel, J. W. 1998. Making sparse Gaussian elimination scalable by static pivoting. In *Supercomputing '98 Proceedings.*

Li, X. S. and Demmel, J. W. 1999. A scalable sparse direct solver using static pivoting. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing.*

Lipton, R. J., Rose, D. J., and Tarjan, R. E. 1979. Generalized nested dissection. *SIAM Journal on Numerical Analysis 16*, 346–358.

Liu, J. W.-H. 1985. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software 11*, 141–153.

Liu, J. W.-H. 1990. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications 11*, 134–172.

Liu, J. W.-H. 1992. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review 34*, 82–109.

Schenk, O., Fichtner, W., and Gartner, K. November 2000. Scalable parallel sparse LU factorization with a dynamical supernode pivoting approach in semiconductor device simulation. Tech. Rep. 2000/10, Integrated Systems Laboratory, Swiss Federal Institute of Technology, Zurich.

Schenk, O., Gartner, K., Fichtner, W., and Stricker, A. 2000. PARDISO: A high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems 789*, 1–9.

Shen, K., Yang, T., and Jiao, X. 2001. S+: Efficient 2D sparse LU factorization on parallel machines. *SIAM Journal on Matrix Analysis and Applications 22,* 1, 282–305.

Tarjan, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*, 146–160.

Received April 2001; revised September 2001 and March 2002; accepted May 2002.