Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO ⋆

Olaf Schenk [a,b,*] Klaus Gärtner [c]

[a] *Department of Computer Science, University of Basel, Klingelbergstrasse 50, CH-4056 Basel, Switzerland*

[b] *IBM Research Devision, T.J.Watson Research Center, P.O.Box 218, Yorktown Heights, NY, 10598*

[c] *Weierstrass Institute for Applied Analysis and Stochastics, Mohrenstr. 39, D-10117 Berlin, Germany*

Abstract

Supernode partitioning for unsymmetric matrices together with complete block diagonal supernode pivoting and asynchronous computation can achieve high gigaflop rates for parallel sparse LU factorization on shared memory parallel computers. The progress in weighted graph matching algorithms helps to extend these concepts further and unsymmetric prepermutation of rows is used to place large matrix entries on the diagonal. Complete block diagonal supernode pivoting allows dynamical interchanges of columns and rows during the factorization process. The level-3 BLAS efficiency is retained and an advanced two-level left–right looking scheduling scheme results in good speedup on SMP machines. These algorithms have been integrated into the recent unsymmetric version of the PARDISO solver. Experiments demonstrate that a wide set of unsymmetric linear systems can be solved and high performance is consistently achieved for large sparse unsymmetric matrices from real world applications.

Key words: Computational sciences, numerical linear algebra, direct solver, unsymmetric linear systems
PACS: 02.60.Cb, 02.60.Dc
1991 MSC: 65F05, 65F50, 65F35

⋆ This work was supported by the Swiss Commission of Technology and Innovation KTI under contract number 5648.1.

* Corresponding author. Tel. +41-61-2671465, Fax: +41-61-2671461.
Email addresses: olaf.schenk@unibas.ch (Olaf Schenk), gaertner@wias-berlin.de (Klaus Gärtner).
URL: informatik.unibas.ch/personen/schenk_o.html (Olaf Schenk).

1 Introduction

The solution of sparse linear systems $Ax = b$, where A is a general sparse $n \times n$ nonsingular matrix, by parallel sparse direct Gaussian elimination is considered. A general sparse matrix is defined to be a matrix that has no special properties, such as symmetry, positive definiteness, diagonal dominance, etc. These matrices arise frequently in the simulation of semiconductor devices, chemical engineering processes, fluid dynamics, in the analysis of circuits and power systems networks, and elsewhere. The solution of these large sparse linear systems is the main computational bottleneck for all these simulations. When partial pivoting is required to maintain numerical stability in direct methods for solving general sparse linear systems, developing high performance parallel software is challenging because partial pivoting causes the computational task-dependency graph to change during execution.

In [12] new permutations and scaling strategies are introduced for sparse Gaussian elimination. The goal is to preprocess the coefficient matrix A so as to obtain an equivalent system with a matrix that is better scaled and more diagonally dominant. This preprocessing reduces the need for partial pivoting, thereby speeding up the factorization process. Evidence of the usefulness of this preprocessing in connection with sparse direct solvers has been provided in [2,17]. However, the amount of partial pivoting after this unsymmetric preprocessing can be still a limitation for the parallel factorization step and static pivoting with iterative refinement, as proposed in [17], can be an efficient alternative to partial pivoting for parallel sparse Gaussian elimination. A problem with the above static pivoting strategy is that it may result in an incorrect factorization and it is not guaranteed that the iterative refinement will always converge to a correct solution x.

This paper addresses the issues of improved scalability and robustness of sparse direct factorization on shared memory multiprocessing architectures. A key design principle for producing high-performance scalable parallel software is to perform partial pivoting as little as possible, whereas the key design principle for robustness is to allow as much pivoting as possible. A compromise is complete block supernode diagonal pivoting, where rows and columns of a supernode can be interchanged without affecting the computational task-dependency graph. While it is not claimed that this alternative approach to partial pivoting or static pivoting will always work, it is shown that the methods implemented in the PARDISO direct solver [21,22] can be successful. The evidence is provided by numerical experiments for a large set of general sparse matrices from real applications and by numerical performance comparisons with some prominent software packages for solving general sparse systems. The paper compares the serial performance

of SuperLU$_{dist}$ [17], MUMPS [4], UMFPACK 3 [8] and WSMP [14] with PARDISO. This paper also contains a parallel performance comparison of PARDISO and WSMP - the general purpose sparse solver that has been shown in [14] to be the best at the time of PARDISO's release. The experiments indicate that the use of unsymmetric row permutations with complete block diagonal supernode pivoting enables the static computation of the task-dependency graph, resulting in an overall factorization strategy that can be both reliable and cost-effective on shared memory multiprocessing architectures.

The paper is organized as follows. In Section 2 the pivoting method and the two-level left-right looking factorization algorithm of the PARDISO solver for unsymmetric sparse linear systems is discussed. The test problems and the numerical experiments are described in Section 3. Finally, in Section 4, the conclusions are presented.

2 Algorithmic features

In this section the algorithms and strategies that are used in the analysis and numerical phase of the computation of the LU factors are described.

Complete block diagonal supernode pivoting

Figure 1 outlines the approach to solve an unsymmetric sparse linear system of equations. According to [17] it is very beneficial to precede the ordering by performing an unsymmetric permutation to place large entries on the diagonal and then to scale the matrix so that the diagonals entries are equal to one. Therefore, in step (1) the unsymmetric row permutation matrix, P_r, is chosen so as to maximize the absolute value of the product of the diagonal entries in $P_r A$. The code used to perform the permutations is taken from MC64, a set of Fortran routines that are included in HSL (formerly known as Harwell Subroutine Library). Further details on the algorithms and implementations are provided in [12]. The diagonal scaling matrices, D_r and D_c, are selected so that the diagonal entries of $A_1 = D_r P_r A D_c$ are 1 in absolute value and its off-diagonal entries are all less than or equal to 1 in absolute value.

In step (2) any symmetric fill-reducing ordering can be computed based on the structure of $P_r A_1 + A_1^T P_r^T$, e.g. minimum degree or nested dissection. All experiments reported in this paper with PARDISO were conducted with a nested dissection algorithm [16].

Like other modern sparse factorization codes [4,7,10,11,14,20], PARDISO relies heavily on supernodes to efficiently utilize the memory hierarchies in the hardware. There are two main approaches in building these supernode. In the first approach, consecutive rows and columns with the same and exactly identical structure in the factors L and U are treated as one supernode. These supernodes are so crucial to high performance in sparse matrix factorization that the criterion for the inclusion of rows and columns in the same supernode can be relaxed [6] to increase the size of the supernodes. This is the second approach and it is called supernode amalgation. In this approach consecutive rows and columns with nearly the same but not identical structures are included in the same supernode, and artificial nonzero entries with a numerical value of 0 are added to maintain identical row and column structures for all members of a supernode. The rationale is that the slight increase in the number of nonzeros and floating-point operations involved in the factorization can be compensated by a higher factorization speed. Both approaches are possible in PARDISO and the first approach has been used in the remainder of the paper.

An interchange among the rows and columns of a supernode, referred to as complete block diagonal supernode pivoting, has no effect on the overall fill-in and this is the mechanism for finding a suitable pivot in PARDISO. However, there is no guarantee that the numerical factorization algorithm would always succeed in finding a suitable pivot within the supernode block. When the algorithm reaches a point where it cannot factor the supernode based on the previously described supernode pivoting, it uses a pivot perturbation strategy similar to [17]. The magnitude of the potential pivot is tested against a constant threshold of $\alpha = \epsilon \cdot ||A_2||_\infty$, where ϵ is the machine precision and $||A_2||_\infty$ is the ∞-norm of the scaled and permuted matrix A_2. Therefore, in step (3), any tiny pivots encountered during elimination are set to $\text{sign}(l_{ii}) \cdot \epsilon \cdot ||A_2||_\infty$ — this trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice it is observed that the diagonal elements are rarely modified for the large class of matrices that has been used in the numerical experiments.

The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed in step (4). Furthermore, when there are a small number of pivot failures, they corrupt only a low dimensional subspace and each perturbation is a $\text{rank} - 1$ update of A_2, so iterative refinement or a Krylov space method such as conjugate gradient squared (CGS) [23] with the perturbed factors L and U can compensate for such corruption with only a few extra iterations. In the numerical experiments in section 3, a CGS algorithm is used in the cases where iterative refinement does not converge.

One type of algorithm in PARDISO that deserves special mention is the parallel scheduling of the left-right looking factorization algorithm. The idea is to elucidate, somewhat, the details of the dynamic two-level scheduling algorithm as described in [21]. The fundamental goal behind the two-level scheduling algorithm on shared-memory multiprocessors is to avoid synchronization and cache conflicts as often as possible. The current version of PARDISO is designed for the shared-address-space paradigm and uses the OpenMP directives for parallelization.

First, a one-level left-right looking algorithm [22] is described resulting in a parallel high performance implementation on shared memory architectures. The pseudo-code of the procedure is shown in Figure 2. This algorithm exploits parallelism by factoring supernodes in a bottom-up traversal of the supernode elimination tree. A centralized pool of tasks that uses ordering information of the supernodes is computed during the analysis phase. This pool contains all leaf supernodes, followed by the hierarchy of parents of these supernodes. In step (1), a process receives the information 'supernode S is ready for factorization' from the centralized pool of tasks in a first critical region of the pseudo-code. The critical section provides mutual exclusion and only one process is allowed to read the supernode number at a time from the pool of tasks. After computing the factorization of all columns of the supernode in step (2), the process enters a second critical region, in which all parents of supernode S are informed of the completion of supernode S.

It should be pointed out that the one-level left-right looking algorithm with a centralized pool of tasks achieves reasonable speedup on SMPs for large sparse matrices where the total factorization time is in the order of 100 seconds or more. Practical experimental evidence has been provided in [22] for various shared memory high-performance multiprocessing architectures. The main limitations of the algorithm show up for small, sparse matrices.

Two main technical reasons limit the SMP scalability for these smaller matrices. The first reason is that ratio of translation lookaside buffer (TLB) misses to supernode factorization time is higher for these smaller matrices if several processors are involved in the factorization process. The TLB contains a finite set of pages which are known as the current working set of the computation. If the computation addresses only memory in the TLB, there is no penalty [1]. Otherwise, a TLB miss occurs, resulting in a large performance penalty. In the one-level algorithm e.g. two leaf supernodes that are direct siblings can be factorized by two different processors. These leaf supernodes are stored contiguously in main memory. If the two proces-

sors are factorizing both supernodes at the same time, then a high number of cache conflicts will occur when these processors write the results back to memory resulting in a high number of TLB misses during the parallel factorization.

The second reason is the amount of synchronization events due to the scheduling of supernodes instead of completely independent subtrees.

Now the two reasons of inefficiency of the preceding one-level method is cured. The two-level scheduling consists in a first level of a dynamic process-to-subtree scheduling. In this level each process computes the complete associated submatrix until the complete submatrix is factorized. In a second level, a dynamic process-to-root-supernode scheduling is being applied, where several processes can factorized one root-supernode by the same time. The processes at this level of the factorization try to compute as much external updates as possible. If some descendants are not yet finished, than the processes will try to compute parts of the factorization of other root supernodes. It is important to note that it is not necessary that all submatrices are completely factorized before a process starts with the process-to-root-supernode scheduling. The primary advantage of this method is that a reasonable dynamic load balancing is achieved at the important end of the factorization process.

The pseudo-code of the algorithm is outlined in Figure 3. In the first level, the parallel structure of the elimination tree is exploited. This elimination tree structure is computed during the preprocessing phase and a set of independent subtrees is dynamically distributed to available processes. Each process factors in step (1) a complete subtree, resulting in significant reduction of TLB misses and parallel synchronization events. This is also called process-to-subtree scheduling because all supernodes in one subtree are computed by one process.

In contrast to the process-to-subtree scheduling, the process-to-root-supernode scheduling is a dynamic one and one root supernode can be factorized by several processes simultaneously. Most of the synchronization events of the two-level left-right looking algorithm occur in the root supernodes and the number of TLB misses and synchronization events is in general significantly smaller compared to the one-level algorithm. Experimental results are given in Section 3, where it is shown that good speedups can also be obtained for small matrices.

3 Experimental results

In this section the test matrices and the numerical experiments are described. The names and the sizes of the unsymmetric test matrices that are used in the experiments are shown in Table 1. Most of these matrices are in the public domain [9,19] and freely available. Further, all matrices were generated by real world applications. The table also contains the dimension, the number of nonzeros, and the related application area. These matrices are representative of problems from a variety of applications, but they share the property that they are difficult to solve with a direct method without any pivoting or preorderings.

The numerical experiments were performed on one and eight processor IBM 1 GHz Power 3 machines. In these machines all processors have 32 KB level-1 caches and two-processor shared 1.4 MB level-2 caches. All solver packages are compiled in 32-bit mode with the -O3 optimization option of the AIX Fortran or C compilers and are linked with the IBM's Engineering and Scientific Subroutine Library (ESSL) for the basic linear algebra subprograms (BLAS) that are optimized for RS6000 processors.

3.1 Numerical accuracy

The numerical behavior of the complete block diagonal supernode pivoting method in PARDISO is illustrated in Table 2. In all cases, an artificial right-hand side b was used in the runs, so that the system $Ax = b$ had the known solution $x = (x_i)$ with $x_i = 1, 1 \leq i \leq n$. Runs were also performed with other choices of b, with similar results. The iterative refinement was stopped when the componentwise relative backward error, Berr = $\max_i \frac{|Ax-b|_i}{(|A|\cdot|x|+|b|)_i}$, [5] was close to machine precision or when Berr does not converge at least by a factor of 2 during one iteration. Table 2 shows the number of steps of iterative refinement or conjugate gradient square iterations for static pivoting and complete block diagonal supernode pivoting in PARDISO with default options. The true error, Err=$||x_{true} - x||_\infty$, the backward error, Berr, and the numbers of perturbed pivots are also reported.

Some comments on the results in Table 2 are in order. First, one of the ground rules for the experiments was that all input parameters of PARDISO were fixed and not modified to accommodate the demands of individual matrices. These input parameters are referred to as PARDISO's default options. These options were determined during a series of pre-experiments in an attempt to fix the parameters (scaling, preordering, pivot failures) for unsymmetric sparse linear systems. For the numerical experiments with

PARDISO, all unsymmetric matrices are scaled and preordered with the MC64 subroutines, pivot failures are handled as described in Section 2, and nested dissection is used.

Secondly, block diagonal supernode pivoting in PARDISO is performed with the LAPACK DGETC2 subroutine. The DGETC2 routine computes an LU factorization with complete pivoting of an $n_s \times n_s$ matrix, where n_s is the number of columns in a supernode. Static pivoting is obtained by changing the DGETC2 subroutine to an LU factorization subroutine with diagonal pivoting. While this static pivoting in PARDISO leads to nonconvergence of iterative refinement for six matrices, the block diagonal supernode pivoting results in backward error convergence for 24 out of 25 matrices. The matrix av41092 can be solved with other non-default solver options but for the experiments we chose to report results with a consistent fixed set of options.

Finally, it should be noted that the number of perturbed pivots has a great influence on the number of iterative refinement steps. A higher number of perturbed pivots resulted, in general, in incorrect factors L and U with an increase in the number of refinement or conjugate gradient square iterations.

3.2 Serial performance of some general sparse solvers

Table 3 and 4 list the performance numbers of some state-of-the art packages for solving large sparse systems of linear equations on a single IBM Power 3 processor. This table is shown to contrast the performance of PARDISO with other well-known software packages. The sparse solvers compared in this section are SuperLU$_{dist}$ [17], MUMPS 4.1.6 [4], UMFPACK 3 [8], WSMP [14] and PARDISO. The packages SuperLU$_{dist}$ and MUMPS 4.1.6 are designed for distributed memory computers using MPI, whereas the target architecture for WSMP and PARDISO is a shared memory system using Pthreads or OpenMP, respectively, and UMFPACK is a sequential code.

The default options for the different solvers are those reported in [14]. The symbol F_N indicates a failure of iterative refinement. Some of the failures reported in the table can be fixed by changing the default options in the code. However, as noted above, the options used to run the experiments in Table 3 and 4 are such that they are best for the test suite as a whole.

The smallest pivoting threshold for MUMPS, UMFPACK and WSMP has been chosen so that all completed factorizations yielded a backward error that is close to machine precision. As a result, the threshold value is 0.01

for MUMPS and WSMP, and 0.1 for UMFPACK. SuperLU$_{dist}$ does not have an option for partial pivoting since it is significantly more complex to implement numerical pivoting on distributed memory architecture, hence the threshold is indicated as static pivoting. PARDISO uses complete block diagonal pivoting; hence the threshold is indicated as supernode pivoting.

In addition to the pivoting approach used, there are other significant differences between the solvers. By default, PARDISO and SuperLU$_{dist}$ use the maximal matching algorithm [12] to maximize the product of the magnitudes of the diagonal entries for all matrices. MUMPS uses it only if the structural symmetry in the original matrix is less than 50%, WSMP uses a similar preprocessing [15] only on matrices if the structural symmetry is less than 80%, and UMFPACK 3 does not use it at all. Secondly, by default, SuperLU$_{dist}$, WSMP, MUMPS and PARDISO use a symmetric permutation computed on the structure of $A + A^T$. SuperLU$_{dist}$ uses the multiple minimum degree [18], WSMP and PARDISO use a nested dissection ordering [13,16], and MUMPS an approximate minimum degree algorithm [3]. UMFPACK uses a column approximate minimum degree algorithm to compute a fill-in reducing preordering [8]. The third difference is that WSMP is the only solver that reduces the coefficient matrix into a block triangular form, while all other solver do not.

Note from Table 3 that the partial pivoting solvers (WSMP, UMFPACK 3, MUMPS) can reduce the backward error of all matrices to machine precision, whereas SuperLU$_{dist}$ and PARDISO cannot. However, is seems that PARDISO can solve in general more matrices than SuperLU$_{dist}$ due to the complete block diagonal supernode pivoting method. From the numerical experiments summarized in Table 3 it appears that WSMP has the smallest overall factorization time with the solver default options, whereas PARDISO has the second smallest overall factorization time.

Table 4 compares the analysis phase and the solution time for an iterative refinement process that yields a backward error close to machine precision for WSMP, MUMPS and PARDISO. Two observations can be drawn from the table. First, it can be observed that the analysis phase of MUMPS is usually much shorter compared to that of WSMP and PARDISO [1] . The reason is clear because the approximate minimum degree variant used in MUMPS is significantly faster than their multilevel nested dissection counterparts used in WSMP and PARDISO. Furthermore MUMPS can reuse a significant amount of information as a byproduct of the minimum degree ordering process whereas WSMP and PARDISO must perform a full separate symbolic factorization to compute the structure of the factors. Secondly, it can

[1] Timing results for SuperLU are omitted since a detailed comparison between SuperLU and MUMPS is also given in [2].

be observed that the restricted supernode block diagonal pivoting method in PARDISO can lead to a higher number of iterative refinement steps to reach the desired degree of accuracy compared with the other solvers.

Finally, Table 3 also compares the number of nonzeros in the factors for each solver. A noteworthy observation from this table is that multilevel nested dissection leads in general to a smaller number of nonzeros, thus resulting in a faster factorization process at the cost of a higher, more time consuming analysis phase. It is also very important to note that the ordering algorithms employed in the different solvers greatly affects the numerical factorization. Unfortunately, it is not possible to run the factorization for all solvers with exactly the same ordering since the analysis phase and the factorization are tightly coupled e.g. in WSMP. WSMP's algorithms work only with a permutation to a block triangular form, which is not implemented in the other solvers. Furthermore, the solver interface of the unsymmetric solver WSMP does not allow the user to specify a special ordering that should be used during the factorization. In order to eliminate the impact of these different orderings Table 5 shows the Gflop/s [2] for each solver package. Although difference other than the factorization algorithm itself greatly affect the performance, it is easy to see the broad picture that emerge from the table. Nearly all of the boldface and most of the underlined entries are in the MUMPS, WSMP and PARDISO's columns.

However, due to the ordering and and the reduction into a block triangular form that is employed into WSMP, the solver has in general the smallest factorization time in seconds among all five solvers on a sequential architecture. WSMP needs, in most of the examples, fewer operations than PARDISO does and it seems that the algorithm based on [13] produces orderings with a smaller fill-in compared to [16], which is used in PARDISO.

3.3 Parallel performance

For the parallel performance and scalability, the LU factorization of PARDISO is compared with that of WSMP in Table 6. WSMP uses the Pthreads library and PARDISO uses the OpenMP parallel directives. In contrast to SuperLU$_{dist}$ and MUMPS, WSMP and PARDISO are designed for a shared-address-space paradigm and no additional constraints and overhead may occur if these similar two solvers are compared in parallel. The solver WSMP was run in parallel by setting the environment variable SPINLOOPTIME and YIELDLOOPTIME to '500'.

[2] Gflop/s is defined by operation count divided by the time for the numerical factorization.

The observation that can be drawn from the table is that the factorization times are affected by the preprocessing and WSMP is, in most cases, faster on a single Power 3 processor. However, the two-level scheduling in PARDISO provides better scalability and hence better performance with eight Power 3 processors.

4 Concluding remarks

The experiments presented in this paper indicate that complete block diagonal supernode pivoting in conjunction with the two-level scheduling method achieves high gigaflop rates and scalability for parallel sparse unsymmetric LU factorization on shared memory parallel computers. As discussed in the paper, complete block diagonal pivoting can be an alternative to partial or static pivoting for some unsymmetric matrices. The primary advantages is a static dependency of tasks that yields a scalability improvement on SMPs.

The focus of the experimental comparison is mainly on the WSMP and PARDISO packages and their different approaches. The robustness of a partial pivoting method and the dynamic directed acyclic task dependency graphs in WSMP and a complete block diagonal supernode pivoting with an undirected graph partitioning in PARDISO, where the efficient dynamic scheduling on pre-computed communication graphs results in better speedup. The experiments also demonstrate that consistently high-performance is achieved by WSMP and PARDISO when compared to other well-known packages.

For different application areas the PARDISO approach results in sufficient robustness. Further improvements can be expected with better reordering algorithms to reduce the operation count of PARDISO, and by the integration of partial threshold pivoting for matrices where pivoting among supernode rows and columns combined with perturbed pivots is not sufficient . The different techniques used in WSMP and PARDISO may stimulate further improvements but it may be hard to reach the robustness and the minimal operation counts of WSMP for unsymmetric matrices and the better scalability of PARDISO on SMPs.

Acknowledgments

The authors thank the anonymous referees for the useful comments.

The authors also wish to thank Anshul Gupta, IBM T.J. Watson Research Center, for providing his large benchmark set of unsymmetric matrices, and Iain Duff and Jacko Koster for the opportunity to use the MC64 graph matching subroutines. Parts of this work were performed while the first author was an academic visiting scientist at the IBM T.J. Watson Research Center: The hospitality and support of IBM are greatly appreciated.

References

[1] R.C. Agarwal, F.G.Gustavson, and M.Zubair. Exploiting functional parallelism of power2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5):563–576, September 1994.

[2] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 27(4):338–421, 2002.

[3] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis and Applications*, 17(4):886–905, 1996.

[4] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Analysis and Applications*, 23(1):15–41, 2001.

[5] Mario Arioli, James W. Demmel, and Iain S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Analysis and Applications*, 10:165–190, 1989.

[6] C.C. Ashcraft and R.G. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.

[7] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Analysis and Applications*, 18(1):140–158, 1997.

[8] Timothy A. Davis. UMFPACK V3.2: an unsymmetric-pattern multifrontal methods with a column pre-orderiung strategy. Technical Report TR-02-2002, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 2002.

[9] Timothy A. Davis. University of Florida Sparse Matrix Collection, University of Florida, Gainesville, FL. Available online from http://www.cise.ufl.edu/d̃avis/sparse.

[10] J. Demmel, J. Gilbert, and X. Li. An asynchronous parallel supernodal algorithm to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999.

[11] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W.-H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[12] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Analysis and Applications*, 20(4):889–901, 1999.

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

[14] A. Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Transactions on Mathematical Software*, 28(3):301–324, 2002.

[15] A. Gupta and L. Ying. On algorithms for finding maximum matchings in bipartite graphs. Technical Report RC 21576 (97320), IBM T. J. Watson Research Center, Yorktown Heights, NY, October 25, 1999.

[16] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[17] X.S. Li and J.W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceeding of the 9th SIAM conference on Parallel Processing for Scientic Computing*, San Antonio, Texas, March 22-34,1999.

[18] J.W.H. Liu. Modification of the Minimum-Degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985.

[19] Matrix Market. National Institute of Standards and Technology, Gaithersburg, MD. Available online from http://math.nist.gov/MatrixMarket.

[20] E.G. Ng and B.W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, 14:1034–1056, 1993.

[21] O. Schenk and K. Gärtner. Two-level scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems. *Parallel Computing*, 28:187–197, 2002.

[22] O. Schenk, K. Gärtner, and W. Fichtner. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT*, 40(1):158–176, 2000.

[23] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 10:36–52, 1989.

(1) Row/column equilibration $A_1 \leftarrow D_r \cdot P_r \cdot A \cdot D_c$, where D_r and D_c are diagonal matrices and P_r is a row permutation that maximizes the magnitude of the diagonal entries.

(2) Find a symmetric permutation P_{fill} to preserve sparsity: $A_2 \leftarrow P_{fill} \cdot A_1 \cdot P_{fill}^T$ and P_{fill} based on $\hat{A} = A_1 + A_1^T$.

(3) Level-3 BLAS factorization $A_2 = Q_r L U Q_c$ with diagonal block supernode pivoting permutations Q_r and Q_c. The growth of diagonal elements is controlled with:

    if $(|l_{ii}| < \epsilon \cdot ||A_2||_\infty)$ **then**

        set $l_{ii} = sign(l_{ii}) \cdot \epsilon \cdot ||A_2||_\infty$

    endif

(4) Solve $Ax = b$ using the block L and U factors, the permutation matrices P_{fill}, P_r, Q_r and Q_c and iterative refinement.

(5) If iterative refinement fails, try CGS with the perturbed factors L and U as a preconditioner.

Fig. 1. Pseudo-code of the complete block diagonal supernode pivoting algorithm for general unsymmetric sparse matrices.

parallel for node = 1, numbers of supernodes

(1) **synchronization:** get node S from a centralized pool of task that is initialized with supernodes in a bottom-up traversal of the supernode elimination tree

(2) **factorization:** computation of L_s and U_s with a left-looking factorization algorithm

(3) **synchronization:** inform all supernode parents of node S with a right-looking algorithm that L_s and U_s are factorized

end for

Fig. 2. Pseudo-code of the one-level left-right looking algorithm.

```
parallel for subtree = 1, numbers of independent subtrees
(1)     factorization: perform one-level, left-right looking algorithm
        on the subtree with synchronizations only in the root nodes
        of the elimination tree
end for
parallel for root node = 1, numbers of root supernodes
(2)     factorization: perform a dynamic one-level, left-right looking
        algorithm on the root nodes of the elimination tree
end for
```

Fig. 3. Pseudo-code of the two-level left-right looking algorithm.

Table 1
Unsymmetric test matrices with their order (N), number of nonzeros (NNZ), and the application area of origin.

Number	Matrix	N	NNZ	Application
1	af23560	23560	484256	Fluid dynamics
2	av41092	41092	1683902	Finite element analysis
3	bayer01	57735	277774	Chemistry
4	bbmat	38744	1771722	Fluid dynamics
5	comp2c	16783	578665	Linear programing
6	e40r0000	17281	553956	Fluid dynamics
7	e40r1000	17281	553956	Fluid dynamics
8	e40r5000	17281	553956	Fluid dynamics
9	ecl32	51993	380415	Circuit Simulation
10	epb3	84617	463625	Thermodynamics
11	fidap011	16614	1091362	Fluid dynamics
12	fidapm11	22294	623554	Fluid dynamics
13	lhr34c	35152	764014	Chemical engineering
14	invextr1	30412	1793881	Fluid dynamcis
15	mixtank	29957	1995041	Fluid dynamics
16	nasarb	54870	2677324	Structural engineering
17	onetone1	36057	341088	Circuit Simulation
18	onetone2	36057	227628	Circuit Simulation
19	raefsky3	21200	1488768	Fluid dynamics
20	raefsky4	19779	1316789	Fluid dynamics
21	tib	18510	1451491	Circuit simulation
22	twotone	120750	1224224	Circuit simulation
23	venkat50	62424	1717792	Structural engineering
24	wang3-old	26064	177168	Semicond. dev. simulation
25	wang4	26068	177196	Semicond. dev. simulation

Table 2
The numerical behavior of the PARDISO solver (default options) with static pivoting and complete block diagonal supernode pivoting. #ref indicates the number of steps of iterative refinement, Err the error, Berr the backward error, and #piv the number of perturbed pivots. A * after the matrix name indicates that supernode pivoting is necessary in PARDISO to obtain convergence and CGS indicates that a conjugate gradient square algorithm has been used to improve the solution. F_N indicates a failure of the method to factor a matrix satisfactorily.

Matrices	static pivoting				block supernode pivoting			
	#ref	Err	Berr	#piv	#ref	Err	Berr	#piv
av23560	2	4.0e-13	3.3e-16	0	1	2.7e-13	2.7e-16	0
av41092	F_N	F_N	F_N	F_N	F_N	F_N	F_N	F_N
bayer01	3	1.1e-05	3.3e-16	0	1	1.1e-05	6.1e-16	0
bbmat	4	8.3e-09	8.4e-09	2	4	8.4e-09	4.8e-16	2
comp2c	3	3.6e-05	1.7e-16	0	2	5.7e-05	4.7e-16	0
e40r0000*	F_N	F_N	F_N	F_N	9	3.6e-10	1.2e-16	6
e40r1000	2	1.6e-10	1.9e-16	0	2	1.6e-10	1.9e-16	0
e40r5000*	F_N	F_N	F_N	F_N	2	6.2e-10	2.7e-16	7
ecl32	2	4.1e-10	5.7e-16	0	2	4.1e-10	5.7e-16	0
epb3	2	1.2e-11	3.5e-16	0	2	1.1e-11	3.4e-16	0
fidap011	1	1.5e-05	6.0e-16	0	1	1.5e-05	6.7e-16	0
fidapm11*	F_N	F_N	F_N	F_N	3 CGS	1.3e-05	6.2e-16	15
lhr34c*	F_N	F_N	F_N	F_N	7	4.1e-02	7.9e-16	1
invextr1	9	1.4e-03	6.7e-16	4	9	1.4e-03	6.7e-16	4
mixtank*	F_N	F_N	F_N	F_N	6	2.4e-05	2.7e-16	12
nasarb	1	2.4e-09	1.4e-16	0	1	4.8e-09	5.1e-16	0
onetone1	1	1.6e-10	1.4e-16	0	1	1.6e-10	1.4e-16	0
onetone2	1	8.0e-11	4.7e-16	0	1	7.7e-11	4.7e-16	0
raefsky3	1	3.0e-10	3.9e-16	0	1	3.6e-12	3.9e-16	0
raefsky4	1	3.0e-11	5.7e-16	0	1	1.3e-11	5.7e-16	0
tib	3	1.5e-11	6.7e-16	1	3	1.5e-11	6.7e-16	1
twotone	3	1.1e-10	1.5e-16	0	3	1.0e-10	1.2e-16	2
venkat50	1	9.0e-12	7.7e-16	0	1	1.3e-12	7.7e-16	0
wang3-old	1	9.3e-12	4.1e-16	0	1	2.4e-12	4.1e-16	0
wang4	1	9.3e-11	6.1e-16	0	1	2.4e-11	6.3e-16	0

Table 3
LU factorization times and operation count (ops) for the numerical factorization. The best time and operation count are shown in boldface, the second best time and operation count are underlined. The last row shows the pivot threshold that yielded a residual norm close to machine precision. F_N indicates that the numerical results were inaccurate with default options.

Matrices	SuperLU$_{dist}$		MUMPS		UMFPACK 3		WSMP		PARDISO	
	time (sec.)	ops $\times 10^9$	time (sec.)	ops $\times 10^9$	time (sec.)	ops $\times 10^9$	time (sec.)	ops $\times 10^9$	time (sec.)	ops $\times 10^9$
af23560	4.09	4.94	**1.62**	**2.55**	2.69	3.45	1.72	3.01	1.89	3.33
av41092	F_N	F_N	4.95	8.41	45.5	37.0	**1.84**	**1.72**	F_N	F_N
bayer01	0.93	0.03	0.50	0.12	0.45	**0.02**	0.38	0.41	**0.32**	0.12
bbmat	76.2	24.5	20.5	41.3	28.2	39.0	**10.6**	**20.3**	20.3	27.9
comp2c	13.2	**0.44**	4.25	4.32	143.	113	**0.99**	0.51	3.15	2.09
e40r0000	0.38	0.25	0.32	**0.17**	1.89	2.16	**0.22**	0.22	0.26	0.21
e40r1000	0.39	0.26	0.32	**0.17**	1.91	2.11	**0.22**	0.25	0.26	0.26
e40r5000	F_N	F_N	**0.32**	**0.17**	2.01	2.09	0.37	0.31	0.37	0.45
ecl32	53.1	73.8	25.1	64.5	69.4	111.	**11.2**	22.6	12.1	**22.4**
epb3	1.57	0.65	1.20	1.16	1.91	1.33	**0.69**	**0.41**	0.80	0.44
fidap011	F_N	F_N	3.72	7.00	5.53	8.50	**1.77**	**2.96**	2.04	3.65
fidapm11	F_N	F_N	4.96	9.66	13.5	20.0	**2.18**	**3.13**	4.76	8.93
lhr34c	3.33	0.24	0.92	0.64	0.90	0.17	**0.44**	**0.15**	0.83	0.53
invextr1	33.7	37.0	15.5	35.6	64.2	8.93	**3.82**	**5.35**	7.18	12.3
mixtank	49.7	79.7	24.7	64.4	138.	242.	**7.78**	**14.5**	9.29	**17.6**
nasarb	7.04	10.2	5.66	9.41	18.0	28.1	**3.36**	**5.60**	4.08	7.02
onetone1	3.15	**1.34**	1.55	2.28	1.93	2.33	**1.24**	1.40	1.79	2.21
onetone2	0.99	0.23	0.52	0.51	**0.32**	**0.08**	0.34	0.19	0.44	0.43
raefsky3	1.84	2.68	1.82	2.90	5.14	7.87	**1.35**	**2.35**	1.36	2.63
raefsky4	7.84	10.5	5.28	10.9	8.03	12.8	**2.27**	**4.12**	2.56	4.75
tib	0.44	0.03	0.15	0.04	5.41	0.20	0.13	0.04	**0.11**	0.03
twotone	224.	**4.83**	15.1	29.3	10.8	10.7	**4.99**	5.94	24.7	37.4
venkat50	2.16	2.27	1.95	2.31	3.78	4.03	**1.35**	**1.80**	1.41	1.83
wang3-old	9.74	14.1	5.86	13.7	15.3	24.2	3.60	6.49	**2.81**	**4.58**
wang4	6.09	8.77	4.69	10.5	19.2	30.7	3.26	6.22	**1.68**	**3.02**
Thresh	static		0.01		0.1		0.01		supernode	

Table 4
Time in seconds for analysis and forward/backward substitutions including iterative refinement on a single 1 GHz IBM power 3 processor for MUMPS, WSMP, and PARDISO, respectively. The number of iterative refinement steps is shown in parentheses.

	MUMPS		WSMP		PARDISO	
Matrices	Analysis	Solve	Analysis	Solve	Analysis	Solve
af23560	0.37	1.24 (1)	1.24	0.21 (2)	0.97	0.22 (1)
av41092	7.23	1.04 (3)	21.2	0.64 (2)	F_N	F_N
bayer01	0.71	0.48 (2)	2.30	0.43 (2)	1.25	0.29 (1)
bbmat	0.96	1.31 (1)	4.78	0.71 (2)	4.93	1.34 (4)
comp2c	5.54	0.47 (2)	1.90	0.11 (1)	3.58	0.35 (2)
e40r0000	0.30	0.13 (1)	0.99	0.08 (1)	0.56	0.30 (9)
e40r1000	0.30	0.13 (1)	1.26	0.08 (1)	0.60	0.08 (2)
e40r5000	0.30	0.18 (1)	1.70	0.09 (1)	1.29	0.24 (2)
ecl32	0.76	1.85 (2)	2.36	0.67 (2)	1.72	0.64 (2)
epb3	0.41	0.26 (1)	1.98	0.62 (2)	1.32	0.31 (2)
fidap011	0.35	0.37 (2)	1.69	0.20 (1)	1.44	0.28 (1)
fidapm11	0.35	0.61 (1)	1.65	0.24 (1)	2.45	1.20 (3 CGS)
lhr34c	1.37	0.45 (2)	2.35	0.39 (2)	2.67	0.30 (7)
invextr1	1.60	0.37 (1)	3.86	0.84 (2)	6.69	1.60 (6)
mixtank	0.96	2.30 (1)	4.70	0.98 (1)	3.34	3.40 (6)
nasarb	0.69	1.42 (1)	4.92	0.89 (1)	3.42	0.40 (1)
onetone1	0.88	0.34 (2)	2.22	0.51 (2)	1.32	0.26 (1)
onetone2	0.40	0.26 (2)	1.45	0.36 (1)	0.93	0.25 (1)
raefsky3	0.36	0.49 (1)	2.50	0.37 (1)	1.41	0.37 (1)
raefsky4	0.85	0.59 (1)	2.64	0.37 (1)	1.59	0.37 (1)
tib	1.46	0.14 (2)	0.50	0.13 (3)	0.79	0.13 (3)
twotone	1.37	0.63 (1)	5.77	1.11 (2)	6.13	1.63 (3)
venkat50	0.77	0.69 (2)	3.85	0.49 (1)	1.75	0.89 (1)
wang3-old	0.41	0.54 (2)	1.22	0.50 (2)	0.64	0.38 (1)
wang4	0.23	0.49 (2)	1.30	0.53 (2)	0.66	0.46 (1)

Table 5
Gflop/s rate for the LU factorization for each solver on one 1 GHz IBM Power 3 processors. The best Gflop/s rate is shown in boldface, the second best Gflop/s rate is underlined. The symbol F_N indicates a failure of iterative refinement.

Matrices	SuperLU$_{dist}$	MUMPS	UMFPACK 3	WSMP	PARDISO
af23560	1.20	1.57	1.28	1.75	**1.76**
av41092	F_N	**1.69**	0.81	0.93	F_N
bayer01	0.03	0.24	0.04	**1.07**	0.37
bbmat	0.32	**2.01**	1.38	1.91	1.37
comp2c	0.03	**1.01**	0.79	0.51	0.66
e40r0000	0.65	0.53	**1.14**	1.00	0.80
e40r1000	0.66	0.53	1.10	**1.13**	1.00
e40r5000	F_N	0.53	1.03	0.83	**1.21**
ecl32	1.38	**2.56**	1.59	2.01	1.85
epb3	0.41	**0.96**	0.69	0.59	0.55
fidap011	F_N	**1.88**	1.53	1.67	1.78
fidapm11	F_N	**1.94**	1.48	1.43	1.87
lhr34c	0.07	**0.69**	0.18	0.34	0.63
invextr1	1.09	**2.29**	0.13	1.40	1.71
mixtank	1.60	**2.60**	1.75	1.86	1.89
nasarb	1.44	1.66	1.56	1.66	**1.72**
onetone1	0.42	**1.47**	1.20	1.12	1.23
onetone2	0.23	0.98	0.25	0.55	**0.99**
raefsky3	1.45	1.59	1.53	1.74	**1.93**
raefsky4	1.33	**2.06**	1.59	1.81	1.85
tib	0.06	0.22	0.03	0.17	**0.27**
twotone	0.02	**1.94**	0.99	1.19	1.51
venkat50	1.05	1.18	1.06	**1.33**	1.29
wang3-old	1.44	**2.33**	1.58	1.80	1.62
wang4	1.44	**2.24**	1.59	1.90	1.79

Table 6
Operation count (ops), LU factorization time in seconds, and speedup (S) of WSMP and PARDISO on one (T_1) and eight (T_8) 1 GHz IBM Power 3 processors. The best time and speedup with eight processors is shown in boldface, the best time with one processor is underlined.

	WSMP				PARDISO			
Matrices	ops x10^9	T_1 (s)	T_8 (s)	S $\frac{T_1}{T_8}$	ops x10^9	T_1 (s)	T_8 (s)	S $\frac{T_1}{T_8}$
af23560	3.01	1.72	0.52	3.30	3.33	1.89	**0.32**	**5.90**
av41092	1.72	1.84	**0.83**	**2.21**	F_N	F_N	F_N	F_N
bayer01	0.41	0.38	0.39	1.00	0.12	0.32	**0.09**	**3.55**
bbmat	20.3	10.6	**2.42**	4.38	27.9	20.3	3.35	**6.05**
comp2c	0.51	0.99	**0.39**	2.53	2.09	3.15	0.53	**5.94**
e40r0000	0.22	0.22	**0.11**	2.00	0.21	0.26	0.13	**2.00**
e40r1000	0.22	0.25	0.10	2.50	0.23	0.26	0.10	**2.60**
e40r5000	0.31	0.37	0.16	2.31	0.45	0.37	**0.13**	**2.84**
ecl32	22.6	11.2	2.85	3.92	22.4	12.1	**1.72**	**7.03**
epb3	0.41	0.69	0.51	1.35	0.44	0.80	**0.42**	**1.90**
fidap011	2.96	1.77	0.57	3.10	3.65	2.04	**0.38**	**5.36**
fidapm11	3.13	2.18	**0.66**	3.30	8.93	4.76	0.69	**6.89**
lhr34c	0.15	0.44	0.41	1.07	0.53	0.83	**0.23**	**3.60**
invextr1	5.35	3.82	**1.06**	3.60	12.3	7.18	1.29	**5.56**
mixtank	14.50	7.78	2.18	3.56	17.6	9.29	**1.22**	**7.61**
nasarb	5.60	3.36	0.95	3.53	7.02	4.08	**0.64**	**6.37**
onetone1	1.40	1.24	0.61	2.03	2.21	1.79	**0.59**	**3.03**
onetone2	0.19	0.34	0.28	1.21	0.43	0.44	**0.16**	**2.75**
raefsky3	2.35	1.35	0.40	3.37	2.63	1.36	**0.25**	**5.44**
raefsky4	4.12	2.27	0.76	2.98	4.75	2.56	**0.40**	**6.40**
tib	0.23	0.13	0.10	1.30	0.03	0.11	**0.04**	**2.75**
twotone	5.94	4.99	**3.23**	1.54	37.4	24.7	3.63	**6.80**
venkat50	1.80	1.35	0.44	3.06	1.83	1.41	**0.37**	**3.81**
wang3-old	6.49	3.60	0.99	3.65	4.58	2.81	**0.44**	**6.38**
wang4	6.22	3.26	0.86	3.79	3.02	1.68	**0.32**	**5.25**