RC 24062 (W0609-114) September 26, 2006
Computer Science

IBM Research Report

A Shared- and Distributed-Memory Parallel General Sparse Direct Solver

Anshul Gupta

IBM T. J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights, NY 10598

anshul@watson.ibm.com

LIMITED DISTRIBUTION NOTICE

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

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

A Shared- and Distributed-Memory Parallel General Sparse Direct Solver

Anshul Gupta

IBM T. J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights, NY 10598

anshul@watson.ibm.com

IBM Research Report RC 24062 (W0609-114)

September 26, 2006

Abstract

An important recent development in the area of solution of general sparse systems of linear equations has been the introduction of new algorithms that allow complete decoupling of symbolic and numerical phases of sparse Gaussian elimination with partial pivoting. This enables efficient solution of a series of sparse systems with the same nonzero pattern but different coefficient values, which is a fairly common situation in practical applications. This paper reports on a shared- and distributed-memory parallel general sparse solver based on these new symbolic and unsymmetric-pattern multifrontal algorithms.

1 Introduction

Direct solution of general sparse systems of linear equations of the form $Ax = b$ involves four phases: *analysis* comprising ordering for fill-in reduction and symbolic factorization, *numerical factorization* of the sparse coefficient matrix A into triangular factors L and U using Gaussian elimination with partial pivoting, *forward and backward elimination* to solve for x using the triangular factors L and U and the right-hand side vector b, and *iterative refinement* of the computed solution. A numerically stable factorization of general sparse matrices involves pivoting, or interchange of rows or columns during factorization. As a result, conventional solvers usually need to combine the numerical factorization with at least some symbolic analysis, which can be inefficient and hard to parallelize. Parallel general solvers, in the past, have either included some symbolic analysis in their numerical factorization phase [10, 3] or have attempted to get around this problem by static pivoting [11] or by simplifying and symmetrizing the data structures [1]. Each of these approaches has its numerical or performance disadvantages, particularly for matrices with significantly unsymmetric structure.

We have recently introduced new symbolic and numerical algorithms [6] that allow complete decoupling of symbolic and numerical phases of sparse Gaussian elimination with partial pivoting. This enables efficient solution of multiple sparse systems with the same nonzero pattern but different coefficient values, which is a

fairly common situation in practical applications. In this paper, we describe a parallel direct solver for general sparse systems of linear equations based on these new algorithms that has recently been included in the Watson Sparse Matrix Package (WSMP) [7]. This solver utilizes both shared- and distributed- memory parallelism in the same program and is designed for a hierarchical parallel computer with network-interconnected SMP nodes. We compare the WSMP solver with two similar well known solvers: MUMPS [1, 2] and SuperLU$_{Dist}$ [11]. We show that the WSMP solver achieves significantly better performance than both these solvers based on traditional algorithms and is numerically more robust than SuperLU$_{Dist}$. We had earlier shown [8] that MUMPS and SuperLU$_{Dist}$ are amongst the fastest distributed-memory general sparse solvers available. A detailed description of the various features and algorithms employed by these packages can be found in [8].

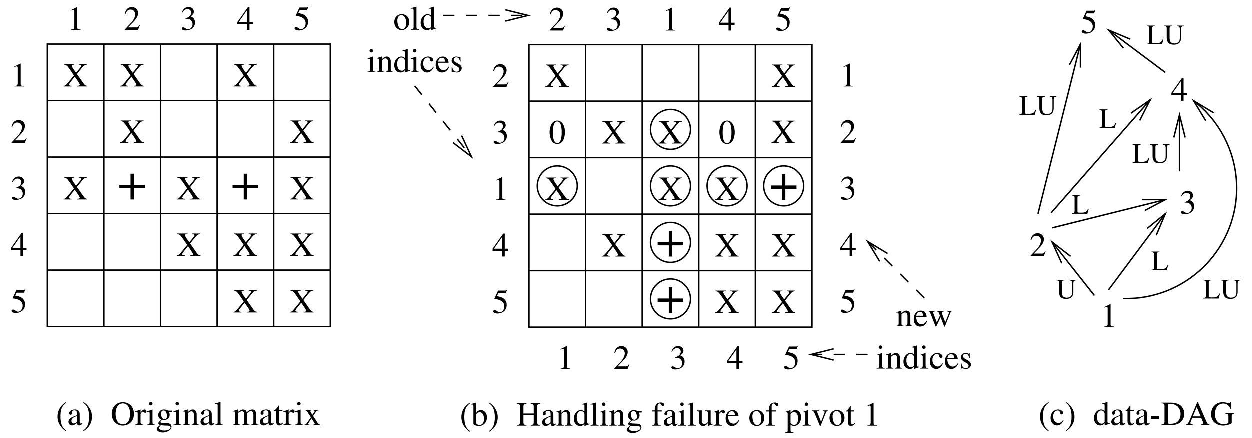
The process of factoring a sparse matrix can be expressed by a directed acyclic task-dependency graph, or task-DAG in short. The vertices of this directed acyclic graph (DAG) correspond to the tasks of factoring rows or columns or groups of rows and columns of the sparse matrix and the edges correspond 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 output data of task i is required as input by task j. In this paper, we define task i as the task of computing column i of L and row i of U. Once the tasks are defined, the task-DAG is unique to a sparse matrix for a given permutation of rows and columns; however, the data-DAG is a function of the sparse factorization algorithm. Multifrontal algorithms [3, 4, 12] 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.

Hadfield [10] introduced a parallel sparse LU factorization code based on the unsymmetric pattern multifrontal method. A significant drawback of this implementation was that partial pivoting during factorization would change the row/column order of the sparse matrix. Therefore, the data-DAG needed to be generated during numerical factorization, thus introducing considerable symbolic processing overhead. Our symbolic algorithms inexpensively compute an a-priori minimal task-dependency graph and near-minimal data-dependency graph for factoring a general sparse matrix that are valid for any amount of pivoting induced by the numerical values during LU factorization. This enables the symbolic processing phase to be completely separated from numerical factorization. As a result, the symbolic computation needs to be performed only once for matrices with the same initial structure but different numerical values, and hence, potentially different pivoting sequences during numerical factorization.

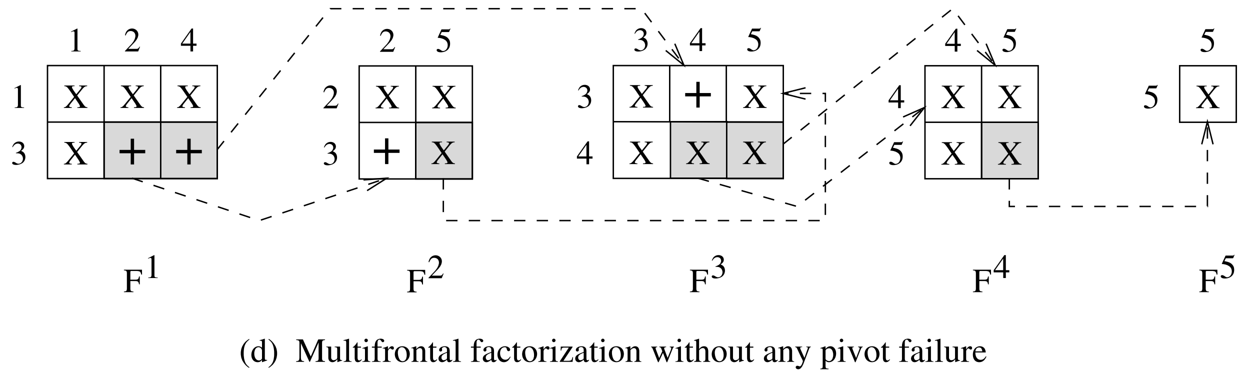
The remainder of the paper is organized as follows. Section 2 gives an overview of WSMP's sparse LU factorization algorithm. Section 3 describes the parallel implementation of this algorithm. Section 4 contains performance and scalability results of this algorithm and a comparison of these results with those of MUMPS [1] and SuperLU$_{Dist}$ [11]. Section 5 contains concluding remarks.

2 Overview of WSMP's sparse LU algorithm

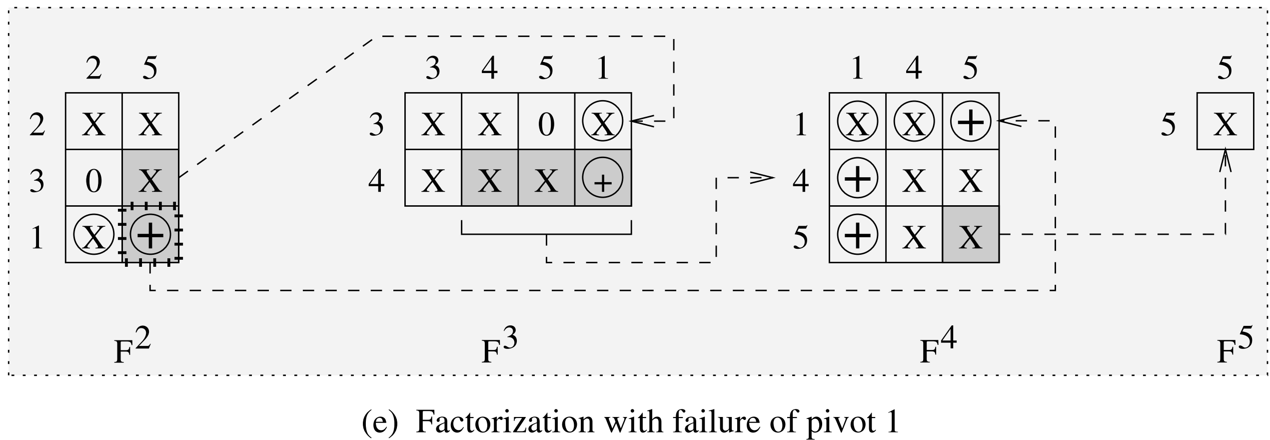
Assuming that the reader is familiar with the basics of multifrontal methods, we briefly describe WSMP's unsymmetric pattern multifrontal algorithm. Figure 1 shows a sparse matrix (a), its data-DAG (c) for factorization and illustrates the unsymmetric multifrontal factorization process both without (d) and with (e) pivoting. In the data-DAG of Figure 1(c), each vertex corresponds to a row-column pair of the corresponding sparse matrix. In practice, each vertex of the data-DAG corresponds to a supernode $[q : r]$, which is a set



(a) Original matrix (b) Handling failure of pivot 1 (c) data-DAG



(d) Multifrontal factorization without any pivot failure



(e) Factorization with failure of pivot 1

Figure 1: An example factorization to show how the failure of pivot 1 is handled by a symmetric permutation of row and column 1 to merge them with their LU-parent supernode, 4. An 'X' denotes a nonzero in the coefficient matrix and a '+' denotes a fill-in. The circled 'X' and '+' are created due to pivoting. A '0' denotes a fill-in predicted by the original symbolic factorization that has a value of zero due to pivoting-related movement of rows and columns.

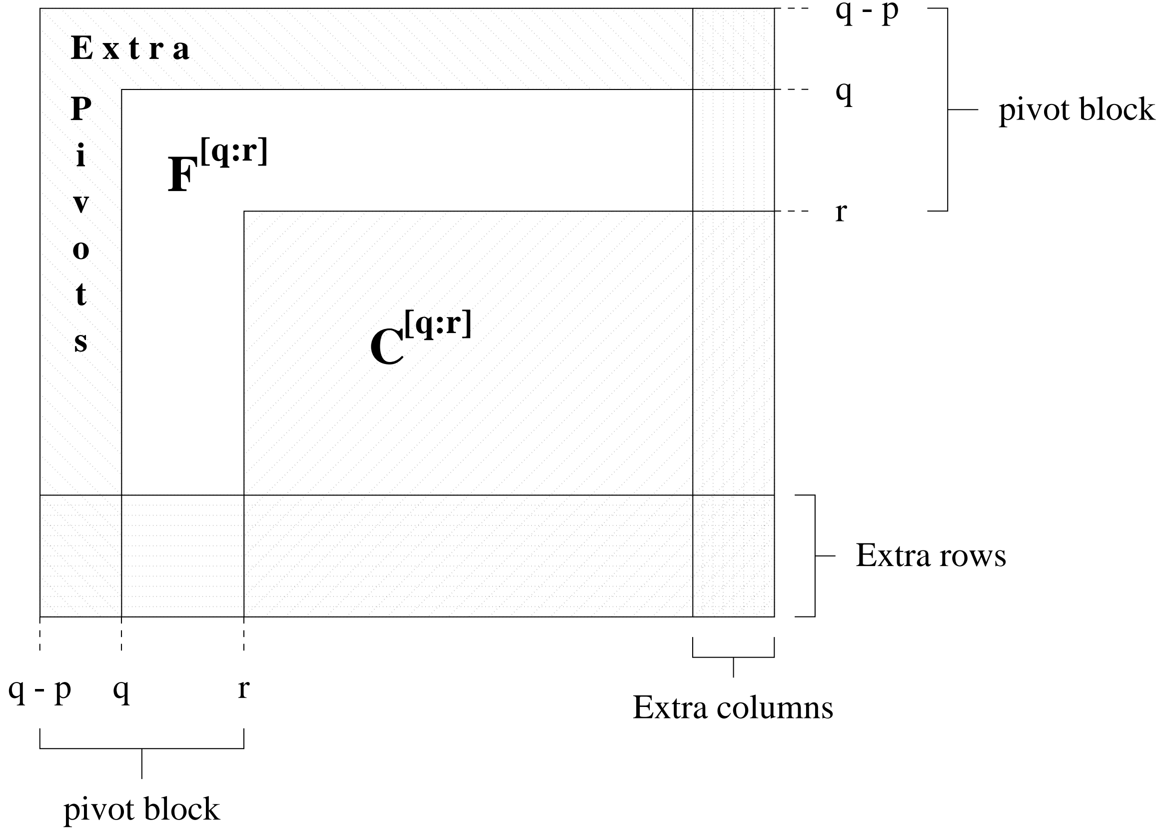


Figure 2: Organization of a typical frontal matrix for a supernode. The failed pivots from the LU-children of the supernode are appended at the beginning of the frontal matrix and the extra rows and columns inherited from U- and L-descendents, respectively, are appended at the end.

of consecutive rows and columns from q to r with the same nonzero structure in the factors L and U. The data-DAG has three types of edges, namely L, U, and LU edges. The type of an edge determines whether data is moved along the rows or along the columns (or both) from a child to a parent in the data-DAG (please refer to [6] for more details).

A fundamental data structure in our unsymmetric multifrontal algorithm is the frontal matrix. A frontal matrix is associated with each supernode. Fig. 2 shows the organization of a typical frontal matrix for a supernode $g = [q : r]$. The core of this frontal matrix is a $\mid$ Struct($L_{*,q}$) $\mid \times \mid$ Struct($U_{q,*}$) $\mid$ portion, where Struct($L_{*,q}$) and Struct($U_{q,*}$) are predicted by the symbolic factorization. In the absence of pivoting, the first $r - q + 1$ rows and columns of this matrix would be factored and would be saved as parts of U and L, respectively. The remaining trailing submatrix constitutes the contribution matrix C^g, whose contents would be absorbed into the frontal matrices of the parents of g in the data-DAG.

In the presence of numerical pivoting, extra pivots as well as other rows and columns may be added to the frontal matrix depending on the labels and pivot failures of the children of g in the data-DAG. Extra pivots (row-column pairs with the same indices) are added to F^g if some of the pivots of g's LU-children fail to satisfy the pivoting criterion. The LU-children of g themselves may have inherited some or all of these failed pivots from their own LU-children. Therefore, failed pivots from any of the LU-descendents of g can end up in its frontal matrix. If p such pivots are added, then the size of the pivot block increases from $r - q + 1$ to

$r - q + p + 1$.

The frontal matrix F^g can similarly inherit extra rows corresponding to failed pivots in its U-descendents whose LU-parents are greater than g and extra columns corresponding to failed pivots in its L-descendents whose LU-parents are greater than g. Irrespective of their new indices, these extra rows and columns are always appended at the end of the original rows and columns of F^g and a sorted list of their indices is maintained at each supernode. Eventually, these are assembled into the extra pivots of the frontal matrices of the LU-parents of the supernodes where these pivots failed. The row and column structures predicted by symbolic factorization are kept intact for future factorizations of matrices with the same nonzero pattern. The additions to these structures due to pivoting, which depend on the nonzero values in the matrix being factored, are maintained separately and are discarded before each new factorization.

Figure 1(d) shows the frontal matrices, their contribution blocks (shaded portions), and the movement of data from children's contribution blocks to their parents frontal matrices for a factorization that proceeds without pivoting. Parts (b) and (e) of the same figure illustrate the scenario in which pivot 1 fails. As Figure 1(e) shows, this pivot failure leads to the addition of an extra row at the end of F^2, an extra column at the end of F^3, and an extra pivot at the beginning of F^4. The extra column of C^2 and the extra row of C^3 are assembled into F^4. Figure 1(b) illustrates how the handling of this pivot failure is equivalent to moving row and column 1 to a position between rows and columns 3 and 4 in the original matrix.

3 Parallel implementation

In this section, we describe the shared- and distributed-memory parallel implementation of the algorithm described in Sections 2. WSMP is designed to make the best use of the computational resources of most modern parallel computers. These machines, typically, are either shared-memory multiprocessors or are clusters of nodes consisting of shared-memory multiprocessors. WSMP can run on multiple nodes using MPI processes and each process uses threads (the Pthreads library) to utilize all the CPUs on the node.

The symbolic preprocessing step, among other things, generates a static data-DAG that defines the communication and computation pattern of the multifrontal LU factorization. This static information is used to generate a static mapping of the data-DAG onto the processes. The work required to process certain supernodes could change during execution due to pivoting. However, such changes are usually relatively small and are randomly distributed among the processes. Therefore, they rarely pose a serious load-imbalance problem. Dynamic load-balancing would have introduced extra communication overhead and other complexities in the code. With these factors in mind, we chose static load-balancing at the process level. However, multiple threads running on each process keep their loads balanced dynamically. Each process maintains a dynamic heap of tasks that are ready for execution. The threads dynamically pick tasks from the heap and add more tasks as they become available. While processing those portions of the DAG where enough independent tasks cannot be found to keep all processors busy, multiple processors are used to perform the level-3 BLAS operations on supernodes in parallel. More details of the SMP-parallel component of the factorization can be found in [5].

While tasks are assigned dynamically to threads, they are assigned statically to MPI processes. The work associated with each supernode of the data-DAG is computed and each supernode is then assigned an executing parent, based on the longest weighted path from the root supernode. This imposes a tree on the data-DAG, and a recursive procedure starting at the root maps supernodes onto MPI processes statically, based on the amount of work in each subtree at each stage.

With the exception of the root supernode, a typical task in WSMP's multifrontal factorization involves

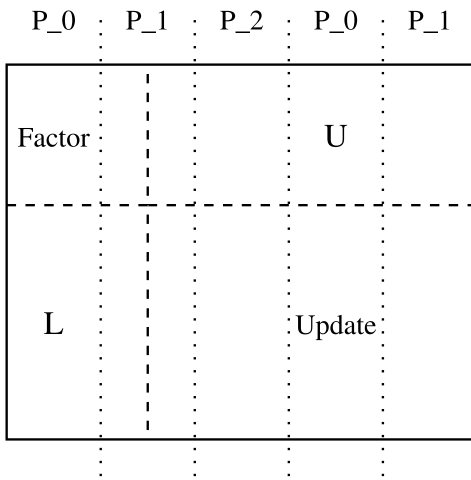
partial factorization and update of a dense rectangular frontal matrix and distributing the updated part of the matrix among the parents of the supernode corresponding to the factored frontal matrix. The root supernode involves full factorization of a dense square matrix. Depending on its size and location relative to the root, a task may be mapped onto one or multiple processes. When mapped on a single process, a task may be performed by a single or by multiple threads. If required, appropriate shared-memory parallel dense linear algebra routines are employed for partial factorization and updates. Henceforth, we will focus on describing process-level parallelization of WSMP's unsymmetric sparse factorization algorithm.

When a task is mapped onto multiple processes, the group of processes on which the task is mapped is viewed as a virtual grid. The virtual grid can be one-dimensional or two-dimensional, depending on the number of processes. In the current implementation, the grids are chosen such that they are one-dimensional with multiple columns for less than 6 processes and are two-dimensional for 6 or more processes. Moreover, the number of process rows in all grids is always a power of 2. This permits rapid broadcast and reduction operations among the processes in a process-column. All grid sizes are not allowed. For example, a 5-process grid is 1×5, a 6-process grid is 2×3, a 7-process grid is not allowed, and an 8-process grid is 2×4. The root supernode is always mapped onto the largest permissible grid with number of processes less than or equal to the total number of MPI processes that the program is running on. As we move away from the root supernode in the data-DAG and as more tasks can be executed in parallel, the process grids onto which these tasks are mapped become progressively smaller. Eventually, the tasks are mapped onto single processes.

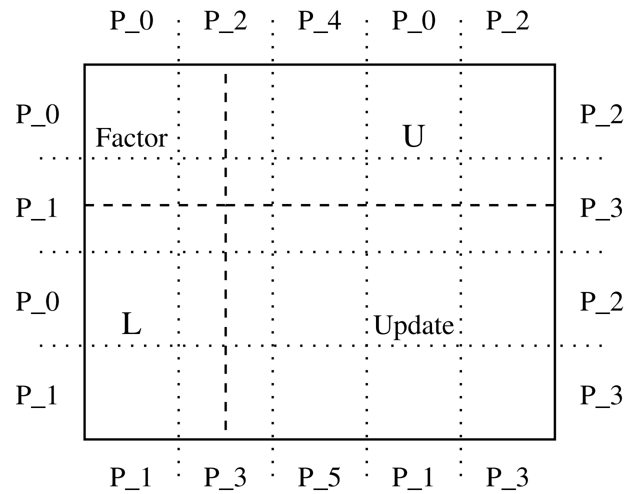
In addition to a serial/multithreaded partial factorization kernel, four types of message-passing parallel factorization kernels are employed for the four scenarios for mapping frontal matrices onto process grid, as shown in Figure 3. Efficient implementation of pivoting for numerical stability is a key requirement of these kernels.

With a non-root frontal matrix mapped onto a one-dimensional grid (Figure 3(a)), if a process can find a pivot among its columns, then no communication is required for pivoting. Otherwise, a column interchange involving communication with another process is required. When a non-root frontal matrix is mapped onto a two-dimensional grid (Figure 3(b)), then finding a pivot may require all processes in a column of the grid to communicate to find the pivot. That is the reason why the process grids have fewer rows than columns and the number of rows is a power of two so that fast logarithmic time communication patterns can be used for pivot searches along columns. Furthermore, this communication is avoided at every single pivot step by exchanging data corresponding to more than one column in each communication step. The frontal matrix corresponding to the root supernode never requires column interchanges because this matrix is factored completely. Therefore, pivoting is always free of communication for this matrix on a one-dimensional grid (Figure 3(c)). On a two-dimensional grid (Figure 3(d)), pivoting on the root supernode involves communication among the processes along process columns only. Once a pivot block has been factored, it is communicated along the pivot row and the pivot column, which are updated and are then communicated along the columns and the rows of the grid, respectively, to update the remainder of the matrix. The computation then moves to the next pivot block in a pipelined manner and continues until the supernode has been factored completely or no more pivots can be found.

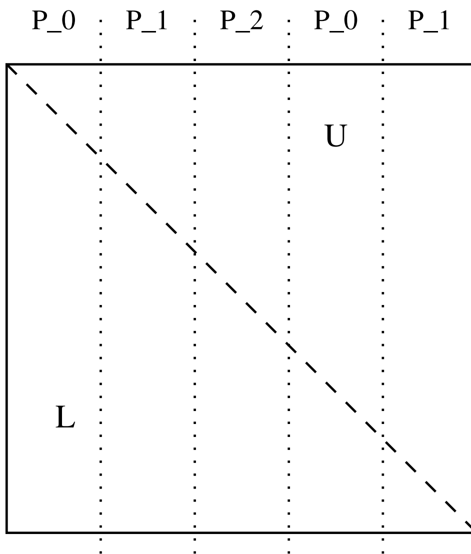
After all rows and columns corresponding to a supernode have been eliminated, or it is determined that no more suitable pivots can be found, the update matrix is distributed among the parent supernodes for inclusion into their frontal matrices. This update matrix includes the unfactored rows and columns of the supernode, if any. This is a crucial communication step that must be performed efficiently. In order to limit communication during this step, the rows and columns of the frontal/update matrices are distributed among the processes of a grid based on the binary representation of the global indices of the rows and columns in the original matrix.



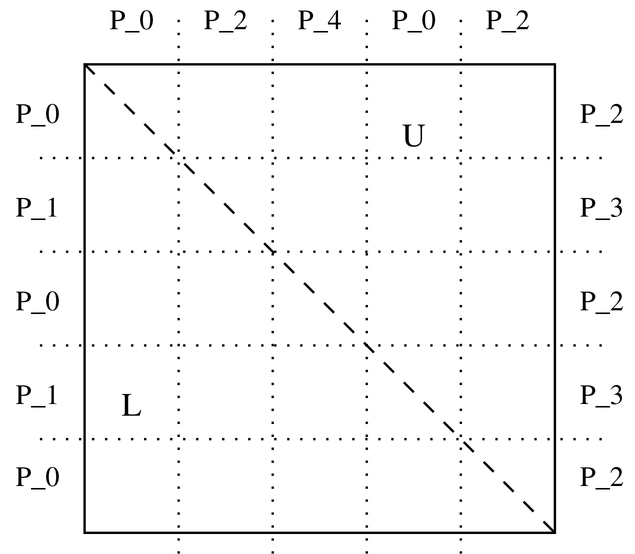
(a) A partial factor and update task mapped onto a 3−process 1−dimensional grid.



(b) A partial factor and update task mapped onto a 6−process 2−dimensional grid.



(c) A factorization task corresponding to the root node mapped onto a 3−process 1−dimensional grid.



(d) A factorization task corresponding to the root node mapped onto a 6−process 2−dimensional grid.

Figure 3: The four scenarios for mapping frontal matrices onto process grids.

This scheme is described in detail in [9] in the context of symmetric matrices. This distribution avoids an all-to-all communication between the processes of the child and parent grids. A process from the child grid usually needs to send data to only a small predetermined subset of processes of the parent grid.

4 Performance results and comparisons

In this section, we present detailed performance results of WSMP (version 6.7) and compare them with those of MUMPS [1] (version 4.6) and SuperLU$_{Dist}$ [11] (version 2.0) on a suite of 25 diverse test matrices. MUMPS is a multifrontal solver like WSMP, but works with an elimination tree and symmetric data structures derived by padding the matrix with zeros to make it structurally symmetric. SuperLU uses a supernodal right looking factorization algorithm with static pivoting; i.e., instead of swapping rows or columns for numerical stability, it merely perturbs very small or zero diagonal entries to a suitable larger value to allow the factorization to continue. It relies on iterative refinement to recover the accuracy lost due to this perturbation. A detailed description of the various algorithmic features of these packages can be found in [8]. The experiments were performed on a 32-node IBM SP3 running AIX 5.2. Each node had four 450 MHz Power3 CPUs and 4 Gbytes of RAM. All codes were compiled with -O3 optimization option in 64-bit mode and linked with ESSL BLAS. A maximum of 4 Gbytes of memory is available to each process.

In order to keep the comparison as fair as possible, the following preprocessing steps were applied to the matrices before passing them on the symbolic and numerical factorization phases of each code. WSMP used its built-in maximum-matching code to maximize the product of the diagonal entries for all matrices, which were then scaled to ensure that the magnitude of each diagonal entry was 1.0. This was followed by a symmetric permutation for fill-reduction based on a distributed nested-dissection ordering applied to the structure of $A + A^T$, where A is the matrix obtained after row permutation based on maximum matching. For MUMPS and SuperLU, the same preprocessing step was applied, followed by a fill-reducing symmetric ordering using WSMP's serial nested-dissection algorithm. MUMPS' and SuperLU's built-in unsymmetric permutation, scaling, and ordering options were switched off. As a result, MUMPS and SuperLU always worked on identical matrices. On a single CPU, all three solvers worked on identical matrices. However, on more than 1 CPU, the permutation generated by WSMP's serial ordering algorithm applied to MUMPS and SuperLU is different from that generated by WSMP's distributed ordering algorithm. These differences are small and random. Therefore, we expect that the large number of results reported (a total of 100, four on each of the 25 matrices) smooths out the random differences and we can derive useful conclusions from the general patterns that can be observed from these results.

All results in this section are presented using bar charts[1], where the lengths of the bars are normalized with respect an appropriate quantity in each figure. Each bar is subdivided into a solid and a hollow portion. The solid portion corresponds to the factorization time. The solution phase is depicted by the hollow portion of the bars and includes the time taken by the solvers' built-in iterative refinement to reduce the norm of the residual to 10^{-15}. This is important for a fair comparison of the solvers because they apply different pivoting techniques, which result in different trade-offs between the accuracy and speed of factorization. We want to measure the overall effectiveness of the algorithmic and design choices made in different solvers in solving real-life problems in a practical setting.

As described earlier, our target machine is a cluster of shared-memory multiprocessors connected with a high-speed switch. The fact that each node is a multiprocessor is typical of todays parallel computing

[1] Readers interested in actual times can obtain them at *http://www.cs.umn.edu/ agupta/wsmp*.

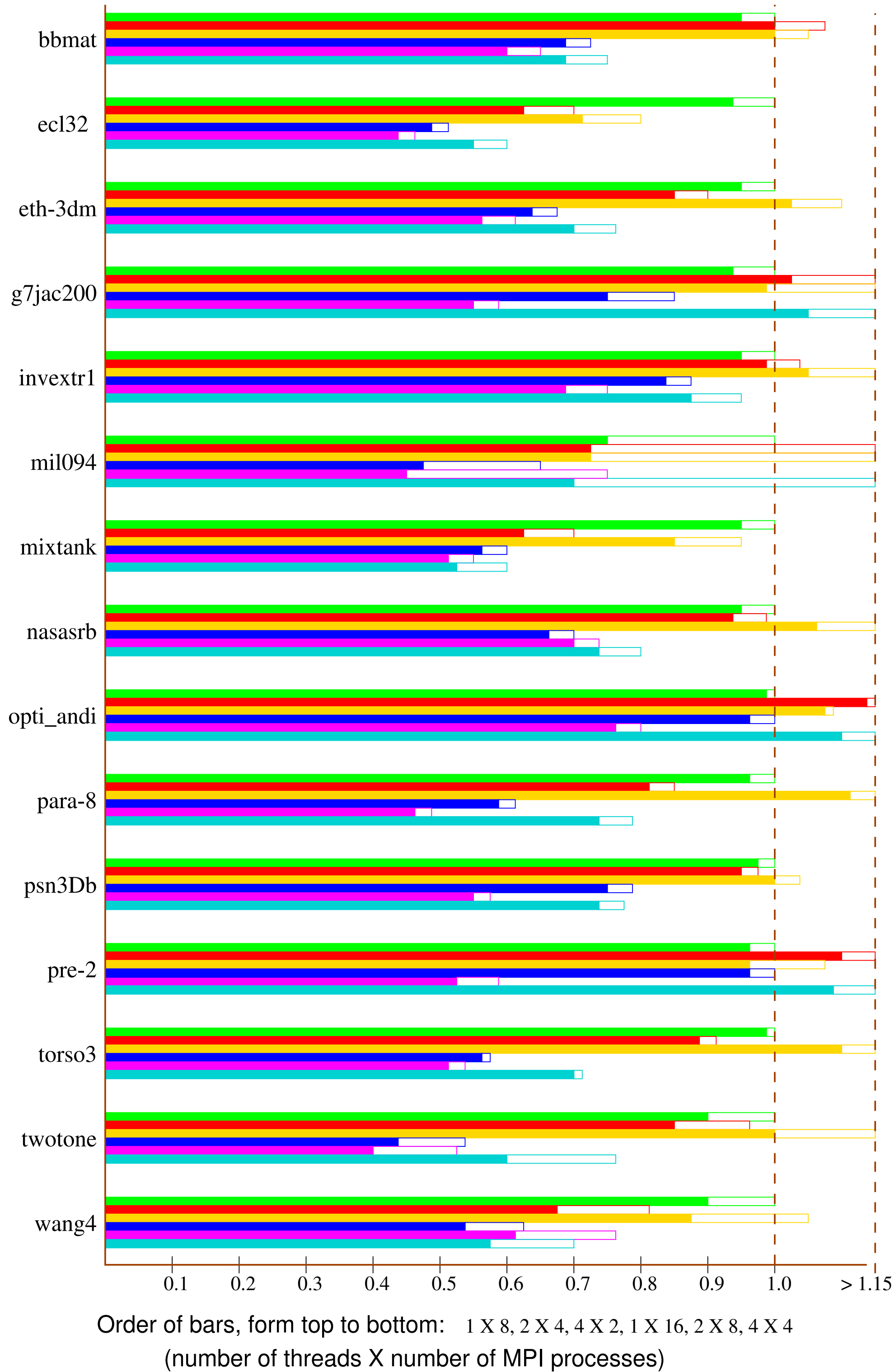


Figure 4: WSMP factor and solve times on 8 and 16 CPUs with various combinations of threads and MPI processes for the 15 largest matrices of the test suite. All times are normalized with respect to the 1×8 time; i.e., the time on 8 MPI processes with a single thread each. The solid portion of each bar represents factorization and the hollow portion represents solve.

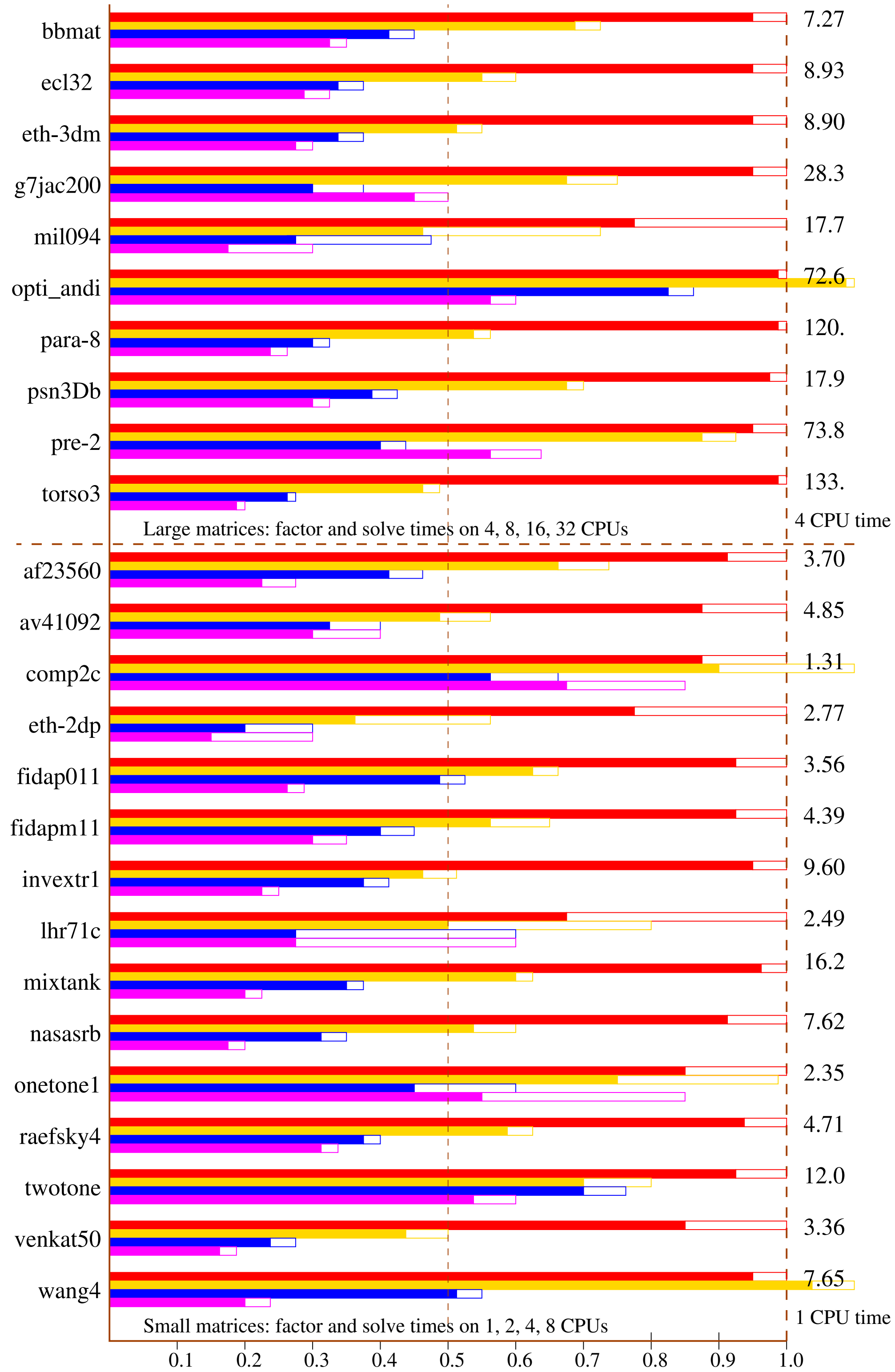


Figure 5: WSMP scalability results. For the 10 largest problems, factor and solve times on 4, 8, 16, and 32 CPUs are shown; for the remaining 15 problems, the times on 1, 2, 4, and 8 CPUs are shown. The solid portion of each bar represents factorization and the hollow portion represents solve.

environments. On a machine like this, a user has a choice of using a combination of shared-memory and message-passing parallelism. MUMPS and SuperLU$_{Dist}$ were initially designed to work only in distributed-memory parallel mode; however, they can be liked with threaded BLAS. We tried using multiple combinations of threads and MPI processes for MUMPS and SuperLU and found that overall, both packages deliver their best performance with a single thread and as many MPI processes as the number of CPUs. WSMP is designed specifically for clusters of multiprocessors. In Figure 4 we present the results of factoring and solving the 15 largest systems in our test suite on 8 and 16 CPUs with 1, 2, and 4 threads per MPI process. These results show that on 8 CPUs, the best performance is obtained with 2 threads and 4 MPI processes in 60% of the cases. On 16 CPUs, the best performance is obtained with 2 threads and 8 MPI processes in 80% of the cases. Using 4 threads per MPI process appears to be the worst case for most matrices. Based on these observations, we conduct the remainder of the experiments as follows. For MUMPS and SuperLU, we always use as many MPI processes as the number of CPUs. The MPI processes are run on individual nodes, thus using only one of the four CPUs on each node, and having all memory of each node available to them. It was observed that using two processes per node yielded similar timing results, but the solvers ran out of memory for some matrices. Using four processes per node degraded the performance for some matrices and the solvers ran out of memory in many cases. For WSMP, we use one thread per MPI process for up to 4 CPUs, and 2 threads per MPI process for more that 4 CPUs. Just like MUMPS and SuperLU the MPI processes are run on individual nodes.

Figure 5 shows how the factorization and solution times of WSMP scale with the number of CPUs. For this, and the remainder of the results in the paper, we have divided the test matrices into two categories. For the 10 biggest problems (in terms of their serial factorization and solution time), we present results on 4 to 32 CPUs, and for the remaining matrices, we present the results on 1 to 8 CPUs. In Figure 5, the lengths of the bars are normalized with respect to that for 4 CPUs for the large problems, and for 1 CPU for the remaining problems. The actual time in seconds corresponding to the unit length bars are given in the column on the right side of the figure. Notice that the scalability is generally good, with a few exceptions, where the speedups are poor or even less than 1.0 in some cases. We will soon see that such problem cases exist for MUMPS and SuperLU too. In fact WSMP seems to have fewer problem cases than other solvers.

Figure 6 shows a comparison of the times taken by WSMP and MUMPS to factor and solve our test problems. In each case, the total solution time of WSMP is considered to be one unit and all other times are normalized with respect to this. The structural symmetry of each matrix (after preprocessing) is shown in the column marked **SS** on the left. Some key observations from this figure are as follows. The structural symmetry does not seem to play an important role in the relative performance of WSMP and MUMPS. Barring the problem cases for WSMP (e.g., *opti_andi, g7jac200*) and MUMPS (e.g., *comp2c, eth-2dp, mil094*), the run time of MUMPS on the smallest number of CPUs for each matrix is between a factor of 1.0 and 1.5 of the WSMP time. However, WSMP appears to scale much better, and as a result, the heights of the bars for each matrix usually increases as the number of CPUs is increased. In particular, the solve phase of WSMP seems to scale significantly better than that of MUMPS, even though both solvers usually require the same number of iterations of refinement. Among a total of 100 observations reported, WSMP is slower than MUMPS in 8% of the cases, and is more than twice as fast in about half the cases.

Figure 7 shows a similar comparison between WSMP and SuperLU$_{Dist}$. SuperLU seems to have a number of problem cases. For example, for *comp2c, eth-2dp, lhr71c, mil094, onetone1, pre-2*, and *twotone*, it is between 8 and 250 times slower than WSMP. It also fails to produce the correct solution for 6 out of 25 problems. For the remaining problems, the scalability of WSMP and SuperLU appears to be similar; i.e., the lengths of the bars show small random fluctuations for a given matrix as the number of CPUs is changed.

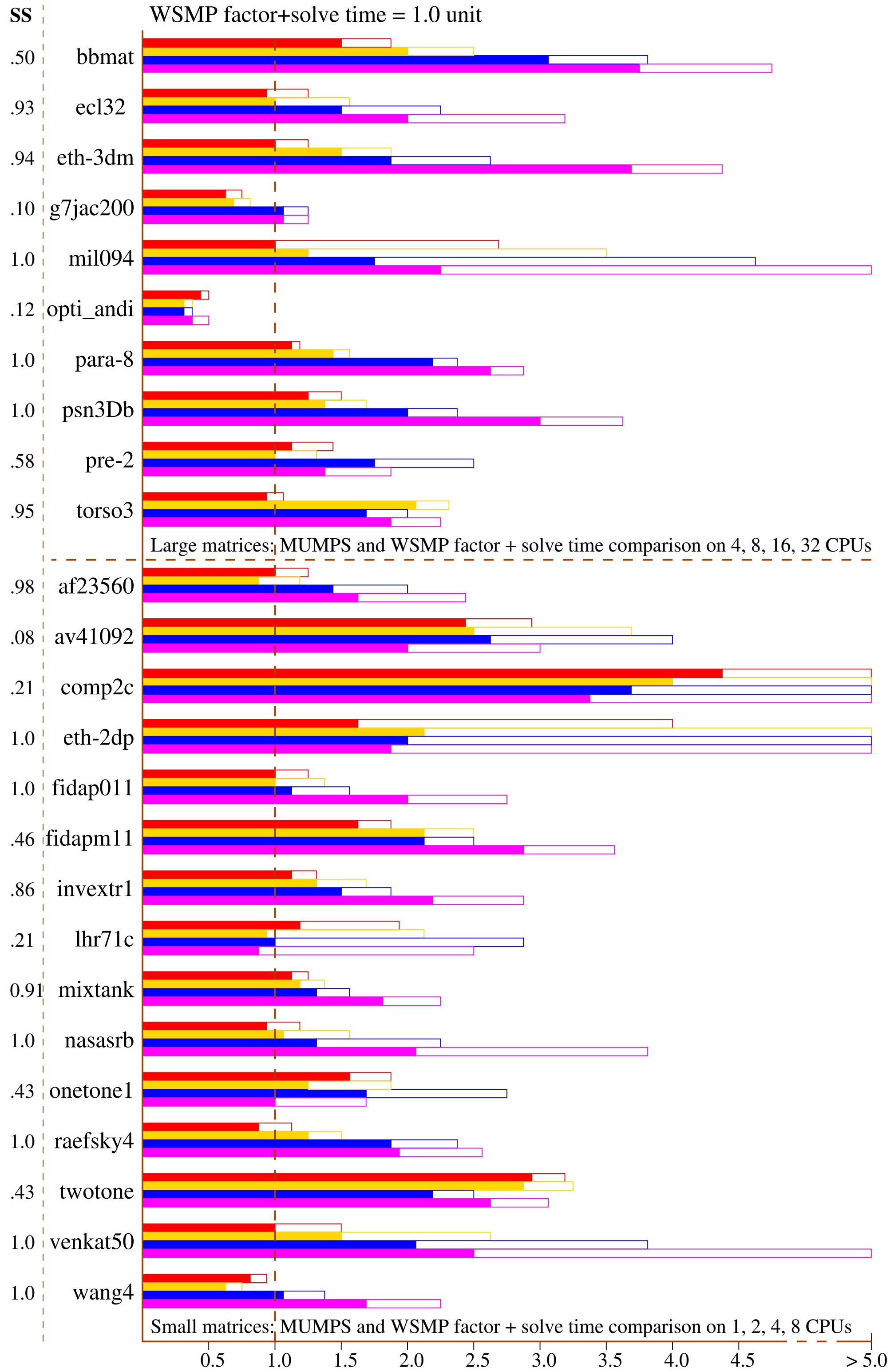


Figure 6: Parallel factor and solve time of MUMPS with respect to that of WSMP. The solid portion of each bar represents factorization and the hollow portion represents solve. The first column lists the structural symmetry of the preprocessed matrix.

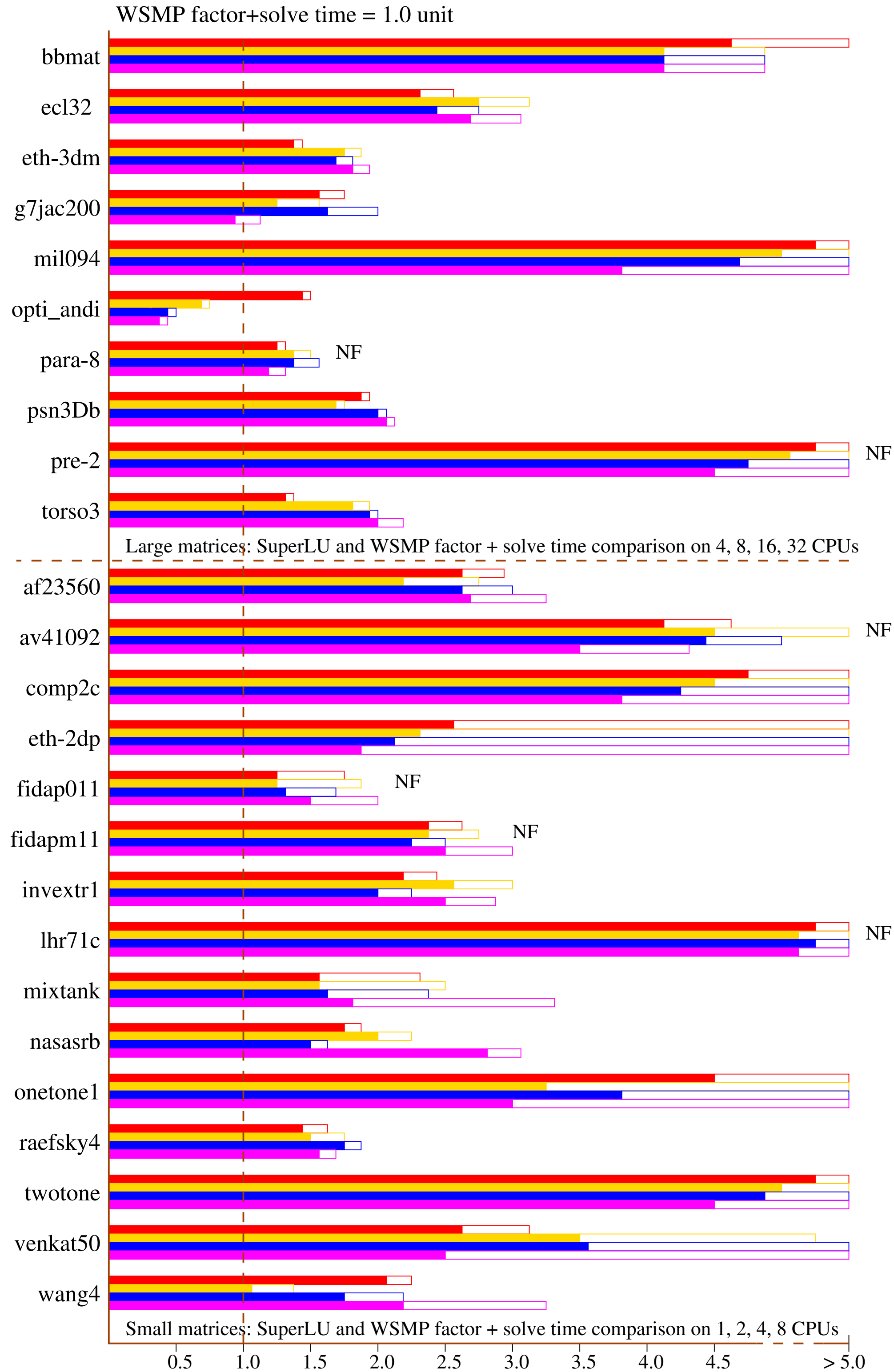


Figure 7: Parallel factor and solve time of MUMPS with respect to that of WSMP. The cases marked "NF" are those in which SuperLU$_{Dist}$ generated an incorrect solution, even after iterative refinement.

While unlike MUMPS, the solve phase of SuperLU is not inherently slow, it stills takes a considerable portion of the total time for many problems due to the extra iterative refinement steps required to recover the accuracy lost in static pivoting. Among a total of 100 observations reported, WSMP is slower than SuperLU in 3% of the cases, and is more than twice as fast in about two-thirds of the cases.

5 Concluding remarks

In this paper, we introduce WSMP's shared- and distributed-memory parallel sparse direct solver for general matrices and compare its performance with that of MUMPS and SuperLU$_{Dist}$ on a suite of 25 test matrices on up to 32 CPUs. On an average, WSMP appears to compare extremely favorably with respect to both other distributed-memory parallel solvers, which themselves are the among best such solvers available. There are several algorithmic and design choices that contribute to WSMP's performance. Some of these key features are: (1) new static dependency graphs [6] that permit effective static mapping of tasks to processes (minimizes communication), (2) dynamic threshold pivoting (avoids numerical failures and costly iterative refinement), (3) unsymmetric pattern multifrontal algorithm (avoids extra memory and operations due to artificial symmetrization of the structure, uses memory hierarchy effectively through level 3 BLAS), (4) two-dimensional process grids with mapping based on binary representation of global indices (minimizes communication), and (5) a hierarchical shared- and distributed-memory parallel design (enhances scalability).

References

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

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

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

[4] Duff, I.S., Reid, J.K.: The multifrontal solution of unsymmetric sets of linear equations. SIAM Journal on Scientific and Statistical Computing **5**(3), 633–641 (1984)

[5] Gupta, A.: A high-performance GEPP-based sparse solver. In: Proceedings of PARCO (2001). *http://www.cs.umn.edu/~agupta/doc/parco-01.ps*

[6] Gupta, A.: Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. SIAM Journal on Matrix Analysis and Applications **24**(2), 529–552 (2002)

[7] Gupta, A.: 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 (November 20, 2000). *http://www.cs.umn.edu/~agupta/wsmp*

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

[9] Gupta, A., Karypis, G., Kumar, V.: Highly scalable parallel algorithms for sparse matrix factorization. IEEE Transactions on Parallel and Distributed Systems **8**(5), 502–520 (May 1997)

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

[11] Li, X.S., Demmel, J.W.: SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Transactions on Mathematical Software **29**(2), 110–140 (2003)

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