Analysis and comparison of two general sparse solvers for distributed memory computers[1]

Patrick R. Amestoy[2], Iain S. Duff[3], Jean-Yves L'Excellent[4] and Xiaoye S. Li[5]

ABSTRACT
This paper provides a comprehensive study and comparison of two state-of-the-art direct solvers for large sparse sets of linear equations on large-scale distributed-memory computers. One is a multifrontal solver called `MUMPS`, the other is a supernodal solver called `SuperLU`.
We describe the main algorithmic features of the two solvers and compare their performance characteristics with respect to uniprocessor speed, interprocessor communication, and memory requirements. For both solvers, preorderings for numerical stability and sparsity play an important role in achieving high parallel efficiency.
We analyse the results with various ordering algorithms. Our performance analysis is based on data obtained from runs on a 512-processor Cray T3E using a set of matrices from real applications. We also use regular 3D grid problems to study the scalability of the two solvers.

Keywords: MPI, distributed memory architecture, sparse matrices, matrix factorization, multifrontal methods, supernodal methods.

AMS(MOS) subject classifications: 65F05, 65F50.

---

[1] This work has been partially supported by the France-Berkeley Fund.
Current reports available by anonymous ftp to ftp.numerical.rl.ac.uk in directory pub/reports. This report is in file adelRAL01003.ps.gz. Report also available through URL http://www.numerical.rl.ac.uk/reports/reports.html. Also published as Technical Report TR/PA/00/90 from CERFACS, 42 Ave G. Coriolis, 31057 Toulouse Cedex, France.
[2] amestoy@enseeiht.fr, ENSEEIHT-IRIT, 2 rue Camichel, Toulouse, France.
[3] I.Duff@rl.ac.uk, Rutherford Appleton Laboratory, and CERFACS, France. The work of this author was supported in part by the EPSRC Grant GR/M78502.
[4] excelle@enseeiht.fr, NAG Ltd, Wilkinson House, Oxford OX2 8DR, England.
[5] xiaoye@nersc.gov, NERSC, Lawrence Berkeley National Lab, MS 50F, 1 Cyclotron Rd., Berkeley, CA 94720. The work of this author was supported in part by the National Science Foundation Cooperative Agreement No. ACI-9619020 and NSF Grant No. ACI-9813362.

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

December 20, 2000

Contents

1 Introduction

We consider the direct solution of sparse linear equations on distributed memory computers where communication is by message passing, normally using `MPI`. We study in detail two state-of-the-art solvers, `MUMPS` (Amestoy, Duff, L'Excellent and Koster 1999, Amestoy, Duff and L'Excellent 2000) and `SuperLU` (Li and Demmel 1999). The first uses a multifrontal approach with dynamic pivoting for stability while the second is based on a supernodal technique with static pivoting and iterative refinement. We discuss the detailed algorithms used in these two codes in Section 3.

Two very important factors affecting the performance of both codes are the use of preprocessing to preorder the matrix so that the diagonal entries are large relative to the off-diagonals and the strategy used to compute an ordering for the rows and columns of the matrix to preserve sparsity. We discuss these aspects in detail in Section 4.

We compare the performance of the two codes in Section 5, where we show that such a comparison can be fraught with difficulties even though the authors of both codes are involved in the study. In Section 6, regular grids problems are used to further illustrate and analyse the difference between the two approaches. We had originally planned a comparison of more sparse codes but, given the difficulties we have found in assessing codes that we know well, we have for the moment shelved this more ambitious project. However, we feel that the lessons that we have learned in this present exercise are both invaluable to us in our future wider study and have given us some insight into the behaviour of sparse direct codes which we feel is useful to share with a wider audience at this stage. In addition to valuable information on the comparative merits of multifrontal versus supernodal approaches, we have examined the parameter space for such a comparison exercise and have identified several key parameters that influence to a differing degree the two approaches.

2 Test environment

Throughout this paper, we will use a set of test problems to evaluate the performance of our algorithms. Our test matrices come from the forthcoming Rutherford-Boeing Sparse Matrix Collection (Duff, Grimes and Lewis 1997) [1], the industrial partners of the PARASOL Project[2], Tim Davis' collection[3], SPARSEKIT2[4] and the EECS Department of UC Berkeley[5]. The PARASOL test matrices are available from Parallab, Bergen, Norway[6]. Two smaller matrices (GARON2 and LNSP3937) are included in our set of matrices but will be used only in Section 4.1 to illustrate differences in the numerical behaviour of the two solvers.

Note that, for most of our experiments, we do not consider symmetric matrices in our test set because `SuperLU` cannot exploit the symmetry and is unable to produce an $\mathbf{LDL}^T$ factorization. However, since our test examples in Section 6 are symmetric, we do

[1] Web page `http://www.cse.clrc.ac.uk/Activity/SparseMatrices/`
[2] EU ESPRIT IV LTR Project 20160
[3] Web page `http://www.cise.ufl.edu/~davis/sparse/`
[4] Web page `http://math.nist.gov/MatrixMarket/data/SPARSKIT/`
[5] Matrix ECL32 is included in the Rutherford-Boeing Collection
[6] Web page `http://www.parallab.uib.no/parasol/`

Real Unsymmetric Assembled (RUA)				
Matrix name	Order	No. of entries	StrSym$^{(*)}$	Origin
BBMAT	38744	1771722	0.54	Rutherford-Boeing (CFD)
ECL32	51993	380415	0.93	EECS Department of UC Berkeley
INVEXTR1	30412	1793881	0.97	PARASOL (Polyflow S.A.)
FIDAPM11	22294	623554	1.00	SPARSKIT2 (CFD)
GARON2	13535	390607	1.00	Davis collection (CFD)
LHR71C	70304	1528092	0.00	Davis collection (Chem Eng)
LNSP3937	3937	25407	0.87	Rutherford-Boeing (CFD)
MIXTANK	29957	1995041	1.00	PARASOL (Polyflow S.A.)
RMA10	46835	2374001	1.00	Davis collection (CFD)
TWOTONE	120750	1224224	0.28	Rutherford-Boeing (circuit sim)
WANG4	26068	177196	1.00	Rutherford-Boeing (semiconductor)

Table 2.1: Test matrices. $^{(*)}$ StrSym is the number of nonzeros matched by nonzeros in symmetric locations divided by the total number of entries (so that a structurally symmetric matrix has value 1.0).

show results with both the symmetric and unsymmetric factorization versions of `MUMPS`. Matrices MIXTANK and INVEXTR1 have been modified because of out-of-range (underflow) values in matrix files. To keep the same sparsity pattern, we did not want to replace those underflow values by zeros. Instead, we have replaced all entries with an exponent smaller than -300 to numbers with the same mantissa but with an exponent of -300. For each linear system, the right-hand side vector is generated so that the true solution is a vector of all ones.

All results presented in this paper have been obtained on the Cray T3E-900 (512 DEC EV-5 processors, 256 Mbytes of memory per processor, 900 peak Megaflop rate per processor) from NERSC at Lawrence Berkeley National Laboratory. We will also refer to experiments on a 35 processor IBM SP2 (66.5 MHertz processor with 128 Mbytes of physical memory and 512 Mbytes of virtual memory and 266 peak Megaflop rate per processor) at GMD in Bonn, Germany, used during the PARASOL Project. The performance characteristics of the two machines are listed in Table 2.2.

Computer	CRAY T3E-900	IBM SP2
Frequency of the processor	450 MHertz	66 MHertz
Peak uniproc. performance	900 Mflops	264 Mflops
Effective uniproc. performance	340 Mflops	150 Mflops
Peak communication bandwidth	300 Mbytes/sec	36 Mbytes/sec
Latency	4 μsec	40 μsec
Bandwidth/Effective performance	0.88	0.24

Table 2.2: Characteristics of the CRAY T3E-900 and the IBM SP2. The factorization of matrix WANG4 using `MUMPS` was used to estimate the effective uniprocessor performance of the computers.

3 Description of the algorithms used

In this section, we briefly describe the main characteristics of the algorithms used in the solvers and highlight the major differences between them. For a complete description of the algorithms, the reader should consult previous papers by the authors of these algorithms (Amestoy et al. 1999, Amestoy et al. 2000, Li and Demmel 1998, Li and Demmel 1999).

Both algorithms can be described by a computational tree whose nodes represent computations and whose edges represent transfer of data. In the case of the multifrontal method, `MUMPS`, some steps of Gaussian elimination are performed on a dense frontal matrix at each node and the Schur complement (or contribution block) that remains is passed for assembly at the parent node. In the case of the supernodal code, `SuperLU`, the distributed memory version uses a right-looking formulation which, having computed the factorization of a block of columns corresponding to a node of the tree, then immediately sends the data to update the block columns corresponding to ancestors in the tree.

Both codes can accept any pivotal ordering and both have a built-in capability to generate an ordering based on an analysis of the pattern of $A + A^T$, where the summation is performed symbolically. However, for the present version of `MUMPS`, the symbolic factorization is markedly less efficient if an input ordering is given since different logic is used in this case. The default ordering used by `MUMPS` is approximate minimum degree (AMD) (Amestoy, Davis and Duff 1996*a*) while the default for `SuperLU` is multiple minimum degree (MMD) (Liu 1985). However, in our experiments using a minimum degree ordering, we considered only the AMD ordering since both codes can generate this using the subroutine `MC47` from HSL (2000). It is usually far quicker than MMD and produces a symbolic factorization close to that produced by MMD. We also use nested dissection orderings (ND). Sometimes we use the `ON-MeTiS` ordering from `MeTiS` (Karypis and Kumar 1998), and sometimes the nested dissection/`haloamd` ordering from `SCOTCH` (Pellegrini, Roman and Amestoy 1999) depending on which performs better on each particular problem. In addition, it is sometimes very beneficial to precede the ordering by performing an unsymmetric permutation to place large entries on the diagonal and then scaling the matrix so that the diagonals are all of modulus one and the off-diagonals have modulus less than or equal to one. We use the `MC64` code of HSL to perform this preordering and scaling (Duff and Koster 1999) and indicate clearly when this is done. The effect of using this preordering of the matrix is discussed in detail in Section 4.1. Finally, when `MC64` is not used, our matrices are always scaled.

In both approaches, a pivot order is defined by the analysis and symbolic factorization stages. In `MUMPS`, the modulus of the prospective pivot is compared with the largest modulus of an entry in the row and is only accepted if this is greater than a threshold value, typically between 0.001 and 0.1 (our default value is 0.01). Note that, although `MUMPS` can choose pivots from off the diagonal, the largest entry in the column might be unavailable for pivoting at this stage if all entries in its row are not fully summed. This threshold pivoting strategy is common in sparse Gaussian elimination and helps to avoid excessive growth in the size of entries during the matrix factorization and so directly reduces the bound on the backward error. If a prospective pivot fails the test, all that happens is that it is kept in the Schur complement and is passed to the parent node. Eventually all rows with entries in the column will be available for pivoting, at the root if not before, so that a pivot can be chosen from the column. Thus the numerical factorization can respect the threshold criterion but at the cost of increasing the size of the frontal matrices and causing more work and fill-in than were forecast. For the SuperLU approach, a static pivoting strategy is used and we keep to the pivotal sequence chosen in the analysis. The magnitude of the potential pivot is tested against a threshold of $\epsilon^{1/2}||A||$, where ϵ is the machine precision and $||A||$ is the norm of A. If it is less than this value it is immediately set to this value (with the same sign) and the modified entry is used as pivot. This corresponds to a half-precision perturbation to the original matrix entry. The result is that the factorization is not exact and iterative refinement may be needed. Note that, after iterative refinement, we obtained an accurate solution in all the cases that we tested. If problems were still to occur then extended precision BLAS (Li, Demmel, Bailey, Henry, Hida, Iskandar, Kahan, Kapur, Martin, Tung and Yoo 2000) could be used.

3.1 MUMPS main parallel features

The parallelism within MUMPS is at two levels. The first uses the structure of the assembly tree, exploiting the fact that computations at nodes that are not ancestors or descendents are independent. The initial parallelism from this source (*tree parallelism*) is the number of leaf nodes but this reduces to one at the root. The second level is in the subdivision of the elimination operations through blocking of the frontal matrix. This blocking gives rise to *node parallelism* and is either by rows (referred to as *1D-node parallelism*) or by rows and columns (at the root and referred to as *2D-node parallelism*). Node parallelism depends on the size of the frontal matrix which, because of delayed pivots, is only known at factorization time. Therefore, this is determined dynamically. Each tree node is assigned a processor *a priori*, but the subassignment of blocks of the frontal matrix is done dynamically.

Most of the machine dependent parameters in MUMPS that control the efficiency of the code are designed to take into account both the uniprocessor and multiprocessor characteristics of the computers. Because of the dynamic distributed scheduling approach, we do not need as precise a description of the performance characteristics of the computer as for approaches based on static scheduling such as PaStiX (Henon, Ramet and Roman 1999). Most of the machine dependent parameters in MUMPS are associated with the block sizes involved in the parallel blocked factorization algorithms of the dense frontal matrices. Our main objective is to maintain a minimum granularity to efficiently exploit the potential of the processor while providing sufficient tasks to exploit the available parallelism. Our target machines differ in several respects. The most important ones are illustrated in Table 2.2. We found that smaller granularity tasks could be used on the CRAY T3E than on the IBM SP2 because of the relatively faster rate of communication to Megaflop rate on the CRAY T3E than on the IBM SP2 (see Table 2.2). That is to say that the communication is relatively more efficient on the CRAY T3E.

Dynamic scheduling is a major and original feature of the approach used in MUMPS. A critical part of this algorithm is when a process associated with a tree node decides to reassign some of its work, corresponding to a partitioning of the rows, to a set of so-called *worker* processes. We call such a node a one-dimensional parallel node. In earlier versions of MUMPS, a fixed block size is used to partition the rows and work is distributed to processes starting with the least loaded process. (The load of a process is determined by the amount of work (number of operations) allocated to it and not yet processed, which can be determined very cheaply.) Since the block size is fixed, it is possible for a process in charge of a one-dimensional parallel node to give additional work to processes that are already more loaded than itself. This can happen near the leaf nodes of the tree where sparsity provides enough parallelism to keep all processes busy. On the other hand, insufficient tasks might be created to provide work to all idle processes. This situation is more likely to occur close to the root of the tree.

In the new algorithm (available since Version 4.1 of MUMPS), the block size for the one-dimensional partitioning can be dynamically adjusted by the process in charge of the node. Early in the processing of the tree (that is, near the leaves) this gives a relatively bigger block size so reducing the number of worker processes; whereas close to the root of the tree the block size will be automatically reduced to compensate for the lack of parallelism in the assembly tree. We bound the block size for partitioning a one-dimensional parallel node by an interval. The lower bound is needed to maintain a minimum task granularity and control the volume of messages. The upper bound of the interval is less critical (it is by default chosen to be about eight times the lower bound) but it is used in estimating the maximum size of the communication buffers and of the factors and so should not be too large.

This "all dynamic" strategy of both partitioning and distributing work onto the processors could cause some trouble on a large number of processors (more than 128). In that case, it can be quite beneficial to take into account some "global" information to help the local decisions. For example one could restrict the choice of worker processes to a set of candidate processors determined statically during the analysis phase. This notion, commonly used in the design of static scheduling algorithms such as that in Henon et al. (1999), could reduce the overhead of the dynamic scheduling algorithm, reduce the increase in the communication volume when increasing the number of processors, and improve the local decision. The tuning of the parameters controlling the block size for 1D partitioning would then be easier and the estimation of the memory required during factorization would be more accurate. On a large number of processors, both performance and software improvements could thus be expected. This feature is not available in the current Version 4.1 of MUMPS but will be implemented in a future release. We will see that by adding this feature, one could address some of the current limitations of the MUMPS approach, see Section 5.2.

The solution phase is also performed in parallel and uses asynchronous communications both for the forward elimination and the back substitution. In the case of the forward elimination, the tree is processed from the leaves to the root, in a similar way to the factorization, while the back substitution requires a different algorithm that processes the tree from the root to the leaves. A pool of ready-to-be-activated tasks is used. We do not change the distribution of the factors as generated in the factorization phase. Hence, type 2 and 3 node parallelism are also used in the solution phase.

3.2 SUPERLU main parallel features

SuperLU also uses two levels of parallelism although more advantage is taken of the node parallelism through blocking of the supernodes. Because the pivotal order is fully determined at the analysis phase, the assignment of blocks to processors can be done statically *a priori* before the factorization commences. A *2D block-cyclic* layout is used and the execution can be pipelined since the sequence is predetermined. The matrix partitioning is based on the notion of an *unsymmetric supernode* introduced in Demmel, Eisenstat, Gilbert, Li and Liu (1999). The supernode is defined over the matrix factor L. A supernode is a range $(r : s)$ of columns of L with the triangular block just below the diagonal being full, and the same nonzero structure elsewhere (this is either full or zero). This supernode partition is used as the block partition in *both* row and column dimensions, that is the diagonal blocks are square. If there are N supernodes in an n-by-n matrix, there will be N^2 blocks of non-uniform size. Figure 3.1 illustrates such a block partition. The off-diagonal blocks may be rectangular and need not be full. Furthermore, the columns in a block of U do not necessarily have the same row structure. We call a dense subvector in a block of U a *segment*. The P processes are also arranged as a 2D mesh of dimension $P_r \times P_c = P$. By block-cyclic layout, we mean block (I, J) (of L or U) is mapped onto the process at coordinate $((I-1) \bmod P_r, (J-1) \bmod P_c)$ of the process mesh. During the factorization, block $L(I, J)$ is only needed by the processes on the process row $((I-1) \bmod P_r)$. Similarly, block $U(I, J)$ is only needed by the processes on the process column $((J-1) \bmod P_c)$. This partitioning and mapping can be controlled by the user. First, the user can set the *maximum block size* parameter. The symbolic factorization algorithm identifies supernodes, and chops the large supernodes into smaller ones if their sizes exceed this parameter. The supernodes may be smaller than this parameter due to sparsity and the blocks are then defined by the supernode boundaries. (That is, supernodes can be smaller than the maximum block size but never larger.) Our experience has shown that a good value for this parameter on the IBM SP2 is around 40, while on the Cray T3E it is around 24. Second, the user can set the shape of the process grid, such as 2×3 or 3×2. The more square the grid, the better the performance expected. This rule of thumb was used on the Cray T3E to define the grid shapes.

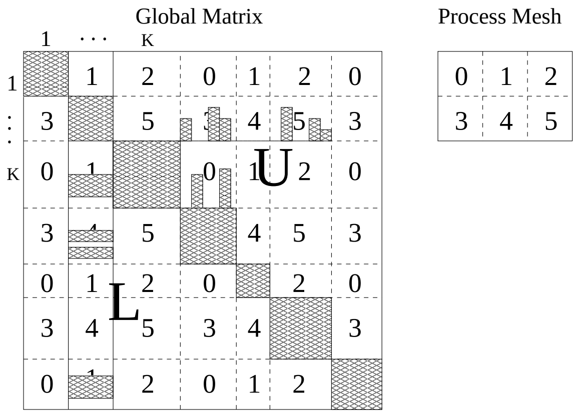

Figure 3.1: The 2D block-cyclic layout used in `SuperLU`.

In this 2D mapping, each block column of L resides on more than one process, namely, a column of processes. For example in Figure 3.1, the second block column of L resides on the column processes {1, 4}. Process 1 only owns two nonzero blocks, which are not contiguous in the global matrix.

The main numerical kernel involved during numerical factorization is a block update corresponding to the rank-k update to the Schur complement:

$$A(I,J) \leftarrow A(I,J) - L(I,K) \times U(K,J) ,$$

see Figure 3.2. In the earlier versions of SuperLU, this computation was based on Level 2.5 BLAS. That is, we call the Level 2 BLAS routine GEMV (matrix-vector product) but with multiple vectors (segments), and the matrix $L(I,K)$ is kept in cache across these multiple calls. This to some extent mimics the Level 3 BLAS GEMM (matrix-matrix product) performance. However, the difference between Level 2.5 and Level 3 is still quite large on many machines, e.g. the IBM SP2. This motivated us to modify the kernel in the following way in order to use Level 3 BLAS. For best performance, we distinguish two cases corresponding to the two shapes of a $U(K,J)$ block.

- The segments in $U(K,J)$ are of same height, as shown in Figure 3.2 (a).
  Since the nonzero segments are stored contiguously in memory, we can call GEMM directly, without performing operations on any zeros.

- The segments in $U(K,J)$ are of different heights, as shown in Figure 3.2 (b).
  In this case, we first copy the segments into a temporary working array T, with some leading zeros padded if necessary. We then call GEMM using $L(I,K)$ and T (instead of $U(K,J)$). We perform some extra floating-point operations for those padding zeros. The copying itself does not incur a run time cost, because the data must be loaded in the cache anyway. The working storage T is bounded by the maximum block size, which is a tunable parameter. For example, we usually use 40×40 on the IBM SP2 and 24×24 on the Cray T3E.

Depending on the matrix, this Level 3 BLAS kernel improved the uniprocessor factorization time by about 20% to 40% on the IBM SP2. A performance gain was also observed on the Cray T3E. It is clear that the extra operations are well offset by the benefit of the more efficient Level 3 BLAS routines.

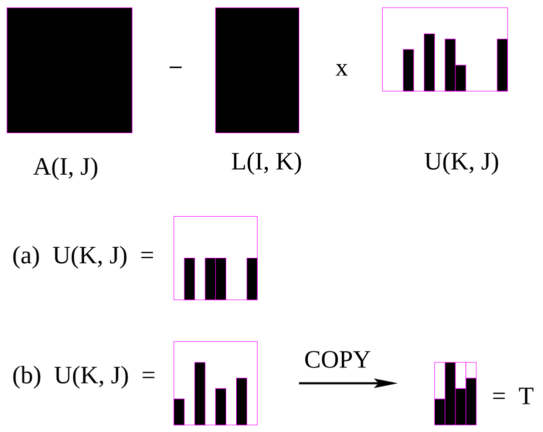

Figure 3.2: Illustration of the numerical kernels used in `SuperLU`.

The current factorization algorithm has two limitations to parallelism. Here we explain, by examples, what the problems are and speculate how the algorithm may be improved in the future. In the following matrix notation, the zero blocks are left blank. For each nonzero block we mark in $\boxed{\textbf{box}}$ the process which owns the block.

- Parallelism from the sparsity.
  Consider a matrix with 4-by-4 blocks mapped onto a 2-by-2 process mesh

$$\begin{bmatrix} \boxed{0} & \boxed{1} & \boxed{0} & \boxed{1} \\ & \boxed{3} & \boxed{2} & \boxed{3} \\ \boxed{0} & \boxed{1} & \boxed{0} & \\ & \boxed{3} & & \boxed{3} \end{bmatrix}.$$

  Although node 2 is the parent of node 1 in the elimination tree (associated with $A^T + A$), not all processes in column 2 depend on column 1. Only process 1 depends on the L block on process 0. Process 3 could start factorizing column 2 at the same time as process 0 is factorizing column 1, before process 1 starts factorizing column 2. But the current algorithm requires all the column processes to factorize the column synchronously, thereby introducing idle time for process 3. We can relax this constraint by allowing the diagonal process (3 in this case) to factorize the diagonal block and then send the factored block down to the off-diagonal processes (using `mpi_isend`), even before the off-diagonal processes are ready for this column. This would eliminate some artificial interprocess dependencies and potentially reduce the length of the critical path.

  Note that this kind of independence comes from not only the sparsity but also the 2D process-to-matrix mapping. An even more interesting study would be to formalize these 2D task dependencies into a task graph, and perform some optimal scheduling on it.

- Parallelism from the directed acyclic elimination graphs (Gilbert and Liu 1993) often referred to as *elimination dags* or edags.
  Consider another matrix with 6-by-6 blocks mapped onto a 2-by-3 process mesh

$$\begin{bmatrix} \boxed{0} & \boxed{1} & & \boxed{0} & & \boxed{2} \\ & \boxed{4} & & \boxed{3} & & \\ & & \boxed{2} & \boxed{0} & & \boxed{2} \\ & \boxed{4} & & \boxed{3} & & \boxed{5} \\ \boxed{0} & & & & \boxed{1} & \boxed{2} \\ \boxed{3} & & \boxed{5} & & & \boxed{5} \end{bmatrix}.$$

  Columns 1 and 3 are independent in the elimination dags. The column process sets {0, 3} and {2, 5} could start factorizing columns 1 and 3 simultaneously. However, since process 2 is also involved in the update task of block $(5, 6)$ associated with Step 1 and our algorithm gives precedence to all the tasks in Step 1 over any task in Step 3, process 2 does not factorize column 3 immediately. We may change this task precedence by giving the factorization task of a later step higher priority than the update tasks of the previous steps, because the former is more likely to be on the critical path. This would exploit better the task independence coming from the elimination dags.

We expect the above improvements will have a large impact for very sparse and/or very unsymmetric matrices, and for the orderings that give wide and bushy elimination trees, such as nested dissection.

The triangular solution algorithm is also designed around the same distributed 2D data structure. The forward substitution proceeds from the bottom of the elimination tree to the root, whereas the back substitution proceeds from the root to the bottom. The algorithm is based on a sequential variant called "inner product" formulation. The execution of the program is completely *message-driven*. Each process is in a self-scheduling loop, performing appropriate local computation depending on the type of the message received. The entirely asynchronous approach enables large overlap between communication and computation and helps to overcome the much higher communication to computation ratio in this phase.

3.3 First comments on the algorithmic differences

Both approaches use Level 3 BLAS to perform the elimination operations. However, in MUMPS the frontal matrices are always square. It is possible that there are zeros in the frontal matrix especially if there are delayed pivots or the matrix structure is markedly asymmetric but the present implementation takes no advantage of this sparsity and all the counts measured assume the frontal matrix is dense. It is shown in Amestoy and Puglisi (2000) that one can detect and exploit the structural asymmetry of the frontal matrices. With this new algorithm, significant gains both in memory and in time to perform the factorization can be obtained. For example, using MUMPS with the new algorithm, the number of operations to factorize matrices LHR71C and TWOTONE would be reduced by 30% and 37%, respectively. The approach, tested on a shared memory multifrontal code MA41 (Amestoy and Duff 1993) from HSL (2000), is however not yet available in the current version of MUMPS. In SuperLU, advantage is taken of sparsity in the blocks and usually the dense matrix blocks are smaller than those used in MUMPS. In addition, SuperLU uses a more sophisticated data structure to keep track of the irregularity in sparsity. Thus, the uniprocessor Megaflop rate of SuperLU is much worse than that of MUMPS. This performance penalty is to some extent alleviated by the reduction in floating-point operations because of the better exploitation of sparsity. As a rule of thumb, MUMPS will tend to perform particularly well when the matrix structure is close to symmetric while SuperLU can better exploit asymmetry. We note that, even if the same ordering is input to the two codes, the computational tree generated in each case will be different. In the case of MUMPS, the assembly tree generated by MC47 is used to drive the MUMPS factorization phase, while, for SuperLU, the directed acyclic computational graphs (*dags*) are built implicitly.

In Figures 3.3 and 3.4, we use a vampir trace (Nagel, Arnold, Weber, Hoppe and Solchenbach 1996) to illustrate the typical parallel behaviour of both approaches. These traces correspond to a zoom in the middle of the factorization phase of matrix BBMAT on 8 processors of the CRAY T3E. Black areas correspond to time spent in communications and related MPI calls. Each line between two processes corresponds to one message transfer. From the plots we can see that SuperLU has distinct phases for local computation and interprocess communication, whereas for MUMPS, it is hard to distinguish when the process performs computation and when it transfers a message. This is due to the asynchronous scheduling algorithm used in MUMPS which may have a better chance of overlapping communication with computation.

4 Impact of preprocessing and numerical issues

In this section, we first study the impact on both solvers of the preprocessing of the matrix. In this preprocessing, we first use row or column permutations to permute large entries onto the diagonal. In Section 4.1, we report and compare both the structural and the numerical impact of this preprocessing phase on the performance and accuracy of our solvers. After this phase, a symmetric ordering (minimum degree or nested dissection) is used and we study the relative influence of these orderings on the performance of the solvers in Section 4.2. We also comment on the relative cost of the analysis phase of the two solvers.

4.1 Use of a preordering to place large entries onto the diagonal and the cost of the analysis phase

Duff and Koster (1999) developed an algorithm for permuting a sparse matrix so that the diagonal entries are large relative to the off-diagonal entries. They have also written a computer code, MC64 (available from HSL (2000)), to implement this algorithm. Here, we use option 5 of MC64 which maximizes the product of the modulus of the diagonal entries and then scales the permuted matrix so that it has diagonal entries of modulus one and all off-diagonals of modulus less than or equal to one.

The importance of this preordering and scaling is clear. For MUMPS it should limit the amount of numerical pivoting during the factorization, which increases the overall cost of the factorization. For SuperLU, we expect such a permutation to be even more crucial, reducing the amount of small pivots that are modified and set to $\varepsilon^{1/2}||A||$.

The MC64 code of Duff and Koster (1999) is quite efficient and so should normally require little time relative to the matrix factorization even if the latter is executed on many processors while MC64 runs on only one processor. Results in this section will show that it is not always the case. Moreover, matrices which are unsymmetric but have a symmetric or nearly symmetric structure are a very common problem class. The problem with these is that MC64 performs an unsymmetric permutation and will tend to destroy the symmetry of the pattern. Since both codes use a symmetrized pattern for the sparsity ordering (see Section 4.2) and MUMPS uses one also for the symbolic and numerical factorization, the overheads in having a markedly unsymmetric pattern can be high. Conversely, when the initial matrix is very unsymmetric (as for example LHR71C) the unsymmetric permutation may actually help to increase structural symmetry thus giving a second benefit to the subsequent matrix factorization.

We show the effects of using MC64 on some examples in Table 4.1. In Table 4.4, we illustrate the relative cost of the main steps of the analysis phase when MC64 is used to preprocess the matrix.

We see in Table 4.1 that, for very unsymmetric matrices (LHR71C and TWOTONE), MC64 is really needed by MUMPS and SuperLU to factorize these matrices efficiently. Both matrices have zeros on the diagonal. Because of the static pivoting approach used by SuperLU, unless these zeros are made nonzero by fill-in and are then large enough, they will be perturbed

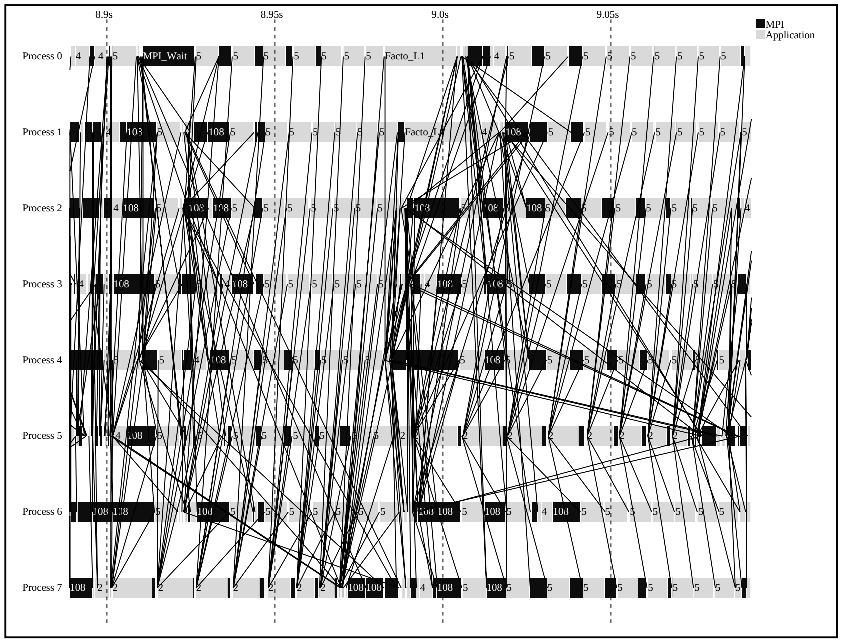

Figure 3.3: Illustration of the asynchronous behaviour of the `MUMPS` factorization phase.

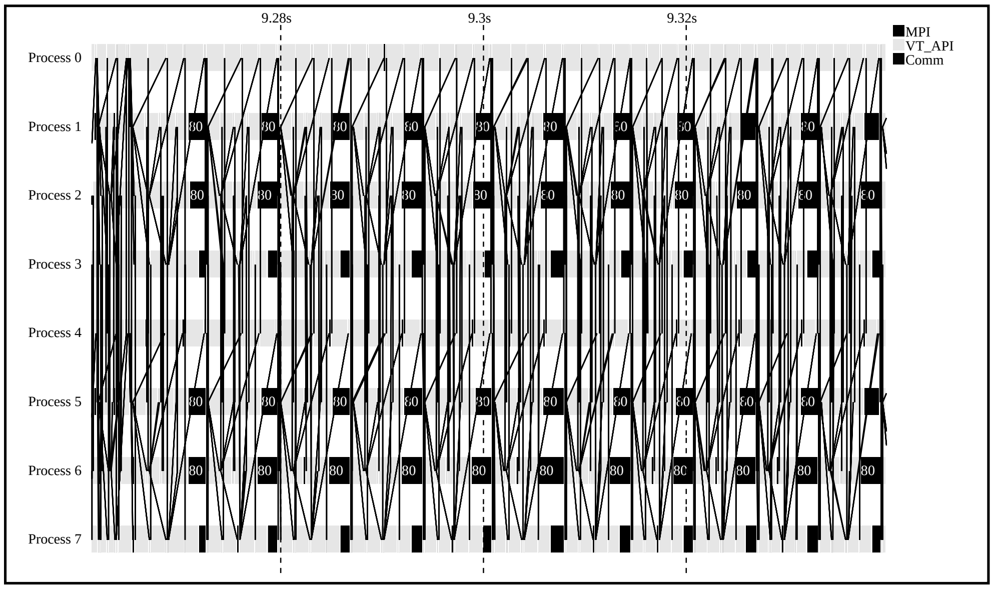

Figure 3.4: Illustration of the relatively more synchronous behaviour of the `SuperLU` factorization phase.

Matrix	Solver	Ordering	StrSym	Nonzeros in factors ($\times 10^6$)	Flops ($\times 10^9$)
BBMAT	MUMPS	AMD	0.54	46.1	41.5
	—	MC64+AMD	0.50	44.3	36.9
	SuperLU	AMD	0.54	41.2	34.0
	—	MC64+AMD	0.50	40.2	31.2
ECL32	MUMPS	AMD	0.93	42.9	64.6
	—	MC64+AMD	0.93	42.9	64.6
	SuperLU	AMD	0.93	42.4	68.3
	—	MC64+AMD	0.93	42.7	68.4
INVEXTR1	MUMPS	AMD	0.97	31.2	35.8
	—	MC64+AMD	0.86	33.6	38.6
	SuperLU	AMD	0.97	24.8	22.6
	—	MC64+AMD	0.86	28.4	28.0
FIDAPM11	MUMPS	AMD	1.00	16.1	9.7
	—	MC64+AMD	0.46	29.4	28.5
	SuperLU	AMD	1.00	14.0	8.9
	—	MC64+AMD	0.46	24.8	22.0
GARON2	MUMPS	AMD	1.00	2.4	0.3
	—	MC64+AMD	0.83	2.7	0.4
	SuperLU	AMD	1.00	2.1	0.3
	—	MC64+AMD	0.83	2.5	0.4
LHR71C	MUMPS	AMD$^{(*)}$	0.00	285.8	1431.0
	—	MC64+AMD	0.21	11.8	1.4
	SuperLU	AMD$^{(*)}$	0.00	222.5	—
	—	MC64+AMD	0.21	7.6	0.5
LNSP3937	MUMPS	AMD	0.87	0.3	0.02
	—	MC64+AMD	0.55	0.4	0.03
	SuperLU	AMD	0.87	0.2	0.02
	—	MC64+AMD	0.55	0.3	0.03
MIXTANK	MUMPS	AMD	1.00	39.1	64.4
	—	MC64+AMD	0.91	45.7	81.5
	SuperLU	AMD	1.00	38.4	64.1
	—	MC64+AMD	0.91	41.2	64.6
RMA10	MUMPS	AMD	1.00	8.9	1.4
	—	MC64+AMD	0.90	9.7	1.6
	SuperLU	AMD	1.00	8.9	1.5
	—	MC64+AMD	0.90	9.3	1.5
TWOTONE	MUMPS	AMD	0.28	235.0	1221.1
	—	MC64+AMD	0.43	22.1	29.3
	SuperLU	AMD	0.28	65.3	159.0
	—	MC64+AMD	0.43	11.9	8.0
WANG4	MUMPS	AMD	1.00	11.6	10.5
	—	MC64+AMD	1.00	11.6	10.5
	SuperLU	AMD	1.00	10.7	9.1
	—	MC64+AMD	1.00	10.7	9.1

Table 4.1: Impact of permuting large entries onto the diagonal (using MC64) on the size of the factors and the number of operations. $^{(*)}$ estimation given by the analysis (not enough memory to perform factorization). StrSym denotes the structural symmetry after ordering.

during factorization and a factorization of a nearby matrix is obtained. In the case of MUMPS, the dramatically higher fill-in obtained without MC64 makes it also necessary to use MC64. For MUMPS, the main benefit from using MC64 is more structural than numerical. The permuted matrix has in fact a larger structural symmetry (see column 4 of Table 4.1) so that a symmetric permutation can be obtained on the permuted matrix that is more efficient in preserving sparsity. SuperLU benefits in a similar way from symmetrization because the computation of the symmetric permutation is based on the same assumption even if SuperLU preserves better the asymmetric structure of the factors by performing a symbolic analysis on a directed acyclic graph and exploiting asymmetry in the factorization phase (compare, for example, results with MUMPS and SuperLU on matrices LHR71C, MIXTANK and TWOTONE).

		SuperLU		MUMPS	
Matrix	Iter.	No MC64	MC64	No MC64	MC64
BBMAT		Err=2.1e-03	Err=5.6e-01	Err= 1.3e-06	Err=6.5e-08
	0	Berr=4.0e-09	1.3e-05	Berr=7.4e-11	1.2e-11
	1	Berr=7.7e-16	4.6e-11	Berr=3.2e-16	3.2e-16
	2	Berr=5.2e-16	9.7e-15	Berr=3.2e-16	2.7e-16
	3	Berr=	4.7e-16		
	4	Berr=	5.0e-16		
		Err= 2.5e-09	Err=2.4e-09	Err= 3.0e-09	Err=3.5e-09
LNSP3937		Err=1.6e-01	Err=2.7e-11	Err=9.2e-07	Err=3.6e-11
	0	Berr=1.6e-07	3.5e-12	Berr=4.3e-08	1.5e-12
	1	Berr=1.5e-08	2.2e-16	Berr=4.7e-16	2.4e-16
	2	Berr=5.7e-10	2.5e-16	Berr=2.1e-16	2.0e-16
	3	Berr=1.6e-11			
	4	Berr=4.2e-13			
	5	Berr=1.1e-14			
	6	Berr=3.2e-16			
	7	Berr=3.2e-16			
		Err=1.0e-11	Err=2.2e-11	Err=6.3e-12	Err=6.4e-12
GARON2		Err=9.2e-07	Err=3.7e-12	Err=1.7e-11	3.4e-12
	0	Berr=2.5e-10	2.4e-15	1.6e-15	2.1e-15
	1	Berr=3.4e-16	3.8e-16	2.2e-16	2.3e-16
	2	Berr=3.4e-16	3.4e-16	2.0e-16	1.8e-16
		Err=2.9e-12	Err=3.3e-12	Err=1.6e-12	Err=1.3e-12

Table 4.2: Illustration of the convergence of iterative refinement.

The use of MC64 can also improve the quality of the factors and the numerical behaviour of the factorization phase, and can reduce the number of steps of iterative refinement required to reduce the backward error to machine precision. This is illustrated in Table 4.2 where we show the number of steps of iterative refinement required to reduce the componentwise relative backward error, $Berr = \max_i \frac{|r|_i}{(|A|\cdot|x|+|b|)_i}$ (Arioli, Demmel and Duff 1989), to machine precision ($\varepsilon \approx 2.2 \times 10^{-16}$ on the CRAY T3E). Iterative refinement will stop when either the required accuracy is reached or the convergence rate is too slow ($Berr$ does not decrease by at least a factor of two). The true error is reported as $Err = \frac{||x_{true} - x||}{||x_{true}||}$. This table illustrates the impact of the use of MC64 on the quality of

WITHOUT MC64						
Matrix	Solver	WITHOUT Iter. Ref.		WITH Iterative Refinement		
		Berr	Err	Nb	Berr	Err
BBMAT	MUMPS	7.4e-11	1.3e-06	2	3.2e-16	3.0e-09
	SuperLU	4.0e-09	2.1e-03	2	5.2e-16	2.5e-09
ECL32	MUMPS	3.6e-13	3.0e-11	2	3.1e-16	1.4e-11
	SuperLU	2.4e-14	2.6e-11	2	2.9e-16	7.0e-11
INVEXTR1	MUMPS	4.4e-08	8.9e-01	2	8.3e-06	2.8e-05
	SuperLU	1.7e-07	1.0e-01	3	8.0e-16	1.3e-05
FIDAPM11	MUMPS	3.6e-11	1.7e-09	2	2.8e-16	1.2e-12
	SuperLU	1.7e-06	1.9e-04	4	3.1e-16	1.8e-12
GARON2	MUMPS	1.6e-15	1.7e-11	2	2.0e-16	1.6e-12
	SuperLU	2.5e-10	9.2e-07	2	3.4e-16	2.9e-12
LHR71C	MUMPS	Not enough memory				
	SuperLU	Not enough memory				
LNSP3937	MUMPS	4.3e-08	9.2e-07	3	2.1e-16	6.3e-12
	SuperLU	1.6e-07	1.6e-01	7	3.2e-16	1.0e-11
MIXTANK	MUMPS	1.9e-12	4.8e-09	2	5.9e-16	1.4e-11
	SuperLU	3.6e-09	4.4e-04	3	4.8e-16	2.8e-11
RMA10	MUMPS	1.2e-13	8.3e-13	2	5.0e-16	1.2e-12
	SuperLU	2.2e-06	3.8e-05	3	4.2e-16	9.2e-13
TWOTONE	MUMPS	5.0e-07	1.3e-05	3	1.3e-15	2.1e-11
	SuperLU	1.0e+00	6.6e+126	1	1.0e+00	2.6e+220

WITH MC64						
Matrix	Solver	WITHOUT Iter. Ref.		WITH Iterative Refinement		
		Berr	Err	Nb	Berr	Err
BBMAT	MUMPS	1.2e-11	6.5e-08	2	2.7e-16	3.5e-09
	SuperLU	1.3e-05	5.6e-01	4	5.0e-16	2.4e-09
ECL32	MUMPS	5.6e-12	5.6e-10	2	3.0e-16	1.6e-11
	SuperLU	2.9e-14	1.3e-11	2	3.5e-16	1.7e-11
INVEXTR1	MUMPS	6.7e-16	1.6e-05	2	6.3e-16	5.6e-06
	SuperLU	1.0e-05	9.8e-01	3	6.8e-16	1.2e-05
FIDAPM11	MUMPS	4.4e-12	2.3e-10	2	3.6e-16	6.8e-13
	SuperLU	1.3e-01	7.8e-01	12	3.5e-16	1.1e-12
GARON2	MUMPS	2.1e-15	3.4e-12	2	1.8e-16	1.3e-12
	SuperLU	2.4e-15	3.7e-12	2	3.4e-16	3.3e-12
LHR71C	MUMPS	1.1e-05	9.9e+00	3	3.2e-13	1.0e+00
	SuperLU	7.1e-04	3.9e+06	2	8.9e-07	4.2e+07
LNSP3937	MUMPS	1.5e-12	3.6e-11	2	2.0e-16	6.4e-12
	SuperLU	3.5e-12	2.7e-11	2	2.5e-16	2.2e-11
MIXTANK	MUMPS	4.8e-12	2.3e-08	2	4.2e-16	4.0e-11
	SuperLU	8.2e-03	8.7e-01	5	5.1e-16	3.1e-11
RMA10	MUMPS	2.1e-12	3.4e-11	2	5.0e-16	1.0e-12
	SuperLU	1.3e-06	3.9e-05	3	4.9e-16	1.1e-12
TWOTONE	MUMPS	3.2e-13	1.6e-10	2	1.6e-15	2.3e-11
	SuperLU	1.0e-06	9.0e-03	4	6.1e-16	1.6e-11

Table 4.3: Comparison of the numerical behaviour, backward error (Berr) and forward error (Err), of the solvers. Nb indicates the number of steps of iterative refinement.

the initial solution obtained with both solvers prior to iterative refinement. Additionally, it shows that, thanks to numerical partial pivoting, the initial solution is almost always more accurate with MUMPS than with SuperLU and is usually markedly so. These observations are further confirmed on a larger number of test matrices in Table 4.3. The same stopping criterion was applied for these runs as for the runs in Table 4.2. In the case of MUMPS, MC64 can also result in a reduction in the number of off-diagonal pivots and in the number of delayed pivots. For example on the matrix INVEXTR1 the number of off-diagonal pivots drops from 1520 to 109 and the number of delayed pivots drops from 2555 to 42. One can also see in Table 4.2 (e.g., BBMAT) that MC64 does not always improve the numerical accuracy of the solution obtained with SuperLU.

As expected, we see that, for matrices with a fairly symmetric pattern (e.g., matrix FIDAPM11 in Table 4.1), the use of MC64 leads to a significant decrease in symmetry which, for both solvers, results in a significant increase in the number of operations during factorization. We additionally recollect that the time spent in MC64 can dominate the analysis time of either solver (see Table 4.4), even for matrices such as FIDAPM11 and INVEXTR1 for which it does not provide any gain for the subsequent steps. Thus, for both solvers, the default should be to not use MC64 on fairly symmetric matrices. In practice, the default option of the MUMPS package is such that MC64 is automatically invoked when the structural symmetry is found to be less than 0.5. For SuperLU, zeros on the diagonal and numerical issues must also be considered so that an automatic decision during the analysis phase is more difficult.

We finally compare, in Figure 4.1, the time spent by the two solvers during the analysis phase when reordering is based only on AMD (MC64 is not invoked). Since the time spent

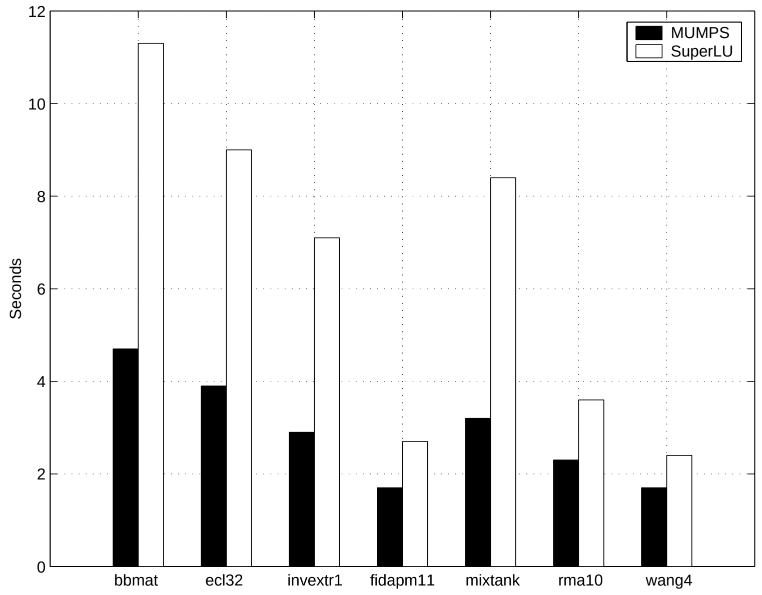

Figure 4.1: Time comparison of the analysis phases of MUMPS and SuperLU. MC64 preprocessing is NOT used and AMD ordering is used.

in AMD is very similar in both cases, this gives a good estimation of the cost difference

Matrix	Solver	Preprocess.	Total	MC64	AMD
BBMAT	MUMPS	AMD	4.7	—	3.0
	—	MC64+AMD	7.2	2.1	3.1
	SuperLU	AMD	11.3	—	2.8
	—	MC64+AMD	11.8	2.0	2.9
ECL32	MUMPS	AMD	3.9	—	2.3
	—	MC64+AMD	4.5	0.5	2.3
	SuperLU	AMD	9.0	—	2.1
	—	MC64+AMD	14.1	0.6	2.1
INVEXTR1	MUMPS	AMD	2.9	—	1.2
	—	MC64+AMD	47.2	42.6	1.5
	SuperLU	AMD	7.1	—	1.2
	—	MC64+AMD	45.8	36.8	1.5
FIDAPM11	MUMPS	AMD	1.7	—	0.6
	—	MC64+AMD	13.1	10.4	1.6
	SuperLU	AMD	2.7	—	0.5
	—	MC64+AMD	14.1	9.1	1.4
GARON2	MUMPS	AMD	0.4	—	0.1
	—	MC64+AMD	0.8	0.4	0.1
	SuperLU	AMD	0.8	—	0.1
	—	MC64+AMD	1.2	0.4	0.1
LHR71C	MUMPS	AMD	47.5	—	39.4
	—	MC64+AMD	34.0	31.0	2.0
	SuperLU	AMD	120.6	—	35.0
	—	MC64+AMD	32.0	26.9	1.8
LNSP3937	MUMPS	AMD	0.1	—	0.1
	—	MC64+AMD	0.2	0.1	0.1
	SuperLU	AMD	0.1	—	0.1
	—	MC64+AMD	0.3	0.1	0.1
MIXTANK	MUMPS	AMD	3.2	—	0.8
	—	MC64+AMD	5.8	2.2	0.9
	SuperLU	AMD	8.4	—	0.8
	—	MC64+AMD	11.0	2.2	0.9
RMA10	MUMPS	AMD	2.3	—	0.4
	—	MC64+AMD	4.6	2.3	0.5
	SuperLU	AMD	3.6	—	0.5
	—	MC64+AMD	6.1	2.3	0.6
TWOTONE	MUMPS	AMD	12.7	—	8.7
	—	MC64+AMD	8.8	1.7	4.8
	SuperLU	AMD	21.4	—	7.9
	—	MC64+AMD	12.0	1.7	4.4
WANG4	MUMPS	AMD	1.7	—	0.8
	—	MC64+AMD	2.0	0.2	0.8
	SuperLU	AMD	2.4	—	0.7
	—	MC64+AMD	2.6	0.2	0.7

Table 4.4: Influence of permuting large entries onto the diagonal (using MC64) on the time (in seconds) for the analysis phase of MUMPS and SuperLU.

of the analysis phase of the two solvers. Note that SuperLU is not currently tied to any specific ordering code and does not take advantage of all the information available from an ordering algorithm. A tighter coupling with an ordering, as is the case with MUMPS and AMD, should reduce the analysis time for SuperLU. However, during the analysis phase of SuperLU, all the asymmetric structures needed for the factorization are computed and the directed acyclic graph (Gilbert and Liu 1993) of the unsymmetric matrix must be built and mapped onto the processors. With MUMPS, the main data structure handled during analysis is the assembly tree which is produced directly as a by-product of the ordering phase. No further data structures are introduced during this phase. Dynamic scheduling will be used during factorization so that only a simple massage of the tree and a partial mapping of the computational tasks onto the processors are performed during analysis.

4.2 Use of orderings to preserve sparsity

On matrices for which MC64 is not used we show, in Table 4.5, the impact of the choice of the symmetric permutation on the fill-in and floating-point operations for the factorization. As was observed in Amestoy et al. (1999), the use of nested dissection can significantly improve the performance of MUMPS. We see here that SuperLU will also, although to a lesser extent, benefit from the use of a nested dissection ordering. We examine the influence of the ordering on the performance further in Section 5. We also notice that, for both orderings, SuperLU exploits the asymmetry of the matrix somewhat better than MUMPS (see BBMAT with structural symmetry 0.53). We expect the asymmetry of the problem to be better exploited by MUMPS when the approach described in Amestoy and Puglisi (2000) is implemented.

Matrix	Ordering	Solver	NZ in **LU** $\times 10^6$	Flops $\times 10^9$
BBMAT	AMD	MUMPS	46.1	41.5
		SuperLU	41.2	34.0
	ND	MUMPS	35.8	25.7
		SuperLU	33.9	23.5
ECL32	AMD	MUMPS	42.9	64.6
		SuperLU	42.4	68.3
	ND	MUMPS	24.8	20.9
		SuperLU	24.3	20.7
INVEXTR1	AMD	MUMPS	31.2	35.9
		SuperLU	24.2	21.3
	ND	MUMPS	16.2	8.1
		SuperLU	13.3	5.9
MIXTANK	AMD	MUMPS	39.1	64.4
		SuperLU	38.2	64.4
	ND	MUMPS	19.6	13.2
		SuperLU	18.6	12.9

Table 4.5: Influence of the symmetric sparsity orderings on the fill-in and floating-point operations on the factorization of unsymmetric matrices. (MC64 is not used.)

5 Performance analysis on general matrices

5.1 Performance of the numerical phases

In this section, we compare the performance and study the behaviour of the numerical phases (factorization and solve) of the two solvers.

For the sake of clarity, we will only report results with the best (in terms of factorization time) sparsity ordering for each approach. When the best ordering for MUMPS is different from that for SuperLU, results with both orderings will be provided. This means that results with both nested dissection and minimum degree orderings are given that illustrate the different sensitivity of the codes to the choice of the ordering. We note that, even when the same ordering is given to each solver, they will not usually perform the same number of operations. In general, SuperLU performs fewer operations than MUMPS because it exploits better the asymmetry of the matrix although the execution time is less for MUMPS because of the Level 3 BLAS effect.

Although results are very often matrix dependent, we will try, as much as possible, to identify some general properties of the two solvers. We should point out that the maximum dimension of our unsymmetric test matrices is only 120750 (see Table 2.1).

5.1.1 Study of the factorization phase

We show in Table 5.1 the factorization time of both solvers. On the smaller matrices, we only report in Table 5.2 results with up to 64 processors.

We observe that MUMPS is usually faster than SuperLU and is significantly so on a small number of processors. We believe there are two reasons. First, MUMPS handles symmetric and more regular data structures better than SuperLU, because MUMPS uses Level 3 BLAS kernels on bigger blocks than those used within SuperLU. As a result, the Megaflop rate of MUMPS on one processor is on average about twice that of the SuperLU factorization. This is also evident in the results on smaller test problems in Table 5.2 and from the results on 3D grid problems in Section 6. Note that, even on the matrix TWOTONE, for which SuperLU performs three times fewer operations than MUMPS, MUMPS is over 2.5 times faster than SuperLU on four processors. On a small number of processors, we also notice that SuperLU does not always fully benefit from the reduction in the number of operations due to the use of a nested dissection ordering (see BBMAT with SuperLU using 4 processors).

Furthermore, one should notice that, with matrices that are structurally very asymmetric, SuperLU can be much less scalable than MUMPS. For example, on matrix LHR71C in Table 5.2, speedups of 2.5 and 8.3 are obtained with SuperLU and MUMPS, respectively. This is due to the two parallel limitations of the current SuperLU algorithm described in Section 3.2. First, SuperLU does not fully exploit the parallelism of the elimination dags. Second, the pipelining mechanism does not fully benefit from the sparsity of the factors (a blocked column factorization should be implemented). This also explains why SuperLU does not fully benefit, as in the case for MUMPS, from the better balanced tree generated by a nested dissection ordering.

We see that the ordering very significantly influences the performance of the codes (see results with matrices BBMAT and ECL32) and, in particular, MUMPS generally outperforms SuperLU, even on a large number of processors, when a nested dissection ordering is used. On the other hand, if we use the minimum degree ordering, SuperLU can be faster than

MUMPS on a large number of processors. We also see that, on most of our unsymmetric problems, neither solver provides enough parallelism to benefit from using more than 128 processors. The only exception is matrix ECL32 using the AMD ordering (requiring 64×10^9 flops for the factorization), for which only SuperLU continues to decrease the factorization time up to 512 processors. Our lack of other large unsymmetric systems gives us few data points in this regime but one might expect that, independently of the ordering, the 2D distribution used in SuperLU should provide better scalability (and hence eventually better performance) on a large number of processors than the mixed 1D and 2D distribution used in MUMPS. To further analyse the scalability of our solvers, we consider three dimensional regular grid problems in Section 6.

Matrix	Ord.	Solver	Number of processors								
			1	4	8	16	32	64	128	256	512
BBMAT	AMD	MUMPS	—	45.7	24.0	16.5	13.7	11.9	11.2	9.1	12.6
		SuperLU	—	66.1	38.1	22.8	14.6	11.2	8.9	9.9	9.1
	ND	MUMPS	—	39.4	22.8	13.2	11.9	9.9	9.2	9.4	11.6
		SuperLU	—	137.8	74.9	41.2	25.2	17.3	12.4	14.3	14.7
ECL32	AMD	MUMPS	—	54.6	32.0	23.8	17.6	15.6	15.1	16.0	16.5
		SuperLU	—	107.4	58.4	35.8	20.6	14.9	11.1	10.9	8.9
	ND	MUMPS	—	24.7	14.1	9.7	7.7	6.9	7.0	7.0	8.9
		SuperLU	—	49.0	28.2	16.7	12.0	9.9	8.8	9.9	9.5
INVEXTR1	ND	MUMPS	31.8	13.2	6.5	4.5	3.9	3.8	4.4	5.4	6.3
		SuperLU	68.2	23.1	13.3	9.1	6.7	5.7	4.7	6.1	5.8
MIXTANK	ND	MUMPS	40.8	13.0	7.8	5.6	4.4	3.9	4.2	4.2	5.4
		SuperLU	88.1	28.8	14.6	10.1	7.0	5.3	4.5	5.6	5.5
TWOTONE	MC64	MUMPS	—	40.3	22.6	18.6	14.7	14.4	14.3	14.0	14.3
	+AMD	SuperLU	—	106.2	61.8	32.7	25.7	21.0	16.2	21.2	18.5

Table 5.1: Factorization time (in seconds) of large test matrices on the CRAY T3E. "—" indicates not enough memory.

Matrix	Ordering	Solver	Number of processors					
			1	4	8	16	32	64
FIDAPM11	AMD	MUMPS	31.6	11.7	8.4	6.5	5.7	5.7
		SuperLU	58.6	14.3	9.7	6.0	4.5	4.4
LHR71C	MC64+AMD	MUMPS	13.3	4.3	2.9	1.7	1.5	1.6
		SuperLU	34.7	17.8	13.0	12.5	11.5	14.0
RMA10	AMD	MUMPS	8.1	3.1	2.2	2.1	2.0	2.1
		SuperLU	11.6	5.1	3.7	3.6	3.1	3.8
WANG4	AMD	MUMPS	30.6	11.1	7.0	5.2	4.3	3.9
		SuperLU	56.3	19.4	13.9	7.9	5.8	5.6

Table 5.2: Factorization time (in seconds) of small test matrices on the CRAY T3E. "—" indicates not enough memory.

To better understand the performance differences observed in Tables 5.1 and 5.2 and to identify the main characteristics of our solvers we show, in Table 5.3, the average communication volume. The speed of communication can depend very much on the number and the size of the messages and we also indicate the maximum size of the messages and the average number of messages. To overlap communication by computation, MUMPS uses fully asynchronous communications (during both sends and receives). The use of non-blocking sends during the more synchronous scheduled approach used by SuperLU also enables overlapping between communication and computation.

Matrix	Ord	Solver	Number of processors								
			4			16			64		
			Max	Vol.	#Mess	Max	Vol.	#Mess	Max	Vol.	#Mess
BBMAT	AMD	MUMPS	4.9	44	3240	3.3	63	1700	2.9	20	2257
		SuperLU	0.18	81	23412	0.09	61	34176	0.05	35	35035
	ND	MUMPS	2.2	7	2214	2.8	43	1441	1.5	48	3228
		SuperLU	0.17	82	30698	0.09	62	45598	0.04	36	50925
ECL32	AMD	MUMPS	9.7	91	5451	3.7	117	2585	2.9	54	2743
		SuperLU	0.32	90	27437	0.16	67	37486	0.09	39	34981
	ND	MUMPS	8.5	37	3663	2.5	60	1981	1.5	29	2679
		SuperLU	0.25	56	28966	0.13	42	41172	0.07	24	41271
INVEXTR1	ND	MUMPS	2.2	13	2320	1.1	18	1314	1.5	7	1550
		SuperLU	0.15	31	17774	0.08	23	25824	0.05	13	27123
FIDAPM11	AMD	MUMPS	2.5	28	3000	2.4	22	1471	2.4	6	1323
		SuperLU	0.15	27	14768	0.08	20	19114	0.04	12	15621
LHR71C	MC64	MUMPS	1.0	1	96	1.1	1	342	1.1	1	377
	+AMD	SuperLU	0.04	21	72932	0.03	15	95653	0.02	8	91640
MIXTANK	ND	MUMPS	3.5	30	3138	1.7	33	1650	1.2	11	1616
		SuperLU	0.19	40	13667	0.11	30	19635	0.05	18	19064
RMA10	AMD	MUMPS	0.7	3	114	0.7	2	302	0.7	1	337
		SuperLU	0.06	18	11346	0.03	13	14124	0.02	7	10883
TWOTONE	MC64	MUMPS	8.8	61	5076	2.9	139	4144	2.1	49	2762
	+AMD	SuperLU	0.26	27	120006	0.15	20	153995	0.05	11	104906
WANG4	AMD	MUMPS	3.9	16	3483	1.5	27	1682	1.5	8	1215
		SuperLU	0.19	24	27728	0.10	18	34495	0.05	10	27561

Table 5.3: Maximum size of the messages (Max in Mbytes), average volume of communication (Vol. in Mbytes) and number of messages per processor (#Mess) for large matrices during factorization.

From the results in Table 5.3, it is difficult to make any definitive comment on the average volume of communication. Overall it is broadly comparable with sometimes MUMPS and sometimes SuperLU having lower volume, occasionally by a significant amount. However, although the average volume of messages with 64 processors can be comparable with both solvers, there is between one and two orders of magnitude difference in the average number of messages and therefore in the average size of the messages. This is due to the much larger number of messages involved in a fan-out approach (SuperLU) compared to a multifrontal approach (MUMPS). Note that, with MUMPS, the number of messages includes the messages (one integer) required by the dynamic scheduling algorithm to update the load on the processes.

The average volume of communication per processor of each solver depends very much on the number of processors. While, with SuperLU, increasing the number of processors will generally decrease the communication volume per processor it is not always the case with MUMPS. Note that adding some global information to the local dynamic scheduling algorithm of MUMPS will help to increase the granularity of the level 2 node subtasks without losing parallelism (see Section 3.1) and thus can result in a decrease in the average volume of communication on a large number of processors.

5.1.2 Study of the solve phase

We already discussed in Section 4.1 the difference in the numerical behaviour of the two solvers, showing that, in general, SuperLU will involve more steps of iterative refinement than MUMPS to obtain the same accuracy in the solution.

In this section, we focus on the time spent to obtain the solution. We apply enough steps of iterative refinement to ensure that the componentwise relative backward error ($Berr$) is less than $\sqrt{\varepsilon} = 1.48 \times 10^{-8}$. Each step of iterative refinement involves not only a forward and a backward solve but also a matrix-vector product with the original matrix. With MUMPS, the user can provide the input matrix in a very general distributed format (Amestoy et al. 1999). This functionality was used to parallelize the matrix-vector products. With SuperLU, the parallelization of the matrix-vector product was easier because the input matrix is duplicated on all the processors.

In Table 5.4, we report both the time to perform one solution step (using the factorized matrix to solve $\mathbf{A}x = b$) and when necessary ($Berr$ greater than $\sqrt{\varepsilon}$) the time to improve the solution using iterative refinement (lines with "+ IR"). With SuperLU, except on ECL32 and MIXTANK which did not require any iterative refinement, one step of iterative refinement was required and was always enough to reduce the backward error to $\sqrt{\varepsilon}$. With MUMPS, iterative refinement was only required on the matrix INVEXTR1 and the backward error was already so close to $\sqrt{\varepsilon}$ (on one processor $Berr = 3.06 \times 10^{-8}$) that on 4 and 8 processors no step of iterative refinement was required ($Berr$ for the initial solution was already equal to 1.17×10^{-8}). In this case, the time reported in the row "+ IR" corresponds to the time to perform the computation of the backward error. We first observe (compare, for example, Tables 5.1 and 5.4) that, on a small number of processors (less than 8), the solve phase is almost two orders of magnitude less costly than the factorization. On a large number of processors, because our solve phases are relatively less scalable than the factorization phases, the difference drops to one order of magnitude. On applications for which a large number of solves might be required per factorization this could become critical for the performance and might have to be addressed in the future. We show solution times for our smaller matrices in Table 5.5 where we have not run iterative refinement.

The performance reported in Tables 5.4 and 5.5 would appear to suggest that the regularity in the structure of the matrix factors generated by the factorization phase of MUMPS is responsible for a faster solve phase than that of SuperLU for up to 256 processors. On 512 processors, the solve phase of SuperLU is sometimes faster than that of MUMPS although in all cases the fastest solve time is recorded by MUMPS usually on a fewer number of processors. The cost of iterative refinement can significantly increase the cost of obtaining a solution. With SuperLU, because of static pivoting, it is more likely that iterative refinement will be required to obtain an accurate solution on numerically difficult matrices (see BBMAT, INVEXTR1 and TWOTONE). With MUMPS, the use of partial pivoting during the factorization will reduce the number of matrices for which iterative refinement is required. (In our set, only INVEXTR1 requires iterative refinement.) For both solvers, the use of MC64 to preprocess the matrix can also be considered to reduce the number of steps of iterative refinement and even avoid the need to use it in some cases (see Section 4.1).

Matrix	Order.	Solver	Number of processors								
			1	4	8	16	32	64	128	256	512
BBMAT	`AMD`	`MUMPS`	—	0.53	0.38	0.31	0.32	0.32	0.36	0.40	0.56
		`SuperLU`	—	1.77	1.59	1.05	1.00	0.80	0.70	0.70	0.66
		— + (IR)	—	3.38	2.10	1.60	1.27	1.05	0.90	0.89	0.79
	`ND`	`MUMPS`	—	0.38	0.37	0.26	0.29	0.31	0.35	0.37	0.54
		`SuperLU`	—	2.12	1.74	1.28	1.12	0.99	0.82	0.85	0.68
		— + (IR)	—	4.91	2.69	2.41	1.47	1.32	1.04	1.04	0.87
ECL32	`AMD`	`MUMPS`	—	0.80	0.50	0.40	0.41	0.40	0.45	0.52	0.83
		`SuperLU`	—	2.09	1.99	1.54	1.46	1.10	0.98	0.73	0.57
	`ND`	`MUMPS`	—	0.53	0.35	0.30	0.28	0.28	0.43	0.39	0.48
		`SuperLU`	—	1.76	1.96	1.38	1.41	1.05	0.93	0.68	0.53
INVEXTR1	`ND`	`MUMPS`	0.59	0.31	0.20	0.18	0.18	0.18	0.25	0.26	0.37
		— + (IR)	1.52	0.16	0.11	0.31	0.30	0.29	0.32	0.39	0.55
		`SuperLU`	1.45	0.77	0.73	0.55	0.51	0.46	0.36	0.34	0.28
		— + (IR)	2.69	1.58	1.11	0.90	0.74	0.67	0.54	0.52	0.44
MIXTANK	`ND`	`MUMPS`	0.67	0.27	0.19	0.16	0.16	0.15	0.19	0.24	0.35
		`SuperLU`	1.47	0.90	0.82	0.65	0.58	0.49	0.33	0.30	0.24
TWOTONE	`MC64`	`MUMPS`	—	1.03	0.92	0.97	0.98	0.98	1.03	1.13	1.41
	`+AMD`	`SuperLU`	—	3.26	3.02	2.52	2.24	1.84	1.56	1.38	1.21
		— + (IR)	—	25.84	11.13	12.63	4.18	3.64	2.27	1.84	1.55

Table 5.4: Solve time (in seconds) for large matrices on the CRAY T3E. " — + (IR) " shows the time spent improving the initial solution using iterative refinement. "—" indicates not enough memory.

Matrix	Ord.	Solver	Number of processors					
			1	4	8	16	32	64
FIDAPM11	`AMD`	`MUMPS`	0.48	0.25	0.24	0.21	0.20	0.20
		`SuperLU`	1.14	0.70	0.55	0.52	0.50	0.40
LHR71C	`MC64+AMD`	`MUMPS`	0.92	0.56	0.32	0.24	0.23	0.22
		`SuperLU`	2.39	2.48	2.76	2.19	2.02	1.83
RMA10	`AMD`	`MUMPS`	0.43	0.23	0.22	0.21	0.22	0.23
		`SuperLU`	0.79	0.66	0.54	0.52	0.37	0.31
WANG4	`AMD`	`MUMPS`	0.57	0.29	0.21	0.19	0.17	0.16
		`SuperLU`	1.01	1.01	0.77	0.88	0.85	0.65

Table 5.5: Solve time (in seconds) for small matrices on the CRAY T3E.

5.2 Memory usage

In this section, we study the memory used during factorization as a function of both the solver used and the number of processors, see Table 5.6.

We want first to point out that, because of the dynamic scheduling approach and the threshold pivoting used in MUMPS, the analysis phase cannot fully predict the space that will be required on each processor and an upper bound is therefore used for the memory allocation. With the static task mapping approach used in SuperLU, the memory used can be predicted during the analysis phase. In this section, we only compare the memory actually used by the solvers during the factorization phase. This includes reals, integers and communication buffers. Storage for the initial matrix is, however, not included but we have seen, in Amestoy et al. (1999), that the input matrix can also be provided in a general distributed format and can be handled very efficiently by the solver. This option is available in MUMPS. In SuperLU the initial matrix is currently duplicated on all processors[7].

Matrix	Ordering	Solver	Number of processors					
			4		16		64	
			Avg.	Max.	Avg.	Max.	Avg.	Max.
BBMAT	AMD	MUMPS	147	176	52	65	32	40
		SuperLU	113	114	50	51	33	34
	ND	MUMPS	114	118	44	53	28	35
		SuperLU	124	128	60	61	43	44
ECL32	AMD	MUMPS	190	212	55	64	32	41
		SuperLU	113	115	42	44	24	25
	ND	MUMPS	132	139	39	44	25	28
		SuperLU	79	81	33	34	21	22
INVEXTR1	ND	MUMPS	65	85	23	28	17	22
		SuperLU	47	48	22	22	15	16
FIDAPM11	AMD	MUMPS	65	67	25	30	16	19
		SuperLU	38	39	16	16	10	10
LHR71C	MC64	MUMPS	54	48	22	25	16	20
	+AMD	SuperLU	49	51	27	29	21	21
MIXTANK	ND	MUMPS	84	87	29	31	19	21
		SuperLU	55	56	23	23	14	15
RMA10	AMD	MUMPS	39	42	17	25	11	21
		SuperLU	32	33	15	16	10	11
TWOTONE	MC64	MUMPS	167	180	57	67	42	60
	+AMD	SuperLU	66	80	35	41	24	24
WANG4	AMD	MUMPS	69	82	22	23	15	20
		SuperLU	33	34	14	14	8	9

Table 5.6: Memory used during factorization (in Megabytes, per processor).

We notice, in Table 5.6, a significant reduction in the memory required when increasing the number of processors. We also see that, in general, SuperLU usually requires less memory than MUMPS although this is less apparent when many processors are used showing the better memory scalability of MUMPS. One can observe that there is little difference

[7]For MUMPS, note that the storage reported still includes another internal copy of the initial matrix in a distributed *arrowhead* form, necessary for the assembly operations during the multifrontal algorithm.

between the average and maximum memory usage showing both algorithms are well balanced, with SuperLU the better of the two.

Note that memory scalability can be critical on globally addressable platforms where parallelism increases the total memory used. On purely distributed machines such as the T3E, the main factor remains the memory used per processor which should allow large problems to be solved when enough processors are available.

6 Performance analysis on 3-D grid problems

To further analyse and understand the scalability of our solvers, we report in this section on results obtained for the 11-point discretization of the Laplacian operator on three-dimensional (NX, NY, NZ) grid problems.

We consider a set of 3D cubic (NX=NY=NZ) and rectangular (NX, NX/4, NX/8) grids on which a nested dissection ordering is used. The size of the grids used, the number of operations and the timings are reported in Table 6.1. When increasing the number of processors, we have tried as much as possible to maintain a constant number of operations per processor while keeping as much as possible the same shape of grids. It was not possible to satisfy all these constraints, thus the number of operations per processor is not completely constant.

Nprocs	Grid size			$\mathbf{LDL}^T$ factorization		**LU** factorization			
				MUMPS-SYM		MUMPS-UNS		SuperLU	
	NX	NY	NZ	flops $\times 10^9$	time	flops $\times 10^9$	time	flops $\times 10^9$	time
Cubic grids (nested dissection)									
1	29			3.6	18.8	7.2	24.0	7.2	57.0
2	33			8.0	20.8	16.0	29.5	15.9	62.3
4	36			13.4	19.9	26.8	28.1	26.8	53.3
8	41			30.1	18.5	60.1	33.9	60.0	61.5
16	46			59.1	20.7	118.1	34.4	117.9	62.7
32	51			112.7	24.3	225.3	46.3	224.9	65.7
64	57			222.7	30.3	445.1	67.3	444.7	76.1
128	64			444.2	51.6	887.8	113.9	886.4	80.7
Rectangular grids (nested dissection)									
1	96	24	12	2.2	13.2	4.5	16.6	4.5	31.1
2	110	28	13	4.8	13.1	9.5	17.5	9.6	36.6
4	120	30	15	9.0	12.0	17.9	17.0	17.9	35.4
8	136	34	17	18.4	13.8	36.8	19.5	36.6	33.0
16	152	38	19	36.5	13.3	72.8	24.6	72.7	42.2
32	168	42	21	67.8	14.9	135.5	27.5	135.3	43.7
64	184	46	23	118.2	19.3	236.2	35.8	236.0	51.3
128	208	52	26	243.1	27.4	485.8	53.6	485.6	60.7

Table 6.1: Factorization time (in seconds) on Cray T3E. **LU** factorization is performed for MUMPS-UNS and SuperLU, $\mathbf{LDL}^T$ for MUMPS-SYM.

Since all our test matrices are symmetric, we can use MUMPS to compute either an $\mathbf{LDL^T}$ factorization, referred to as MUMPS-SYM, or an **LU** factorization, referred to as MUMPS-UNS.

SuperLU will compute an **LU** factorization. Note that, for a given matrix, the unsymmetric solvers (SuperLU and MUMPS-UNS) perform roughly twice as many operations as MUMPS-SYM.

To overcome the problem of the number of operations per processor being non-constant, we first report in Figures 6.1 and 6.2 the Megaflop rate per processor for our three approaches on cubic and rectangular grids, respectively. In our context, the Megaflop rate is meaningful because on those grid problems the number of operations is almost identical for MUMPS-UNS and SuperLU (see Table 6.1), thus it corresponds to the absolute performance of the approach used for a given problem. We first notice that on up to 8 processors, and independently of the grid shape, MUMPS-UNS is about twice as fast as SuperLU and also has a much higher Megaflop rate than MUMPS-SYM. On 128 processors on both rectangular and cubic grids, all three solvers have similar Megaflop rates per processor.

In Figures 6.3 and 6.4, we show the parallel efficiency on cubic and rectangular grids respectively. The efficiency of a solver on p processors is computed as the ratio of its Megaflop rate per processor on p processors over its Megaflop rate on 1 processor.

In terms of efficiency, SuperLU is generally more efficient on cubic grids than MUMPS-UNS even on a relatively small number of processors. MUMPS-SYM is relatively more efficient than MUMPS-UNS and the MUMPS-SYM efficiency is very comparable to that of SuperLU. On a large number of processors SuperLU is significantly more efficient than MUMPS-UNS. The peak ratio between the methods is reached on cubic grids (128 processors) for which SuperLU is about three and two times more efficient than MUMPS-UNS and MUMPS-SYM, respectively.

Finally, we report in Table 6.2 a quantitative evaluation of the overhead due to parallelism on cubic grids, using the analysis tool vampir (Nagel et al. 1996). In the rows "computation", we report the percentage of the time spent doing numerical factorization. MPI calls and idle time due to communications or synchronization are reported in rows "overhead" of the table.

Nprocs	Grid size	MUMPS-SYM	MUMPS-UNS	SuperLU
4				
	(NX=36)			
	computation	69%	76%	87%
	overhead	31%	24%	13%
16				
	(NX=46)			
	computation	67%	69%	75%
	overhead	33%	31%	25%
64				
	(NX=57)			
	computation	50%	36%	56%
	overhead	50%	64%	44%

Table 6.2: Percentage of the factorization time (cubic grids) spent in computation and in overhead due to communication and synchronization.

Table 6.2 shows that SuperLU has less overhead than either version of MUMPS. We also observe a better parallel behaviour of MUMPS-SYM with respect to MUMPS-UNS, as analysed in

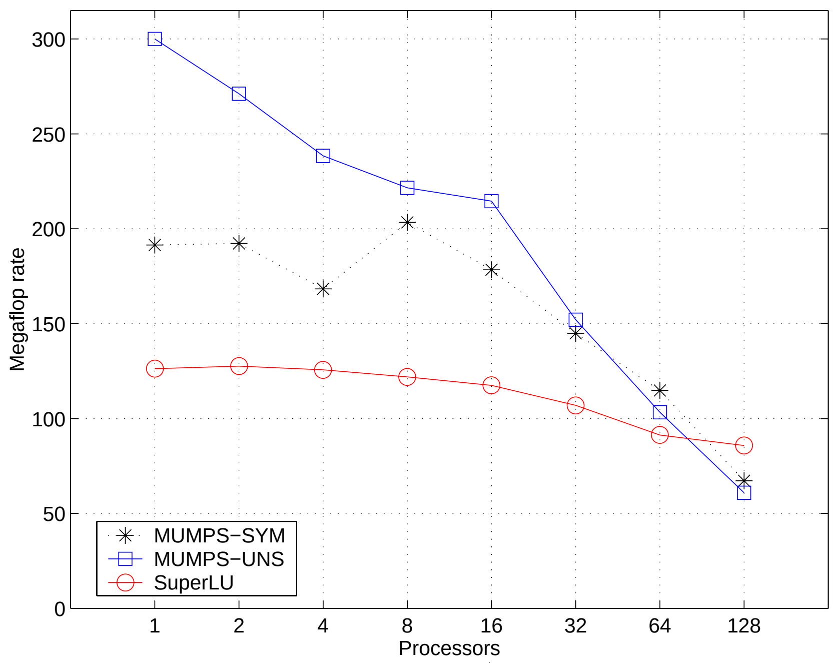

Figure 6.1: Megaflop rate per processor (cubic grids, nested dissection).

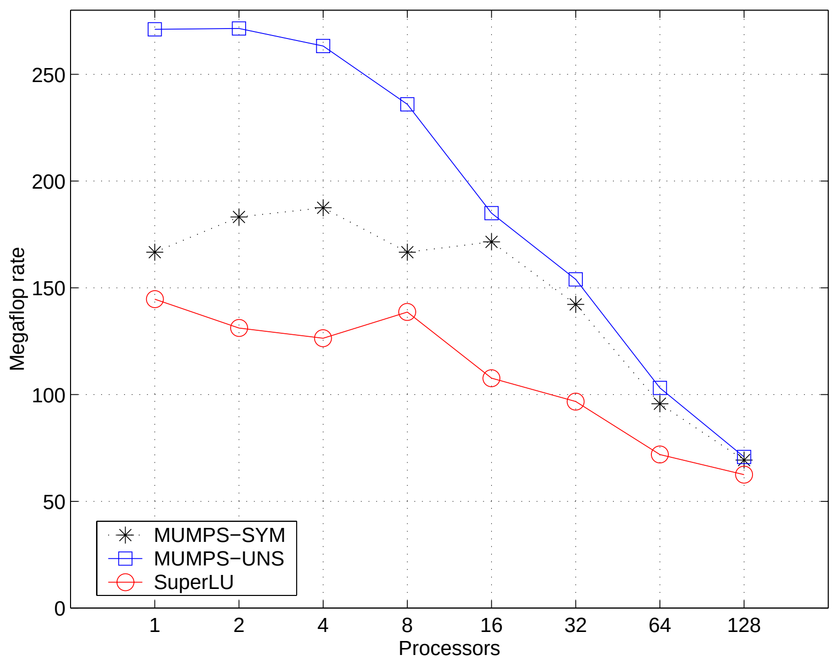

Figure 6.2: Megaflop rate per processor (rectangular grids, nested dissection).

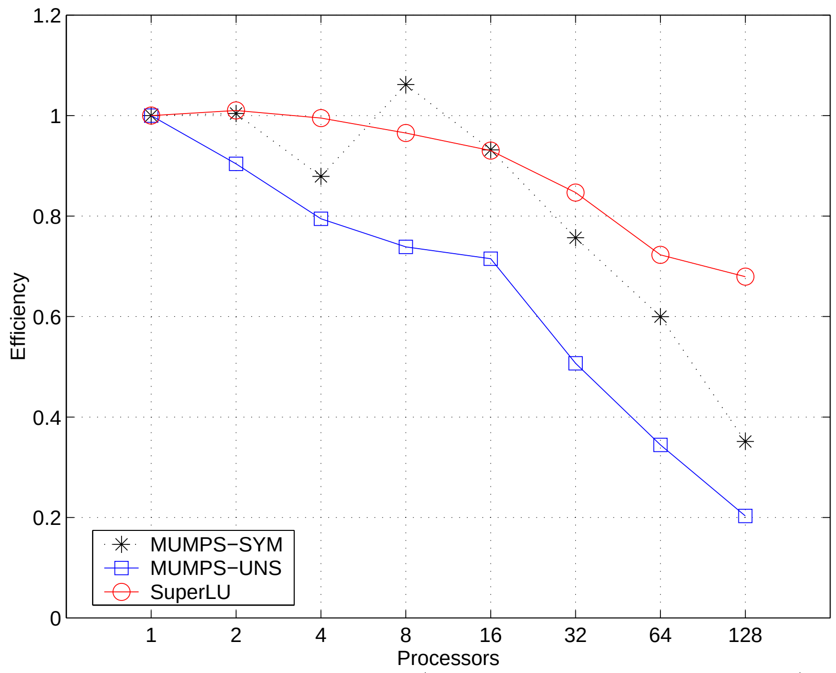

Figure 6.3: Parallel efficiency (cubic grids, nested dissection).

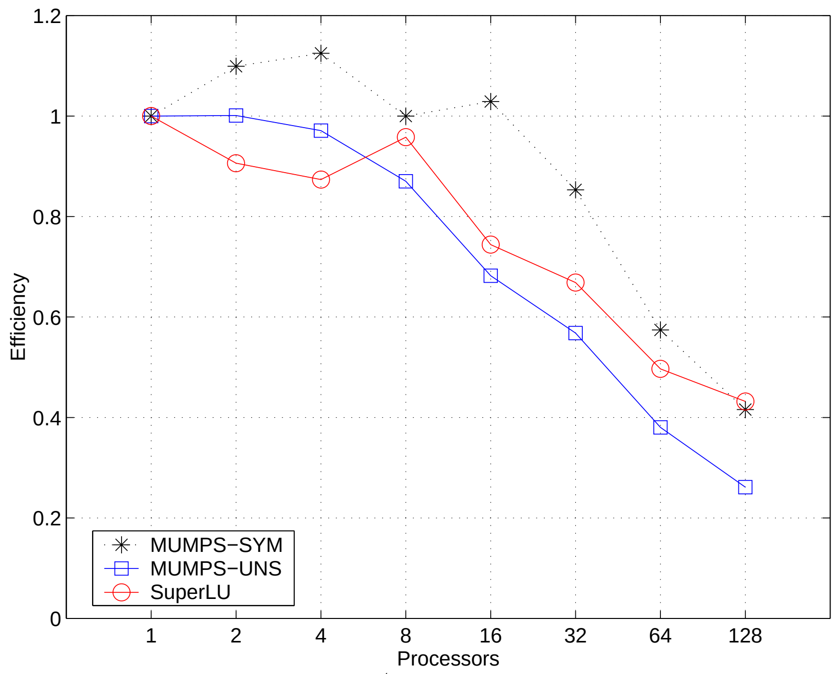

Figure 6.4: Parallel efficiency (rectangular grids, nested dissection).

Amestoy et al. (2000), which is mainly due to the fact that node level parallelism provides relatively more parallelism in a symmetric context.

7 Concluding remarks

In this paper, we have presented a detailed analysis and comparison of two state-of-the-art parallel sparse direct solvers—a multifrontal solver `MUMPS` and a supernodal solver `SuperLU`. Our analysis is based on experiments using a massively parallel distributed-memory machine—the Cray T3E, and a dozen matrices from different applications. Our analysis addresses the efficiency of the solvers in many respects, including the role of preordering steps and their costs, the accuracy of the solution, sparsity preservation, the total memory required, the amount of interprocessor communication, the times for factorization and triangular solves, and scalability. We found that both solvers have strengths and weaknesses. We summarize our observations as follows.

- Both solvers can benefit from a numerical preordering scheme implemented in `MC64`, although `SuperLU` benefits to a greater extent than `MUMPS`. For `MUMPS`, this helps reduce the number of off-diagonal pivots and the number of delayed pivots. For `SuperLU`, this may reduce the need for small diagonal perturbations and the number of iterative refinements. However, since this permutation is asymmetric, it may destroy the structural symmetry of the original matrix, and cause more fill-in and operations. This tends to introduce a greater performance penalty for `MUMPS` than for `SuperLU` although recent work by Amestoy and Puglisi (2000) might affect this conclusion. This is why by default, `MUMPS` does not use `MC64` on fairly symmetric matrices.

- `MUMPS` usually provides a better initial solution; this is due to the effect of dynamic versus static pivoting. With one step of iterative refinement, `SuperLU` usually obtains a solution with about the same level of accuracy.

- Both solvers can accept as input any fill-in reducing ordering, which is applied symmetrically to both the rows and columns. `MUMPS` performs better with nested dissection than minimum degree, because it can exploit the better tree parallelism provided by a nested dissection ordering, whereas `SuperLU` does not exploit this level of parallelism and its parallel efficiency is less sensitive to different orderings.

- Given the same ordering, `SuperLU` preserves the sparsity and the asymmetry of the L and U factors better. `SuperLU` usually requires less memory than `MUMPS`, and more so with smaller numbers of processors. On 64 processors, `MUMPS` requires 25–30% more memory on average.

- Although the total volume of communication is comparable for both solvers. `MUMPS` requires many fewer messages, especially with large numbers of processors. The difference can be up to two orders of magnitude. This is partly intrinsic to the algorithms (multifrontal versus fan-out), and partly due to the 1D (`MUMPS`) versus 2D (`SuperLU`) matrix partitioning.

- `MUMPS` is usually faster in both factorization and solve phases. The speed penalty for `SuperLU` partly comes from the code complexity in order to preserve the irregular sparsity pattern, and is partly due to more communication messages. With more processors, `SuperLU` shows better scalability, because its 2D partitioning scheme does a better job in keeping all the processors busy despite the fact that it introduces more messages.

As we said in the introduction, we started this exercise with the intention of comparing a wider range of sparse codes. However, as we have demonstrated in the preceding sections, the task of conducting such a comparison is very complex. We do feel though that the experience we have gained in this task will be useful in extending the comparisons in the future.

In the following tables, we summarize the major characteristics of the parallel sparse direct codes of which we are aware. A clear description of the terms used in the tables is given by Heath, Ng and Peyton (1991).

Code	Technique	Scope	Availability	Reference
CAPSS	Multifrontal	SPD	`www.netlib.org/scalapack`	(Heath and Raghavan 1997)
MUMPS	Multifrontal	SYM/UNS	`www.enseeiht.fr/apo/MUMPS`	(Amestoy et al. 1999)
PaStiX	Fan-in	SPD	see caption§	(Henon et al. 1999)
PSPASES	Multifrontal	SPD	`www.cs.umn.edu/~mjoshi/pspases`	(Gupta, Karypis and Kumar 1997)
SPOOLES	Fan-in	SYM/UNS	`www.netlib.org/linalg/spooles`	(Ashcraft and Grimes 1999)
SuperLU	Fan-out	UNS	`www.nersc.gov/~xiaoye/SuperLU`	(Li and Demmel 1999)
S+	Fan-out†	UNS	`www.cs.ucsb.edu/research/S+`	(Fu, Jiao and Yang 1998)
WSMP‡	Multifrontal	SYM	IBM product	(Gupta 2000)

Table 7.1: Distributed memory codes.
§ `www.dept-info.labri.u-bordeaux.fr/~ramet/pastix`
† Uses QR storage to statically accommodate any LU fill-in
‡ Only object code for IBM is available. No numerical pivoting performed.

Code	Technique	Scope	Availability	Reference
GSPAR	Interpretative	UNS	Grund	(Borchardt, Grund and Horn 1997)
MA41	Multifrontal	UNS	`www.cse.clrc.ac.uk/Activity/HSL`	(Amestoy and Duff 1993)
MA49	Multifrontal QR	RECT	`www.cse.clrc.ac.uk/Activity/HSL`	(Amestoy, Duff and Puglisi 1996*b*)
PanelLLT	Left-looking	SPD	Ng	(Ng and Peyton 1993)
PARDISO	Left-right looking	UNS	Schenk	(Schenk, Gärtner and Fichtner 2000)
PSLDLT†	Left-looking	SPD	SGI product	(Rothberg 1994)
PSLDU†	Left-looking	UNS	SGI product	(Rothberg 1994)
SuperLU	Left-looking	UNS	`www.nersc.gov/~xiaoye/SuperLU`	(Demmel et al. 1999)

Table 7.2: Shared memory codes
† Only object code for SGI is available

Acknowledgments

We want to thank James Demmel, Jacko Koster and Rich Vuduc for very helpful discussions. We are grateful to Chiara Puglisi for her comments on an early version of this article and her help with the presentation. We also want to thank John Reid for his comments on the first version of this paper.

References

Amestoy, P. R. and Duff, I. S. (1993), 'Memory management issues in sparse multifrontal methods on multiprocessors', *Int. J. Supercomputer Applics* **7**, 64–82.

Amestoy, P. R. and Puglisi, C. (2000), An unsymmetrized multifrontal LU factorization, Technical Report RT/APO/00/3, ENSEEIHT-IRIT. Also Lawrence Berkeley National Laboratory Report LBNL-46474.

Amestoy, P. R., Davis, T. A. and Duff, I. S. (1996*a*), 'An approximate minimum degree ordering algorithm', *SIAM J. Matrix Analysis and Applications* **17**(4), 886–905.

Amestoy, P. R., Duff, I. S. and L'Excellent, J.-Y. (2000), 'Multifrontal parallel distributed symmetric and unsymmetric solvers', *Comput. Methods in Appl. Mech. Engrg.* **184**, 501–520.

Amestoy, P. R., Duff, I. S. and Puglisi, C. (1996*b*), 'Multifrontal QR factorization in a multiprocessor environment', *Numerical Linear Algebra with Applications* **3**(4), 275–300.

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y. and Koster, J. (1999), A fully asynchronous multifrontal solver using distributed dynamic scheduling, Technical Report RAL-TR-1999-059, Rutherford Appleton Laboratory.

Arioli, M., Demmel, J. W. and Duff, I. S. (1989), 'Solving sparse linear systems with sparse backward error', *SIAM J. Matrix Analysis and Applications* **10**, 165–190.

Ashcraft, C. and Grimes, R. (1999), SPOOLES: An object-oriented sparse matrix library, *in* 'Proceedings of the Ninth SIAM Conference on Parallel Processing'. See `http://www.netlib.org/linalg/spooles`.

Borchardt, J., Grund, F. and Horn, D. (1997), Parallel numerical methods for large systems of differential-algebraic equations in industrial applications, Technical Report 382, Weierstraß-Institut für Angewandte Analysis und Stochastik, Berlin.

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

Duff, I. S. and Koster, J. (1999), On algorithms for permuting large entries to the diagonal of a sparse matrix, Technical Report RAL-TR-1999-030, Rutherford Appleton Laboratory. Also appeared as Report TR/PA/99/13, CERFACS, Toulouse, France. To appear in *SIAM Journal on Matrix Analysis and Applications.*

Duff, I. S., Grimes, R. G. and Lewis, J. G. (1997), The Rutherford-Boeing Sparse Matrix Collection, Technical Report RAL-TR-97-031, Rutherford Appleton Laboratory. Also Technical Report ISSTECH-97-017 from Boeing Information & Support Services, Seattle and Report TR/PA/97/36 from CERFACS, Toulouse.

Fu, C., Jiao, X. and Yang, T. (1998), 'Efficient sparse LU factorization with partial pivoting on distributed memory architectures', *IEEE Trans. Parallel and Distributed Systems* **9**(2), 109–125.

Gilbert, J. R. and Liu, J. W. H. (1993), 'Elimination structures for unsymmetric sparse LU factors', *SIAM J. Matrix Analysis and Applications* **14**, 334–354.

Gupta, A. (2000), WSMP: Watson Sparse Matrix Package Part I – direct solution of symmetric sparse systems Version 1.0.0β, Technical Report TR RC-21886, IBM research division, T.J. Watson Research Center, Yorktown Heights. http://www.cs.umn.edu/~agupta/wsmp.html.

Gupta, A., Karypis, G. and Kumar, V. (1997), 'Highly scalable parallel algorithms for sparse matrix factorization', *IEEE Trans. Parallel and Distributed Systems* **8**(5), 502–520.

Heath, M. T. and Raghavan, P. (1997), 'Performance of a fully parallel sparse solver', *Int. J. Supercomputer Applications* **11**(1), 49–64.

Heath, M. T., Ng, E. G. Y. and Peyton, B. W. (1991), 'Parallel algorithms for sparse linear systems', *SIAM Review* **33**, 420–460.

Henon, P., Ramet, P. and Roman, J. (1999), A mapping and scheduling algorithm for parallel sparse fan-in numerical factorization, *in* 'EuroPar'99 Parallel Processing', Lecture Notes in Computer Science, No. 1685, Springer-Verlag, Berlin, Heidelberg, New York, pp. 1059–1067.

HSL (2000), 'A collection of Fortran codes for large scale scientific computation'. http://www.cse.clrc.ac.uk/Activity/HSL.

Karypis, G. and Kumar, V. (1998), METIS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*, University of Minnesota.

Li, X. S. and Demmel, J. W. (1998), Making sparse Gaussian elimination scalable by static pivoting, *in* 'Proceedings of Supercomputing', Orlando, Florida.

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', San Antonio, Texas.

Li, X. S., Demmel, J. W., Bailey, D. H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kapur, A., Martin, M. C., Tung, T. and Yoo, D. J. (2000), Design, Implementation and Testing of Extended and Mixed Precision BLAS, Tech. Rep. LBNL-45991, Lawrence Berkeley National Laboratory, Berkeley. (also LAPACK Working Note #149). Submitted to *ACM Transactions on Mathematical Software*.

Liu, J. W. H. (1985), ‘Modification of the minimum degree algorithm by multiple elimination’, *ACM Trans. Math. Softw.* **11**(2), 141–153.

Nagel, W. E., Arnold, A., Weber, M., Hoppe, H.-C. and Solchenbach, K. (1996), ‘VAMPIR: Visualization and Analysis of MPI Resources’, *Supercomputer* **12**(1), 69–80.

Ng, E. G. and Peyton, B. W. (1993), ‘Block sparse Cholesky algorithms on advanced uniprocessor computers’, *SIAM J. Scientific Computing* **14**, 1034–1056.

Pellegrini, F., Roman, J. and Amestoy, P. (1999), Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering, *in* ‘Proceedings of Irregular’99, San Juan’, Lecture Notes in Computer Science **1586**, Springer-Verlag, pp. 986–995.

Rothberg, E. (1994), Efficient sparse Cholesky factorization on distributed-memory multiprocessors, *in* J. G. Lewis, ed., ‘Proceedings 5th SIAM Conference on Linear Algebra’, SIAM Press, Philadelphia, p. 141.

Schenk, O., Gärtner, K. and Fichtner, W. (2000), ‘Efficient sparse lu factorization with left–right looking strategy on shared memory multiprocessors’, *BIT* **40**(1), 158–176.