Analysis, Tuning and Comparison of Two General Sparse Solvers for Distributed Memory Computers*

Patrick R. Amestoy†, Iain S. Duff‡, Jean-Yves L'Excellent§, and Xiaoye S. Li¶

July 18, 2000

Abstract

We describe the work performed in the context of a Franco-Berkeley funded project between NERSC-LBNL located in Berkeley (USA) and CERFACS-ENSEEIHT located in Toulouse (France). We discuss both the tuning and performance analysis of two distributed memory sparse solvers (`SuperLU` from Berkeley and `MUMPS` from Toulouse) on the 512 processor Cray T3E from NERSC (Lawrence Berkeley National Laboratory). This project gave us the opportunity to improve the algorithms and add new features to the codes. We then quite extensively analyse and compare the two approaches on a set of large problems from real applications. We further explain the main differences in the behaviour of the approaches on artificial regular grid problems. As a conclusion to this activity report, we mention a set of parallel sparse solvers on which this type of study should be extended.

*The project is supported by the Franco-Berkeley Fund. This project also utilized resources of the National Energy Research Scientific Computing Center (NERSC) which is supported by the Director, Office of Advanced Scientific Computing Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

†amestoy@enseeiht.fr, ENSEEIHT-IRIT, 2 rue Camichel, 31071 Toulouse, France. Visiting NERSC.

‡I.Duff@rl.ac.uk, CERFACS, 42 Ave G Coriolis, F-31527 Toulouse Cedex 1, France.

§jeanyves@nag.co.uk, ENSEEIHT-IRIT, France now at NAg Ltd, Wilkinson House, Oxford OX2 8DR, England.

¶xiaoye@nersc.gov, NERSC, Lawrence Berkeley National Lab, MS 50F, 1 Cyclotron Rd., Berkeley, CA 94720. The research of this author was supported in part by the National Science Foundation Cooperative Agreement No. ACI-9619020 and NSF Grant No. ACI-9813362.

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 the two codes, `MUMPS` [3, 6], and `SuperLU` [23]. The first uses a multifrontal approach with dynamic pivoting for stability while the second is based on a supernodal technique with static pivoting. We discuss these two codes in more detail in Section 2.

We compare the performance of the two codes in Section 6, where we show that such a comparison can be fraught with difficulties even when, as in this case, the code authors are involved in the study. In Section 7, regular grids problems are used to further illustrate and analyse the difference between the two approaches. We had originally thought to do a comparison of more sparse codes but, given the documented difficulties 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.

Two of the most important parameters are the use of a 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. We discuss these aspects in detail in Sections 4.1 and 4.2 respectively.

From our investigations, we have identified ways in which both codes can be improved. Most of these improvements have been implemented in the framework of this project and are discussed in Section 5. Future work will involve implementation of the remaining points and an extended comparison with other sparse parallel direct codes. For the record, we list the codes that we know about (including commercial codes) in Section 8.

2 Description of algorithms used

In this section, we briefly describe the main characteristics of the algorithms that we are comparing and highlight the major differences between them.

Both 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`, at each node, some steps of Gaussian elimination are performed on a dense frontal matrix 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 capacity 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 than in the case of the native ordering. The standard ordering used by MUMPS is the approximate minimum degree (AMD) ordering [1] while SuperLU uses the multiple minimum degree ordering (MMD) [24] on this symmetrized pattern. However, in the experiments using a minimum degree code in the following sections, we consider only the AMD ordering since both codes can generate this using the HSL routine MC47, it is usually far quicker than MMD, and it produces a symbolic factorization close to that produced by MMD. We also use a nested dissection ordering (ND) that varies from problem to problem. Sometimes we use the ON-MeTiS ordering from MeTiS, sometimes the MFR ordering from Christian Damhaug of Veritas, and sometimes the nested dissection/haloamd ordering from SCOTCH [27]. Additionally, in some cases it is 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 [22] to perform this preordering and scaling [14] and indicate clearly when this is done in the forthcoming results. 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 row and column scaled (each row/column is divided by its maximum value).

In both approaches a pivot order is defined by the analysis and symbolic factorization stages (which can be the same for both algorithms) but, in both cases, numerical considerations might prevent strict adherence to this order during numerical factorization. In the case of MUMPS, the modulus of the prospective pivot is compared with the largest modulus of an entry in the column and it is only accepted if this is greater than a threshold value, typically a value between 0.001 and 0.1 (our default value is 0.01). Note that, even though MUMPS can choose pivots from off the diagonal, the largest entry in the column might not be available for pivoting at this stage because all entries in its row may not be fully summed. This threshold pivoting strategy is common in sparse Gaussian elimination and helps to avoid excessive growth in the matrix factorization and so directly reduces the bound on the backward error. If a prospective pivot fails the test and cannot be used within the partial factorization at a node, all that happens is that it is kept in the Schur complement and is passed to the parent node. Eventually all of the columns will be available for pivoting, at the root if not before, so that a pivot can be chosen from that column. Thus the numerical factorization can respect the threshold criterion but at a cost of increasing the size of the frontal matrices and potentially causing more work and fill-in than were forecast. For the SuperLU approach, a static pivoting strategy is used and we keep rigorously 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 1-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 single precision perturbation to the original matrix. The result is that the factor is not exact and iterative refinement may then need to be used. Note that, after iterative refinement, we obtain an accurate solution in all the cases that we tested. If problems were still to occur then extended precision BLAS could be used.

Both approaches use higher level BLAS to effect the elimination operations. However, in MUMPS the frontal matrices are always square and are assumed dense and Level 3 BLAS are used. It is possible that there are zeros in the frontal matrix especially if there are delayed pivots or the matrix structure is markedly unsymmetric but the present implementation

Figure 1: Illustration of the asynchronous behaviour of the `MUMPS` factorization phase.

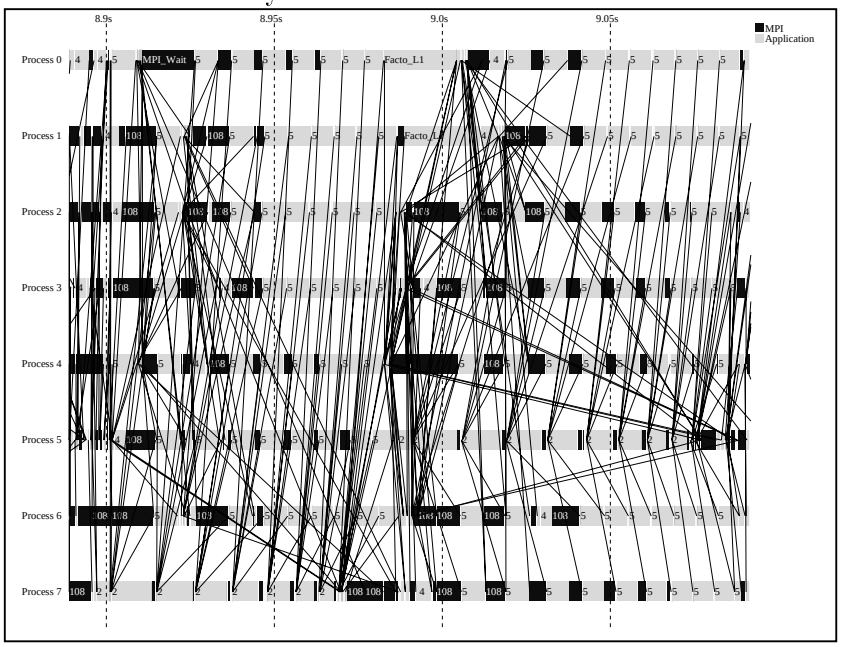
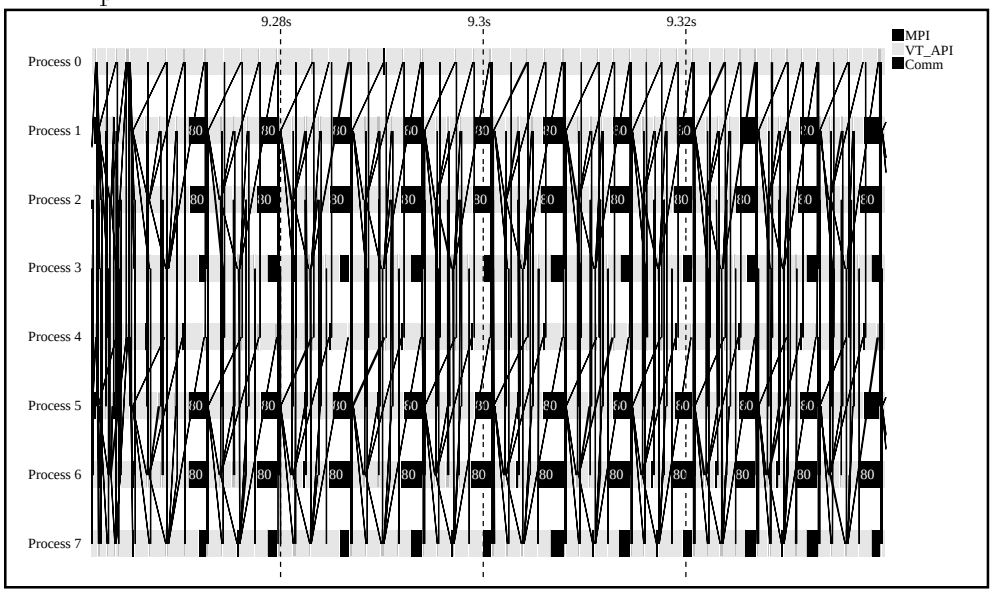


Figure 2: Illustration of the relatively more synchronous behaviour of the `SuperLU` factorization phase.



takes no advantage of this sparsity and all the counts measured assume the frontal matrix is dense. 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 not as good as that of MUMPS. The effect of these characteristics in reducing performance is to some extent balanced 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.

The parallelism within MUMPS is at two levels. The first uses the structure of the assembly tree, exploiting the fact that nodes which 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 giving rise to *node parallelism*, is either one-dimensional (referred to as *1D-node parallelism*) or two-dimensional (at the root and referred to as *2D-node parallelism*). Node parallelism depends on the order 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.

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.

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, a directed acyclic computational graph (*dag*) needs to be built.

In Figures 1 and 2, we use a vampir trace [25] 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 the 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 message. This is due to the asynchronous scheduling algorithm used in MUMPS which may have better chance of overlapping communication with computation.

3 Test environment

Throughout this paper, we will use a set of test problems to illustrate the performance of our algorithms. Most 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,

although there has been a few 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 that we used during the PARASOL Project.

Note that Release 1.3.0.3 of the Cray operating system was used to obtain most results in this paper. Some results (see Section 5) have been obtained with a more recent version of the operating system (Release 1.3.0.4). As far as the performance of the two solvers is concerned, the main difference between these two releases is that with the older release the default size of the internal `MPI` buffer was unlimited while, with the most recent release, the default size is 4 Kbytes. The impact of the size of the `MPI` buffer on the performance of the two solvers is discussed in Section 5.

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.)
NASASRB.RUA	54870	2677324	1.00	NASA (CFD)
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)
Real Symmetric Assembled (RSA)				
Matrix name	Order	No. of entries		Origin
BMWCRA_1	148770	5396386		PARASOL (MSC.Software)
BMW3_2	227362	5757996		PARASOL (MSC.Software)
CRANKSG2	63838	7106348		PARASOL (MSC.Software)
NASASRB.RSA	54870	1366097		NASA (CFD)
HOOD	220542	5494489		PARASOL (INPRO)
SHIP_003	121728	4103881		PARASOL (Det Norske Veritas)

Table 1: Test matrices. $^{(*)}$ StrSym is the number of nonzeros matched by nonzeros in symmetric locations divided by the total number of entries (that is, a symmetric matrix has value 1.0).

Our test matrices come from the forthcoming Rutherford-Boeing Sparse Matrix Collection [13] [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

[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://iftp.cs.umn.edu/pub/sparse/`

[5] Matrix included in Rutherford-Boeing Collection

are available from Parallab (Bergen, Norway)[6]. There are larger matrices available from this Web site but it is difficult to process them symbolically on a single processor.

Note that matrices MIXTANK and INVEXTR1 have been modified because of out-of-range values (causing underflow). To maintain, as much as possible, the same numerical behaviour during the factorization all entries with an exponent smaller than -300 have been set to numbers with the same mantissa but with an exponent of -300. (In fact the out-of-range underflow values in the initial matrix had an exponent equal to either -308 or -309.)

For each linear system, the right-hand side vector is generated so that the true solution is a vector of all ones.

4 Impact of preprocessing and numerical issues

In this section, we first study the impact of the preprocessing of the matrix on both solvers. In this preprocessing, we use column permutations to permute large entries onto the diagonal. 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 will 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 comments on the cost of the analysis phase

In [14], Duff and Koster developed an algorithm for permuting a sparse matrix so that the diagonal is large relative to the off-diagonals. More precisely, when the matrix is reordered and scaled, the resulting matrix has diagonal entries all equal to one in modulus with off-diagonal entries all of modulus less than or equal to one. They have also written a computer code, `MC64`, to implement this algorithm. It is this code that we have used in our subsequent studies. 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. In the case of the multifrontal scheme, the analysis phase chooses a pivotal sequence based purely on the structure of the matrix and assumes that diagonal pivoting is possible even in the unsymmetric case. During the factorization phase, the pivot chosen by the analysis is checked for numerical stability using a threshold pivoting criterion. If it passes the test, it is used as pivot and the factorization follows exactly the route forecast by the analysis. If, however, it fails the test, then the pivot is not chosen. The size of the Schur complement matrix, the so called contribution block, sent to the parent node will then increase, potentially increasing both storage and operation count for the factorization above that forecast by the analysis. Intuitively, the less the number of times we need delay pivoting in this way, the better we retain the good performance forecast by the analysis. If the initial matrix has large

[6] Web page `http://www.parallab.uib.no/parasol/`

diagonals (relative to the off-diagonals), it does not guarantee that subsequent reduced matrices will have this property but it should help.

For the SuperLU code, such a permutation can be even more crucial. If, for example, the (1,1) entry were zero, the analysis in the original algorithm would fail. After the modifications discussed later in Section 5, the analysis can process this matrix and static pivoting in the SuperLU code would replace this entry by $\varepsilon^{1/2}||A||$ where ε is the machine precision. A factorization will still be produced, although of a modified matrix. Thus the effect of this preordering can be very dramatic for SuperLU as we see in the results in Table 3.

The MC64 code of [14] 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 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 2. A more complete set of results is provided in Table 17 of the Appendix. In Figure 3, 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 2 that, on very unsymmetric matrices (LHR71C and TWOTONE), using MC64 is really necessary to factor these matrices efficiently. Both matrices have zeros in 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 in the SuperLU factorization and only a factorization of a nearby matrix is obtained. Although MUMPS can factor a matrix with zeros on the diagonal, the fill-in obtained without MC64 makes the use of MC64 necessary in this case also. In the case of 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) 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 nonsymmetric 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).

The use of MC64 can also improve the quality of the factors, the numerical behaviour of the factorization phase, and the number of steps of iterative refinement required to improve the accuracy of the solution. This is illustrated in Table 3 where we show the number of steps of iterative refinement required to reduce the componentwise relative backward error, $Berr = \max_i \frac{|r|_i}{(|A|\cdot|x|+|b|)_i}$ [7], to machine precision ($\approx$ 2.2e-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

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
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
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
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
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

Table 2: Influence of permuting large entries onto the diagonal (using MC64) on the size of factors and the number of operations to perform factorization. StrSym denotes the structural symmetry after ordering. $^{(*)}$ Only the estimation given by the analysis is provided (there was not enough memory to perform the factorization).

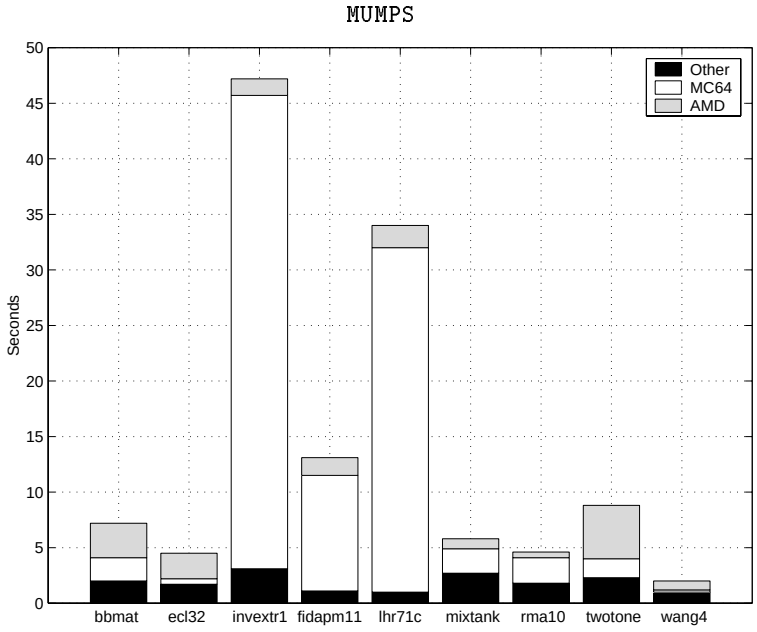
as $Err = \frac{||x_{true} - x||}{||x_{true}||}$. This table illustrates the impact of the use of MC64 on the quality of the initial solution obtained with both solvers prior to iterative refinement. Additionally, it shows that, thanks to numerical partial pivoting, the initial solution is generally more accurate with MUMPS than with SuperLU. These two observations are further confirmed on a larger number of test matrices in Table 18 of the Appendix. In the case of the MUMPS solver, MC64 can also result in a reduction in the number of off-diagonal pivots and in the number of delayed pivots. For example on 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 3 (results on matrix BBMAT) that MC64 does not always improve the numerical accuracy of the solution obtained with SuperLU.

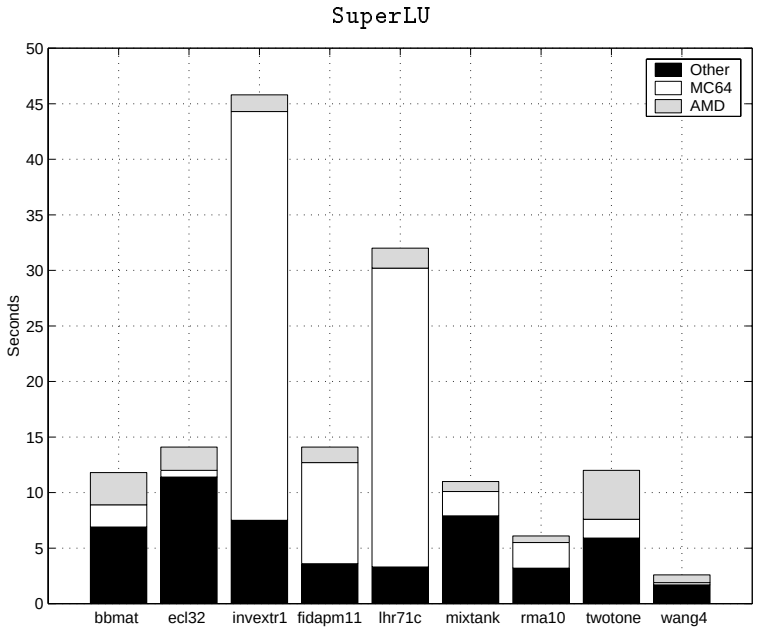
		SuperLU		MUMPS	
Matrix	Iter.	No MC64	MC64	No MC64	MC64
BBMAT		Err=2.1e-03	Err=5.6e-01	Err= 1.3e-06	Err=6.5e-08
	0	Berr=4.0e-09	1.3e-05	Berr=7.4e-11	1.2e-11
	1	Berr=7.7e-16	4.5e-11	Berr=3.2e-16	3.2e-16
	2	Berr=5.2e-16	9.7e-15	Berr=3.2e-16	2.7e-16
	3	Berr=	4.7e-16		
		Err= 2.5e-09	Err=2.4e-09	Err= 3.0e-09	Err=3.5e-09
LNSP3937		Err=1.6e-01	Err=2.6e-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	1.9e-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-10	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 3: Illustration of the convergence of iterative refinement.

As one might expect, we see that, for matrices with a fairly symmetric pattern (see results in Table 2 on matrix FIDAPM11), 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 Figure 3), 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.

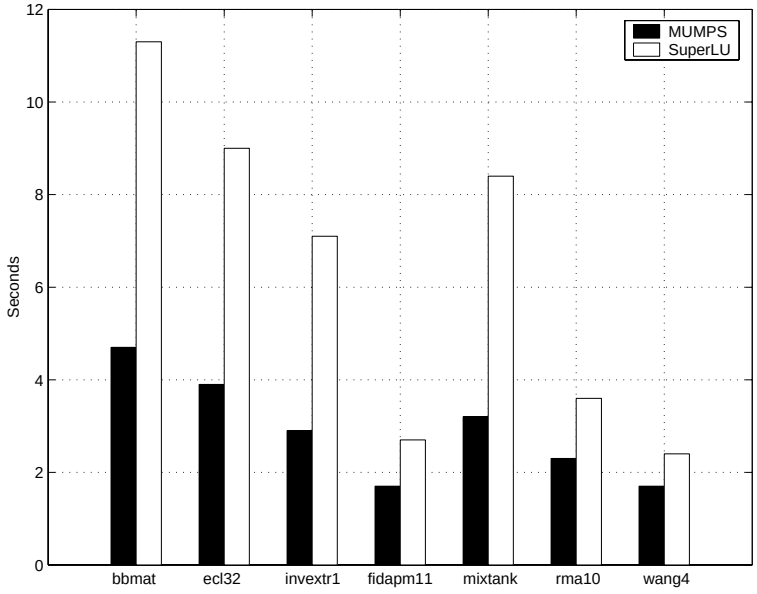
Figure 3: Analysis time study





We finally compare, in Figure 4, the time spent by the two solvers during the analysis phase when reordering is based only on AMD (MC64 is not invoked). Since the

Figure 4: Time comparison of the analysis phases of MUMPS and SuperLU. MC64 preprocessing is NOT used and AMD ordering is used.



time spent in AMD is very similar in both cases, this gives a good estimation of the cost difference of the analysis phase of the two solvers. During the SuperLU analysis phase, all the unsymmetric structures involved during the factorization are computed and the directed acyclic graph [17] of the unsymmetric matrix must be built and mapped onto the processors. (Path searches in the directed acyclic graph are used to reduce communications.)

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, the impact of the choice of the symmetric permutation on the cost of the numerical phase. As was observed in [3], 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 this further in Section 5.1.3. We also notice that, independently of the ordering used, SuperLU exploits the asymmetry of the matrix better than MUMPS. (See results with matrix BBMAT of structural symmetry 0.53 as well as results on more unsymmetric matrices provided in Table 2.)

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
NASASRB.RUA	AMD	MUMPS	24.2	9.5
		SuperLU	23.9	9.5
	ND	MUMPS	21.2	6.9
		SuperLU	21.0	6.8

Table 4: Influence of the symmetric sparsity reorderings on the cost of the factorization phase of unsymmetric matrices. (`MC64` is not used.)

5 Tuning our sparse solvers

In addition to enabling an in-depth comparison of our approaches to the solution of sparse linear systems on distributed memory computers, the Franco-Berkeley collaboration gave us the opportunity of introducing new features and improving our algorithms, much of these motivated by the in-depth study. We discuss both code tuning and algorithmic improvement of the two sparse solvers in this section before continuing with our comparison, because we will use the improved versions of our codes in the results described in the following sections.

Porting a code to a new platform always provides a good opportunity for improving its performance not only on the target platform but also on previously implemented environments. For both solvers, the first phase in the optimisation on a new platform consists in adjusting a set of machine dependent parameters to fit the target machine. These parameters are used to balance the parallel machine's speed of computation and communication, and the algorithm's degree of parallelism. In the case of `MUMPS`, the porting of the code to the 512 processor CRAY T3E-900 gave us the opportunity to study the behaviour of the code on a larger number of processors than used in our previous work [3, 6]. From our set of machine dependent parameters we choose appropriate parameters to address this issue. Other algorithmic modifications were motivated by having more processors available to us than formerly.

When porting `SuperLU` to the IBM SP2, we found that there is a rather big performance gap between Level 2.5 and Level 3 BLAS. This motivated us to refine the numerical kernel to always use Level 3 BLAS, see Section 5.1.1. Even on the same machine, we found that `MPI` programming environment changes (for example the internal buffer size) may result in a dramatic performance difference, see Section 5.1.2. This enabled us to identify possible enhancements to the `SuperLU` code to make its performance more robust.

We will occasionally, and only as complementary information, use `MPI` routine names since it might help readers to understand which functionality of `MPI` has been used. However, knowledge of `MPI` syntax is certainly not required to understand the algorithms. Users interested in `MPI` should refer to the `MPI` user manual or to [12].

5.1 Description of the modifications made to `SuperLU`

`SuperLU` has a machine dependent parameter to control the granularity of the local computation tasks and of the messages. This is called the *maximum block size*. Setting it appropriately strikes a good balance between computation, communication and degree of parallelism. For a larger value, we expect to have more local computation and less communication. This is good for machines with relatively faster computation than communication, such as the IBM SP2. Our experience showed that a good value for this parameter on the IBM SP2 is around 40, while on the Cray T3E a good value is around 24.

Another parameter is called relaxation, by which we mean that we relax the criteria used to define supernodes. That is, we may amalgamate several columns into a supernode even if their row structures are different so long as they do not differ by too much. More precisely, at the bottom of the elimination tree of $A^T + A$, we perform the amalgamation as follows: node i is merged with its parent node j when the subtree rooted at j has less

than or equal to r nodes, where, r is a small integer and can be set by the user. Setting $r = 1$ disables amalgamation. For this amalgamated supernode, we will explicitly store some zeros and perform operations on them. By doing so, we will have a larger block size and will reduce the irregularity in the nonzero structure of the matrix. On the IBM SP2, we can afford more aggressive amalgamation, and r is usually set to 10. On the Cray T3E, we use a smaller number such as 6.

In the following subsections, we describe several algorithmic improvements that were made or were discovered but still remain to be done in future developments.

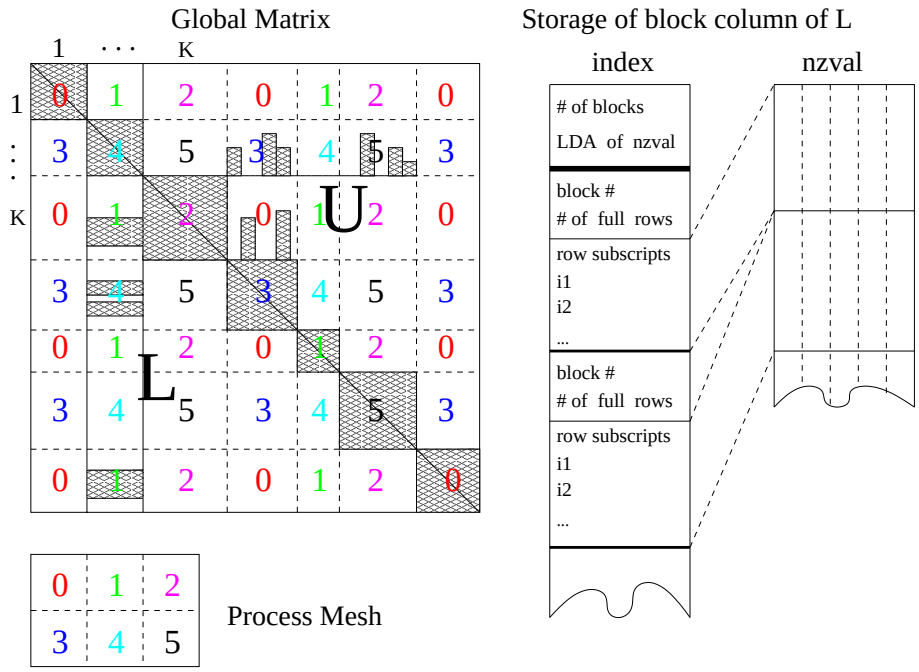
5.1.1 Exploitation of Level 3 BLAS

In order to understand the numerical kernel routines used in `SuperLU`, we first explain how we partition the matrix into blocks of submatrices, and how the blocks are assigned to the processes.

Our matrix partitioning is based on the notion of an *unsymmetric supernode* introduced in [11]. 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. If there are N supernodes in an n-by-n matrix, there will be N^2 blocks of non-uniform size. Figure 5 illustrates such a block partition. The off-diagonal blocks may be rectangular and may 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 mod P_r, J mod P_c) of the process mesh. During the factorization, block $L(I, J)$ is only needed by the processes on the process row (I mod P_r). Similarly, block $U(I, J)$ is only needed by the processes on the process column (J mod 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 formed along the supernode boundaries. 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 is the performance expected.

In this 2D mapping, each block column of L resides on more than one process, namely, a column of processes. For example in Figure 5, 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 schema on the right of Figure 5 depicts the data structure to store the nonzero blocks on a process. Besides the numerical values stored in a Fortran-style array `nzval` in column major order, we need the information to interpret the location and row subscript of each nonzero. This is stored in an integer array `index`, which includes the information for the whole block column and for each individual block in it. Note that many off-diagonal blocks are zero and hence not stored. Neither do we store the zeros in a nonzero block. Both lower and upper triangles of the diagonal block are stored in the L data structure. A process owns parts of $\lceil N/P_c \rceil$ block columns of L, so it needs $\lceil N/P_c \rceil$ pairs of `index/nzval` arrays.

Figure 5: The 2D block-cyclic layout and the data structure used in `SuperLU`.



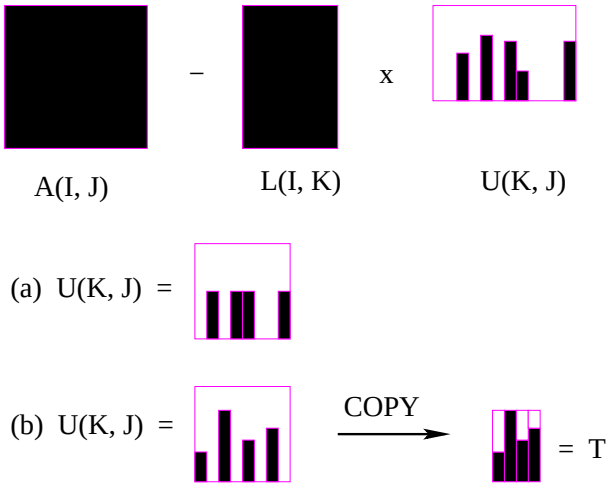
The main numerical kernel is a block update corresponding to the rank-k update to the Schur complement: $A(I,J) \leftarrow A(I,J) - L(I,K) \times U(K,J)$, see Figure 6. In the earlier versions of SuperLU, this computation was based on Level 2.5 BLAS. That is, we call the Level 2 BLAS routine GEMV (matrix-vector 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, such as 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 6 (a).
  Since the segments are stored contiguously in memory, we can call GEMM directly.

- The segments in $U(K,J)$ are of different heights, as shown in Figure 6 (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 will perform some extra floating-point operations for those padding zeros. The copying itself does not incur a run time cost, because we need to access this part of data anyway and, once accessed, it will stay in cache throughout the GEMM computation. 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 matrices, this Level 3 BLAS kernel improves the uniprocessor factorization speed by about 20% to 40% on the IBM SP2. A similar 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 6: Illustration of the numerical kernels used in `SuperLU`.



5.1.2 Pipelining and nonblocking send and receive

In this subsection, we first describe in detail how the parallel factorization algorithm utilizes the pipeline effect. Then we discuss how to improve the performance robustness by introducing immediate sends and receives. The following notation will be used in Figures 7 and 8, and throughout the discussion. Matlab notation is used for integer ranges and submatrices.

- Process IDs
  - $PROC_c(K)$: the set of column processes that own block column K
    For example, in Figure 5, $PROC_c(K) = \{2, 5\}$.
  - $PROC_r(K)$: the set of row processes that own block row K
    For example, in Figure 5, $PROC_r(K) = \{0, 1, 2\}$.
  - P_{K+1} : the process in $PROC_c(K+1) \cap PROC_r(K+1)$
  - me : the process rank as illustrated in Figure 5
- Tasks labelled in Figure 8
  - F (...) : **Factorize** a block column or a block row[7]
  - S (...) : **Send** a block column or a block row

[7] There is also communication involved in this task, but it is negligible, and so is omitted in the discussion.

– R (...) : **Receive** a block column or a block row
– U$^{(K)}$(...) : **Update** the trailing submatrix using $L(:, K)$ and $U(K, :)$

Figure 7 outlines the parallel sparse LU factorization algorithm. There are three steps in the K-th iteration of the loop. In step (1), only processes $PROC_c(K)$ participate in factoring block column $L(K : N, K)$. In step (2), only processes $PROC_r(K)$ participate in factoring block row $U(K, K + 1 : N)$. The rank-b update by $L(K + 1 : N, K)$ and $U(K, K+1 : N)$ in step (3) represents most of the work and also exhibits more parallelism than the other two steps, where b is the block size of the K-th block column/row.

For ease of understanding, the algorithm presented in Figure 7 has been simplified. The actual implementation uses a *pipelined* organization so that processes $PROC_c(K+1)$ will start step (1) of iteration $K + 1$ as soon as the rank-b update (step (3)) of iteration K to block column $K + 1$ finishes, before completing the update to the trailing matrix $A(K+1 : N, K+2 : N)$ owned by $PROC_c(K+1)$. Figure 8 illustrates this idea using Steps K and $K + 1$ of the algorithm. In the figure, we show the activities of the four process groups along the time line. The path marked with the dashed line represents the critical path, that is, the parallel runtime could be reduced only if the critical path is shortened. The block factorization tasks "F (...)" are usually on the critical path, whereas the update tasks "U (...)" are often overlapped with the other tasks. There is lack of parallelism for the "F (...)" tasks in Steps (1) and (2), because only one set of column processes or row processes participate in these tasks. This pipelining mechanism alleviates this problem. For instance, on 64 processors of the Cray T3E, we observed speedups of between 10% and 40% over the non-pipelined implementation.

Currently, for the message transfer tasks "S (...)" and "R (...)", we use MPI's standard send and receive operations, `mpi_send` and `mpi_recv`. In Figure 8, we see idle time (longer send) during the sending of "S ($L(:, K + 1)$)" for process P_{K+1} on the critical path. This could happen if the sender and receiver are required to handshake before proceeding. That is, process P_{K+1} posts `mpi_send` long before processes $PROC_r(K)$ post the matching `mpi_recv`, and the sender must be blocked to wait for `mpi_recv`. This handshaking is not always required with `mpi_send`. Here is how the MPI standard defines its semantics [30, pp. 32]:

> "... It does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. ... The message might be copied directly into the matching receiver buffer, or it might be copied into a temporary system buffer. In the first case, the send call will not complete until a matching receive call occurs. In the second case, the send call may return ahead of the matching receive call, allowing a single-threaded process to continue with its computation. The MPI implementation may make either of these choices. It might block the sender or it might buffer the data. "

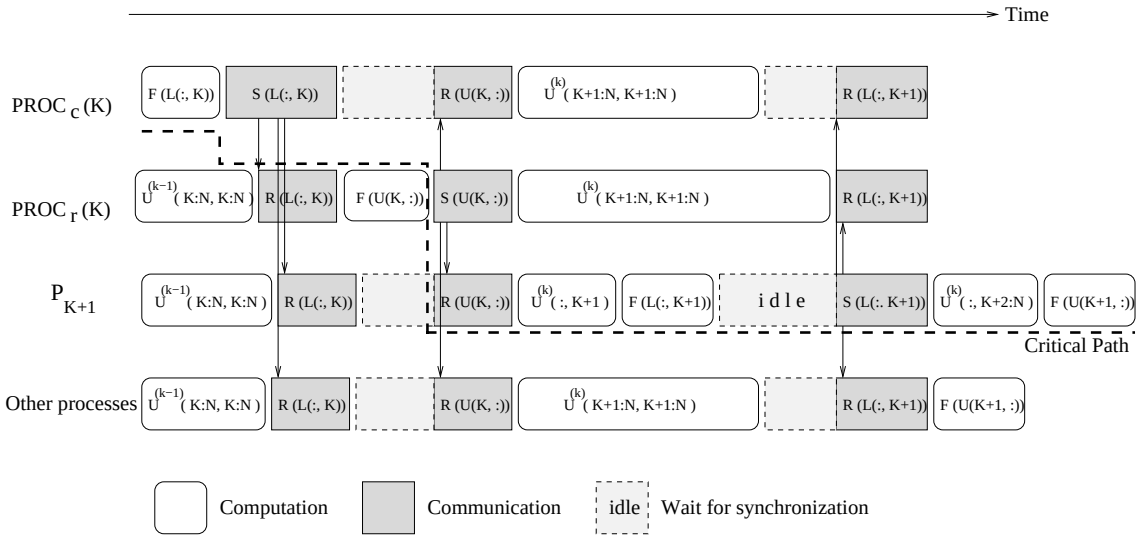
Very often, an MPI implementor chooses to use two different protocols depending on the length of the message:

- *Short protocol (eager protocol)* for small messages.
  The sender copies the data into the system buffer, and returns immediately without

Figure 7: Outline of the parallel factorization algorithm used in `SuperLU`.

for block $K = 1$ **to** N **do**
  (1) **if** ($me \in PROC_c(K)$)
    Factorize block column $L(K:N, K)$
    Send $L(K:N, K)$ to the processes in my row who need it
  else
    Receive $L(K:N, K)$ from one of the processes $PROC_c(K)$ (if I need it)
  endif
  (2) **if** ($me \in PROC_r(K)$)
    Factorize block row $U(K, K+1:N)$
    Send $U(K, K+1:N)$ to processes in my column who need it
  else
    Receive $U(K, K+1:N)$ from one of the processes $PROC_r(K)$ (if I need it)
  endif
  (3) **for** $J = K+1$ to N **do**
    for $I = K+1$ to N **do**
      if ($me \in PROC_r(I)$ and $me \in PROC_c(J)$
        and $L(I,K) \neq 0$ and $U(K,J) \neq 0$)
        Update trailing submatrix $A(I,J) \leftarrow A(I,J) - L(I,K) \cdot U(K,J)$
      endif
    end for
  end for
end for

Figure 8: Illustration of the pipeline at Steps K and $K+1$ during the `SuperLU` factorization.



waiting for the matching receive. The additional copying usually increases the message transfer overhead, however, in many asynchronous algorithms it may be effectively smeared by overlapping the computation with the communication. This is exactly what we observed from the `SuperLU` performance.

- *Long protocol* for large messages.
  The sender first sends a "request-to-send" message to the receiver, then waits for the receiver to send back a "ready-to-receive" message. The sender now transmits the message data directly into the receiver's user space without buffering. This protocol requires handshaking of the sender and receiver, but the message transfer overhead is smaller than for the short protocol because we do not pay the extra cost of copying.

Whether a message is short or long is determined by the size of the MPI system buffer. For example, on the Cray T3E, the user may determine the size of the system buffer by setting an environment variable `MPI_BUFFER_MAX`. If a message length exceeds this value, the long protocol will be used. When the short protocol is used, the idle time shown in Figure 8 would disappear.

We have seen big differences in performance between setting `MPI_BUFFER_MAX` to unlimited and to 4 Kbytes. Table 5 shows the timing differences. The most dramatic is on 2 processors, where the difference is 74%.

Nprocs	1	2	4	8	16	32	64	128
Grid size	29	33	36	41	46	51	57	64
unlimited	57	62	53	62	63	66	76	81
4 Kbytes	58	108	92	103	104	102	119	111

Table 5: `SuperLU` factorization time in seconds for the cubic grid problems with `MPI_BUFFER_MAX` set to unlimited and to 4Kbytes.

It is somewhat unpleasant that the performance of our code depends on the MPI system buffer size. We plan to make some algorithmic changes so that the code performance is less dependent on the underlying MPI implementation and is more portable. Our proposal is to introduce the nonblocking send and receive, `mpi_isend` and `mpi_irecv` as follows.

- For the sender, we simply replace `mpi_send` by `mpi_isend`. This could also eliminate the idle time during the send "S $(L(:, K+1))$" shown in Figure 8.

- For the receiver, we will post `mpi_irecv` much earlier than we actually need the data. For example, for processes $PROC_r(K)$, we could post "R $(L(:, K+1))$" before "U $(A(K+1:N, K+1:N))$". That is, as soon as we have received a message using `mpi_wait`, we will post the `mpi_irecv` for the next message, before performing the local computation with the just-arrived message.

Although it is not hard to implement this idea, we need to provide an extra buffer on the receiving process to take care of one outstanding message. We still have to experiment with this scheme and see how sensitive the performance is to the size of the system buffer.

5.1.3 Exploiting more parallelism from the sparsity and the elimination dags

One simple sparsity ordering strategy for unsymmetric matrices is to apply a symmetric ordering algorithm on a symmetrized matrix. In the earlier versions of `SuperLU`, we usually adopted the sparsity ordering strategies based on the structure of A^TA, whether using minimum degree or nested dissection. After the ordering, we computed an elimination tree also based on the structure of A^TA, and followed by another column reordering so that the nodes of the tree are numbered in a postorder. The rationale of performing an ordering on A^TA is based on the following observation [16]. The structure of the symbolic Cholesky factor of A^TA is an upper bound on the structure of the LU factors for any row permutations (corresponding to partial pivoting). So minimizing the upper bound tends to minimize the actual nonzero structure of L and U. But since in the distributed memory version of `SuperLU`, we use a static pivoting strategy in place of partial pivoting (i.e. the pivots are chosen on the diagonal), the Cholesky factor of A^TA is much too loose an upper bound for the actual structure.

When pivots are chosen on the diagonal, the symmetrized matrix $A^T + A$ is better. The Cholesky factor of $A^T + A$ gives a tighter bound on the LU factors than that of A^TA. For example, using an `AMD` ordering on the structure of $A^T + A$, we have seen more than twice the fill-in reduction and even more reduction in operations, compared with using an `AMD` ordering on A^TA.

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 **box** the process which owns the block.

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

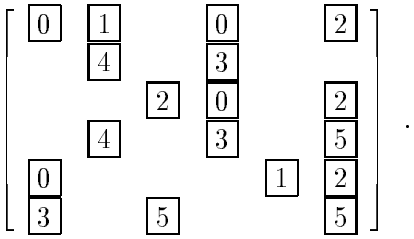
$$\begin{bmatrix} \boxed{0} & \boxed{1} & \boxed{0} & \boxed{1} \\ & \boxed{3} & \boxed{2} & \boxed{3} \\ \boxed{0} & \boxed{1} & \boxed{0} & \\ & \boxed{3} & & \boxed{3} \end{bmatrix}.$$

  Although node 2 is the parent of node 1 in the elimination tree (associated with $A^T + A$), not all processes in column 2 depends on column 1. Only process 1 depends on the L block on process 0. Process 3 could start factoring column 2 at the same time as process 0 is factoring column 1, before process 1 starts factoring 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 factor 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 false 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 elimination dags.
  Consider another matrix with 6-by-6 blocks mapped onto the 2-by-3 process mesh



  Columns 1 and 3 are independent in the elimination dags. The column process sets $\{0, 3\}$ and $\{2, 5\}$ could start factoring 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 to any task in Step 3, process 2 does not factor column 3 immediately. We may change this task precedence by giving the factor task of a later iteration higher priority than the update tasks of the previous iterations, because the former is more like on the critical path. This would exploit better the task independencies 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.

5.2 Description of the modifications made to MUMPS

The initial version of MUMPS will be referred to as *Version 4.0* whereas the modified version will be referred to as *Version 4.1*.

Most of our machine dependent parameters that control the efficiency of the code are designed to take into account both the uniprocessor and multiprocessor characteristics of the computers. Because of our 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 [21]. 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 parallelism available. We found that smaller granularity tasks could be used on the CRAY T3E than on the IBM SP2 because node parallelism is relatively more effective on the CRAY T3E than on the IBM SP2. This can be explained by the relatively faster rate of communication to Megaflop rate on the CRAY T3E than on the IBM SP2 (see Table 6). That is to say that the communication is relatively more efficient on the CRAY T3E.

Our first algorithmic modification was to modify our initial dynamic scheduling algorithm to better control the distribution of the tasks to the processors. The critical

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 peak	4 μsec	40μsec
Bandwidth/Effective performance	0.88	0.24

Table 6: Performance of the CRAY T3E-900 and the IBM SP2. The factorization of matrix WANG4 was used to estimate the effective uniprocessor performance of the computers.

part of this algorithm is when a process associated with a tree node decides to reassign some of its work, corresponding to a one-dimensional partitioning of the rows, to a set of so-called *worker* processes. We call such a node a one-dimensional parallel node. In `MUMPS` Version 4.0, 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.) 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, 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 should give 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 not critical (it is by default chosen at 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 strategy has been implemented for unsymmetric matrices. For symmetric matrices, we already have an irregular partitioning of the non-fully summed rows and this would make such a dynamic adjustment of the block size difficult to implement in an efficient way. In addition such a change would be unlikely to produce the same benefits as in the unsymmetric (fixed block size) case. We leave further study of this as an open problem for future consideration.

The second and main algorithmic modification of `MUMPS` is the introduction of asynchronous immediate receives (`mpi_irecv`) during the factorization phase. In Version 4.0, the communications are fully asynchronous and are based on an immediate send (`mpi_isend`). The receiver normally matches the asynchronous send with a test for the availability of the message, potentially followed by an effective reception of the message (`mpi_recv`). A problem with this mechanism occurs when messages are much larger than the `MPI` internal buffer size (whose default size on the CRAY T3E is 4 Kbytes).

In this case, independently of the time difference between the issue of the send and the issue of the receive, almost all the data to be exchanged will start to be sent only when the receive process actually issues a receive instruction providing user space required for the communication to proceed. This can very significantly affect the potential algorithmic overlapping between computation and communication. However, if we can use an immediate receive (`mpi_irecv`), which can be interpreted as having a separate "spawned" process implementing the reception, the reception can progress in parallel with the process that issued the `mpi_irecv`, so that potentially the receive can have completed (that is the complete message is available in the user space of the process issuing the `mpi_irecv`) at the time when we test for the availability of the message. Note that by doing so we have also overlapped the copying from the `MPI` buffer to the user space.

In this paragraph, we recall what is specific to a CRAY T3E implementation of `MPI` communications that will be relevant to the analysis of the performance of our sparse solvers. We focus on the differences between standard asynchronous communications (`mpi_send` and `mpi_recv`) and immediate communications (`mpi_isend` and `mpi_irecv`) since it corresponds to the two communication patterns respectively used by `SuperLU` and `MUMPS`. On the CRAY T3E, only an `MPI` receive buffer (no `MPI` send buffer) of default size 4 Kbytes (Release 1.3.0.4) is used to implement the communications. When the message to be sent is smaller than the size of the `MPI` receive buffer then the sender directly writes into the `MPI` receive buffer of the destination process. Note that this will occur independently of the way the send is actually performed (`mpi_send` or `mpi_isend`) and the receiver performs the reception (`mpi_recv` or `mpi_irecv`). As long as space is available in the `MPI` receive buffer, the send process will not be blocked and, when the receive process actually issues a reception, only coping from the `MPI` buffer to the user space will be involved. For large messages (larger than the `MPI` receive buffer) then the `MPI` buffer is not used. The communication of the effective data will only start when the receiver posts the receive instruction (`mpi_recv` or `mpi_irecv`). Note that an `MPI` buffer of unlimited size (as was the case in an earlier version of the CRAY T3E MPI implementation, Release 1.3.0.3) would then result in causing communications based on standard sends and receives (`mpi_send` and `mpi_recv`) to perform very similarly to asynchronous immediate communications (`mpi_isend` and `mpi_irecv`).

We illustrate, in Table 7, the impact of the size of the `MPI` buffer on the performance of our algorithm on a large matrix of our test set. The standard receive (`mpi_recv`) is used to match the immediate send `mpi_isend`. One first sees that, on our example, the size of

Size (in Bytes) of the `MPI` buffer								
0	128	512	1K	4K (*)	64K	512K	2Mega	8Mega
37.7	37.0	37.4	38.3	37.6	32.8	28.3	26.4	26.4

Table 7: Influence of the `MPI` buffer size on the time (in seconds) for the factorization of matrix CRANKSG2 on 8 processors of the CRAY T3E. `mpi_recv` is used to match `mpi_isend`. (*) default value on the CRAY T3E.

.

the `MPI` buffer strongly influences the factorization time. Secondly, with the default size of the `MPI` buffer (4 Kbytes), the use of a standard receive (`mpi_recv`) to match an immediate asynchronous send (`mpi_isend`) does not lead to a good overlapping of communication with

computation. As was explained before, matching the immediate sends (`mpi_isend`) with immediate receives (`mpi_irecv`) should address both issues (that is, independence with respect to `MPI` buffer size and communication overlapping).

Although, in the context of `MUMPS`, the use of an immediate receive seems quite natural, we explain in Section 5.2.1, why it required more algorithmic developments than might have been expected. The main issue with using an immediate receive in our very asynchronous environment is that we cannot tell *a priori* which message we are receiving. That is, the `mpi_irecv` request must be sent to receive any type of message from any source. In our implementation, we avoid some possible added complications by restricting ourselves to a single `mpi_irecv` pending request. In Section 5.2.1, we explain this in more detail, discuss the algorithmic issues, illustrate the difference in the behaviour of the code (with and without `mpi_irecv`), and analyse the performance gains.

Note that Table 7 refers to the internal `MPI` buffers. There can also be a buffer defined by the user which may be a separate staging area or may be directly in the working space of the user process. In the following, we will always prefix the term buffer by `MPI` when we mean the `MPI` internal buffers so that the use of the term buffer, without this prefix, will always refer to the user-defined buffer space. This will apply also to the use of the term in the Tables and Algorithms.

The performance improvement due to the modifications described in this section (including the use of immediate receive with the default size of 4 Kbytes for the `MPI` buffer) is illustrated in Table 8. We see that, on medium size symmetric and unsymmetric matrices from our set, and using both a minimum degree based ordering (`AMD`) and a nested dissection ordering (`MeTiS`), the gains mainly come from the better parallel behaviour of the modified version of the code.

Matrix	Ordering	Version	Number of processors						
			1	2	4	8	16	32	64
HOOD	MeTiS	4.0	—	—	25.1	12.1	6.3	3.7	3.5
		4.1	—	—	15.0	8.2	4.4	4.0	2.4
NASASRB.RSA	AMD	4.0	26.7	18.4	10.2	6.4	6.3	6.1	6.1
		4.1	26.7	17.5	8.8	5.4	4.2	3.8	3.7
INVEXTR1	MeTiS	4.0	—	23.1	18.0	15.6	14.9	14.4	14.5
		4.1	—	23.1	11.8	8.5	7.5	7.0	6.8
NASASRB.RUA	AMD	4.0	—	23.1	18.0	15.6	14.9	14.4	14.5
		4.1	—	23.1	11.8	8.5	7.5	7.0	6.8
WANG4	AMD	4.0	30.6	19.0	11.2	7.3	5.5	5.0	4.4
		4.1	30.6	18.9	11.1	7.0	5.2	4.3	3.9

Table 8: Time (in seconds) for factorization using two versions of the `MUMPS` solver. — means not enough memory.

In the next section, when we study in detail the benefit coming from our main algorithmic modification (that is, the use of asynchronous immediate receives), we will use larger test matrices and perform a more extensive performance analysis.

5.2.1 Introducing immediate receives during factorization

In this section, we describe the modifications required for our introduction of asynchronous immediate receives in `MUMPS` Version 4.0. For the sake of clarity, we first describe how we modify the reception of messages involved during dynamic scheduling. We then show how to modify this solution to handle all type of receptions involved in the `MUMPS` code. We study the performance obtained on large symmetric and unsymmetric matrices and illustrate, using `vampir` traces, the gains obtained on one of our largest problems.

As we mentioned in the previous sections, in `MUMPS` Version 4.0, communications are asynchronous and based on immediate sends with explicit buffering in user space. A Fortran module was designed for this purpose and is briefly described in [6]. On the destination process, the reception of the messages will be the key point for the synchronization and scheduling of the work. In fact, message reception can be invoked in the following three situations :

1. *Dynamic scheduling*:
   Blocking and non-blocking receives are used to drive the scheduling of the tasks on each process.

2. *Task ordering:*
   A process may have to receive and treat a "late" message to be able to finish its current task.

3. *Insufficient space in send buffer:*
   To avoid deadlock, the corresponding process tries to receive messages until space becomes available in its local send buffer.

We recall that to avoid the drawback of centralised scheduling, distributed dynamic scheduling is used. A pool of tasks private to each process is used to implement dynamic scheduling. All tasks ready to be activated by a process are stored in the pool of tasks local to the process. Each process then executes Algorithm 1.

Algorithm 1 *Dynamic scheduling*

While not all nodes processed
  If [local pool empty] **Then**
    blocking receive for a message; process the message
  Else If [message available (– *mpi_iprobe* –)] **then**
    receive and process message
  Else
    extract work from the pool, and process it
  End If
End While

To modify the dynamic scheduling algorithm in the context of immediate receives, we introduce in Algorithm 2 the procedure *Try_to_receive_and_process_message* for which the parameter `blocking` indicates whether we want to wait for the arrival of a message or not.

Algorithm 2 *Dynamic scheduling with immediate receive*

While not all nodes processed
    `blocking` = false
    If [local pool empty] `blocking` = true
    Try_to_receive_and_process_message (blocking)
    If [no message received and pool not empty] **Then**
        extract work from the pool and process it
    End If
End While

The procedure *Try_to_receive_and_process_message* is described by Algorithm 3. We differentiate between the cases of a receive request pending and of the blocking wait for a message. Finally, we always receive all short messages related to dynamic scheduling (one integer holding the updated load of the other processes) that are ready to be received before reactivating an immediate receive request. We remind the reader that the load of a process is defined as the total number of operations ready to be performed. In fact, since a maximum of one `mpi_irecv` request is pending at a given time, part of the benefit of issuing `mpi_irecv` might be lost if one does not force, at this point of the algorithm, the reception of these short and trivial to process messages (in fact, they are really used to emulate an `mpi_put` in a portable way). If we take the example of a large message following a single dynamic scheduling message then, until the small message is processed, it is not possible to have the `mpi_irecv` active on the large message. Thus we postpone the start of the `mpi_irecv` on the large message which might cause a delay in the sending process because of situation 3. A delay in the sender could then cascade causing a blocking receive because of a "late message" as in situation 2. An immediate receive request will thus be issued each time a message is received and processed.

Actually, Algorithm 3 should also be designed to handle messages corresponding to situation 2 (task ordering). During task ordering, we need to perform a blocking receive on a so called "late message". Such cases are illustrated in [4, 6] and are due to the fact that, although the algorithm is asynchronous, we still have to maintain a partial order between the tasks. Our asynchronous algorithm has been designed so that, when a "late message" needs to be received, we can guarantee that this message has already been sent. A blocking receive on this message can thus safely be performed. Note that in Algorithm 3, the parameter `blocking` only specifies that we are blocked until the reception of any message.

The main difficulty introduced by the use of a blocking receive on a given message in Algorithm 3 is that a "wrong" message might already be in our receive buffer because of an asynchronous pending receive request. Algorithm 4 shows how we have modified Algorithm 3 to solve this problem. An additional parameter `LateMessage` has been introduced to characterise the expected message. Combined with `blocking` set to true, `LateMessage` indicates the type of message (process source and message label) that is expected. Setting `LateMessage` to "any message" will enable us to perform a blocking receive on any message as required by the dynamic scheduling Algorithm 2.

For the sake of clarity, two new local variables have been introduced in Algorithm 4.

Algorithm 3 *Try_to_receive_and_process_message (blocking)*

If [Receive request pending] **Then**
  If [blocking] **Then**
    Wait for the end of pending receive request (– *mpi_wait* –)
    Process message
  Else If [Message in buffer (– *mpi_test* –)] **Then**
      Process message
  End If
Else
  If [blocking] **Then**
    Blocking receive for any message (– *mpi_recv* –)
    Process message
  Else If [Message ready to be received (– *mpi_iprobe* –)] **Then**
      Receive message in buffer (– *mpi_recv* –)
      Process message
  End If
End If
If [No receive request pending] **Then**
  process all ready-to-be-received messages related to dynamic scheduling
  Reactivate an immediate receive request (– *mpi_irecv* –)
End If

`MessRecv` indicates that a message has been received during the pending receive request. `RightMessage` is true when the message received is the expected one (that is has the same characteristics as `LateMessage`). Comments are in parentheses using small and slanted fonts. Note that, if `LateMessage` is true in a call to Algorithm 4, then `blocking` must also be true.

Between lines 5 and 8 of the algorithm we are, in the case mentioned before, of having already received a message in our local buffer which is not the expected one. Since we know that the expected message has been sent we can do a blocking probe on the expected message (line 7) and force the current process to wait for the availability of the late message before processing the current message in the buffer. This will enable us to perform a non-blocking probe on the expected message at line 16 and *conclude at line 18 that the message must have been processed if it is not ready to be received.* In fact, we must also guarantee that the expected message has not be stored in the receive buffer by an immediate receive request. Therefore, we must be sure that between lines 7 and 16 another immediate receive has not been issued. The only place which could cause the activation of an immediate receive is at line 13 where Algorithm 4 might be called recursively. A receive can be issued during the processing of almost any message giving rise to a situation 2 (task ordering) or 3 (insufficient space in the send buffer). To avoid such an occurrence, we suspend the activation of immediate receives at line 12 and only reactivate it again at line 15. At line 31, we must then test whether activation of an immediate receive is authorised.

One final minor problem introduced by the use of immediate receive is that it must now be the responsibility of the sending process to decide if the receive buffer of the destination

Algorithm 4 *Try_to_receive_and_process_message(* `blocking`, `LateMessage` *)*

```
0. MessRecv=false; RightMessage=true
1. If [ Receive request pending ] Then
2.    If [ blocking ] Then
3.       Wait for the end of pending request; (– mpi_wait – )
4.       MessRecv=true ( – message is in buffer – )
5.       If [ The message received ≠ LateMessage ] Then
6.          RightMessage=false
7.          Blocking probe for expected message ( – mpi_probe – )
8.       End If
9.    Else If [ Message in buffer ] MessRecv=true
10.   End If
11.   If [ MessRecv ] Then
12.      If [ Not RightMessage ] Suppress activation of immediate receive
13.      Process the message already in buffer
14.      If [ Not RightMessage ] Then
15.         Re-authorise activation of immediate receive
16.         If [ LateMessage ready to be received (– mpi_iprobe – ) ] Then
17.            Receive and process it
18.         Else (– expected message is already received and processed – )
19.         End If
20.      End If
21.   End If
22. Else
23.   If [ blocking ] Then
24.      Blocking receive for any message ( – mpi_recv –)
25.      Process message
26.   Else If [ Message ready to be received ( – mpi_iprobe –) ] Then
27.            Receive message in buffer (– mpi_recv – )
28.            Process message
29.   End If
30. End If
31. If [ No receive request pending and Immediate receive authorised ] Then
32.   Receive and process all ready-to-be-received messages related to dynamic scheduling
33.   Reactivate an immediate receive request (– mpi_irecv – )
34. End If
```

process is large enough to process it. In Version 4.0 of the code, the destination process always checked the size of the message to be received before receiving it. The maximum size of the receive buffer can only be *estimated* during the analysis phase because of the delay in selecting pivots caused by numerical pivoting for stability.

Using immediate receive, we explain in the following why one can expect better overlapping of communication and computation. In this context, a message is said to be large if it is significantly larger than the internal `MPI` system buffers. For large messages, we see two reasons for obtaining an improvement. (For short messages we do not expect much improvement.) First, with no immediate receive, if a large message is to be received then the message might actually finish being transferred/sent only when the receiver actually performs the reception. Second, using immediate receive, the space in the send buffer becomes free earlier. Less idle time in the sending process, as in situation 3, might be expected if the send buffer is not saturated.

Matrix	Ordering	`mpi_irecv`	Number of processors				
			4	8	16	32	64
BMWCRA_1	`MeTiS`	OFF	—	—	24.7	20.4	11.4
		ON	—	—	22.7	16.6	9.6
BMW3_2	`MeTiS`	OFF	—	24.6	16.4	9.2	6.2
		ON	—	22.6	15.9	8.2	5.7
CRANKSG2	`MeTiS`	OFF	—	37.6	22.1	13.3	8.9
		ON	—	26.4	18.1	11.3	8.0
SHIP_003	`MeTiS`	OFF	—	—	37.3	24.5	17.9
		ON	—	—	30.8	21.1	15.7
BBMAT	`AMD`	OFF	46.0	25.7	19.9	17.2	12.9
		ON	45.2	24.7	18.0	15.2	12.5
ECL32	`AMD`	OFF	56.7	38.4	26.5	19.9	15.3
		ON	54.0	35.4	23.4	18.4	15.7
INVEXTR1	`AMD`	OFF	37.7	26.9	19.4	21.6	20.0
		ON	36.8	25.6	19.5	21.3	18.9
MIXTANK	`AMD`	OFF	57.3	36.7	25.4	23.2	17.1
		ON	52.9	33.5	24.1	19.7	16.9

Table 9: Influence of the use of `mpi_irecv` on the time (in seconds) for factorization of `MUMPS` Version 4.1. — means not enough memory

On our largest test matrices we show, in Table 9, the impact of using immediate receive during the factorization phase. Version 4.1 of `MUMPS` with the same tuning of machine dependent parameters has been used to get all the results (with and without immediate receive) reported in Table 9. The default size of our send buffer is twice the size of the largest message. Note that the performance of Version 4.1 reported in Section 6 is slightly better. In fact, the results shown in this section were obtained with the most recent release of the CRAY operating system (1.3.0.4) for which the default size of the `MPI` buffers is 4 Kbytes. All results provided in Section 5 were obtained using an older release (Release 1.3.0.3) for which the size of the `MPI` buffer was unlimited. Although `MUMPS` with `mpi_irecv` is not very sensitive to the size of the `MPI` buffers, this is a good reason for the small performance difference.

One can see that relatively larger gains are obtained on a smaller number of processors. Symmetric matrices seems to benefit more from this modification. Node parallelism involves a relatively larger number of messages on symmetric matrices than on unsymmetric matrices (see results on 64 processors in Table 26 of the Appendix) that might saturate more the send buffer and the internal `MPI` buffers.

Matrix	Ordering	Number of processors				
		4	8	16	32	64
Maximum message size						
BMWCRA_1	MeTiS	—	—	7.3	2.7	2.2
BMW3_2	MeTiS	—	10.0	2.2	1.2	1.3
CRANKSG2	MeTiS	—	7.0	4.0	2.1	1.6
SHIP_003	MeTiS	—	—	5.6	2.6	2.1
BBMAT	AMD	4.8	3.0	2.6	2.5	2.4
ECL32	AMD	12.9	6.1	3.4	3.4	3.2
INVEXTR1	AMD	7.2	5.1	3.4	2.8	2.5
MIXTANK	AMD	16.0	7.3	4.3	3.9	3.9
Average Communication volume						
BMWCRA_1	MeTiS	—	—	34	33	18
BMW3_2	MeTiS	—	25	25	17	9
CRANKSG2	MeTiS	—	31	30	23	18
SHIP_003	MeTiS	—	—	65	60	34
BBMAT	AMD	35	35	33	32	19
ECL32	AMD	64	63	56	45	26
INVEXTR1	AMD	60	43	33	28	15
MIXTANK	AMD	77	115	109	93	51

Table 10: Maximum message size (in Mbytes) and average volume of communication per processor (in Mbytes) during factorization. – means not enough memory.

In Table 10, we show the maximum size of the messages and the average volume of communication. One can see that, because of node level parallelism, the maximum size of the messages generally reduces when increasing the number of processors [3]. It explains why, for a fixed problem, larger relative gains are obtained in Table 9 on a smaller number of processors. We also see that the total volume of messages can also be a good indicator of the gain that can be expected from the use of immediate receives.

To further analyse the gain due to the use of immediate receives, we show in Figures 9 and 10 the execution traces for the factorization of matrix CRANKSG2 (using 8 processors of the CRAY T3E). Messages have been suppressed to see better the proportion of execution time used by `MPI` communications. One can see that `MPI` takes significantly more time when immediate receive is off than when it is on. The summary chart of the same traces in Figure 11 shows that using immediate receives reduces the time spent in `MPI` calls by almost a factor of three.

To conclude this study, we show (compare the results in Tables 7 and 11), as one might expect from the previous discussion, that the new code based on immediate receives (`mpi_irecv`) is very much less sensitive to the size of the internal `MPI` buffer than the initial version based on standard receives (`mpi_recv`).

Figure 9: Immediate receive OFF; Trace of the factorization phase of matrix CRANKSG2 using 8 processors of the CRAY T3E. Black areas correspond to the time spent in `MPI`.

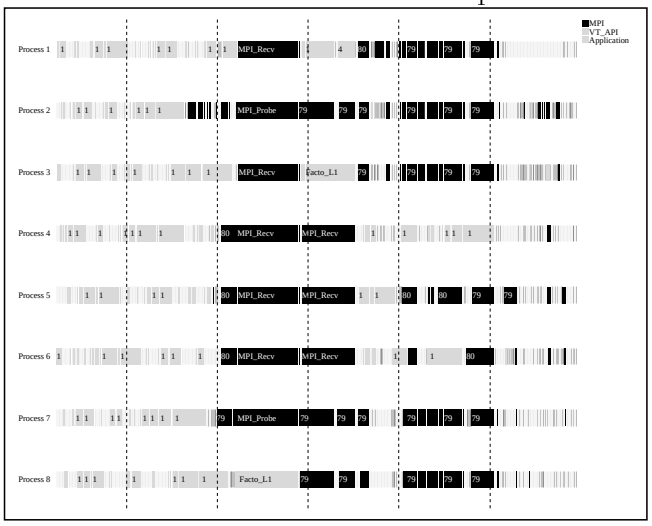


Figure 10: Immediate receive ON; Trace of the factorization phase of matrix CRANKSG2 using 8 processors of the CRAY T3E. Black areas correspond to the time spent in `MPI`.

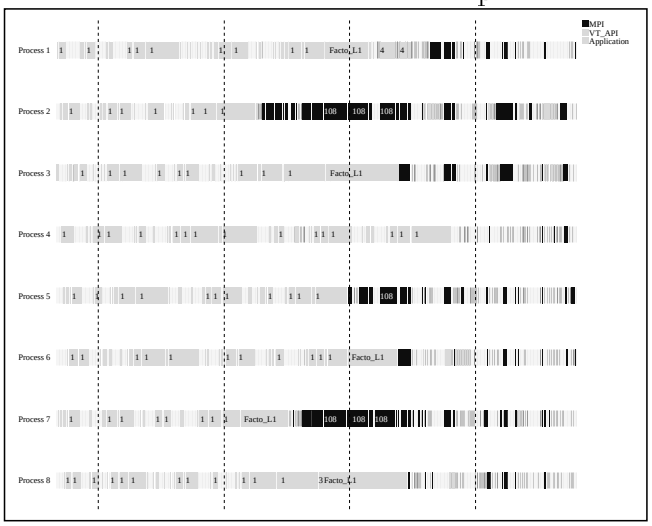
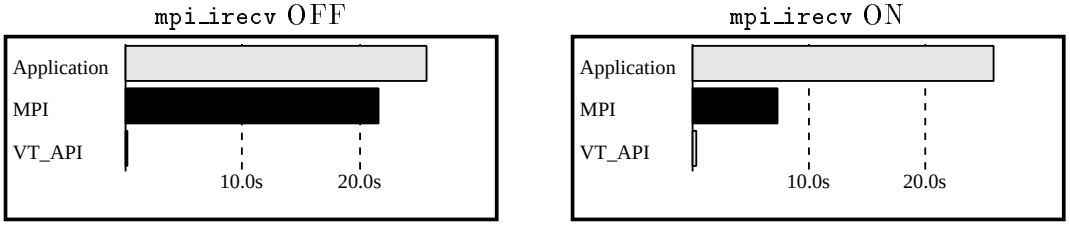


Figure 11: Summary chart on the use of immediate receive during the `MUMPS` factorization phase (Matrix CRANKSG2, ND ordering, 8 processors of the CRAY T3E).



Size (in Bytes) of the `MPI` buffer								
0	128	512	1K	4K (*)	64K	512K	2Mega	8Mega
27.1	27.3	26.5	26.6	26.4	26.2	26.2	26.4	26.2

Table 11: Influence of the `MPI` buffer size on the time (in seconds) for the factorization of matrix CRANKSG2 on 8 processors of the CRAY T3E. `mpi_irecv` is used to match `mpi_isend`. (*) default value on the CRAY T3E.

6 Performance analysis on general matrices

6.1 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 all the results provided in this section we have used Release 1.3.0.3 of the Cray operating system (with `MPI` internal buffers of unlimited size). On the most recent release of the operating system (Release 1.3.0.4) the default size for `MPI` internal buffers is 4 Kbytes. This could result in a performance degradation for both solvers although relatively more for `SuperLU` than for `MUMPS` (see results and discussion in Section 5).

The same orderings will always be used so that we can guarantee that the same input problem is given to the numerical phases. We recall that this does not mean that both solvers will 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 that we discuss later. Both nested dissection and minimum degree orderings will be considered since we want to show that the two solvers have very different sensitivity to the choice of the ordering.

A complete set of results is available in the Appendix. We report, in this section, on results for the largest problems from our set. Both symmetric and unsymmetric matrices will be considered although we mainly focus on unsymmetric ones because only `MUMPS` has an option to exploit symmetry. 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 1).

6.1.1 Study of the factorization phase

We show in Table 12 the time spent during the factorization phase of both solvers. On a relatively small number of processors (less than 32) `MUMPS` is generally faster than `SuperLU` for two reasons.

First, `MUMPS` handles symmetric and more regular data structures better than `SuperLU`, because `MUMPS` uses Level 3 BLAS kernels on relatively bigger blocks than those arising in the Level 3 BLAS kernels of `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 23 and from the results on 3D grid problems in Table 29 of the Appendix. Note that, even on 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 relatively small number of processors, we also notice that `SuperLU` does not always fully benefit from the reduction in the number of operations (see Table 20) due to the use of a nested dissection ordering (see the results with `SuperLU` on matrix BBMAT using 4 processors).

Furthermore, one should notice that, on matrices that are structurally very unsymmetric, `SuperLU` can be much less scalable than `MUMPS`. For example, on matrix LHR71C (see the results in Table 23 of the Appendix), speedups of 1.8 and 8.3 are obtained with `SuperLU` and `MUMPS`, respectively. This is due to two parallel limitations of the current `SuperLU` algorithm described in Section 5.1.3. 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 in MUMPS, from the better balanced tree generated by a nested dissection ordering.

On a larger number of processors (greater than or equal to 32), we see that the ordering very significantly influences the performance of the codes and, in particular, the performance of MUMPS which generally outperforms SuperLU using a nested dissection ordering even on a large number of processors. On the other hand, if we use the minimum degree ordering, SuperLU is generally faster than MUMPS on a large number of processors. We also see that, on most of our unsymmetric problems, neither of the solvers 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, independent 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

Matrix	Order.	Solver	Number of processors							
			1	4	16	32	64	128	256	512
Unsymmetric matrices										
BBMAT	AMD	MUMPS	—	45.7	16.5	13.7	11.9	11.2	9.1	12.6
		SuperLU	—	66.1	22.8	14.6	11.2	8.9	9.9	9.1
	ND	MUMPS	—	39.4	13.2	11.9	9.9	9.2	9.4	11.6
		SuperLU	—	137.8	41.2	25.2	17.3	12.4	14.3	14.7
ECL32	AMD	MUMPS	—	54.6	23.8	17.6	15.6	15.1	16.0	16.5
		SuperLU	—	107.4	35.8	20.6	14.9	11.1	10.9	8.9
	ND	MUMPS	—	24.7	9.7	7.7	6.9	7.0	7.0	8.9
		SuperLU	—	49.0	16.7	12.0	9.9	8.8	9.9	9.5
INVEXTR1	AMD	MUMPS	—	36.0	21.7	19.2	19.1	18.8	16.6	18.6
		SuperLU	—	56.3	17.2	11.0	8.3	6.0	7.1	6.6
	ND	MUMPS	31.8	13.2	4.5	3.9	3.8	4.4	5.4	6.3
		SuperLU	68.2	23.1	9.1	6.7	5.7	4.7	6.1	5.8
MIXTANK	AMD	MUMPS	—	53.3	24.3	20.4	17.0	15.9	16.4	18.2
		SuperLU	—	80.9	23.7	14.4	9.8	6.7	7.0	6.5
	ND	MUMPS	40.8	13.0	5.6	4.4	3.9	4.2	4.2	5.4
		SuperLU	88.1	28.8	10.1	7.0	5.3	4.5	5.6	5.5
TWOTONE	MC64	MUMPS	—	40.3	18.6	14.7	14.4	14.3	14.0	14.3
	+AMD	SuperLU	—	106.2	32.7	25.7	21.0	16.2	21.2	18.5
Symmetric matrices										
SHIP_003	ND	MUMPS	—	—	29.3	17.3	14.1	12.6	11.8	14.8
		SuperLU$^{(*)}$	—	—	—	45.1	31.3	24.2	23.0	19.8
BMWCRA_1	ND	MUMPS	—	—	21.9	12.8	8.8	7.1	6.5	7.9

Table 12: Factorization time of large test matrices on the CRAY T3E. "—" indicates not enough memory. $^{(*)}$SuperLU computes an **LU** factorization of matrix SHIP_003 and does not exploit the symmetry in the matrix.

analyse the scalability of our solvers, we consider three dimensional regular grid problems in Section 7. On our relatively larger symmetric problems (SHIP_003 and BMWCRA_1), we see that performance gains can be obtained on up to 256 processors with MUMPS but on up to 512 processors with SuperLU. This again gives an indication of the better scalability of SuperLU compared to MUMPS.

To better understand the performance differences observed in Table 12 and to identify the main characteristics of our solvers, we show, in Figures 12, 13, and 14, the average communication volume on 4 and 64 processors respectively. The speed of communication can depend very much on the size of the messages and we thus indicate, in Figure 15, the average message size on 64 processors. A complete set of results is provided in the Appendix (see Table 26). 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.

Figure 12 shows that, on 4 processors and with nested dissection, SuperLU involves significantly more communication per processor than MUMPS. On 4 processors and with the minimum degree ordering, the volume of communication of the two solvers is globally "comparable". On 16 processors (Figure 13), the volume of communication of our two solvers using nested dissection is of the same order while, when using a minimum degree ordering, MUMPS involves more communication than SuperLU. On 64 processors (Figure 14) and on matrices for which MUMPS continues subdividing nodes to provide more parallelism than on 16 processors (see for example MIXTANK with AMD, ECL32 with AMD, or BBMAT with nested dissection), the communication volume of MUMPS remains or becomes greater than that of SuperLU.

Figure 15 shows that, 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 size of the messages. This is due to the much larger number of messages involved in a fan-out approach (SuperLU) with respect to a multifrontal approach (MUMPS). Note that, with MUMPS, the number of messages does include the messages (one integer) required by the dynamic scheduling algorithm to update the load of 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 (see Figure 16) it is not always the case with MUMPS (see Figure 17). This should contribute to a better efficiency of SuperLU with respect to MUMPS on a large number of processors.

Figure 12: Average communication volume (4 processors).

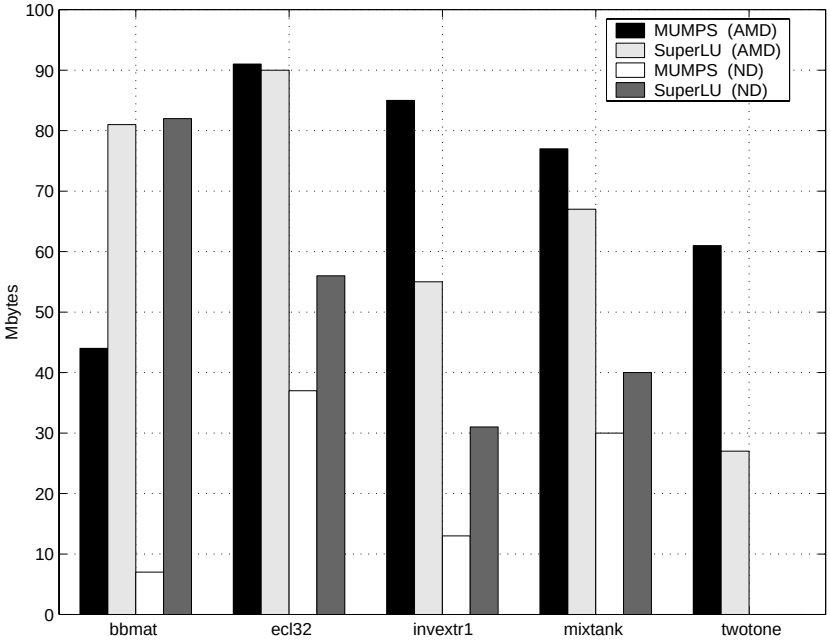


Figure 13: Average communication volume (16 processors).

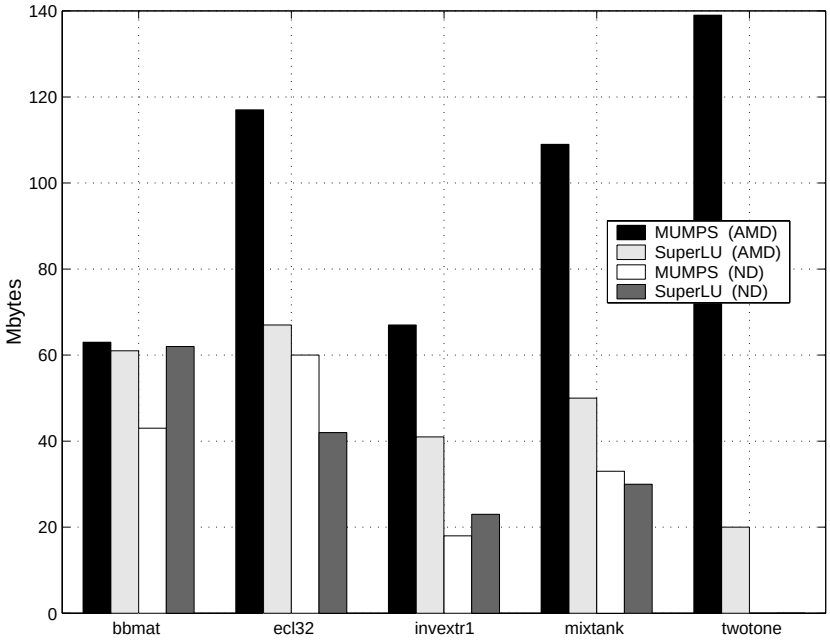


Figure 14: Average communication volume (64 processors).

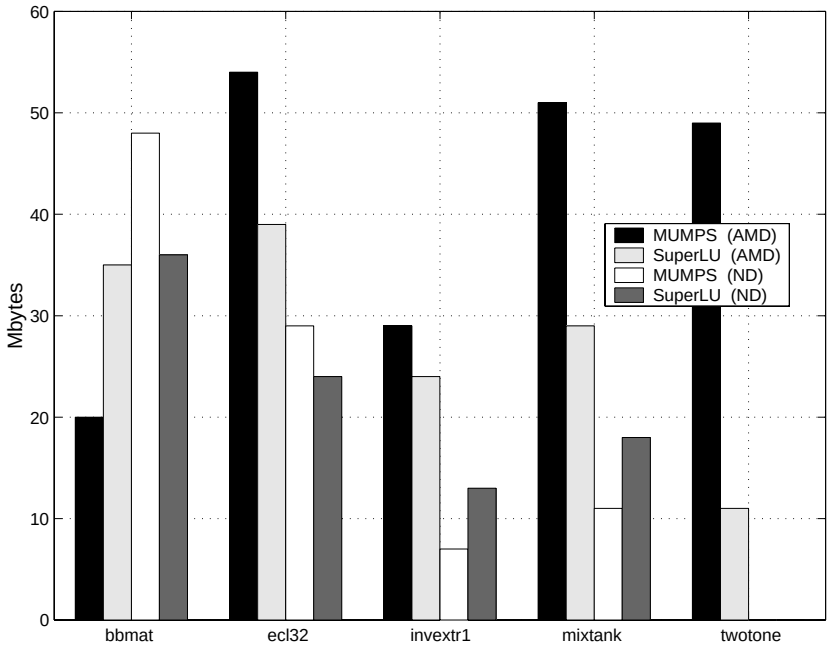


Figure 15: Average message size (64 processors).

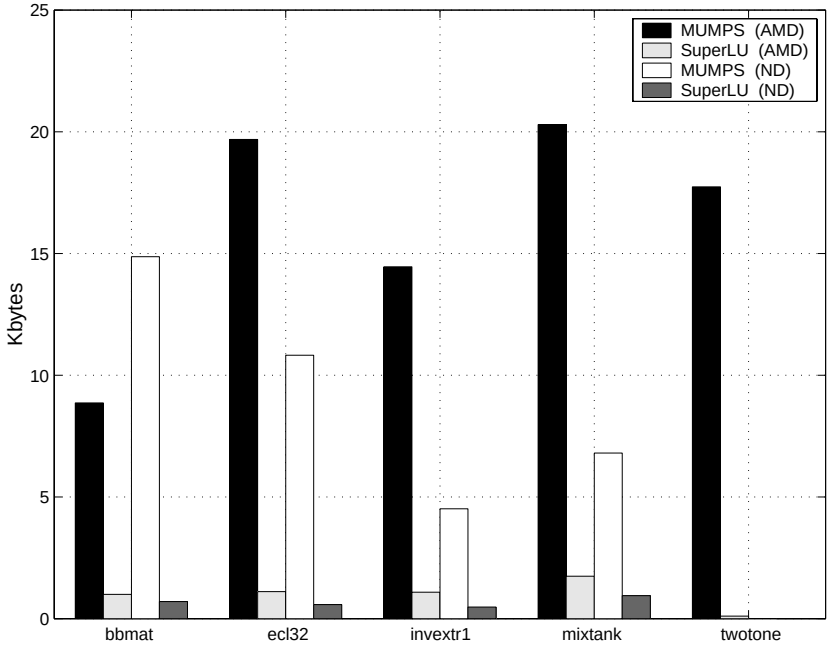


Figure 16: Average volume of communication (`SuperLU` and nested dissection).

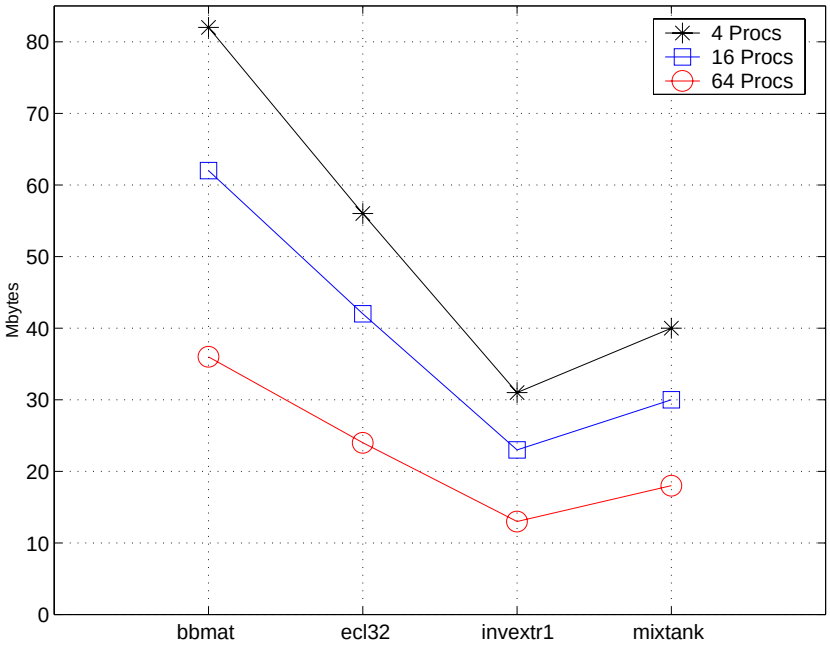
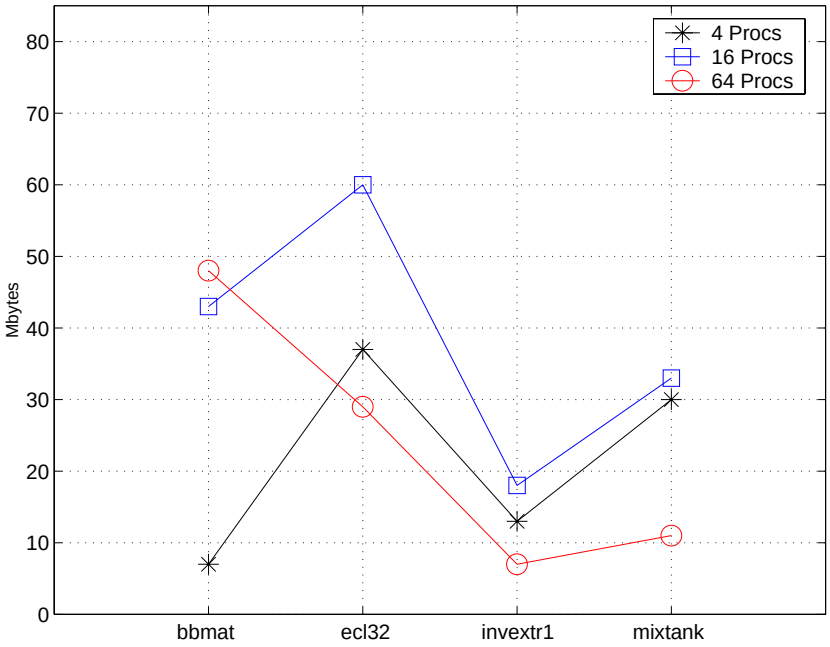


Figure 17: Average volume of communication (`MUMPS` and nested dissection).



6.1.2 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. A complete set of results is provided in the Appendix (Tables 24 and 25). We apply iterative refinement improve the solution only when the backward error ($Berr$) is larger than $\sqrt{\varepsilon}$. The iterative refinement will then stop when the backward error has reached the required accuracy. The iterative refinement procedure involves not only forward and backward solves but also products with the original matrix to compute the backward error used in the stopping criterion. With MUMPS, the user can provide the input matrix in a very general distributed format [3]. 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 onto all the processors.

We first observe (compare the results in Tables 12 and 13) 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.

The performance reported in Table 13 shows that the regularity in the structure of the matrix factors generated by the factorization phase of MUMPS in general leads to a faster solve phase than that of SuperLU for up to 256 processors. On 512 processors, the solve phase of SuperLU is usually 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, on numerically difficult matrices (see BBMAT, INVEXTR1 and MIXTANK) iterative refinement will be required to obtain an accurate solution. 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 might 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	16	32	64	128	256	512
Unsymmetric matrices										
BBMAT	AMD	MUMPS	—	0.53	0.31	0.32	0.32	0.36	0.40	0.56
		SuperLU	—	1.77	1.05	1.00	0.80	0.70	0.70	0.66
		— + (IR)	—	3.38	1.60	1.27	1.05	0.90	0.89	0.79
	ND	MUMPS	—	0.38	0.26	0.29	0.31	0.35	0.37	0.54
		SuperLU	—	2.12	1.28	1.12	0.99	0.82	0.85	0.68
		— + (IR)	—	4.91	2.41	1.47	1.32	1.04	1.04	0.87
ECL32	AMD	MUMPS	—	0.80	0.40	0.41	0.40	0.45	0.52	0.83
		SuperLU	—	2.09	1.54	1.46	1.10	0.98	0.73	0.57
	ND	MUMPS	—	0.53	0.30	0.28	0.28	0.43	0.39	0.48
		SuperLU	—	1.76	1.38	1.41	1.05	0.93	0.68	0.53
INVEXTR1	AMD	MUMPS	—	0.57	0.46	0.41	0.40	0.58	0.54	0.60
		— + (IR)	—	1.42	1.02	0.96	0.94	1.03	1.16	1.38
		SuperLU	—	0.91	0.56	0.54	0.41	0.38	0.34	0.32
		— + (IR)	—	1.57	0.86	0.76	0.61	0.55	0.51	0.48
	ND	MUMPS	0.59	0.31	0.18	0.18	0.18	0.25	0.26	0.37
		— + (IR)	2.10	0.48	0.50	0.48	0.47	0.51	0.62	0.92
		SuperLU	1.45	0.77	0.55	0.51	0.46	0.36	0.34	0.28
		— + (IR)	2.69	1.58	0.90	0.74	0.67	0.54	0.52	0.44
MIXTANK	AMD	MUMPS	—	0.56	0.39	0.41	0.40	0.41	0.49	0.59
		SuperLU	—	1.10	0.69	0.64	0.51	0.41	0.36	0.31
	ND	MUMPS	0.67	0.27	0.16	0.16	0.15	0.19	0.24	0.35
		SuperLU	1.47	0.90	0.65	0.58	0.49	0.33	0.30	0.24
TWOTONE	MC64	MUMPS	—	1.03	0.97	0.98	0.98	1.03	1.13	1.41
	+AMD	SuperLU	—	3.26	2.52	2.24	1.84	1.56	1.38	1.21
		— + (IR)	—	25.84	12.63	4.18	3.64	2.27	1.84	1.55
Symmetric matrices										
SHIP_003	ND	MUMPS	—	—	0.87	0.69	0.71	0.66	0.74	0.88
		SuperLU	—	—	—	1.45	1.18	1.04	0.89	0.70
BMWCRA_1	ND	MUMPS	—	—	0.80	0.55	0.43	0.40	0.47	0.61

Table 13: Solve time for large matrices. " — + (IR) " : time spent to improve the initial solution using iterative refinement. Note that, on the symmetric matrix SHIP_003, SuperLU uses the LU factors to compute the solution.

6.2 Memory usage

We study, in this section, the memory used during factorization as a function of both the algorithm and the number of processors. The best ordering on a subset of the large test problems is used to illustrate our discussion. A complete set of results is reported in the Appendix (see Table 27).

We want first to point out that, because of the dynamic scheduling approach and the partial pivoting with threshold 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 [3], that the input matrix can also be provided in a general distributed format and can be handled very efficiently by the solver.

We notice, in Figures 18 and 19, the significant reduction in the required memory when increasing the number of processors. We also see that, in general, `SuperLU` usually requires less memory than `MUMPS` although this is less so when many processors are used showing a better memory scalability of `MUMPS`. One can observe (see also Table 27) that there is little difference between the average and maximum memory usage showing both algorithms are well balanced, with `SuperLU` marginally the better of the two.

Figure 18: Average memory used per processor.

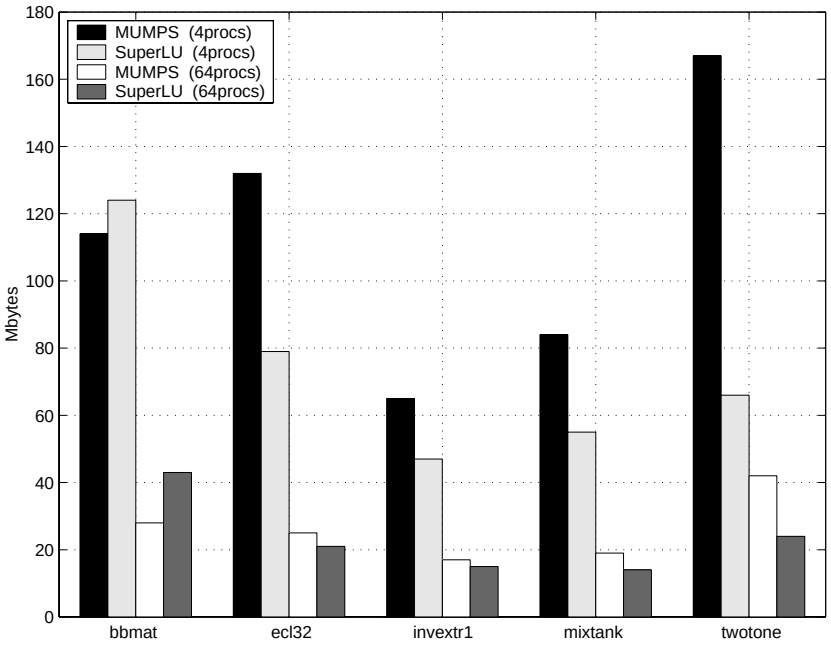
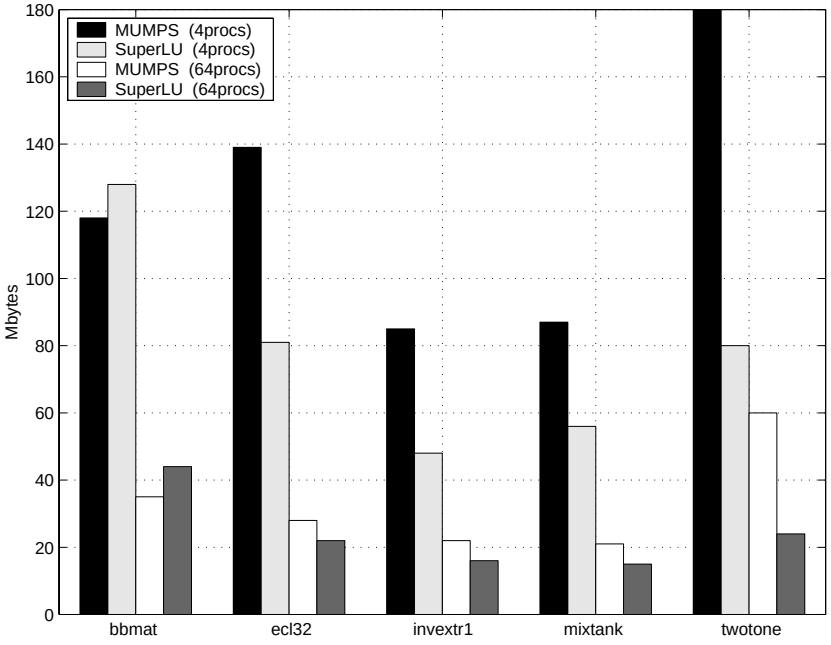


Figure 19: Maximum memory used per processor.



7 Performance analysis on 3-D grid problems

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

We first consider a set of 3D cubic grids (NX=NY=NZ) on which we only consider a nested dissection ordering. Since we are also interested in the behaviour of our solvers using minimum degree, we also consider a set of rectangular grids (NX, NX/4, NX/8) for which both orderings (nested dissection and `AMD`) are applied. The set of grids is defined such that the number of operations (using nested dissection) remains almost constant. The size of the grids used, the number of operations and the complete timings are reported in Table 28 of the Appendix. While 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. This was not always possible (see results in Appendix), so that the number of operations per processor is not completely constant.

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 $\mathbf{LU}$ factorization, referred to as `MUMPS-UNS`. `SuperLU` will compute an $\mathbf{LU}$ 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 20, 22 and 24, the Megaflop rate per processor for our three approaches on cubic grids using nested dissection, rectangular grids using nested dissection, and rectangular grids using `AMD`, 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 results in Appendix).

In Figures 21, 23 and 25 we report the parallel efficiency on cubic grids using nested dissection, rectangular grids using nested dissection, and rectangular grids using `AMD`, respectively. The efficiency of a solver on p processors is defined as the ratio of its Megaflop rate per processor on p processors over its Megaflop rate on 1 processor.

Before discussing the efficiency which is strongly related to the Megaflop rate on one processor, we first briefly discuss the Megaflop rate per processor which corresponds to the absolute performance of the approach used on a given problem. We first notice that on up to 8 processors, and independently of the grid shape and the ordering, `MUMPS-UNS` is about twice as fast as `SuperLU` and also has much higher Megaflop rate than `MUMPS-SYM`. On 128 processors and using nested dissection on both rectangular and cubic grids, all solvers have almost an identical Megaflop rate per processor. However, on 128 processors and using minimum degree `SuperLU` is three times as fast as `MUMPS-UNS`, and slightly faster than `MUMPS-SYM`.

In terms of efficiency, `SuperLU` is generally more efficient that `MUMPS-UNS` even on a relatively small number of processors. `MUMPS-SYM` is relatively more efficient than `MUMPS-UNS` and using nested dissection 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 is reached on rectangular grids ordered with minimum degree (128 processors) for which `SuperLU` is about three times more efficient than both `MUMPS-UNS` and `MUMPS-SYM`. On cubic grids using nested dissection (128 processors), `SuperLU` is also about three times more efficient than `MUMPS`.

Figure 20: Megaflop rate per processor (cubic grids, nested dissection).

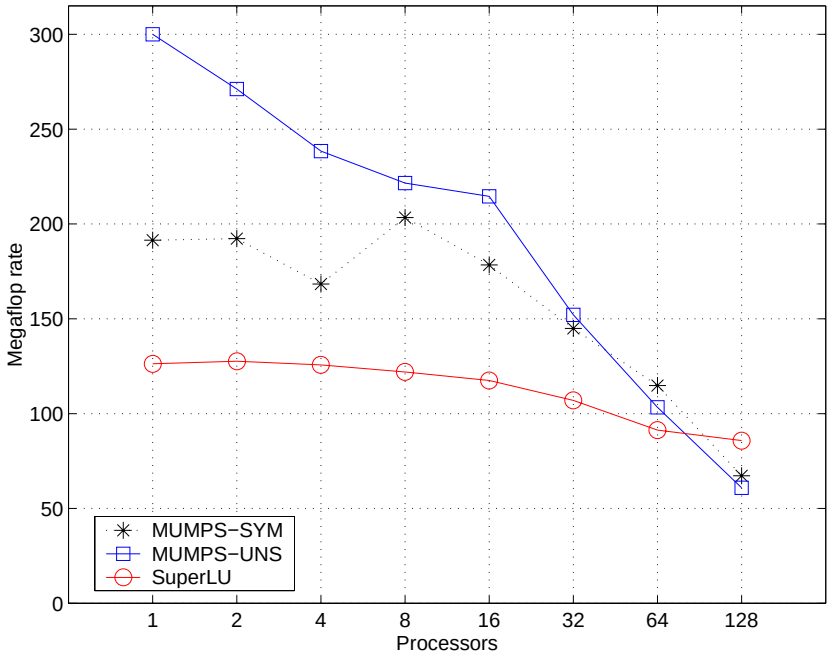


Figure 21: Parallel efficiency (cubic grids, nested dissection).

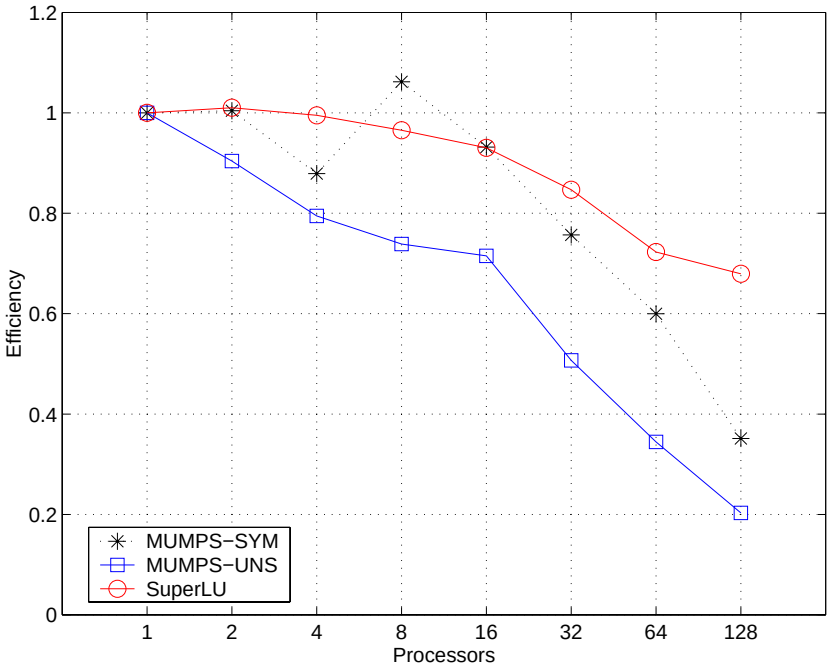


Figure 22: Megaflop rate per processor (rectangular grids, nested dissection).

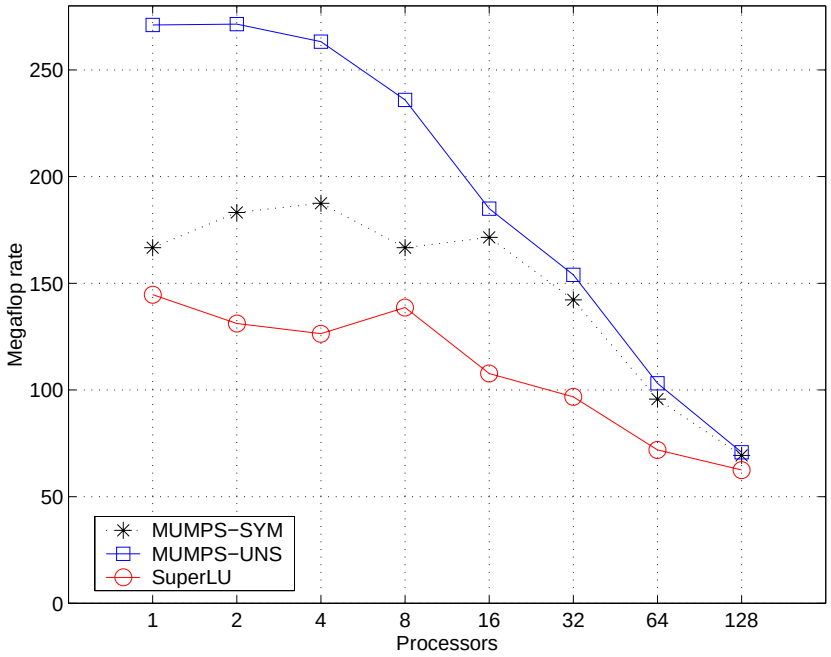


Figure 23: Parallel efficiency (rectangular grids, nested dissection).

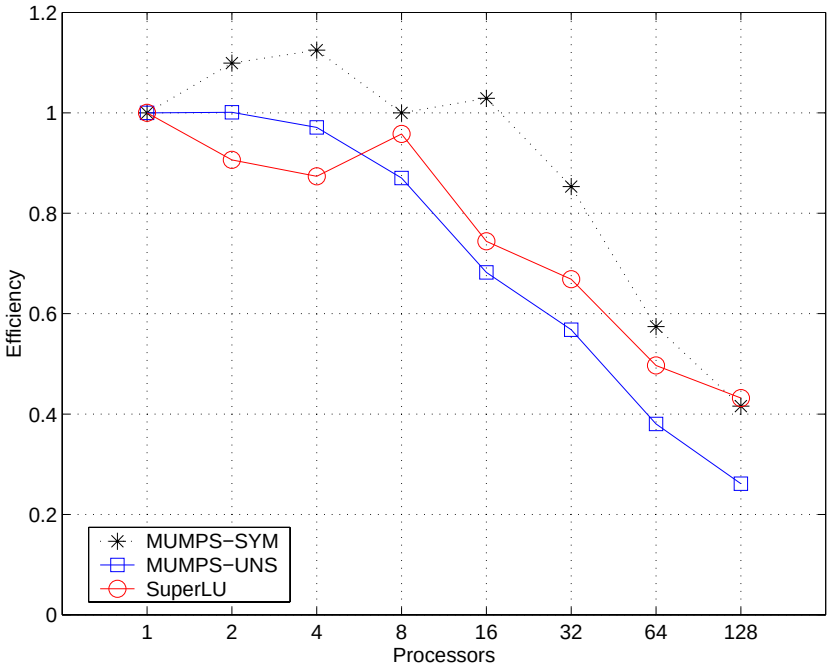


Figure 24: Megaflop rate per processor (rectangular grids, AMD).

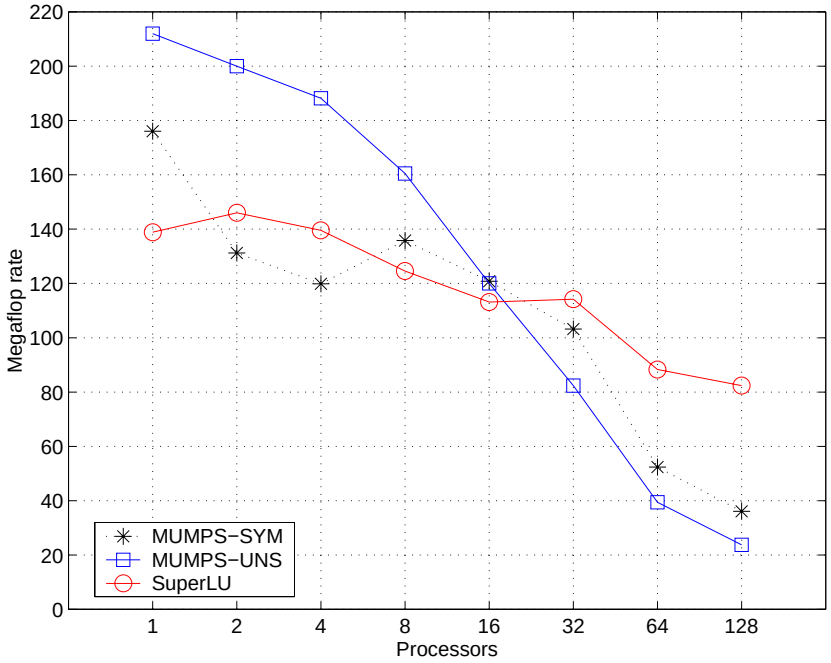
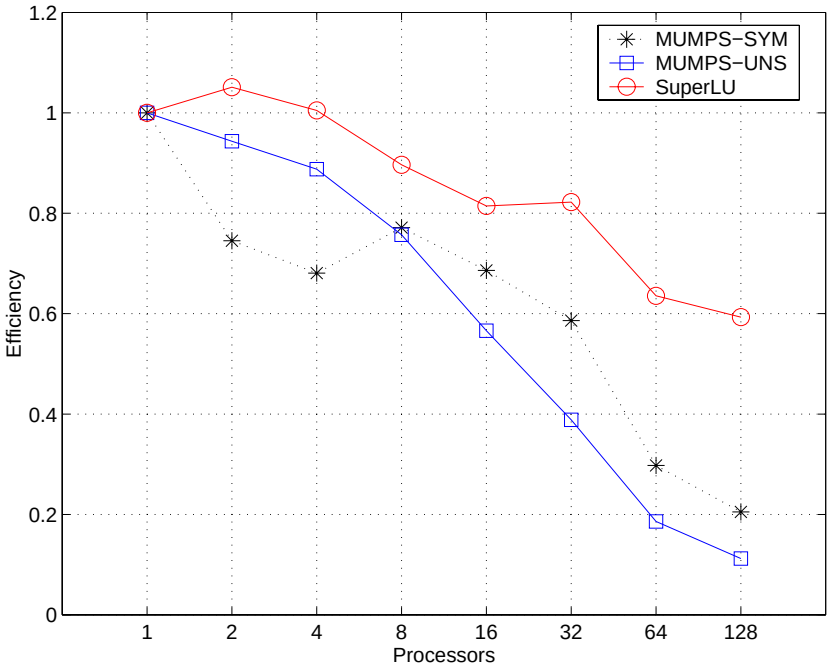


Figure 25: Parallel efficiency (rectangular grids, AMD).



Finally, we report in Table 14, a quantitative evaluation of the overhead due to parallelism. 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.

Table 14 shows that the overhead increases faster with `MUMPS` than with `SuperLU`. The better parallel behaviour of `MUMPS-SYM` with respect to `MUMPS-UNS` can also be observed in this 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 14: Percentage of the factorization time spent in computation and in overhead due to parallelism.

8 Other codes

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 this section, 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 [19].

Code	Technique	Scope	Availability	Ref
CAPSS	Multifrontal	SPD	`www.netlib.org/scalapack`	[20]
MUMPS	Multifrontal	SYM/UNS	`www.enseeiht.fr/apo/MUMPS`	[3]
PaStiX	Fan-in	SPD	see caption(*)	[21]
PSPASES	Multifrontal	SPD	`www.cs.umn.edu/~mjoshi/pspases`	[18]
SPOOLES	Fan-in	SYM/UNS	`www.netlib.org/linalg/spooles`	[8]
SuperLU	Fan-out	UNS	`www.nersc.gov/~xiaoye/SuperLU`	[23]
S+	Fan-out†	UNS	`www.cs.ucsb.edu/research/S+`	[15]

Table 15: Distributed memory codes.
† Uses QR storage to statically accommodate any LU fill-in.
(*) `www.dept-info.labri.u-bordeaux.fr/~ramet/pastix`

Code	Technique	Scope	Availability	Ref
GSPAR	Interpretative	UNS	Grund	[9]
MA41	Multifrontal	UNS	`www.cse.clrc.ac.uk/Activity/HSL`	[2]
MA49	Multifrontal QR	RECT	`www.cse.clrc.ac.uk/Activity/HSL`	[5]
PARDISO	Left-right looking	UNS	Schenk	[29]
PSLDLT	Left-looking	SPD	SGI product	[28]
PSLDU	Left-looking	UNS	SGI product	[28]
SuperLU	Left-looking	UNS	`www.nersc.gov/~xiaoye/SuperLU`	[10]
PanelLLT	Left-looking	SPD	Ng	[26]

Table 16: Shared memory codes

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 report and her help with the presentation.

9 Appendix

The complete set of results is provided in this section. The following notations are used in the tables.

- the componentwise backward error, $Berr = \max_i \frac{|r|_i}{(|A|\cdot|x|+|b|)_i}$ [7],
- the true relative error, $Err = \frac{||x_{true}-x||}{||x_{true}||}$, where x is the computed solution and x_{true} is the exact solution equal to a vector of all ones,
- the double precision machine precision, $\varepsilon \approx$ 2.2e-16 (64 bits).

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
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
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 17: 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.

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-9
	SuperLU	8.7e-08	4.3e-02	3	4.6e-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-8	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	3.1e-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	1.0e+00	1	1.0e+00	1.0e+00

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	4.6e-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	2.0e-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	1.0e+00	2	8.9e-07	1.0e+00
LNSP3937	MUMPS	1.5e-12	3.6e-11	2	2.0e-16	6.4e-12
	SuperLU	3.5e-12	2.7e-11	2	2.2e-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 18: Comparison of the numerical behaviour, backward error (Berr) and forward error (Err), of the solvers. Nb corresponds to the number of steps of iterative refinement. Stopping criterion $Berr < \varepsilon$.

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
LHR71C	MUMPS	AMD	47.5	—	39.4
	—	MC64+AMD	34.0	31.0	2.0
	SuperLU	AMD	121.3	—	35.0
	—	MC64+AMD	32.0	26.9	1.8
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	2.59	0.2	0.7

Table 19: Influence of permuting large entries onto the diagonal (using MC64) on the time (in seconds) for the analysis phase of MUMPS and SuperLU.

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
NASASRB.RUA				
	AMD	MUMPS	24.2	9.5
		SuperLU	23.9	9.5
	ND	MUMPS	21.2	6.9
		SuperLU	21.0	6.8

Table 20: Influence of the symmetric reordering (minimum degree or nested dissection) on the cost of the factorization phase for unsymmetric matrices. (`MC64` is not used.)

Matrix	Solver	NZ in **LU** $\times 10^6$	Flops $\times 10^9$
SHIP_003	MUMPS	57.1	72.1
	SuperLU$^{(*)}$	112.6	145.6
BMWCRA_1	MUMPS	70.3	61.0
BMW3_2	MUMPS	46.2	26.9
CRANKSG2	MUMPS	42.2	42.5
HOOD	MUMPS	27.6	8.2
NASASRB.RSA	MUMPS	10.6	3.4

Table 21: Cost of the $\mathbf{LDL}^T$ factorization phase of `MUMPS` on symmetric matrices using a nested dissection ordering. $^{(*)}$ `SuperLU` performs an unsymmetric **LU** factorization.

Matrix	Order.	Solver	Number of processors								
			1	4	8	16	32	64	128	256	512
Unsymmetric matrices											
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	AMD	MUMPS	—	36.0	26.4	21.7	19.2	19.1	18.8	16.6	18.6
		SuperLU	—	56.3	28.7	17.2	11.0	8.3	6.0	7.1	6.6
	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	AMD	MUMPS	—	53.3	33.5	24.3	20.4	17.0	15.9	16.4	18.2
		SuperLU	—	80.9	38.3	23.7	14.4	9.8	6.7	7.0	6.5
	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
Symmetric matrices											
SHIP_003	ND	MUMPS	—	—	—	29.3	17.3	14.1	12.6	11.8	14.8
		SuperLU	—	—	—	—	45.1	31.3	24.2	23.0	19.8
BMWCRA_1	ND	MUMPS	—	—	—	21.9	12.8	8.8	7.1	6.5	7.9
BMW3_2	ND	MUMPS	—	—	20.5	11.5	7.6	5.2	5.1	5.1	6.6
CRANKSG2	ND	MUMPS	—	—	26.5	15.2	10.4	7.6	7.1	7.6	8.1

Table 22: Factorization phase time study of large test matrices on the CRAY T3E. "—" indicates not enough memory.

Matrix	Ordering	Solver	Number of processors					
			1	4	8	16	32	64
Unsymmetric matrices								
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
NASASRB.RUA	AMD	MUMPS	—	11.8	8.5	7.5	7.0	7.2
		SuperLU	—	18.3	11.2	9.7	7.6	7.7
	ND	MUMPS	28.0	8.2	5.2	3.3	2.8	2.6
		SuperLU		15.1	9.9	7.3	6.0	5.6
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
Symmetric matrices								
HOOD	ND	MUMPS	—	15.0	8.2	4.4	3.4	2.4
NASASRB.RSA	ND	MUMPS	22.1	6.2	3.8	3.3	2.8	2.4

Table 23: Factorization phase time study on the CRAY T3E of small test matrices . "—" indicates not enough memory.

Matrix	Order.	Solver	Number of processors								
			1	4	8	16	32	64	128	256	512
Unsymmetric matrices											
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	AMD	MUMPS	—	0.57	0.41	0.46	0.41	0.40	0.58	0.54	0.60
		— + (IR)	—	1.42	1.03	1.02	0.96	0.94	1.03	1.16	1.38
		SuperLU	—	0.91	0.84	0.56	0.54	0.41	0.38	0.34	0.32
		— + (IR)	—	1.57	1.22	0.86	0.76	0.61	0.55	0.51	0.48
	ND	MUMPS	0.59	0.31	0.20	0.18	0.18	0.18	0.25	0.26	0.37
		— + (IR)	2.10	0.48	0.31	0.50	0.48	0.47	0.51	0.62	0.90
		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	AMD	MUMPS	—	0.56	0.38	0.39	0.41	0.40	0.41	0.49	0.59
		SuperLU	—	1.10	0.96	0.69	0.64	0.51	0.41	0.36	0.31
	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
Symmetric matrices											
SHIP_003	ND	MUMPS	—	—	—	0.87	0.69	0.71	0.66	0.74	0.88
		SuperLU	—	—	—	—	1.45	1.18	1.04	0.89	0.70
BMWCRA_1	ND	MUMPS	—	—	—	0.80	0.55	0.43	0.40	0.47	0.61
BMW3_2	ND	MUMPS	—	—	1.17	0.88	0.71	0.60	0.56	0.64	0.80
CRANKSG2	ND	MUMPS	—	—	0.71	0.46	0.38	0.33	0.35	0.41	0.54

Table 24: Solve phase time study for large matrices. " — + (IR) " shows the time spent improving the initial solution using iterative refinement. Note that, on the symmetric matrix SHIP_003, SuperLU uses the LU factors to compute the solution. Stopping criterion for iterative refinement is $Berr < \sqrt{\varepsilon}$.

Matrix	Ord.	Solver	Number of processors					
			1	4	8	16	32	64
Unsymmetric matrices								
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
NASASRB.RUA	AMD	MUMPS	—	0.41	0.33	0.33	0.31	0.31
		SuperLU		0.95	0.70	0.69	0.61	0.55
	ND	MUMPS	0.82	0.33	0.23	0.18	0.17	0.16
		SuperLU		0.84	0.76	0.60	0.51	0.45
RMA1010	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
Symmetric matrices								
HOOD	ND	MUMPS	—	1.08	0.68	0.58	0.47	0.39
NASASRB.RSA	ND	MUMPS	0.76	0.33	0.28	0.21	0.16	0.15

Table 25: Solve phase time study on the CRAY T3E for small matrices.

Matrix	Ord	Solver	Number of processors								
			4			16			64		
			Max	Vol.	#Mess	Max	Vol.	#Mess	Max	Vol.	#Mess
Unsymmetric matrices											
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	AMD	MUMPS	7.6	85	4169	4.6	67	1967	2.0	29	2006
		SuperLU	0.24	55	15023	0.12	41	21527	0.07	24	21990
	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	91640	0.02	8	91640
MIXTANK	AMD	MUMPS	12.1	77	6071	4.0	109	2660	2.8	51	2513
		SuperLU	0.32	67	11044	0.17	50	16077	0.09	29	16598
	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
Symmetric matrices											
SHIP_003	ND	MUMPS	—	—	—	6.9	175	3586	2.7	94	5752
		SuperLU	—	—	—	—	—	—	0.15	100	33856
BMWCRA_1	ND	MUMPS	—	—	—	10.5	90	2108	3.1	54	3958
BMW3_2	ND	MUMPS	—	—	—	2.3	69	2399	1.6	23	2857
CRANKSG2	ND	MUMPS	—	—	—	5.6	89	2391	2.0	43	3553
HOOD	ND	MUMPS	1.6	4	167	2.0	11	782	2.2	4	1413
NASASRB.RSA	ND	MUMPS	2.5	5	146	1.2	7	609	1.9	2	842

Table 26: 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.

Matrix	Ordering	Solver	Number of processors					
			4		16		64	
			Avg.	Max.	Avg.	Max.	Avg.	Max.
Unsymmetric matrices								
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	AMD	MUMPS	136	171	49	58	27	39
		SuperLU	73	75	30	31	18	19
	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	AMD	MUMPS	206	204	58	61	34	37
		SuperLU	85	85	32	33	19	19
	ND	MUMPS	84	87	29	31	19	21
		SuperLU	55	56	23	23	14	15
NASASRB.RUA	AMD	MUMPS	82	88	33	36	22	26
		SuperLU	72	73	30	31	19	20
	ND	MUMPS	78	81	29	32	20	24
		SuperLU	64	66	28	29	18	19
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
Symmetric matrices								
SHIP_003	ND	MUMPS	—	—	124	105	55	70
		SuperLU	—	—	—	—	93	96
BMWCRA_1	ND	MUMPS	—	—	133	160	59	69
BMW3_2	ND	MUMPS	—	—	90	114	62	69
CRANKSG2	ND	MUMPS	—	—	81	95	57	72
HOOD	ND	MUMPS	145	153	70	81	57	66
NASASRB.RSA	ND	MUMPS	56	59	25	27	20	24

Table 27: Memory used during factorization.

Nprocs	Grid size			$\mathbf{LDL}^T$ factorization		**LU** factorization			
				MUMPSSYM		MUMPSUNS		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
Rectangular grids (minimum degree)									
1	96	24	12	2.5	14.2	3.9	18.4	5.0	36.0
2	110	28	23	3.7	14.1	7.4	18.5	7.3	25.0
4	120	30	15	7.0	14.6	14.0	18.6	13.9	24.9
8	136	34	17	13.8	12.7	27.6	21.5	27.3	27.4
16	152	38	19	28.8	14.9	57.6	30.0	57.2	31.6
32	168	42	21	101.7	30.8	203.2	77.1	202.8	55.5
64	184	46	23	134.8	40.2	269.3	106.7	269.0	47.6
128	208	52	26	328.9	71.2	665.8	218.9	655.7	62.2

Table 28: Factorization time on Cray T3E. **LU** factorization is performed for MUMPSUNS and SuperLU, $\mathbf{LDL}^T$ for MUMPSSYM.

Nprocs	**Cubic grids**			**Rectangular grids**					
	Nested dissection			Nested dissection			Minimum Degree		
	MUMPS		SuperLU	MUMPS		SuperLU	MUMPS		SuperLU
	SYM	UNS		SYM	UNS		SYM	UNS	
1	191	300	126	167	271	145	176	212	139
2	192	271	128	183	271	131	131	200	146
4	168	238	126	187	263	126	120	188	140
8	203	222	122	167	236	139	136	160	125
16	178	215	118	172	185	108	121	120	113
32	145	152	107	142	154	97	103	82	114
64	115	103	91	96	103	72	52	39	88
128	67	61	86	69	71	63	36	24	82

Table 29: Megaflop rate per processor during the factorization phase on Cray T3E.

References

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

[2] P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. of Supercomputer Applics.*, 7:64–82, 1993.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. Technical Report RT/APO/99/2, ENSEEIHT-IRIT, 1999. (submited to SIAM Journal on Matrix Analysis and Applications).

[4] P. R. Amestoy, I. S. Duff, and J. Y. L'Excellent. Parallélisation de la factorisation LU de matrices creuses non-symétriques pour des architectures à mémoire distribuée. *Calculateurs Parallèles Réseaux et Systèmes Répartis*, 10(5):509–520, 1998.

[5] P. R. Amestoy, I. S. Duff, and C. Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Int. Journal of Num. Linear Alg. and Appl.*, 3(4):275–300, 1996.

[6] P. R. Amestoy, I.S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, pages 501–520, 2000.

[7] M. Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM Journal on Matrix Analysis and Applications*, 10:165–190, 1989.

[8] C. Ashcraft and R. G. Grimes. SPOOLES: An object oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 22–24, 1999.

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

[10] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999.

[11] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[12] J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. MPI : A message passing interface standard. *Int. Journal of Supercomputer Applications*, 8:(3/4), 1995.

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

[14] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. Technical Report RAL-TR-1999-030, Rutherford Appleton Laboratory, 1999.

[15] Cong Fu, Xiangmin Jiao, and Tao Yang. Efficient sparse LU factorization with partial pivoting on distributed memory architectures. *IEEE Trans. Parallel and Distributed Systems*, 9(2):109–125, 1998.

[16] A. George and E. Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM J. Sci. Stat. Comput.*, 8:877–898, 1987.

[17] J. R. Gilbert and J. W. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM Journal on Matrix Analysis and Applications*, 14:334–352, 1993.

[18] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. Parallel and Distributed Systems*, 8:502–520, 1997.

[19] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[20] M. T. Heath and P. Raghavan. Performance of a fully parallel sparse solver. *Int. Journal of Supercomputer Applications*, 11(1):49–64, 1997.

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

[22] HSL. A collection of Fortran codes for large scale scientific computation, 2000.

[23] X. S. Li and J. W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 22–24, 1999.

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

[25] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.

[26] E. G. Ng and B. W. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 14:761–769, 1993.

[27] F. Pellegrini, J. Roman, and P. R. Amestoy. 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, pages 986–995, 1999.

[28] E. Rothberg. Efficient sparse Cholesky factorization on distributed-memory multiprocessors. In J.G. Lewis, editor, *Proceedings Fifth SIAM Conference on Applied Linear Algebra*, page 141, Philadelphia, 1994. SIAM Press.

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

[30] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.