RAL-TR-2001-004

Performance and tuning of two distributed memory sparse solvers[1]

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

ABSTRACT

We examine the performance of two codes using direct methods to solve large sparse linear equations on distributed memory computers. The code `SuperLU` is a right-looking supernodal code, written in C, while `MUMPS` is written in Fortran 90 and uses a multifrontal technique. Both codes use `MPI` for message-passing.

In this short paper, we consider the influence of the `MPI` buffer size and send and receive primitives on both codes and compare the performance of the factorization and solution phases of both codes on test examples from the PARASOL Project on a CRAY T3E-900.

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

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

[1] This work has been partially supported by the France-Berkeley Fund. This report is a preprint of a paper that has been accepted for presentation at the Tenth SIAM Conference on Parallel Processing for Scientific Computing that will be held in Norfolk, Virginia from March 12th-14th, 2001. A version of this report will appear on the CD of the Proceedings for that meeting.

Current reports available by anonymous ftp to ftp.numerical.rl.ac.uk in directory pub/reports. This report is in file adelRAL01004.ps.gz. Report also available through URL http://www.numerical.rl.ac.uk/reports/reports.html. Also published as Technical Report TR/PA/00/91 from CERFACS, 42 Ave G. Coriolis, 31057 Toulouse Cedex, France.

[2] amestoy@enseeiht.fr, ENSEEIHT-IRIT, 2 rue Camichel, Toulouse, France.

[3] I.Duff@rl.ac.uk, Rutherford Appleton Laboratory, and CERFACS, France. The work of this author was supported in part by the EPSRC Grant GR/M78502.

[4] excelle@enseeiht.fr, NAG Ltd, Wilkinson House, Oxford OX2 8DR, England.

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

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

December 20, 2000

Contents

1 Introduction

This work was performed in the context of a France-Berkeley funded project between NERSC located in Berkeley (USA) and CERFACS-ENSEEIHT located in Toulouse (France). We consider the direct solution of sparse linear equations on distributed memory computers using two codes, MUMPS (Amestoy, Duff, L'Excellent and Koster 1999, Amestoy, Duff and L'Excellent 2000*a*) and SuperLU (Li and Demmel 1999) The first uses a multifrontal approach with dynamic pivoting for stability while the second is based on a supernodal technique with static pivoting. We discuss both the tuning and performance analysis of the two sparse solvers on the 512 processor Cray T3E from NERSC, Lawrence Berkeley National Laboratory.

Both approaches 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 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. In both approaches, a pivot order is defined by the analysis and symbolic factorization stages. In the case of MUMPS, the modulus of the prospective pivot is compared to the largest modulus of an entry in the row 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. 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 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 half-precision perturbation to the original matrix. The result is that the factor is not exact and iterative refinement may then be needed. 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.

For both solvers it can be very beneficial (in different ways) 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 (Duff and Koster 1999) code of HSL (HSL 2000) to perform this preordering and scaling (Amestoy, Duff, L'Excellent and Li 2000*b*).

Both approaches use Level 3 BLAS to perform the elimination operations. However, in MUMPS the frontal matrices are always square. It is shown in Amestoy and Puglisi (2000) how one can detect and exploit sparsity within the frontal matrices but the present implementation 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.

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 or two-dimensional (at the root node only). During the analysis phase each tree node is statically assigned a processor *a priori*. The subassignment of blocks of the frontal matrix is then done dynamically during factorization. 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.

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.)
MIXTANK	29957	1995041	1.00	PARASOL (Polyflow S.A.)
TWOTONE	120750	1224224	0.28	Rutherford-Boeing (circuit sim)
Real Symmetric Assembled (RSA)				
Matrix name	Order	No. of entries		Origin
CRANKSG2	63838	7106348		PARASOL (MSC.Software)

Table 1.1: 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[1] (Duff, Grimes and Lewis 1997), the industrial partners of the PARASOL Project[2], and the EECS Department of UC Berkeley[3]. Since SuperLU cannot exploit symmetry, most of our test matrices are unsymmetric. The relatively large symmetric matrix CRANKSG2 and the symmetric version of MUMPS will be used only in Section 2 to illustrate the impact of the size of the MPI buffers on the performance of MUMPS.

We first describe in Section 2 how we have tuned and improved our algorithms. We then compare in Section 3 the performance of the two solvers on the Cray T3E-900 (512 DEC EV-5 processors, 256 Mbytes of memory per processor, 900 peak Megaflop rate per processor) from NERSC on a set of large unsymmetric matrices from real applications. We present some concluding remarks in Section 4.

[1] Web page http://www.cse.clrc.ac.uk/Activity/SparseMatrices/

[2] EU ESPRIT IV LTR Project 20160. Matrices are on Web page http://www.parallab.uib.no/parasol/

[3] Matrix ECL32 included in the Rutherford-Boeing Collection

2 Algorithmic improvements and tuning

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 (Amestoy et al. 1999, Amestoy et al. 2000*a*). From our set of machine dependent parameters we chose appropriate parameters to address this issue. Other algorithmic modifications were motivated by having more processors available to us than formerly. The dynamic scheduling approach used in MUMPS was modified (see Amestoy et al. (2000*b*)) to better control the dynamic distribution of the tasks to the processors. When porting SuperLU to an 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. Depending on the matrices, this Level 3 BLAS kernel improves the uniprocessor factorization speed by about 20% to 40% on an IBM SP2. A performance gain was also observed on the Cray T3E. It is clear that the extra operations are well offset by the benefit of the more efficient Level 3 BLAS routines.

Furthermore, even on the same machine, we found that MPI programming environment changes (for example the default internal buffer size) may result in a dramatic performance difference. This enabled us to identify possible enhancements to the SuperLU code to make its performance more robust. We further discuss this issue in the next two subsections.

2.1 Impact of the MPI internal buffer size on the performance of our solvers

Currently the message transfer in SuperLU is performed using MPI standard send and receive operations, mpi_send and mpi_recv. 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 waiting for the matching receive. The additional copying usually increases the message transfer overhead. However, in many asynchronous algorithms the communication may be overlapped with computation. 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. On the Cray T3E, the current default value of the MPI internal buffer is 4 Kbytes and in an earlier MPI implementation (Release 1.3.0.3) the buffer size was unlimited.

Nprocs	1	2	4	8	16	32	64	128
N	29	33	36	41	46	51	57	64
unlimited	57.0	62.3	53.3	61.5	62.7	65.7	76.1	80.7
4 Kbytes	57.0	108.2	92.4	102.5	104.2	101.6	119.3	111.0

Table 2.1: SuperLU factorization time (seconds). Cubic grid problems of dimension $N \times N \times N$ (11-point discretization of the Laplacian operator). MPI_BUFFER_MAX is set to unlimited and to 4Kbytes. (mpi_recv is used to match mpi_send.)

We illustrate, in Tables 2.1 and 2.2, the impact of the size of the MPI buffer on the performance of our codes. One first sees that, in both cases, the size of 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 a send (mpi_send or even mpi_isend) does not lead to a good overlapping of communication with computation.

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 2.2: MUMPS factorization time (seconds). Matrix CRANKSG2 on 8 processors of the CRAY T3E. (mpi_recv is used to match mpi_isend. Number of operations is 4.2×10^{10}.)

.

2.2 Using asynchronous immediate receives to improve the performance

It is somewhat unpleasant that the performance of our codes depend on the MPI system buffer size. However, if we can match immediate sends (mpi_isend) with immediate receives (mpi_irecv) we can hope to address both issues (that is, independence with respect to MPI buffer size and communication overlapping).

In the MUMPS solver, 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. 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 and provides user space for the communication to proceed. This is independent of the type of send used (mpi_send in SuperLU or mpi_isend in MUMPS). 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 proceed 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.

Although, in the context of MUMPS, the use of an immediate receive seems quite natural, we explain in Amestoy et al. (2000*b*) why it has required more algorithmic developments than might have been expected. The main issue with using an immediate receive in our asynchronous algorithmic context 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 have avoided some possible added complications by restricting ourselves to a single mpi_irecv pending request. We show (compare the results in Tables 2.2 and 2.3) that, as one might expect, 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).

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 2.3: MUMPS factorization time (in seconds) of matrix CRANKSG2 using 8 processors of the CRAY T3E. mpi_irecv is used to match mpi_isend.

For the SuperLU solver, we plan to change the communications pattern used so that the code performance is less dependent on the underlying MPI implementation and is more portable. Although this idea might not be hard to implement, we need to provide an extra buffer on the receiving process to take care of one outstanding message. We still have to experiment with the new scheme and see how sensitive the performance is to the size of the system buffer. We plan to report on this later.

3 Performance analysis

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

For the sake of clarity, we will only report results with the best (in terms of factorization time) sparsity ordering for each approach. Both a minimum degree ordering (AMD) (Amestoy, Davis and Duff 1996) based on MC47 from HSL (HSL 2000) and a nested dissection (ND) ordering are considered. (Sometimes we use the ON-MeTiS ordering from MeTiS (Karypis and Kumar 1998), and sometimes the nested dissection/haloamd ordering from SCOTCH (Pellegrini, Roman and Amestoy 1999) depending on which performs better on each particular problem.) For the matrix TWOTONE it is very beneficial to precede the ordering by an unsymmetric permutation to place large entries on the diagonal. We use the MC64 code of HSL to perform this preordering and scaling (Duff and Koster 1999). When the best ordering for MUMPS is different from that for SuperLU, results with both orderings are provided.

We see that MUMPS is usually faster than SuperLU and is significantly so on a small number of processors. We believe there are two reasons. First, MUMPS handles symmetric and more regular

Matrix	Ord.	Flops	Solver	Number of processors						
		$\times 10^9$		1	4	16	64	128	256	512
BBMAT	AMD	41.5	MUMPS	—	45.7	16.5	11.9	11.2	9.1	12.6
		34.0	SuperLU	—	66.1	22.8	11.2	8.9	9.9	9.1
	ND	25.7	MUMPS	—	39.4	13.2	9.9	9.2	9.4	11.6
		23.5	SuperLU	—	137.8	41.2	17.3	12.4	14.3	14.7
ECL32	AMD	64.6	MUMPS	—	54.6	23.8	15.6	15.1	16.0	16.5
		68.3	SuperLU	—	107.4	35.8	14.9	11.1	10.9	8.9
	ND	20.9	MUMPS	—	24.7	9.7	6.9	7.0	7.0	8.9
		20.7	SuperLU	—	49.0	16.7	9.9	8.8	9.9	9.5
INVEXTR1	ND	8.1	MUMPS	31.8	13.2	4.5	3.8	4.4	5.4	6.3
		5.9	SuperLU	68.2	23.1	9.1	5.7	4.7	6.1	5.8
MIXTANK	ND	13.2	MUMPS	40.8	13.0	5.6	3.9	4.2	4.2	5.4
		12.9	SuperLU	88.1	28.8	10.1	5.3	4.5	5.6	5.5
TWOTONE	MC64	29.3	MUMPS	—	40.3	18.6	14.4	14.3	14.0	14.3
	+AMD	8.0	SuperLU	—	106.2	32.7	21.0	16.2	21.2	18.5

Table 3.1: Factorization time (in seconds). "—" indicates not enough memory. Flops corresponds to the number of operations involved during factorization.

data structures better than SuperLU, because MUMPS uses Level 3 BLAS kernels on bigger blocks than those used within SuperLU. As a result, the Megaflop rate of MUMPS on one processor is on average about twice that of the SuperLU factorization. Note that, even on the matrix TWOTONE, for which SuperLU performs three times fewer operations than MUMPS (see column 3), MUMPS is over 2.5 times faster than SuperLU on four processors. On a small number of processors, we also notice that SuperLU does not always fully benefit from the reduction in the number of operations due to the use of a nested dissection ordering (see BBMAT with SuperLU using 4 processors).

We see that the ordering very significantly influences the performance of the codes (see results with matrices BBMAT and ECL32). In particular, MUMPS generally outperforms SuperLU when nested dissection ordering is used 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 solver provides enough parallelism to benefit from using more than 128 processors. The only exception is matrix ECL32 using the AMD ordering (requiring 64×10^9 flops for the factorization), for which SuperLU continues to decrease the factorization time up to 512 processors. Our lack of other large unsymmetric systems gives us few data points in this regime but one might expect that, independently of the ordering, the 2D distribution used in SuperLU should provide better scalability (and hence eventually better performance) on a large number of processors than the mixed 1D and 2D distribution used in MUMPS. To further analyse the scalability of our solvers we have also reported in Amestoy et al. (2000*b*) results obtained on large three dimensional regular grid problems.

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

In Table 3.2, we report both the time to perform one step of solution (use factorized matrix to solve $\mathbf{A}x = b$ and when necessary ($Berr$ greater than $\sqrt{\varepsilon}$) the time to improve the solution using iterative refinement (lines with "+ IR"). With SuperLU, one step of iterative refinement was always enough to reduce the backward error to $\sqrt{\varepsilon}$. With MUMPS, iterative refinement was only required on the matrix INVEXTR1 and the backward error was already so close to $\sqrt{\varepsilon}$ that on 4 and 8 processors iterative refinement was not required. We first observe 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.

The performance reported in Table 3.2 shows that the regularity in the structure of the matrix factors generated by the factorization phase of MUMPS generally leads to a faster solve phase than that of SuperLU for up to 256 processors. On 512 processors, the solve phase of SuperLU is sometimes faster than that of MUMPS. The cost of iterative refinement can significantly increase the cost of obtaining a solution. With SuperLU, because of static pivoting, it is more likely that iterative refinement will be required to obtain an accurate solution on numerically difficult matrices. With MUMPS, the use of partial pivoting during the factorization reduces the number of matrices for which iterative refinement is required.

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

Table 3.2: Solve time (in seconds). " — + (IR) " shows the time spent improving the initial solution using iterative refinement. "—" indicates not enough memory.

4 Concluding remarks

We have described some algorithmic tuning and performance analysis of two state-of-the art distributed sparse direct solvers, SuperLU and MUMPS. Tuning that is planned for MUMPS includes using more global information in the scheduling algorithm and a closer integration of orderings based on nested dissection; for SuperLU, the plan is to exploit more parallelism from the dependency graph and to experiment on the use of immediate primitives to overlap communication and computation.

We started this exercise with the intention of comparing a wider range of sparse codes. However, we have observed that 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 Amestoy et al. (2000*b*), we summarize the major characteristics of the parallel distributed sparse direct codes of which we are aware.

Acknowledgments
We want to thank James Demmel, Jacko Koster, Chiara Puglisi and Rich Vuduc for very helpful discussions.

References

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

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

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

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

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y. and Li, X. S. (2000*b*), Analysis, tuning and comparison of two general sparse solvers for distributed memory computers, Technical Report LBNL-45992, NERSC, Lawrence Berkeley National Laboratory. Shortened version submitted to *ACM Trans. Math. Softw.*

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

Duff, I. S., Grimes, R. G. and Lewis, J. G. (1997), The Rutherford-Boeing Sparse Matrix Collection, Technical Report RAL-TR-97-031, Rutherford Appleton Laboratory. Also

Technical Report ISSTECH-97-017 from Boeing Information & Support Services, Seattle and Report TR/PA/97/36 from CERFACS, Toulouse.

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

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

Li, X. S. and Demmel, J. W. (1999), A scalable sparse direct solver using static pivoting, *in* 'Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing', San Antonio, Texas.

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