MUMPS: A General Purpose Distributed Memory Sparse Solver*

Patrick R. Amestoy[1], Iain S. Duff[2], Jean-Yves L'Excellent[3], and Jacko Koster[4]

[1] ENSEEIHT-IRIT, 2 rue Camichel, 31071 Toulouse cedex, France, and NERSC, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley CA 94720
amestoy@enseeiht.fr
[2] Rutherford Appleton Laboratory, Chilton, Didcot, Oxon, OX11 0QX England, and CERFACS, Toulouse, France
I.Duff@rl.ac.uk
[3] NAG LTD, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR England
jeanyves@nag.co.uk
[4] Parallab, University of Bergen, 5020 Bergen, Norway
jak@ii.uib.no

Abstract. MUMPS is a public domain software package for the multifrontal solution of large sparse linear systems on distributed memory computers. The matrices can be symmetric positive definite, general symmetric, or unsymmetric, and possibly rank deficient. MUMPS exploits parallelism coming from the sparsity in the matrix and parallelism available for dense matrices. Additionally, large computational tasks are divided into smaller subtasks to enhance parallelism. MUMPS uses a distributed dynamic scheduling technique that allows numerical pivoting and the migration of computational tasks to lightly loaded processors. Asynchronous communication is used to overlap communication with computation. In this paper, we report on recently integrated features and illustrate the present performance of the solver on an SGI Origin 2000 and a CRAY T3E.

1 Introduction

MUMPS is a MUltifrontal Massively Parallel Solver [2,3] that is capable of solving systems of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an $n \times n$ symmetric positive definite, general symmetric, or unsymmetric sparse matrix that is possibly rank deficient, $\mathbf{b}$ is the right-hand side vector, and $\mathbf{x}$ is the solution vector to be computed.

The development of the MUMPS software package started as part of the European project PARASOL. Although this project finished in June 1999, the functionality of MUMPS is still being extended and its performance improved. It has been fully integrated in the parallel domain decomposition solver DD that is being developed by Parallab (Bergen, Norway) and that is presented elsewhere in this volume [6].

Several aspects of the algorithms used in MUMPS combine to give us a package which is unique amongst sparse direct solvers. These include:

* This work has been partially supported by the PARASOL Project (EU ESPRIT IV LTR Project 20160).

T. Sørevik et al. (Eds.): PARA 2000, LNCS 1947, pp. 121–130, 2001.

- partial threshold pivoting (to control the growth in the factors) during the numerical factorization of unsymmetric and general symmetric matrices,
- the ability to automatically adapt to processor load variations during the numerical phase,
- high performance, by exploiting the independence of computations due to sparsity and that available for dense matrices, and
- the capability of solving a wide range of problems, using either an **LU** or $\mathbf{LDL}^{\mathrm{T}}$ factorization.

To address all these factors, `MUMPS` is a fully asynchronous algorithm with dynamic data structures and a distributed dynamic scheduling of tasks. Other available codes are usually based on a static mapping of the tasks and data and, for example, do not allow task migration during numerical factorization.

Besides these features, the current version of the `MUMPS` package provides a large range of options, including the possibility of inputting the matrix in assembled format either on a single processor or distributed over the processors. Additionally, the matrix can be input in elemental format (currently only on one processor). `MUMPS` can also determine the rank and a null-space basis for rank-deficient matrices, and can return a Schur complement matrix. It contains classical pre- and postprocessing facilities; for example, matrix scaling, iterative refinement, and error analysis.

In Section 2, we introduce the main characteristics of the algorithms used within `MUMPS` and discuss the sources of parallelism. In Section 3, we discuss the performance on an SGI Origin 2000 and a CRAY T3E. We analyse the effect of new algorithmic aspects to improve the scalability and present preliminary comparisons with some other sparse direct solvers.

2 Distributed Memory Multifrontal Factorization

We refer the reader to the literature (e.g., [7,10]) for an introduction to multifrontal methods. Briefly, the multifrontal factorization of a sparse matrix can be described by an **assembly tree**, where each node corresponds to the computation of a Schur complement matrix, and each edge represents the transfer of this matrix (the contribution block) from the child node to the parent node (or father) in the tree. This parent node assembles (or sums) the contribution blocks from all its child nodes with entries from the original matrix.

Because of numerical pivoting, it is possible that some variables cannot be eliminated from a frontal matrix. The rows and columns that correspond to such variables are added to the contribution block that is sent to the parent node resulting in a larger than predicted frontal matrix. In general this introduces additional (numerical) fill-in in the factors and requires the use of dynamic data structures.

The parallel code solves the system $\mathbf{Ax} = \mathbf{b}$ in three steps:

1. Analysis. One of the processors (the host) computes an approximate minimum degree ordering based on the symmetrized matrix pattern $\mathbf{A}+\mathbf{A}^T$ and carries out the symbolic factorization. The ordering can also be provided by the user. The host computes a mapping of the nodes of the assembly tree to the processors and sends symbolic information to the other processes, each of which estimates the work space required for its part of the factorization and solution.
2. Factorization. The original matrix is first preprocessed and distributed (or redistributed) to the processes. Each process allocates space for contribution blocks and factors. The numerical factorization on each frontal matrix is performed by a process determined by the analysis phase and potentially one or more other processes that are determined dynamically (see below).
3. Solution. The right-hand side vector $\mathbf{b}$ is broadcast from the host to the other processes. They compute the solution vector $\mathbf{x}$ using the distributed factors computed during the factorization phase. The solution vector is then assembled on the host.

Operations at a pair of nodes in the assembly tree where neither is an ancestor of the other are independent. This makes it possible to obtain parallelism from the tree (so-called **tree parallelism**). To reduce communication between processes during the factorization and solution phases, the analysis phase computes a mapping that assigns a complete subtree of the assembly tree to a single process.

Tree parallelism is poor near the root of the tree. It is thus necessary to obtain further parallelism within the large nodes near the root. Therefore, a second level of parallelism (**node parallelism**) is introduced by using a one-dimensional row block partitioning of the frontal matrices for (type 2) nodes near the root that have a large contribution block (see Figure 1). The process (master) that was assigned such a node during the analysis phase, determines a set of (slave) processors dynamically, based on work load estimates that the processes broadcast regularly. The (fully summed) rows of the frontal matrix that contain the potential pivots are assigned to the master process, while the rest of the matrix is partitioned over the slave processes. The master and slave processes then use parallel blocked versions of higher Level BLAS routines for the factorization of the frontal matrix. In case the number of fully summed rows is large relative to the size of the contribution block, MUMPS will split (during the analysis phase) the corresponding tree node into several smaller nodes to avoid the master process from having too much work in the factorization of this node.

Finally, if the frontal matrix of the (type 3) root node is large enough, this matrix is also factorized in parallel. MUMPS uses a two-dimensional block cyclic distribution of the root matrix and subroutines from ScaLAPACK or the vendor equivalent for the actual factorization.

MUMPS allows the input matrix to be held initially on one processor, but also allows the input matrix to be already distributed over the processors. This gives more freedom to the user, but also reduces the time for redistribution of data, because MUMPS can use asynchronous all-to-all (instead of one-to-all)

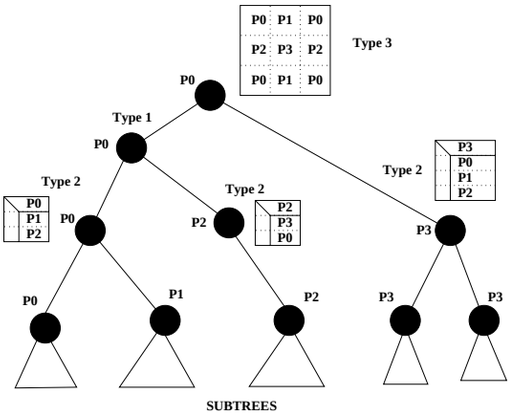

Fig. 1. Example of a (condensed) assembly tree where the leaves represent subtrees with a possible distribution of the computations over the four processes **P0**, **P1**, **P2**, and **P3**.

communications. Additionally, MUMPS can format the input data and sort tasks in parallel. Furthermore, we can expect to solve larger problems since storing the complete matrix on one processor limits the size of the problem that can be solved on a distributed memory computer.

3 Performance Analysis

MUMPS is a portable code that has been used successfully on SGI Cray Origin 2000, IBM SP2, SGI Power Challenge and Cray T3E machines. The software is written in Fortran 90, uses MPI for message passing and the BLAS, LAPACK, BLACS, and ScaLAPACK libraries for (parallel) dense matrix operations.

3.1 Environment

In this paper we report on experiments on the SGI Origin 2000 at Parallab (2×64 R10000 processors, each running at 195 MHertz and with a peak of 400 MFlops per second) and on the Cray T3E from NERSC at Lawrence Berkeley National Laboratory (512 DEV EV-5 processors, each with a peak of 900 MFlops per second).

MUMPS (latest release 4.1) has been tested extensively on problems provided by industrial partners in the PARASOL project. The corresponding matrices are available at `http://www.parallab.uib.no/parasol/`. Other test problems that we have used come from the forthcoming Rutherford-Boeing collection [8] and from the EECS Department of UC Berkeley. Table 1 summarizes some characteristics of the test problems that are used in this paper. The test problems include unsymmetric, symmetric, and symmetric unassembled matrices. The structural

symmetry (column "StrSym") is the ratio of the number of nonzeros that are matched by nonzeros in symmetric locations, divided by the total number of entries in the matrix. For symmetric problems, the column "Entries" represents the number of nonzeros in only the lower triangular part of the matrix.

Table 1. Test matrices.

Real Unsymmetric Assembled				
Matrix name	Order	Entries	StrSym	Origin
ECL32	51993	380415	0.93	EECS Dept. of UC Berkeley
BBMAT	38744	1771722	0.54	Rutherford-Boeing, CFD
INVEXTR1	30412	1793881	0.97	Polyflow (PARASOL)
Real Symmetric Assembled				
Matrix name	Order	Entries	StrSym	Origin
CRANKSG2	63838	7106348	1.00	MSC.Software (PARASOL)
BMWCRA_1	148770	5396386	1.00	MSC.Software (PARASOL)
HOOD	220542	5494489	1.00	INPRO (PARASOL)
BMW3_2	227362	5757996	1.00	MSC.Software (PARASOL)
INLINE_1	503712	18660027	1.00	MSC.Software (PARASOL)
LDOOR	952203	23737339	1.00	INPRO (PARASOL)
Real Symmetric Unassembled				
Matrix name	Order	Entries	elements	Origin
SHIP_003.RSE	121728	9729631	45464	Det Norske Veritas (PARASOL)
SHIPSEC5.RSE	179860	11118602	52272	Det Norske Veritas (PARASOL)

The ordering currently available inside MUMPS is the Approximate Minimum Degree ordering [1]. It has been observed (see for example [3]) that hybrid ND orderings (i.e., orderings based on nested dissection combined with local heuristics for smaller subgraphs) are usually more efficient than purely local heuristics for large problems. Figure 2 illustrates this. The figure shows that the number of floating-point operations for the factorization is significantly smaller with hybrid ND orderings. Furthermore, such orderings can lead to a better load balance in a parallel environment. Since MUMPS has the facility to use orderings that are computed externally (prior to calling MUMPS), the experiments reported in this paper use hybrid ND orderings. (Either METIS [12], a combination of SCOTCH and Halo-AMD [14], or an in-house Det Norske Veritas ordering is used).

Finally, the times reported are measured in seconds and represent elapsed times for the numerical factorization phase, normally the most costly part of the solver.

3.2 Performance on an SGI Origin 2000

Table 2 presents times for the factorization of some large symmetric PARASOL test matrices (assembled and unassembled) on up to 32 processors of the SGI

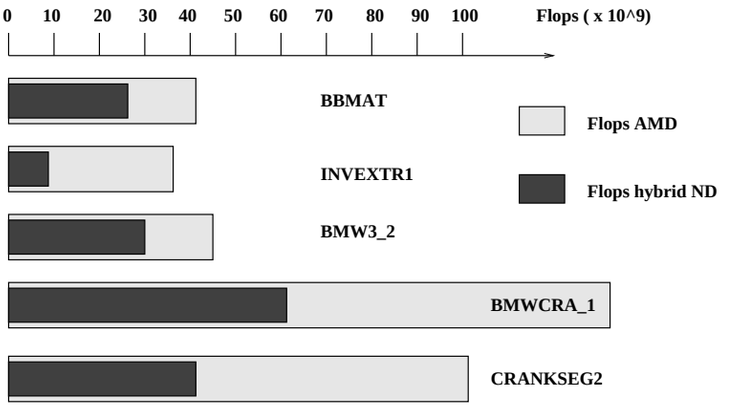

Fig. 2. Number of floating-point operations during MUMPS factorization using hybrid ND and Approximate Minimum Degree orderings.

Origin 2000. Speedups are up to 13.8. We consider the absolute GFlops rate obtained on this machine good for a sparse direct solver.

Table 2. Time (in seconds) for factorization using MUMPS Version 4 on some PARASOL test problems on an SGI Origin 2000.

	Flops	Number of processors					
Matrix	($\times 10^9$)	1	2	4	8	16	32
LDOOR	74.5	416	228	121	68	39	31 (2.4 GFlops)
BMWCRA_1	61.0	307	178	82	58	36	27 (2.3 GFlops)
BMW3_2	28.6	151	96	53	33	18	15 (1.9 GFlops)
INLINE_1	143.2	757	406	225	127	76	55 (2.6 GFlops)
SHIP_003.RSE	73.0	392	237	124	108	51	43 (1.7 GFlops)
SHIPSEC5.RSE	51.7	281	181	103	62	37	29 (1.8 GFlops)

We note that the largest problem that we have solved to date on this platform is a symmetric problem provided by the PARASOL partner MSC.Software of order 943695 with 39.3 million entries in its lower triangular part. The number of entries in the factors is 1.4×10^9 (11.1 Gbytes) and the number of floating-point operations required by the numerical factorization is 5.9×10^{12}; MUMPS required 6.2 hours on 2 processors and 17 Gbytes of memory to factorize the matrix. Note that because of the increase in the total memory requirement, we could not solve the problem on more processors.

3.3 Performance Tuning on a CRAY T3E

While MUMPS was initially developed on machines with a limited number of processors, access to the 512 processor CRAY T3E from NERSC gave us new targets in terms of scalability.

MUMPS has a set of platform-dependent parameters (e.g., block sizes) controlling the uniprocessor and multiprocessor efficiency. First, those parameters were experimentally tuned to meet the characteristics of the architecture. Because of the relatively higher ratio between communication bandwidth and processor performance, the granularity of the tasks arising from type 2 node parallelism (see Section 2) could be reduced, thus allowing also the use of more processors. In this context, task management became even more crucial and the dynamic scheduling strategy was modified to allow the master of a type 2 node to choose dynamically *not only* the set of processors to which generated tasks should be assigned, but also the number of tasks and the size of each task. In this dynamic situation, one rule of thumb is that a process should avoid giving work to a more loaded process.

Table 3. Factorization time (in seconds) for various buffer sizes of the matrix CRANKSG2 on 8 processors of the T3E. (* denotes the default size on the T3E)

	MPI buffer size (bytes)						
type of receive	0	512	1K	4K*	64K	512K	2Mb
standard (MPI_RECV)	37.0	37.4	38.3	37.6	32.8	28.3	26.4
immediate (MPI_IRECV)	27.3	26.5	26.6	26.4	26.2	26.2	26.4

Another important modification was the introduction of immediate communication primitives (MPI_IRECV) on the receiver side, allowing transfer of data to start earlier, as well as more overlap between computation and communication. On the T3E, depending on the size of the MPI internal reception buffer, it can happen that a message reaches a process before the corresponding MPI_RECV is posted. On the other hand, if a message is too large to fit in the MPI internal reception buffer, communication cannot start. Posting an MPI_IRECV earlier allows communication to start as soon as possible, independently of the size of the MPI buffer. This effect is shown in Table 3, where one can see that a significant gain is obtained for the default size of the MPI buffer (26.4 vs. 37.6 seconds).

In Table 4, `MUMPS 4.0` denotes the version of the code where only uniprocessor parameters are tuned for the T3E, and `MUMPS 4.1` denotes the version in which all the modifications described above are integrated. We see in Table 4 that the new version performs significantly better, both for a symmetric and an unsymmetric test problem.

Table 4. Times (in seconds) for factorization by `MUMPS 4.0` and `MUMPS 4.1` on two PARASOL test problems. ('—' denotes not enough memory)

		Number of processors						
Matrix	`MUMPS`	1	2	4	8	16	32	64
HOOD	4.0	—	—	25.1	12.1	6.3	3.7	3.5
(8.2×10^9 flops)	4.1	—	—	15.0	8.2	4.4	4.0	2.4
INVEXTR1	4.0	—	23.1	18.0	15.6	14.9	14.4	14.5
(8.1×10^9 flops)	4.1	—	23.1	11.8	8.5	7.5	7.0	6.8

3.4 Preliminary Comparisons with Other Codes

Table 5 shows a preliminary comparison between the performances of `MUMPS` and `PSPASES` (version 1.0.2) [11]. `PSPASES` solves symmetric positive definite systems and performs a Cholesky ($\mathbf{LL}^T$) factorization based on a static mapping of the computational tasks to the processors. (`MUMPS` uses an $\mathbf{LDL}^T$ factorization.) The table shows that `MUMPS` behaves well in absolute performance and scalability even though `PSPASES` is especially designed for symmetric positive definite problems. Note that on a larger number of processors (128 or more), one should expect `PSPASES` to more significantly outperform `MUMPS` for which the 1D partitioning of the frontal matrices (due to numerical pivoting) will limit the scalability.

Table 5. Time (in seconds) for factorization by `MUMPS` and `PSPASES` of two symmetric test cases (BMWCRA_1 and INLINE_1) on the SGI Origin 2000. The ordering used for both solvers is based on METIS. `PSPASES` requires 2^k, $k > 0$, processors.

		Number of processors						
BMWCRA_1	Flops ($\times 10^9$)	1	2	4	8	16	32	
`PSPASES 1.0.2`	between 62 and 71	—	157	95	66	39	21	
`MUMPS 4.1`	61	307	178	82	58	36	27	
INLINE_1	Flops ($\times 10^9$)	1	2	4	8	16	32	64
`PSPASES 1.0.2`	between 147 and 157	—	574	399	225	128	75	41
`MUMPS 4.1`	143	757	406	225	127	76	55	42

Table 6 shows a preliminary comparison of `MUMPS` and `SuperLU` [13] on the CRAY T3E for two unsymmetric test cases (BBMAT and ECL32). `SuperLU` uses a supernodal right-looking formulation with a static approach. Numerical stability issues are handled with "static pivoting" in the sense that, if a pivot is smaller than $\sqrt{\epsilon} \parallel \mathbf{A} \parallel$, the pivot is set to $\sqrt{\epsilon} \parallel \mathbf{A} \parallel$ and computations can continue exactly as forecast by the analysis. This means that `SuperLU` actually computes the factorization of a modified matrix. However, an accurate solution to the initial system is obtained by using (*i*) iterative refinement, and (*ii*) the `MC64` code [9], used as a preprocessing step to permute large entries onto the diagonal of the matrix.

Although the results presented here are only preliminary, an extensive comparison of MUMPS and SuperLU on a CRAY T3E has now been performed in the context of a France-Berkeley funded project and results of this study are available in [4].

Table 6. Time (in seconds) for factorization by MUMPS and SuperLU for two unsymmetric matrices (BBMAT and ECL32) on the T3E. The same ND ordering is used for both solvers.

		Number of processors						
BBMAT	Flops ($\times 10^9$)	4	16	32	64	128	256	512
SuperLU	23.5	137.8	31.2	25.2	17.3	12.4	14.3	14.7
MUMPS 4.1	25.7	39.4	13.2	11.9	9.9	9.2	9.4	11.6
ECL32	Flops ($\times 10^9$)	4	16	32	64	128	256	512
SuperLU	20.7	49.0	16.7	12.0	9.9	8.8	9.9	9.5
MUMPS 4.1	20.9	24.7	9.7	7.7	6.9	7.0	7.0	8.9

In fact, the only code we are aware of which is comparable to MUMPS in terms of functionality is SPOOLES [5]. For example, it also allows numerical pivoting during the factorization. It uses an object-oriented approach and has the advantage of being very modular and flexible. However, partial experiments performed by a third party show that the performance of SPOOLES is far behind the performance of the solvers mentioned in this section.

4 Concluding Remarks

MUMPS is a general purpose distributed multifrontal solver with a wide range of functionalities. Experimental results show that the solver has good overall performance and scales reasonably well on various parallel architectures.

On-going and future work include algorithmic developments that will improve the basic performance of the code, the tuning of the code on other architectures (e.g., clusters), further comparisons with other codes, and the development of new features required by MUMPS users.

The MUMPS package is freely available (for non-commercial use) and can be obtained from `http://www.enseeiht.fr/apo/MUMPS/`.

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, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods in Appl. Mech. Engrg.*, 184:501–520, 2000. Special issue on domain decomposition and parallel computing.

3. P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. Technical Report RAL-TR-1999-059, Rutherford Appleton Laboratory, 1999. Submitted to *SIAM Journal on Matrix Analysis and Applications*.
4. P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis, tuning and comparison of two general sparse solvers for distributed memory computers. Technical Report LBNL-45992, NERSC, Lawrence Berkeley National Laboratory, June 2000. FBF report.
5. C. Ashcraft and R. G. Grimes. SPOOLES: An object oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing*, 1999.
6. P. E. Bjørstad, J. Koster, and P. Krzyżanowski. Domain decomposition solvers for large scale industrial finite element problems. In *Proceedings of the PARA2000 Workshop on Applied Parallel Computing, June 18-21, Bergen*. Springer-Verlag, 2000.
7. I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.
8. I. S. 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.
9. 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. Submitted to *SIAM Journal on Matrix Analysis and Applications*.
10. I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.
11. A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. on Parallel and Distributed Systems*, 8(5):502–520, 1997.
12. G. Karypis and V. Kumar. MeTiS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota, September 1998.
13. 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.
14. 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.