A fully asynchronous multifrontal solver using distributed dynamic scheduling[1]

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

Technical Report TR/PA/99/28

August 9, 1999

CERFACS
42 Ave G. Coriolis
31057 Toulouse Cedex
France

ABSTRACT
We describe the main features and discuss the tuning of algorithms for the direct solution of sparse linear systems on distributed memory computers developed in the context of PARASOL (ESPRIT IV LTR Project (No 20160)). The algorithms use a multifrontal approach and are especially designed to cover a large class of problems. The problems can be symmetric positive definite, general symmetric, or unsymmetric matrices, all possibly rank deficient, and they can be provided by the user in several formats. The algorithms achieve high performance by exploiting parallelism coming from the sparsity in the problem and that available for dense matrices. The algorithms use a dynamic distributed task scheduling technique to accommodate numerical pivoting and to allow the migration of computational tasks to lightly loaded processors. Large computational tasks are divided into subtasks to enhance parallelism. Asynchronous communication is used throughout the solution process for the efficient overlap of communication and computation.
We illustrate our design choices by experimental results obtained on a Cray SGI Origin 2000 and an IBM SP2 for test matrices provided by industrial partners in the PARASOL project.

Keywords: MPI, distributed memory architecture, sparse matrices, multifrontal direct methods.
AMS(MOS) subject classifications: 65F05, 65F50.

---

[1] Current reports available at http://www.cerfacs.fr/algor/algo_reports.html. Also appeared as Technical Report RAL-TR-1999-059 from Rutherford Appleton Laboratory, Oxfordshire. An earlier version appeared as Technical Report RT/APO/99/02 from ENSEEIHT-IRIT, Toulouse.

[2] amestoy@enseeiht.fr, ENSEEIHT-IRIT, 2 rue Camichel, Toulouse, France.
[3] duff@cerfacs.fr. Also at Atlas Centre, RAL, Oxon OX11 0QX, England.
[4] excelle@cerfacs.fr, CERFACS and ENSEEIHT-IRIT.
[5] J.Koster@rl.ac.uk, Rutherford Appleton Laboratory.

Contents

1 Introduction

We consider the direct solution of large sparse linear systems on distributed memory computers. The systems are 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 work presented in this article, has been performed as Work Package 2.1 within the PARASOL Project. PARASOL is an ESPRIT IV Long Term Research Project (No 20160) for "An Integrated Environment for Parallel Sparse Matrix Solvers". The main goal of this Project, which started on January 1996 and finishes in June 1999, is to build and test a portable library for solving large sparse systems of equations on distributed memory systems. The final library will be in the public domain and will contain routines for both the direct and iterative solution of symmetric and unsymmetric systems.

In the context of PARASOL, we have produced a `MU`ltifrontal `M`assively `P`arallel `S`olver [27, 28] referred to as `MUMPS` in the remainder of this paper. Several aspects of the algorithms used in `MUMPS` combine to give an approach which is unique among sparse direct solvers. These include:

- classical partial numerical pivoting during numerical factorization requiring the use of dynamic data structures,
- the ability to automatically adapt to computer 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, including symmetric, unsymmetric, and rank-deficient systems using either $\mathbf{LU}$ or $\mathbf{LDL}^T$ factorization.

To address all these factors, we have designed a fully asynchronous algorithm based on a multifrontal approach with distributed dynamic scheduling of tasks. The current version of our 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.

Among the other work on distributed memory sparse direct solvers of which we are aware [7, 10, 12, 22, 23, 24], we do not know of any with the same capabilities as the `MUMPS` solver. Because of the difficulty of handling dynamic data structures efficiently, most distributed memory approaches do not perform numerical pivoting during the factorization phase. Instead, they are based on a static mapping of the tasks and data and do not allow task migration during numerical factorization. Numerical pivoting can clearly be avoided for symmetric positive definite matrices. For unsymmetric matrices, Duff and Koster [18, 19] have designed algorithms to permute large entries onto the diagonal and have shown that this can significantly reduce numerical pivoting. Demmel and Li [12] have shown that, if one preprocesses the matrix using the code of Duff and Koster, static pivoting (with possibly modified diagonal values) followed by iterative refinement can normally provide reasonably accurate solutions. They have observed that this preprocessing, in combination with an appropriate scaling of the input matrix, is a key issue for the numerical stability of their approach.

The rest of this paper is organized as follows. We first introduce some of the main terms used in a multifrontal approach in Section 2. Throughout this paper, we study the performance obtained on the set of test problems that we describe in Section 3. We discuss, in Section 4, the main parallel features of our approach. In Section 5, we give initial performance figures and we show the influence of the ordering of the variables on the performance of `MUMPS`. In Section 6, we describe our work on accepting the input of matrices in elemental form. Section 7 then briefly describes the main properties of the algorithms used for distributed assembled matrices. In Section 8, we comment on memory scalability issues. In Section 9, we describe and analyse the distributed dynamic scheduling strategies that will be further analysed in Section 10 where we show how we can modify the assembly tree to introduce more parallelism. We present a summary of our results in Section 11.

Most results presented in this paper have been obtained on the 35 processor IBM SP2 located at GMD (Bonn, Germany). Each node of this computer is a 66 MHertz processor with 128 MBytes of physical memory and 512 MBytes of virtual memory. The SGI Cray Origin 2000 from Parallab (University of Bergen, Norway) has also been used to run some of our largest test problems. The Parallab computer consists of 64 nodes sharing 24 GBytes of physically distributed memory. Each node has two R10000 MIPS RISC 64-bit processors sharing 384 MBytes of local memory. Each processor runs at a frequency of 195 MHertz and has a peak performance of a little under 400 Mflops per second.

All experiments reported in this paper use Version 4.0 of `MUMPS`. The software is written in Fortran 90. It requires MPI for message passing and makes use of BLAS [14, 15], LAPACK [6], BLACS [13], and ScaLAPACK [9] subroutines. On the IBM SP2, we are currently using a non-optimized portable local installation of ScaLAPACK, because the IBM optimized library PESSL V2 is not available.

2 Multifrontal methods

It is not our intention to describe the details of a multifrontal method. We rather just define terms used later in the paper and refer the reader to our earlier publications for a more detailed description, for example [3, 17, 20].

In the multifrontal method, all elimination operations take place within dense submatrices, called **frontal matrices**. A frontal matrix can be partitioned as shown in Figure 1. In this matrix, pivots can be chosen from within the block $\mathbf{F}_{11}$ only. The Schur complement matrix $\mathbf{F}_{22} - \mathbf{F}_{21}\mathbf{F}_{11}^{-1}\mathbf{F}_{12}$ is computed and used to update later rows and columns of the overall matrix. We call this update matrix, the **contribution block**.

$$
\begin{array}{r c}
 & \begin{array}{cc} \text{fully summed columns} & \text{partly summed columns} \\ \downarrow & \downarrow \end{array} \\
\begin{array}{r} \text{fully summed rows} \rightarrow \\ \text{partly summed rows} \rightarrow \end{array} & \begin{bmatrix} \mathbf{F}_{11} & \mathbf{F}_{12} \\ \mathbf{F}_{21} & \mathbf{F}_{22} \end{bmatrix}
\end{array}
$$

Figure 1: Partitioning of a frontal matrix.

The overall factorization of the sparse matrix using a multifrontal scheme can be described by an **assembly tree**, where each node corresponds to the computation of a Schur complement as just described, and each edge represents the transfer of the contribution block from the son node to the father node in the tree. This father node assembles (or sums) the contribution blocks from all its son nodes with entries from the original matrix. If the original matrix is given in assembled format, complete rows and columns of the input matrix are assembled at once, and, to facilitate this, the input matrix is ordered according to the pivot order and stored as a collection of arrowheads. That is, if the permuted matrix has entries in, for example, columns $\{j_1, j_2, j_3\}$ of row i, $i < j_1, j_2, j_3$, and in rows $\{k_1, k_2\}$ of column i, $i < k_1, k_2$, then the arrowhead list associated with variable i is $\{a_{ii}, a_{j_1 i}, a_{j_2 i}, a_{j_3 i}, a_{ik_1}, a_{ik_2}\}$. In the symmetric case, only entries from the lower triangular part of the matrix are stored. We say that we are storing the matrix in **arrowhead form** or by **arrowheads**. For unassembled matrices, complete element matrices are assembled into the frontal matrices and the input matrix need not be preprocessed.

In our implementation, the assembly tree is constructed from the symmetrized pattern of the matrix and a given sparsity ordering. By symmetrized pattern, we mean the pattern of the matrix $\mathbf{A} + \mathbf{A}^T$ where the summation is symbolic. Note that this allows the matrix to be unsymmetric.

Because of numerical pivoting, it is possible that some variables cannot be eliminated from a frontal matrix. The fully summed rows and columns that correspond to such variables are added to the contribution block that is sent to the father node. The assembly of fully summed rows and columns into the frontal matrix of the father node means that the corresponding elimination operations are **delayed**. This will be repeated until elimination steps on the later frontal matrices have introduced stable pivots to the delayed fully summed part. The delay of elimination steps corresponds to an *a posteriori* modification of the original assembly tree structure and in general introduces additional (numerical) fill-in in the factors.

An important aspect of the assembly tree is that operations at a pair of nodes where neither is an ancestor of the other are independent. This gives the possibility for obtaining parallelism from the tree (so-called **tree parallelism**). For example, work can commence in parallel on all the leaf nodes of the tree. Fortunately, near the root node of the tree, where the tree parallelism is very poor, the frontal matrices are usually much larger and so techniques for exploiting parallelism in dense factorizations can be used (for example, blocking and use of higher Level BLAS). We call this **node parallelism**. We discuss further aspects of the parallelism of the multifrontal method in later sections of this paper. Our work is based on our experience of designing and implementing a multifrontal scheme on shared and virtual shared memory computers (for example, [2, 3, 4]) and on an initial prototype distributed memory multifrontal version [21]. We describe the design of our resulting distributed memory multifrontal algorithm in the rest of this paper.

3 Test problems

Throughout this paper, we will use a set of test problems to illustrate the performance of our algorithms. We describe the set in this section.

In Tables 1 and 2, we list our unassembled and assembled test problems, respectively. All except one come from the industrial partners of the PARASOL Project. The remaining matrix, BBMAT, is from the forthcoming Rutherford-Boeing Sparse Matrix Collection [16]. For symmetric matrices, we show the number of entries in the lower triangular part of the matrix. Typical PARASOL test cases are from the following major application areas: computational fluid dynamics (CFD), structural mechanics, modelling compound devices, modelling ships and mobile offshore platforms, industrial processing of complex non-Newtonian liquids, and modelling car bodies and engine components. Some test problems are provided in both assembled format and elemental format. The suffix (RSA or RSE) is used to differentiate them. For those in elemental format, the original matrix is represented as a sum of element matrices

$$\mathbf{A} = \sum \mathbf{A}_i ,$$

where each $\mathbf{A}_i$ has nonzero entries only in those rows and columns that correspond to variables in the ith element. Because element matrices may overlap, the number of entries of a matrix in elemental format is usually larger than for the same matrix when assembled (compare the matrices from Det Norske Veritas of Norway in Tables 1 and 2). Typically there are about twice the number of entries in the unassembled elemental format.

Real Symmetric Elemental (RSE)				
Matrix name	Order	No. of elements	No. of entries	Origin
M_T1.RSE	97578	5328	6882780	Det Norske Veritas
SHIP_001.RSE	34920	3431	3686133	Det Norske Veritas
SHIP_003.RSE	121728	45464	9729631	Det Norske Veritas
SHIPSEC1.RSE	140874	41037	8618328	Det Norske Veritas
SHIPSEC5.RSE	179860	52272	11118602	Det Norske Veritas
SHIPSEC8.RSE	114919	35280	7431867	Det Norske Veritas
THREAD.RSE	29736	2176	3718704	Det Norske Veritas
X104.RSE	108384	6019	7065546	Det Norske Veritas

Table 1: Unassembled symmetric test matrices from PARASOL partner (in elemental format).

In Tables 3, 4, and 5, we present statistics on the factorizations of the various test problems using `MUMPS`. The tables show the number of entries in the factors and the number of floating-point operations (flops) for elimination. For unsymmetric problems, we show both the estimated number, assuming no pivoting, and the actual number when numerical pivoting is used.

The statistics clearly depend on the ordering used. Two classes of ordering will be considered in this paper. The first is an Approximate Minimum Degree ordering (referred to as AMD, see [1]). The second class is based on a hybrid Nested Dissection and minimum degree technique (referred to as ND). These hybrid orderings were generated using ONMETIS [26] or a combination of the graph partitioning tool SCOTCH [29] with a variant of AMD (Halo-AMD, see [30]). For matrices available in both assembled and unassembled format, we used nested dissection based orderings provided by Det Norske Veritas and denote these by MFR. Note that, in this paper, it is not our intention to compare the packages that we used to obtain the orderings; we will only discuss the influence of the type of ordering on the performance of `MUMPS` (in Section 5).

The AMD ordering algorithms are tightly integrated within the `MUMPS` code; the other orderings are passed to `MUMPS` as an externally computed ordering. Because of this tight integration, we observe in Table 3 that the analysis time is smaller using AMD than some

Real Unsymmetric Assembled (RUA)			
Matrix name	Order	No. of entries	Origin
MIXING-TANK	29957	1995041	Polyflow S.A.
INV-EXTRUSION-1	30412	1793881	Polyflow S.A.
BBMAT	38744	1771722	Rutherford-Boeing (CFD)
Real Symmetric Assembled (RSA)			
Matrix name	Order	No. of entries	Origin
OILPAN	73752	1835470	INPRO
B5TUER	162610	4036144	INPRO
CRANKSEG_1	52804	5333507	MacNeal-Schwendler
CRANKSEG_2	63838	7106348	MacNeal-Schwendler
BMW7ST_1	141347	3740507	MacNeal-Schwendler
BMWCRA_1	148770	5396386	MacNeal-Schwendler
BMW3_2	227362	5757996	MacNeal-Schwendler
M_T1.RSA	97578	4925574	Det Norske Veritas
SHIP_001.RSA	34920	2339575	Det Norske Veritas
SHIP_003.RSA	121728	4103881	Det Norske Veritas
SHIPSEC1.RSA	140874	3977139	Det Norske Veritas
SHIPSEC5.RSA	179860	5146478	Det Norske Veritas
SHIPSEC8.RSA	114919	3384159	Det Norske Veritas
THREAD.RSA	29736	2249892	Det Norske Veritas
X104.RSA	108384	5138004	Det Norske Veritas

Table 2: Assembled test matrices from PARASOL partners (except the matrix BBMAT).

AMD ordering					
Matrix	Entries in factors ($\times 10^6$)		Flops ($\times 10^9$)		Time for analysis (seconds)
	estim.	actual	estim.	actual	
MIXING-TANK	38.5	39.1	64.1	64.4	4.9
INV-EXTRUSION-1	30.3	31.2	34.3	35.8	4.6
BBMAT	46.0	46.2	41.3	41.6	8.1
ND ordering					
Matrix	Entries in factors ($\times 10^6$)		Flops ($\times 10^9$)		Time for analysis (seconds)
	estim.	actual	estim.	actual	
MIXING-TANK	18.9	19.6	13.0	13.2	12.8
INV-EXTRUSION-1	15.7	16.1	7.7	8.1	14.0
BBMAT	35.7	35.8	25.5	25.7	11.3

Table 3: Statistics for unsymmetric test problems on the IBM SP2.

user-defined precomputed ordering (in this paper ND or MFR orderings). In addition, the cost of computing the external ordering is not included in these tables.

	AMD ordering			ND ordering	
Matrix	Entries in factors ($\times 10^6$)	Flops ($\times 10^9$)	Time for analysis (seconds)	Entries in factors ($\times 10^6$)	Flops ($\times 10^9$)
OILPAN	10	4	4	10	3
B5TUER	26	13	15	24	12
CRANKSEG_1	40	50	10	32	30
CRANKSEG_2	61	102	14	41	42
BMW7ST_1	27	15	10	25	11
BMWCRA_1	97	128	13	70	61
BMW3_2	51	45	15	45	29

Table 4: Statistics for symmetric test problems on the IBM SP2.

Matrix	Entries in factors ($\times 10^6$)	Flops ($\times 10^9$)
M_T1	29	17
SHIP_003	57	73
SHIPSEC1	37	32
SHIPSEC5	51	52
SHIPSEC8	34	34
THREAD	24	39
X104	24	10

Table 5: Statistics for symmetric test problems, available in both assembled (RSA) and unassembled (RSE) formats (MFR ordering).

4 Parallel implementation issues

In this paper, we assume a one-to-one mapping between processes and processors in our distributed memory environment. A process will thus implicitly refer to a unique processor and, when we say for example that a task is allocated to a process, we mean that the task is also mapped onto the corresponding processor.

As we did before in a shared memory environment [4], we exploit parallelism both arising from sparsity (tree parallelism) and from dense factorizations kernels (node parallelism). To avoid the limitations due to centralized scheduling, where a host process is in charge of scheduling the work of the other processes, we have chosen a distributed scheduling strategy. In our implementation, a pool of work tasks is distributed among the processes that participate in the numerical factorization. A host process is still used to perform the analysis phase (and identify the pool of work tasks), distribute the right-hand side vector, and collect the solution. Our implementation allows this host process to participate in the computations during the factorization and solution phases. This allows the user to run the code on a single processor and avoids one processor being idle during the factorization and solution phases.

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

1. Analysis. The host performs 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 also computes a mapping of the nodes of the assembly tree to the processors. The mapping is such that it keeps communication costs during factorization and solution to a minimum and balances the memory and computation required by the processes. The computational cost is approximated by the number of floating-point operations, assuming no pivoting is performed, and the storage cost by the number of entries in the factors. After computing the mapping, the host sends symbolic information to the other processes. Using this information, each process estimates the work space required for its part of the factorization and solution. The estimated work space should be large enough to handle the computational tasks that were assigned to the process at analysis time plus possible tasks that it may receive dynamically during the factorization, assuming that no excessive amount of unexpected fill-in occurs due to numerical pivoting.

2. Factorization. The original matrix is first preprocessed (for example, converted to arrowhead format if the matrix is assembled) and distributed to the processes that will participate in the numerical factorization. Each process allocates an array 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. The factors must be kept for the solution phase.

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.

4.1 Sources of parallelism

We consider the condensed assembly tree of Figure 2, where the leaves represent subtrees of the assembly tree.

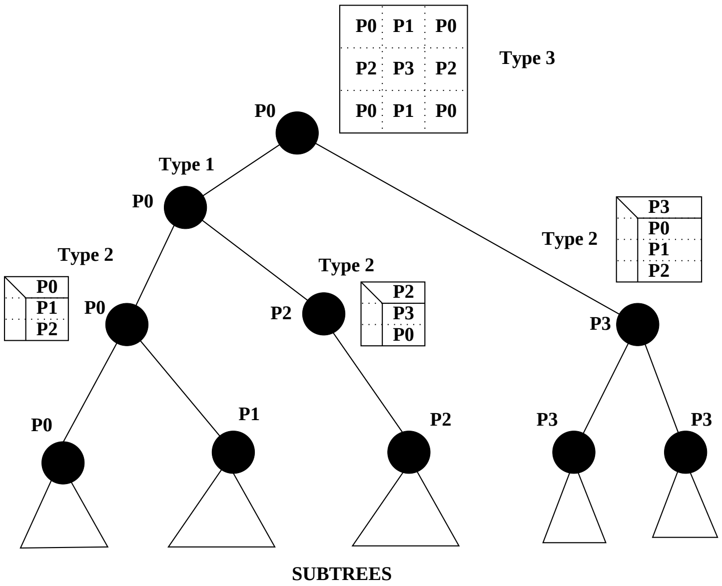

Figure 2: Distribution of the computations of a multifrontal assembly tree over the four processors P0, P1, P2, and P3.

If we only consider tree parallelism, then the transfer of the contribution block from a node in the assembly tree to its father node requires only local data movement when the nodes are assigned to the same process. Communication is required when the nodes are assigned to different processes. To reduce the amount of communication during the factorization and solution phases, the mapping computed during the analysis phase assigns a subtree of the assembly tree to a single process. In general, the mapping algorithm chooses more leaf subtrees than there are processes and, by mapping these subtrees carefully onto the processes, we achieve a good overall load balance of the computation at the bottom of the tree. We have described this in more detail in [5]. However, if we exploit only tree parallelism, the speedups are very disappointing. Obviously it depends on the problem, but typically the maximum speedup is no more than 3 to 5 as illustrated in Table 6. This poor performance is caused by the fact that the tree parallelism decreases while going towards the root of the tree. Moreover, it has been observed (see for example [4]) that often more than 75% of the computations are performed in the top three levels of the assembly tree. It is thus necessary to obtain further parallelism within the large nodes near the root of the tree. The additional parallelism will be based on parallel blocked versions of the algorithms used during the factorization of the frontal matrices.

Nodes of the assembly tree that are treated by only one process will be referred to as nodes of **type 1** and the parallelism of the assembly tree will be referred to as **type 1 parallelism**. Further parallelism is obtained by a one-dimensional (1D) block partitioning of the rows of the frontal matrix for nodes with a large contribution block (see Figure 2). Such nodes will be referred to as nodes of **type 2** and the corresponding parallelism as **type 2 parallelism**. Finally, if the frontal matrix of the root node is large enough, we partition it in a two-dimensional

(2D) block cyclic way. The parallel root node will be referred to as a node of **type 3** and the corresponding parallelism as **type 3 parallelism**.

4.2 Type 2 parallelism

During the analysis phase, a node is determined to be of type 2 if the number of rows in its contribution block is sufficiently large. If a node is of type 2, one process (called the **master**) holds all the fully summed rows and performs the pivoting and the factorization on this block while other processes (called **slaves**) perform the updates on the partly summed rows (see Figure 1).

The slaves are determined dynamically during factorization and any process may be selected. To be able to assemble the original matrix entries quickly into the frontal matrix of a type 2 node, we duplicate the corresponding original matrix entries (stored as arrowheads or element matrices) onto all the processes before the factorization. This way, the master and slave processes of a type 2 node have immediate access to the entries that need to be assembled in the local part of the frontal matrix. This duplication of original data enables efficient dynamic scheduling of computational tasks, but requires some extra storage. This is studied in more detail in Section 8. (Note that for a type 1 node, the original matrix entries need only be present on the process handling this node.)

At execution time, the master of a type 2 node first receives symbolic information describing the structure of the contribution blocks of its son nodes in the tree. This information is sent by the (master) processes handling the sons. Based on this information, the master determines the exact structure of its frontal matrix and decides which slave processes will participate in the factorization of the node. It then sends information to the processes handling the sons to enable them to send the entries in their contribution blocks directly to the appropriate processes involved in the type 2 node. The assemblies for this node are subsequently performed in parallel. The master and slave processes then perform the elimination operations on the frontal matrix in parallel. Macro-pipelining based on a blocked factorization of the fully summed rows is used to overlap communication with computation. The efficiency of the algorithm thus depends on both the block size used to factor the fully summed rows and on the number of rows allocated to a slave process. Further details and differences between the implementations for symmetric and unsymmetric matrices are described in [5].

4.3 Type 3 parallelism

At the root node, we must factorize a dense matrix and we can use standard codes for this. For scalability reasons, we use a 2D block cyclic distribution of the root node and we use ScaLAPACK [9] or the vendor equivalent implementation (routine PDGETRF for general matrices and routine PDPOTRF for symmetric positive definite matrices) for the actual factorization.

Currently, a maximum of one root node, chosen during the analysis, is processed in parallel. The node chosen will be the largest root provided its size is larger than a computer dependent parameter (otherwise it is factorized on only one processor). One process (also called the master) holds all the indices describing the structure of the root frontal matrix.

We call the root node, as determined by the analysis phase, the **estimated root node**. Before factorization, the structure of the frontal matrix of the estimated root node is statically mapped onto a 2D grid of processes. This mapping fully determines to which process an entry of the estimated root node is assigned. Hence, for the assembly of original matrix entries and contribution blocks, the processes holding this information can easily compute exactly the processes to which they must send data to.

In the factorization phase, the original matrix entries and the part of the contribution blocks from the sons corresponding to the estimated root can be assembled as soon as they are available. The master of the root node then collects the index information for all the delayed variables (due to numerical pivoting) of its sons and builds the final structure of the root frontal matrix. This symbolic information is broadcast to all processes that participate in the factorization. The contributions corresponding to delayed variables are then sent by the sons to the appropriate processes in the 2D grid for assembly (or the contributions can be directly assembled locally if the destination is the same process). Note that, because of the requirements of ScaLAPACK, local copying of the root node is required since the leading dimension will change if there are any delayed pivots.

4.4 Parallel triangular solution

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

5 Basic performance and influence of ordering

From earlier studies (for example [25]), we know that the ordering may seriously impact both the uniprocessor time and the parallel behaviour of the method. To illustrate this, we report in Table 6 performance obtained using *only* type 1 parallelism. The results show that using only type 1 parallelism does not produce good speedups. The results also show (see columns "Speedup") that we usually get better parallelism with nested dissection based orderings than with minimum degree based orderings. We thus gain by using nested dissection because of both a reduction in the number of floating-point operations (see Tables 3 and 4) and a better balanced assembly tree.

We now discuss the performance obtained with `MUMPS` on matrices in assembled format that will be used as a reference for this paper. The performance obtained on matrices provided in elemental format is discussed in Section 6. In Tables 7 and 8, we show the performance of `MUMPS` using nested dissection and minimum degree orderings on the IBM SP2 and the SGI Origin 2000, respectively. Note that speedups are difficult to compute on the IBM SP2 because memory paging often occurs on a small number of processors. Hence, the better performance with nested dissection orderings on a small number of processors of the IBM SP2 is due, in part, to the reduction in the memory required by each processor (since there are less entries in the factors). To get a better idea of the true algorithmic speedups (without memory paging effects), we give, in Table 7, the uniprocessor CPU time for one processor, instead of the elapsed time.

Matrix	Time		Speedup	
	AMD	ND	AMD	ND
OILPAN	12.6	7.3	2.91	4.45
BMW7ST_1	55.6	21.3	2.55	4.87
BBMAT	78.4	49.4	4.08	4.00
B5TUER	33.4	25.5	3.47	4.22

Table 6: Influence of the ordering on the time (in seconds) and speedup for the factorization phase, using *only* type 1 parallelism, on 32 processors of the IBM SP2.

When the memory was not large enough to run on one processor, an estimate of the Megaflop rate was used to compute the uniprocessor CPU time. (This estimate was also used, when necessary, to compute the speedups in Table 6.) On a small number of processors, there can still be a memory paging effect that may significantly increase the elapsed time. However, the speedup over the elapsed time on one processor (not given) can be considerable.

Matrix	Ordering	Number of processors					
		$1^{(*)}$	4	8	16	24	32
OILPAN	AMD	37	13.6	9.0	6.8	5.9	5.8
	ND	33	10.8	7.1	5.7	4.6	4.6
B5TUER	AMD	116	155.5	24.1	16.8	16.1	13.1
	ND	108	55.7	21.6	16.8	14.7	10.5
CRANKSEG_1	AMD	456		508.3	162.4	78.4	63.3
	ND	270	228.2	102.0	42.4	39.1	31.9
CRANKSEG_2	AMD	926	-	-	819.6	308.5	179.7
	ND	378	-	316.6	79.7	41.7	35.7
BMW7ST_1	AMD	142	153.4	46.5	21.3	18.4	16.7
	ND	104	105.7	36.7	20.2	12.9	11.7
BMW3_2	AMD	421	-	309.8	74.2	51.0	34.2
	ND	246	-	145.3	42.6	25.8	23.6
MIXING-TANK	AMD	495	-	288.5	70.7	64.5	61.3
	ND	104	32.80	26.1	17.4	14.4	14.8
INV-EXTRUSION-1	AMD	279	-	67.9	63.2	56.5	56.0
	ND	70	25.7	17.5	16.0	13.1	12.4
BBMAT	AMD	320	276.4	68.3	47.8	44.0	39.8
	ND	198	106.4	76.7	35.2	34.6	30.9

Table 7: Impact of the ordering on the time (in seconds) for factorization on the IBM SP2. $^{(*)}$ estimated CPU time on one processor; - means not enough memory.

Table 8 also shows the elapsed time for the solution phase; we observe that the speedups for this phase are quite good.

In the remainder of this paper, we will use nested dissection based orderings, unless stated otherwise.

Factorization phase							
Matrix	Ordering	Number of processors					
		1	2	4	8	16	32
CRANKSEG_2	AMD	566.1	392.2	220.0	115.9	86.4	77.4
	ND	216.9	115.9	72.0	60.3	46.9	38.9
BMW7ST_1	AMD	85.7	56.0	28.2	18.5	15.1	14.2
	ND	63.1	38.5	27.9	19.5	21.1	11.5
BMWCRA_1	AMD	663.0	396.5	238.7	141.6	110.3	76.9
	ND	306.6	182.7	80.9	52.9	41.2	35.5
BMW3_2	AMD	252.7	153.4	81.8	49.4	34.0	27.3
	ND	152.1	93.8	52.5	33.0	22.1	17.0
Solution phase							
Matrix	Ordering	Number of processors					
		1	2	4	8	16	32
CRANKSEG_2	AMD	6.8	5.8	4.4	2.9	2.4	2.3
	ND	4.3	2.7	1.8	1.5	1.1	1.8
BMW7ST_1	AMD	4.2	2.4	2.3	1.9	1.4	1.6
	ND	3.3	2.1	1.7	1.4	1.6	1.5
BMWCRA_1	AMD	11.4	7.2	6.8	3.9	2.8	2.4
	ND	8.3	4.7	2.7	2.1	1.8	2.0
BMW3_2	AMD	6.7	4.1	3.6	2.4	2.1	1.9
	ND	6.3	3.8	2.9	2.4	2.0	2.4

Table 8: Impact of the ordering on the time (in seconds) for factorization and solve phases on the SGI Origin 2000.

6 Elemental input matrix format

In this section, we discuss the main algorithmic changes to handle efficiently problems that are provided in elemental format. We assume that the original matrix can be represented as a sum of element matrices

$$\mathbf{A} = \sum \mathbf{A}_i \ ,$$

where each $\mathbf{A}_i$ has nonzero entries only in those rows and columns that correspond to variables in the ith element. $\mathbf{A}_i$ is usually held as a dense matrix, but if the matrix $\mathbf{A}$ is symmetric, only the lower triangular part of each $\mathbf{A}_i$ is stored.

In a multifrontal approach, element matrices need not be assembled in more than one frontal matrix during the elimination process. This is due to the fact that the frontal matrix structure contains, by definition, all the variables adjacent to any fully summed variable of the front. As a consequence, element matrices need not be split during the assembly process. Note that, for classical fan-in and fan-out approaches [7], this property does not hold since the positions of the element matrices to be assembled are not restricted to fully summed rows and columns.

The main modifications that we had to make to our algorithms for assembled matrices to accommodate unassembled matrices lie in the analysis, the distribution of the matrix, and the assembly process. We will describe them in more detail below.

In the analysis phase, we exploit the elemental format of the matrix to detect supervariables. We define a **supervariable** as a set of variables having the same list of adjacent elements. This is illustrated in Figure 3 where the matrix is composed of two overlapping elements and has three supervariables. (Note that our definition of a supervariable differs from the usual definition, see for example [11]).

Supervariables have been used successfully in a similar context to compress graphs associated with assembled matrices from structural engineering prior to a multiple minimum degree ordering [8]. For assembled matrices, however, it was observed in [1] that the use of supervariables in combination with an Approximate Minimum Degree algorithm was not more efficient.

Matrix	*Graph_size* with supervariable detection	
	OFF	ON
M_T1.RSE	9655992	299194
SHIP_003.RSE	7964306	204324
SHIPSEC1.RSE	7672530	193560
SHIPSEC5.RSE	9933236	256976
SHIPSEC8.RSE	6538480	171428
THREAD.RSE	4440312	397410
X104.RSE	10059240	246950

Table 9: Impact of supervariable detection on the length of the adjacency lists given to the ordering phase.

Table 9 shows the impact of using supervariables on the size of the graph processed by the ordering phase (AMD ordering). *Graph_size* is the length of the adjacency lists of variables/supervariables given as input to the ordering phase. Without supervariable detection,

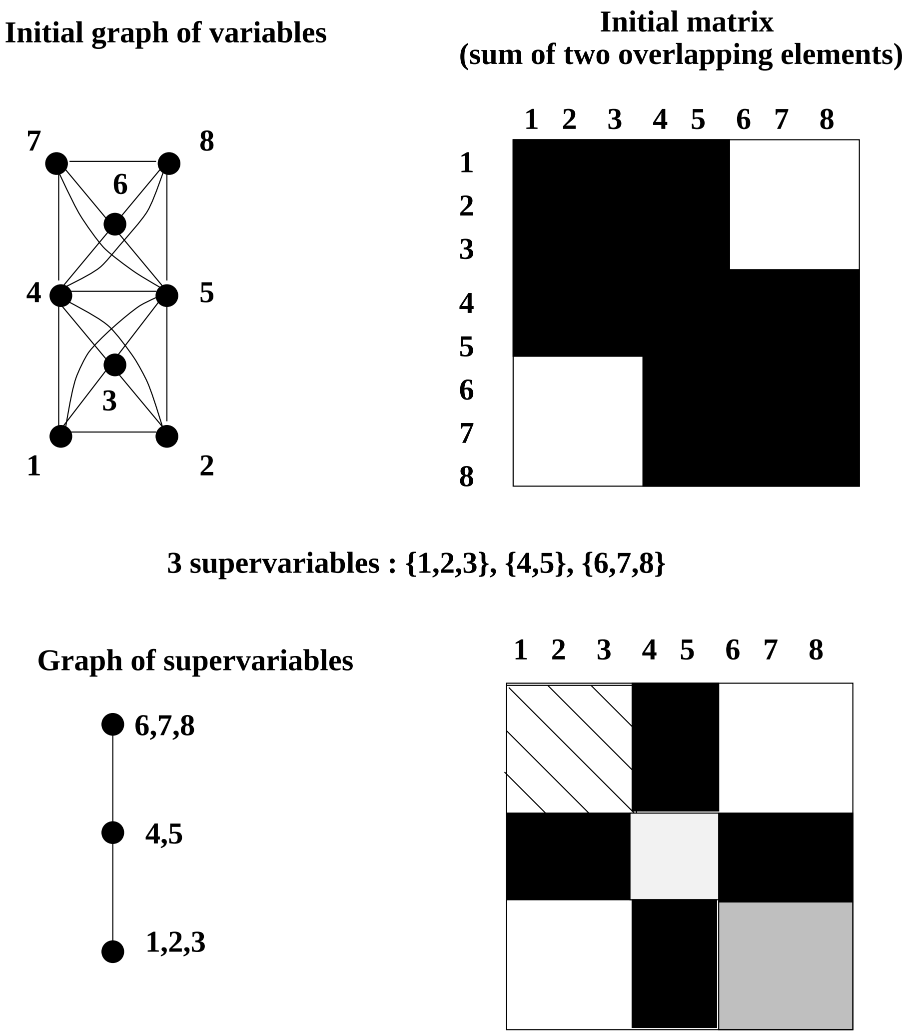

Figure 3: Supervariable detection for matrices in elemental format.

Graph_size is twice the number of off-diagonal entries in the corresponding assembled matrix. The working space required by the analysis phase using the AMD ordering is dominated by the space required by the ordering phase and is *Graph_size* plus an overhead that is a small multiple of the order of matrix. Since the ordering is performed on a single processor, the space required to compute the ordering is the most memory intensive part of the analysis phase. With supervariable detection, the complete uncompressed graph need not be built since the ordering phase can operate directly on the compressed graph. Table 9 shows that, on large graphs, compression can reduce the memory requirements of the analysis phase dramatically.

Table 10 shows the impact of using supervariables on the time for the complete analysis phase (including graph compression and ordering). We see that the reduction in time is not only due to the reduced time for ordering; significantly less time is also needed for building the much smaller adjacency graph of the supervariables.

Matrix	Time for analysis supervariable detection	
	OFF	ON
M_T1.RSE	4.6 (1.8)	1.5 (0.3)
SHIP_003.RSE	7.4 (2.8)	3.2 (0.7)
SHIPSEC1.RSE	6.0 (2.2)	2.6 (0.6)
SHIPSEC5.RSE	10.1 (4.6)	3.9 (0.8)
SHIPSEC8.RSE	5.7 (2.0)	2.6 (0.5)
THREAD.RSE	2.6 (0.9)	1.2 (0.2)
X104.RSE	6.4 (3.5)	1.5 (0.3)

Table 10: Impact of supervariable detection on the time (in seconds) for the analysis phase on the SGI Origin 2000. The time spent in the AMD ordering is in parentheses.

The overall time spent in the assembly process for matrices in elemental format will differ from the overall time spent in the assembly process for the equivalent assembled matrix. Obviously, for the matrices in elemental format there is often significantly more data to assemble (usually about twice the number of entries as for the same matrix in assembled format). However, the assembly process of matrices in elemental format should be performed more efficiently than the assembly process of assembled matrices. First, because we potentially assemble at once a larger and more regular structure (a full matrix). Second, because most input data will be assembled at or near leaf nodes in the assembly tree. This has two consequences. The assemblies are performed in a more distributed way and most assemblies of original element matrices are done at type 1 nodes. (Hence, less duplication of original matrix data is necessary.) A more detailed analysis of the duplication issues linked to matrices in elemental format will be addressed in Section 8. In our experiments (not shown here), we have observed that, despite the differences in the assembly process, the performance of `MUMPS` for assembled and unassembled problems is very similar, provided the same ordering is used. The reason for this is that the extra amount of assemblies of original data for unassembled problems is relatively small compared to the total number of flops.

The experimental results in Tables 11 and 12, obtained on the SGI Origin 2000, show the good scalability of the code for both the factorization and the solution phases on our set of unassembled matrices.

Matrix	Number of processors				
	1	2	4	8	16
M_T1.RSE	92	56	30	18	17
SHIP_003.RSE	392	242	156	120	92
SHIPSEC1.RSE	174	128	65	36	27
SHIPSEC5.RSE	281	176	114	63	43
SHIPSEC8.RSE	187	127	68	36	30
THREAD.RSE	186	120	69	46	37
X104.RSE	56	34	20	16	16

Table 11: Time (in seconds) for factorization of the unassembled matrices on the SGI Origin 2000. MFR ordering is used.

Matrix	Number of processors				
	1	2	4	8	16
M_T1.RSE	3.5	2.1	1.1	1.2	0.8
SHIP_003.RSE	6.9	3.6	3.3	2.5	2.0
SHIPSEC1.RSE	3.8	3.1	2.1	1.6	1.5
SHIPSEC5.RSE	5.5	4.2	2.9	2.2	1.9
SHIPSEC8.RSE	3.8	3.1	2.0	1.4	1.3
THREAD.RSE	2.3	1.9	1.3	1.0	0.8
X104.RSE	2.6	1.9	1.4	1.0	1.1

Table 12: Time (in seconds) for the solution phase of the unassembled matrices on the SGI Origin 2000. MFR ordering is used.

7 Distributed assembled matrix

The distribution of the input matrix over the available processors is the main preprocessing step in the numerical factorization phase. During this step, the input matrix is organized into arrowhead format and distributed according to the mapping provided by the analysis phase. In the symmetric case, the first arrowhead of each frontal matrix is also sorted to enable efficient assembly [5]. If the assembled matrix is initially held centrally on the host, we have observed that the time to distribute the real entries of the original matrix can sometimes be comparable to the time to perform the actual factorization. For example, for matrix OILPAN, the time to distribute the input matrix on 16 processors of the IBM SP2 is on average 6 seconds whereas the time to factorize the matrix is 6.8 seconds (using AMD ordering, see Table 7). Clearly, for larger problems where more arithmetic is required for the actual factorization, the time for factorization will dominate the time for redistribution.

With a distributed input matrix format we can expect to reduce the time for the redistribution phase because we can parallelize the reformatting and sorting tasks, and we can use asynchronous all-to-all (instead of one-to-all) communications. 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. Thus, to improve both the memory and the time scalability of our approach, we should allow the input matrix to be distributed.

Based on the static mapping of the tasks to processes that is computed during the analyis phase, one can *a priori* distribute the input data so that no further remapping is required at the beginning of the factorization. This distribution, referred to as the `MUMPS` **mapping**, will limit the communication to duplications of the original matrix corresponding to type 2 nodes (further studied in Section 8).

To show the influence of the initial matrix distribution on the time for redistribution, we compare, in Figure 4, three ways for providing the input matrix:

1. Centralized mapping: the input matrix is held on one process (the host).
2. `MUMPS` mapping: the input matrix is distributed over the processes according to the static mapping that is computed during the analysis phase.
3. Random mapping: the input matrix is uniformly distributed over the processes in a random manner that has no correlation to the mapping computed during the analysis phase.

The figure clearly shows the benefit of using asynchronous all-to-all communications (required by the `MUMPS` and random mappings) compared to using one-to-all communications (for the centralized mapping). It is even more interesting to observe that distributing the input matrix according to the `MUMPS` mapping does not significantly reduce the time for redistribution. We attribute this to the good overlapping of communication with computation (mainly data reformatting and sorting) in our redistribution algorithm.

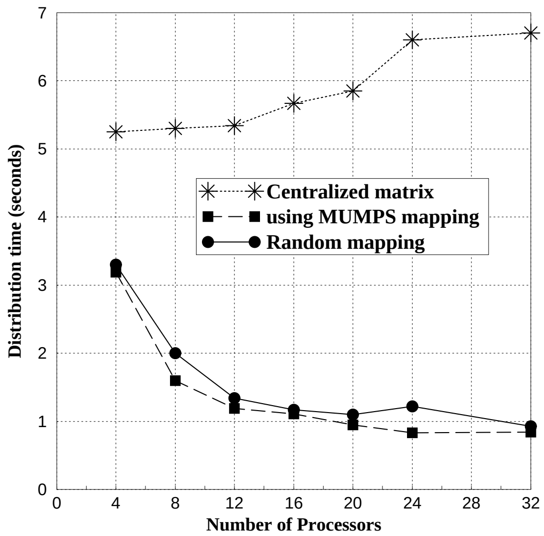

Figure 4: Impact of the initial distribution for matrix OILPAN on the time for redistribution on the IBM SP2.

8 Memory scalability issues

In this section, we study the memory requirements and memory scalability of our algorithms.

Figure 5 illustrates how MUMPS balances the memory load over the processors. The figure shows, for two matrices, the maximum memory required on a processor and the average over all processors, as a function of the number of processors. We observe that, for varying numbers of processors, these values are quite similar.

BMW3_2 BMWCRA_1

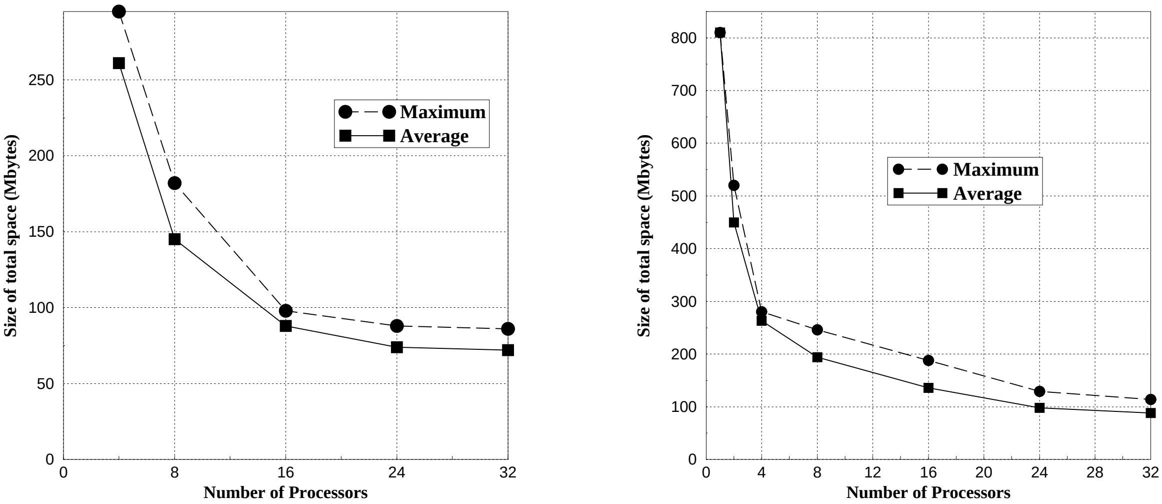

Figure 5: Total memory requirement per processor (maximum and average) during factorization (ND ordering).

Table 13 shows the *average* size per processor of the main components of the working space used during the factorization of the matrix BMW3_2. These components are:

- FACTORS: the space reserved for the factors; a processor does not know after the analysis phase in which type 2 nodes it will participate, and therefore it reserves enough space to be able to participate in all type 2 nodes.
- STACK AREA: the space used for stacking both the contribution blocks and the factors.
- INITIAL MATRIX: the space required to store the initial matrix in arrowhead format.
- COMMUNICATION BUFFERS: the space allocated for both send and receive buffers.
- OTHER: the size of all the remaining workspace allocated per processor.
- TOTAL: the total memory required per processor.

The lines *ideal* in Table 13 are obtained by dividing the memory requirement on one processor by the number of processors. By comparing the actual and ideal numbers, we get an idea how MUMPS scales in terms of memory for some of the components.

Number of processors	1	2	4	8	16	24	32
FACTORS	423	211	107	58	35	31	31
ideal	-	211	106	53	26	18	13
STACK AREA	502	294	172	92	51	39	38
ideal	-	251	126	63	31	21	16
INITIAL MATRIX	69	34.5	17.3	8.9	5.0	4.0	3.5
ideal	-	34.5	17.3	8.6	4.3	2.9	2.2
COMMUNICATION BUFFERS	0	45	34	14	6	6	5
OTHER	20	20	20	20	20	20	20
TOTAL	590	394	243	135	82	69	67
ideal	-	295	147	74	37	25	18

Table 13: Analysis of the memory used during factorization of matrix BMW3_2 (ND ordering). All sizes are in MBytes per processor.

We see that, even if the total memory (sum of all the local workspaces) increases, the average memory required per processor significantly decreases up to 24 processors. We also see that the size for the factors and the stack area are much larger than ideal. Part of this difference is due to parallelism and is unavoidable. Another part, however, is due to an overestimation of the space required. The main reason for this is that the mapping of the type 2 nodes on the processors is not known at analysis and each processor can potentially participate in the elimination of any type 2 node. Therefore, each processor allocates enough space to be able to participate in all type 2 nodes. The working space that is actually used is smaller and, on a large number of processors, we could reduce the estimate for both the factors and the stack area. For example, we have successfully factorized matrix BMW3_2 on 32 processors with a stack area that is 20% smaller than reported in Table 13.

The average working space used by the communication buffers also significantly decreases up to 16 processors. This is mainly due to type 2 node parallelism where contribution blocks are split among processors until a minimum granularity is reached. Therefore, when we increase the number of processors, we decrease (until reaching this minimum granularity) the size of the contribution blocks sent between processors. Note that on larger problems, the average size per processor of the communication buffers will continue to decrease for a larger number of processors. We see, as expected, that the line OTHER does not scale at all since it corresponds to data arrays of size $O(n)$ that need to be allocated on each process. We see that this space significantly affects the difference between TOTAL and *ideal*, especially for larger numbers of processors. However, the relative influence of this fixed size area will be smaller on large matrices from 3D simulations and therefore does not affect the asymptotic scalability of the algorithm.

The imperfect scalability of the initial matrix storage comes from the duplication of the original matrix data that is linked to type 2 nodes in the assembly tree. We will study this in more detail in the remainder of this section. We want to stress, however, that from a user point of view, all numbers reported in this context should be related to the total memory used by the `MUMPS` package which is usually dominated, on large problems, by the size of the stack area.

An alternative to the duplication of data related to type 2 nodes would be to allocate the original data associated with a frontal matrix to only the master process responsible for

Matrix		Number of processors					
		1	2	4	8	12	16
OILPAN	Type 2 nodes	0	4	7	10	17	22
	Total entries	1835	1845	1888	2011	2235	2521
BMW7ST_1	Type 2 nodes	0	4	7	9	13	21
	Total entries	3740	3759	3844	4031	4308	4793
BMW3_2	Type 2 nodes	0	1	3	13	14	21
	Total entries	5758	5767	5832	6239	6548	7120
SHIPSEC1.RSA	Type 2 nodes	0	0	4	11	19	21
	Total entries	3977	3977	4058	4400	4936	5337
SHIPSEC1.RSE	Type 2 nodes	0	1	4	13	19	27
	Total entries	8618	8618	8618	8627	8636	8655
THREAD.RSA	Type 2 nodes	0	3	8	12	23	25
	Total entries	2250	2342	2901	4237	6561	8343
THREAD.RSE	Type 2 nodes	0	2	8	12	15	25
	Total entries	3719	3719	3719	3719	3719	3719

Table 14: The amount of duplication due to type 2 nodes. "Total entries" is the sum of the number of original matrix entries over all processors ($\times 10^3$). The number of type 2 nodes is also given.

the type 2 node. During the assembly process, the master process would then be in charge of redistributing the original data to the slave processes. This strategy introduces extra communication costs during the assembly of a type 2 node and thus has not been chosen. With the approach based on duplication, the master process responsible for a type 2 node has all the flexibility to choose collaborating processes dynamically since this will not involve any data migration of the original matrix. However, the extra cost of this strategy is that, based on the decision during analysis of which nodes will be of type 2, partial duplication of the original matrix must be performed.

In order to keep all the processors busy, we need to have sufficient node parallelism near the root of the assembly tree, `MUMPS` uses a heuristic that increases the number of type 2 nodes with the number of processors used. The influence of the number of processors on the amount of duplication is shown in Table 14. On a representative subset of our test problems, we show the total number of type 2 nodes and the sum over all processes of the number of original matrix entries and duplicates. If there is only one processor, type 2 nodes are not used and no data is duplicated. Figure 6 shows, for four matrices, the number of original matrix entries that are duplicated on all processors, relative to the total number of entries in the original matrix.

Since the original data for unassembled matrices are in general assembled earlier in the assembly tree than the data for the same matrix in assembled format, the number of duplications is often relatively much smaller with unassembled matrices than with assembled matrices. Matrix THREAD.RSE (in elemental format) is an extreme example since, even on 16 processors, type 2 node parallelism does not require any duplication (see Table 14).

To conclude this section, we want to point out that the code scales well in terms of memory usage. On (virtual) shared memory computers, the total memory (sum of local workspaces over all the processors) required by `MUMPS` can sometimes be excessive. Therefore, we are currently investigating how we can reduce the current overestimates of the local stack areas so we can

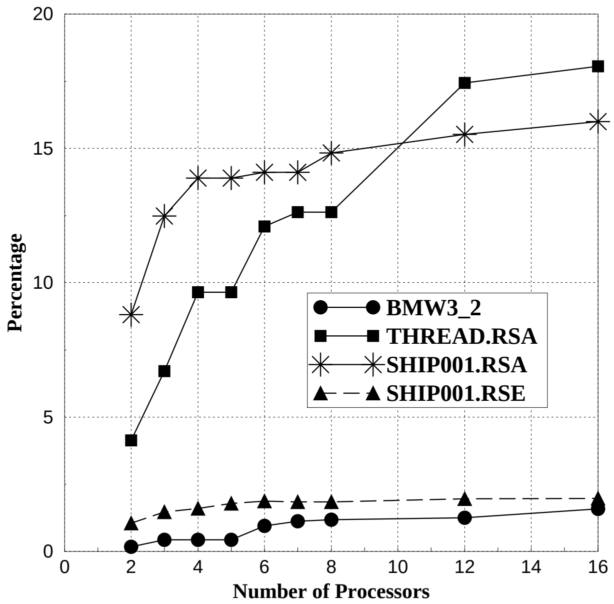

Figure 6: Percentage of entries in the original matrix that are duplicated on all processors due to type 2 nodes.

reduce the total memory required. A possible solution might be to limit the dynamic scheduling of a type 2 node (and corresponding data duplication) to a subset of processors.

9 Dynamic scheduling strategies

To avoid the drawback of centralized scheduling on distributed memory computers, we have implemented distributed dynamic scheduling strategies. We remind the reader that type 1 nodes are statically mapped to processes at analysis time and that only type 2 tasks, which represent a large part of the computations and of the parallelism of the method, are involved in the dynamic scheduling strategy.

To be able to choose dynamically the processes that will collaborate in the processing of a type 2 node, we have designed a two-phase assembly process. Let $Inode$ be a node of type 2 and let $Pmaster$ be the process to which $Inode$ is initially mapped. In the first phase, the (master) processes to which the sons of $Inode$ are mapped, send symbolic data (integer lists) to $Pmaster$. When the structure of the frontal matrix is determined, $Pmaster$ decides a partitioning of the frontal matrix and chooses the slave processes. It is during this phase that $Pmaster$ will collect information concerning the load of the other processors to help in its decision process. The slave processes are informed that a new task has been allocated to them. $Pmaster$ then sends the description of the distribution of the frontal matrix to all collaborative processes of all sons of $Inode$ so that they can send their contribution blocks (real values) in pieces directly to the correct processes involved in the computation of $Inode$. The assembly process is thus fully parallelized and the maximum size of a message sent between processes is reduced (see Section 8).

A pool of tasks private to each process is used to implement dynamic scheduling. All tasks ready to be activated on a given process are stored in the pool of tasks local to the process. Each process executes the following algorithm:

Algorithm 1

while (*not all nodes processed*)
  if *local pool empty* **then**
    blocking receive for a message; process the message
  elseif *message available* **then**
    receive and process message
  else
    extract work from the pool, and process it
  endif
end while

Note that the algorithm gives priority to message reception. The main reasons for this choice are first that the message received might be a source of additional work and parallelism and second, the sending process might be blocked because its send buffer is full (see [5]). In the actual implementation, we use the routine **MPI_IPROBE** to check whether a message is available.

We have implemented two scheduling strategies. In the first strategy, referred to as **cyclic scheduling**, the master of a type 2 node does not take into account the load on the other processors and performs a simple cyclic mapping of the tasks to the processors. In the second strategy, referred to as **(dynamic) flops-based scheduling**, the master process uses information on the load of the other processors to allocate type 2 tasks to the least loaded processors. The load of a processor is defined here as the amount of work (flops) associated with all the active or ready-to-be-activated tasks. Each process is in charge of maintaining local information associated with its current load. With a simple remote memory access procedure, using for example the one-sided communication routine **MPI_GET** included in MPI-2, each process has access to the load of all other processors when necessary. However, MPI-2 is not available on our target computers. To overcome this, we have designed a module based only on symmetric communication tools (MPI asynchronous send and receive). Each process is in charge of both updating and broadcasting its local load. To control the frequency of these broadcasts, an updated load is broadcast only if it is significantly different from the last load broadcast.

When the initial static mapping does not balance the work well, we can expect that the dynamic flops-based scheduling will improve the performance with respect to cyclic scheduling. Tables 15 and 16 show that significant performance gains can be obtained by using dynamic flops-based scheduling. On more than 24 processors, the gains are less significant because our test problems are too small to keep all the processors busy and thus lessen the benefits of a good dynamic scheduling algorithm. We also expect that this feature will improve the behaviour of the parallel algorithm on a multi-user distributed memory computer.

Another possible use of dynamic scheduling is to improve the memory usage. We have seen, in Section 8, that the size of the stack area is overestimated. Dynamic scheduling based on

Matrix & scheduling	Number of processors 16	20	24	28	32
CRANKSEG_2					
cyclic	79.1	47.9	40.7	41.3	38.9
flops-based	61.1	45.6	41.9	41.7	40.4
BMW3_2					
cyclic	52.4	31.8	26.2	29.2	23.0
flops-based	29.4	27.8	25.1	25.3	22.6

Table 15: Comparison of cyclic and flops-based schedulings. Time (in seconds) for factorization on the IBM SP2 (ND ordering).

Matrix & scheduling	Number of processors 4	8	16
SHIP_003.RSE			
cyclic	156.1	119.9	91.9
flops-based	140.3	110.2	83.8
SHIPSEC5.RSE			
cyclic	113.5	63.1	42.8
flops-based	99.9	61.3	37.0
SHIPSEC8.RSE			
cyclic	68.3	36.3	29.9
flops-based	65.0	35.0	25.1

Table 16: Comparison of cyclic and flops-based schedulings. Time (in seconds) for factorization on the SGI Origin 2000 (MFR ordering).

memory load, instead of computational load, could be used to address this issue. Type 2 tasks can be mapped to the least loaded processor (in terms of memory used in the stack area). The memory estimation of the size of the stack area can then be based on a static mapping of the type 2 tasks.

10 Splitting nodes of the assembly tree

During the processing of a parallel type 2 node, both in the symmetric and the unsymmetric case, the factorization of the pivot rows is performed by a single processor. Other processors can then help in the update of the rows of the contribution block using a 1D decomposition (as presented in Section 4). The elimination of the fully summed rows can represent a potential bottleneck for scalability, especially for frontal matrices with a large fully summed block near the root of the tree, where type 1 parallelism is limited. To overcome this problem, we subdivide nodes with large fully summed blocks, as illustrated in Figure 7.

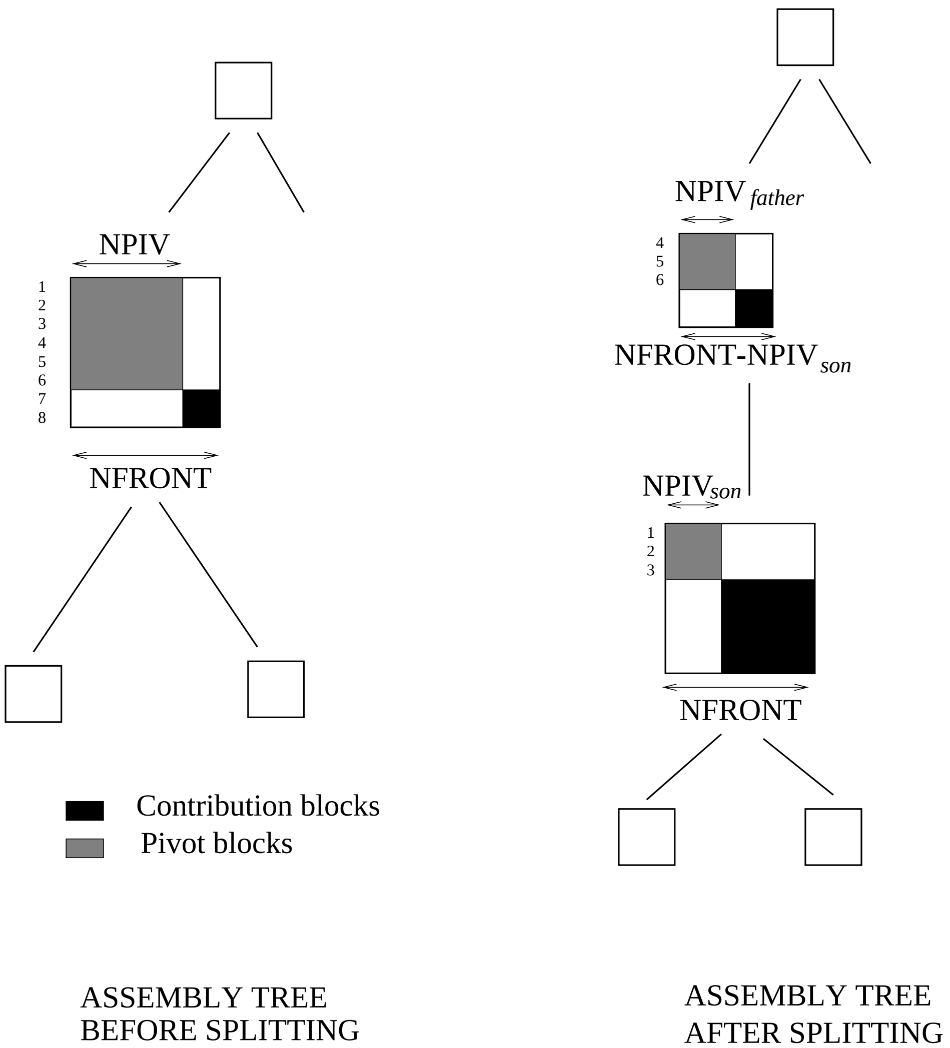

Figure 7: Tree before and after the subdivision of a frontal matrix with a large pivot block.

In this figure, we consider an initial node of size NFRONT with NPIV pivots. We replace this node by a son node of size NFRONT with NPIV_{son} pivots, and a father node of size $\mathrm{NFRONT}-\mathrm{NPIV}_{son}$, with $\mathrm{NPIV}_{father} = \mathrm{NPIV}-\mathrm{NPIV}_{son}$ pivots. Note that by splitting a node, we increase the number of operations for factorization, because we add assembly operations. Nevertheless, we expect to benefit from splitting because we increase parallelism.

We experimented with a simple algorithm that postprocesses the tree after the symbolic factorization. The algorithm considers only nodes near the root of the tree. Splitting large nodes far from the root, where sufficient tree parallelism can already be exploited, would only lead to additional assembly and communication costs. A node is considered for splitting only if its distance to the root, that is, the number edges between the root and the node, is not more than $d_{max} = \log_2(\mathrm{NPROCS} - 1)$.

Let $Inode$ be a node in the tree, and $d(Inode)$ the distance of $Inode$ to the root. For all nodes $Inode$ such that $d(Inode) \leq d_{max}$, we apply the following algorithm.

Algorithm 2 *Splitting of a node*

if NFRONT − NPIV/2 *is large enough* **then**
  1. Compute W_{master} = *number of flops performed by the master of Inode.*
  2. Compute W_{slave} = *number of flops performed by a slave,*
  assuming that NPROCS − 1 *slaves can participate.*
  3. **if** $W_{master} > W_{slave} \cdot (1 + \frac{p \cdot max(1, d(Inode)-1)}{100})$ **then**
    3.1. Split $Inode$ *into nodes* son *and* $father$ *so that* $\mathrm{NPIV}_{son} = \mathrm{NPIV}_{father} = \mathrm{NPIV}/2$.
    3.2. Apply Algorithm 2 recursively to nodes son *and* $father$.
  endif
endif

Algorithm 2 is applied to a node only when NFRONT – NPIV/2 is large enough because we want to make sure that the son of the split node is of type 2. (The size of the contribution block of the son will be NFRONT – NPIV_{son}.) A node is split only when the amount of work for the master (W_{master}) is large relative to the amount of work for a slave (W_{slave}). To reduce the amount of splitting further away from the root, we add, at step 3 of the algorithm, a relative factor to W_{slave}. This factor depends on a machine dependent parameter p, $p > 0$, and increases with the distance of the node from the root. Parameter p allows us to control the general amount of splitting. Finally, because the algorithm is recursive, we may divide the initial node into more than two new nodes.

The effect of splitting is illustrated in Table 17 on both the symmetric matrix CRANKSEG_2 and the unsymmetric matrix INV-EXTRUSION-1. *Ncut* corresponds to the number of type 2 nodes cut. A value $p = 0$ is used as a flag to indicate no splitting. Flops-based dynamic scheduling is used for all runs in this section. The best time obtained for a given number of processors is indicated in bold font. We see that significant performance improvements (of up to 40% reduction in time) can be obtained by using node splitting. The best timings are generally obtained for relatively large values of p. More splitting occurs for smaller values of p, but the corresponding times do not change much.

CRANKSEG_2						
p		Number of processors				
		16	20	24	28	32
0	Time	61.1	45.6	41.9	41.7	40.4
	Ncut	0	0	0	0	0
200	Time	37.9	31.4	30.4	29.5	**25.4**
	Ncut	6	7	9	9	12
150	Time	41.8	**31.3**	31.0	28.9	27.2
	Ncut	7	9	10	12	13
100	Time	39.8	32.3	**28.4**	**28.6**	26.7
	Ncut	9	11	13	14	15
50	Time	**36.7**	33.6	31.4	29.6	27.4
	Ncut	12	13	16	17	21
10	Time	40.8	32.5	29.5	29.8	26.0
	Ncut	16	17	21	28	32

INV-EXTRUSION-1						
p		Number of processors				
		4	8	16	24	32
0	Time	25.9	16.7	14.6	13.5	14.6
	Ncut	0	0	0	0	0
200	Time	25.5	16.7	**13.4**	**12.1**	**12.4**
	Ncut	0	1	3	6	12
150	Time	**24.9**	16.3	13.5	13.4	12.4
	Ncut	1	1	4	11	9
100	Time	24.9	**16.2**	13.7	13.1	13.6
	Ncut	1	2	6	19	24
50	Time	24.9	17.0	13.5	13.6	16.6
	Ncut	1	3	14	25	35
10	Time	24.9	17.5	13.4	14.5	15.8
	Ncut	2	6	17	27	33

Table 17: Time (in seconds) for factorization and number of nodes cut for different values of parameter p on the IBM SP2. Nested dissection ordering and flops-based dynamic scheduling are used.

11 Summary

Tables 18 and 19 show results obtained with `MUMPS 4.0` using both dynamic scheduling and node splitting. Default values for the parameters controlling the efficiency of the package have been used and therefore the timings do not always correspond to the fastest possible execution time. The comparison with results presented in Tables 7, 8, and 11 summarizes well the benefits coming from the work presented in Sections 9 and 10.

Matrix	Number of processors					
	$1^{(*)}$	4	8	16	24	32
OILPAN	33	11.1	7.5	5.2	4.8	4.6
B5TUER	108	82.1	51.9	13.4	13.1	10.5
CRANKSEG_1	270	185.3	92.4	27.3	25.6	20.9
CRANKSEG_2	378	-	-	41.8	31.0	27.2
BMW7ST_1	104	-	29.8	13.7	11.7	11.3
BMW3_2	246	-	-	24.1	24.0	20.4
MIXING-TANK	104	30.8	21.6	16.4	14.7	14.8
INV-EXTRUSION-1	70	24.9	16.3	13.5	13.4	12.4
BBMAT	198	255.4	85.2	34.8	32.8	30.9

Table 18: Time (in seconds) for factorization using `MUMPS 4.0` with default options on IBM SP2. ND ordering is used. $^{(*)}$: uniprocessor CPU or estimated CPU time; - means swapping or not enough memory.

Matrix	Number of processors				
	1	2	4	8	16
CRANKSEG_2	217	112	66	46	29
BMW7ST_1	62	36	25	12	10
BMWCRA_1	307	178	82	58	36
BMW3_2	151	96	53	33	18
M_T1.RSE	92	56	31	19	13
SHIP_003.RSE	392	237	124	108	51
SHIPSEC1.RSE	174	125	63	39	25
SHIPSEC5.RSE	281	181	103	62	37
SHIPSEC8.RSE	187	119	64	35	27
THREAD.RSE	186	125	70	38	24
X104.RSE	56	34	19	12	11

Table 19: Time (in seconds) for factorization using `MUMPS 4.0` with default options on SGI Origin 2000. ND or MFR ordering is used.

The largest problem we have solved to date is a symmetric matrix of order 943695 with more than 39 million entries. The number of entries in the factors is 1.4×10^9 and the number of operations during factorization is 5.9×10^{12}. On one processor of the SGI Origin 2000, the factorization phase required 8.9 hours and on two (non-dedicated) processors 6.2 hours were required. Because of the total amount of memory estimated and reserved by `MUMPS`, we could not solve it on more than 2 processors. This issue will have to be addressed to improve the scalability on globally addressable memory computers and further analysis will be performed on purely distributed memory computers with a larger number of processors. Possible solutions to this have been mentioned in the paper (limited dynamic scheduling and/or memory based dynamic scheduling) and will be developed in the future.

Acknowledgements

We are grateful to Jennifer Scott and John Reid for their comments on an early version of this paper.

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, M. J. Daydé, I. S. Duff, and P. Morère. Linear algebra calculations on a virtual shared memory computer. *Int. Journal of High Speed Computing*, 7:21–43, 1995.

[3] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *Int. J. of Supercomputer Applics.*, 3:41–59, 1989.

[4] 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.

[5] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *To appear in special issue of Comput. Methods in Appl. Mech. Eng. on Domain Decomposition and Parallel Computing*, 1999.

[6] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, second edition*. SIAM Press, 1995.

[7] C. Ashcraft. The fan-both family of column-based distributed Cholesky factorisation algorithms. In J. R. Gilbert and J. W. H Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 159–190. Springer-Verlag NY, 1993.

[8] C. Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM J. Scientific Computing*, 16(6):1404–1411, 1995.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, 1997.

[10] T. Blank, R. Lucas, and J. Tiemann. A parallel solution method for large sparse systems of equations. *IEEE Trans. on Comput.*, 6(6):981–991, 1989.

[11] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. Technical Report Berkeley UCB//CSD-95-883, University of California at Berkeley, 1995.

[12] J. W. Demmel and X. S. Li. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of Supercomputing*, Orlando, Florida, November 1998.

[13] J. Dongarra and R. C. Whaley. LAPACK Working Note 94: A Users' Guide to the BLACS v1.0. Technical Report UT-CS-95-281, University of Tennessee, Knoxville, Tennessee, USA, 1995.

[14] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. Algorithm 679. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[15] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. Algorithm 679. A set of Level 3 Basic Linear Algebra Subprograms: model implementation and test programs. *ACM Transactions on Mathematical Software*, 16:18–28, 1990.

[16] 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, Seattle and Report TR/PA/97/36 from CERFACS, Toulouse.

[17] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.

[18] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(4):889–901, 1999.

[19] 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.

[20] 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.

[21] V. Espirat. Développement d'une approche multifrontale pour machines à mémoire distribuée et réseau hétérogène de stations de travail. Technical report, ENSEEIHT-IRIT, 1996. Rapport de stage 3ieme Année.

[22] A. George, M. T. Heath, J. W. H. Liu, and E. G.-Y. Ng. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988.

[23] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. TR 94-063, University of Minnesota, 1994. To appear in *IEEE Trans. on Parallel and Distributed Systems*, 1997.

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

[25] B. Hendrickson and E. Rothberg. Improving the runtime and quality of nested dissection ordering. *SIAM J. Scientific Computing*, 20(2):468–489, 1998.

[26] 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.

[27] PARASOL. Deliverable 2.1c: MUMPS Version 3.1. A MUltifrontal Massively Parallel Solver. Technical report, February 19, 1999.

[28] PARASOL. Deliverable 2.1b: MUMPS Version 2.0. A MUltifrontal Massively Parallel Solver. Technical report, January 10, 1998.

[29] F. Pellegrini. SCOTCH 3.1 User's guide. Technical Report 1137-96, LaBRI, Université Bordeaux I, August 1996.

[30] 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, Puerto Rico*, 1999.