ESPRIT IV
Domain 4: Long Term Research
Task 4.2.: Reactive Long Term Research

PARASOL

An Integrated Programming Environment for Parallel Sparse Matrix Solvers (Project No. 20160)

Deliverable D 2.1d Final report

A Multifrontal Massively Parallel Solver

MUMPS version 4.0

June 30, 1999

CERFACS (F)
ENSEEIHT-IRIT (F)
Rutherford Appleton Laboratory (GB)

Classification:	
Status:	C
Version:	4.0
Distribution:	EU Commission, Reviewers, Project partners

Contents

Abstract

This document describes work performed in Workpackage 2.1 of the ESPRIT project 20160 (PARASOL) carried out at CERFACS (France), ENSEEIHT-IRIT (France), and Rutherford Appleton Laboratory (England).

1 Roadmap for parallel direct solver MUMPS

This deliverable is associated with Version 4.0 of the `MUMPS` code. There follows the complete roadmap for `MUMPS`.

SOLVER NAME: `MUMPS` or `PSL_MUMPS`

SCHEDULE OVERVIEW:

version	description	time schedule
	Alpha version of code in PVM	1/97
1.0	MPI version using only tree parallelism	5/97
2.0	Node and tree parallelism and distributes original matrix and root node. Uses ScaLAPACK at root node. Still unsymmetric assembled. Better data management enabling solution of larger problems. Using PARASOL interface (host-node paradigm)	2/98
2.1	Version for symmetric positive definite matrices	5/98
2.2	with Fortran 90 interface Additional features include: Ability to handle general symmetric matrices. Hybrid host version (including serial code). Basic version of rank estimate and null-space.	9/98
2.2	with PARASOL interface Includes orderings based on METIS and other graph partitioning strategies.	10/98
3.1	Version with element entry. Includes improved strategy for rank estimation and null-space.	1/99
3.2	Capability of handling distributed matrix input.	2/99
4.0	Final version$^{(*)}$ tuned to interface with other PARASOL codes.	4/99

$^{(*)}$ An additional functionality (not present in the original roadmap), for using `MUMPS` within Schur based methods, has been added to `MUMPS 4.0`.

2 Introduction to MUMPS

`MUMPS` ("MUltifrontal Massively Parallel Solver") is a package for solving linear systems of equations $\mathbf{Ax} = \mathbf{b}$, where the matrix $\mathbf{A}$ is either unsymmetric, symmetric positive definite, or general symmetric. `MUMPS` uses a multifrontal technique which is a direct method based on the LU factorization of the matrix. We refer the reader to the papers [3, 30, 31] for full details of this technique.

Several aspects of our algorithm combine to give us an approach which is unique amongst 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 both the independence due to sparsity and the parallelism coming from dense structure processing, and
- the capability of solving a wide range of problems, including symmetric, unsymmetric, and rank deficient systems using either **LU** or $\mathbf{LDL}^{\mathrm{T}}$ factorization.

To address all these factors we have designed a fully asynchronous algorithm based on a multifrontal approach with distributed dynamic scheduling of the tasks. Most currently available versions of distributed memory parallel solvers are based on a static mapping of the tasks and of the data and do not allow either numerical pivoting or task migration during numerical factorization. Assuredly, among the other work on distributed memory sparse direct solvers of which we are aware [13, 16, 23, 35, 37, 38], we do not know of any with the same capabilities as the MUMPS solver. The current version of our package provides a large range of options (assembled, assembled distributed, and elemental input format, determination of null-space basis and rank deficiency, return of Schur complement matrix, and classical pre and postprocessing facilities) to process a large class of test problems (symmetric definite, general symmetric, unsymmetric, and rank deficient matrices).

In the multifrontal method, all elimination operations take place within a dense submatrix, called a *frontal matrix*. The frontal matrix can be partitioned as

$$\begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix}$$

and pivots at this stage in the elimination can be chosen from within the block F_{11} only. The Schur complement $F_{22} - F_{21}F_{11}^{-1}F_{12}$ is computed and used to update later rows and columns of the matrix. We call this update matrix, the *contribution block*.

The overall factorization of the sparse matrix using a multifrontal scheme, can be described by an *assembly tree*, where the nodes correspond to computations of the Schur complement as just described, and the edges represent the transfer of the contribution block which is assembled (or summed) with other contribution blocks and original matrix entries at the parent node in the tree. The original matrix entries are summed as a complete row and column (or a block of rows and columns) 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 columns $\{j_1, j_2, j_3\}$ of row i, $i < j_1, j_2, j_3$, (and also in rows $\{k_1, k_2\}$ of column i, $i < k_1, k_2$), then the arrowhead list associated to variable i is $\{a_{ii}, a_{j_1,i}, a_{j_2,i}, a_{j_3,i}, a_{i,k_1}, a_{i,k_2}\}$. 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*.

In our implementation, the tree is constructed from the symmetrized pattern of the matrix and a given sparsity ordering. By symmetrized pattern, we mean the pattern of a matrix, where a_{ij} is considered present if a_{ij} or a_{ji} is nonzero, $1 \leq i, j \leq n$. That is, the pattern of the matrix $A + A^T$ where the summation is symbolic. Our factorization does, however, allow the matrix to be unsymmetric.

An important aspect of the assembly tree is that operations at nodes which are not ancestors or descendents of each other are independent thus giving the possibility for obtaining parallelism from the tree (so-called *tree parallelism*). For example, work can commence immediately and 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 throughout the later sections of this report. 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 [33].

In the unsymmetric case, threshold pivoting is used to maintain numerical stability so that it is possible that the pivots selected at the analysis phase are unsuitable. In the numerical factorization phase, we are at liberty to choose pivots from anywhere within the pivot block (including off-diagonal pivots) but it still may be impossible to eliminate all variables from this block. The result is that the Schur complement that is passed to the parent node may be larger than anticipated by the analysis phase and so our data structures may be different from those forecast by the analysis. This implies that we need to allow dynamic scheduling during numerical factorization. In the symmetric positive-definite case only static scheduling is required. However, in this present work, we will use dynamic scheduling for symmetric systems because we want to use our code to solve problems that are not positive definite and it provides more flexibility for load balancing.

3 Experimental environment

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

In Tables 1 and 2, we show both symmetric and unsymmetric test problems. 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 [28]. For symmetric matrices, the number of entries does not include the entries in the strictly upper 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 which correspond to variables in the ith element. Because element matrices may overlap, the number of entries of a matrix in elemental format is often larger than for the same matrix when assembled (compare the matrices from Det Norske Veritas in 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	Nb of elements	Nb 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 various test problems: the number of entries in the factors and the number of floating-point operations for elimination (for unsymmetric problems we show both the estimated and the actual number which may differ because of numerical pivoting).

The statistics clearly depend on the ordering used during elimination. Two classes of ordering will be used in the report. 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 a combination of the graph partitioning tool SCOTCH [44] with Halo-AMD (see [45]), or ONMETIS [41]. 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 in the following. Note that in this report, it is not our intention to compare the ordering packages used. We will discuss the influence of the type of orderings on the performance of MUMPS in Section 6.1.

Real Unsymmetric Assembled (RUA)			
Matrix name	Order	Nb 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	Nb of entries	Origin
OILPAN	73752	1835470	INPRO
B5TUER	162610	4036144	INPRO
MH	220542	5494489	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 partner (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 (ordering based on AMD or ND).

The AMD ordering algorithms are tightly integrated within the MUMPS code; the other orderings are passed to MUMPS via an option that allows the user to specify an externally computed ordering. Because of this tight integration, we observe in Table 3 that the analysis time is smaller using AMD than some user-defined precomputed ordering. In addition, the cost of computing the external ordering is not reported 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
MH	28	8	$6^{(*)}$	28	9
BMW7ST_1	27	15	10	25	11
BMW3_2	51	45	15	45	29
BMWCRA_1	97	128	$6^{(*)}$	70	61
CRANKSEG_1	40	50	10	32	30
CRANKSEG_2	61	102	14	41	42

Table 4: Statistics for symmetric test problems on the IBM SP2 (ordering based on AMD or ND). $^{(*)}$ time obtained on the SGI Origin 2000 because of insufficient disk space 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).

Most results presented in this report have been obtained on a 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) 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.

All experiments reported in this report use Version 4.0 of MUMPS. The software is written in Fortran 90. It requires MPI for message passing and makes use of BLAS [26, 27], LAPACK [11], BLACS [25], and ScaLAPACK [15] 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.

4 Description of the main implementation issues

The main features of the MUMPS package include the solution of the transposed system, input of the matrix in assembled format or elemental format, error analysis, iterative refinement, scaling of the original matrix, estimate of rank deficiency and null space basis, and the possibility for the user to input a given ordering.

The software is written in Fortran 90. It requires MPI for message passing and makes use of BLAS [26, 27], LAPACK, BLACS, and ScaLAPACK [15] subroutines. It has been tested on an IBM-SP2, an SGI Power Challenge, and an SGI Origin 2000.

MUMPS exploits both parallelism arising from sparsity and from dense factorizations kernels. The pool of work tasks is distributed among the processors, but an identified (host) processor is required to perform the analysis phase, distribute the incoming matrix to the other (slave) processors, collect the solution, and generally oversee the computation. The code solves the system $\mathbf{Ax} = \mathbf{b}$ in three main steps:

1. Analysis. The host performs an approximate minimum degree algorithm based on the symmetrized pattern $\mathbf{A} + \mathbf{A}^T$, and carries out symbolic factorization. A mapping of the multifrontal computational graph is then computed, and symbolic information is transferred from the host to the other processors. Using this information, the processors estimate the memory necessary for factorization and solution.
2. Factorization. The host sends appropriate entries (or elements) of the original matrix to the other processors that are responsible for the numerical factorization. The numerical factorization on each frontal matrix is conducted by a *master* processor (determined by the analysis phase) and one or more *slave* processors (determined dynamically). Each processor allocates an array for contribution blocks and factors; the factors must be kept for the solution phase.
3. Solution. The right-hand side $\mathbf{b}$ is broadcast from the host to the other processors. These processors compute the solution $\mathbf{x}$ using the (distributed) factors computed during Step 2, and the solution is assembled on the host.

MUMPS allows the host processor to participate in computations during the factorization and solve phases, just like a slave processor. This may lead to memory imbalance since the host already stores the initial matrix, but it also allows us to run MUMPS on a single processor and avoids one processor being idle during the factorization and solve phases.

For both the symmetric and the unsymmetric algorithms used in the code, we have chosen a fully asynchronous approach with dynamic scheduling of the computational tasks. Asynchronous communication is used to enable overlapping between communication and computation. Dynamic scheduling was initially chosen to accommodate numerical pivoting in the factorization. The other important reason for this choice is that, with dynamic scheduling, the algorithm can adapt itself at execution time to remap work and data to more appropriate processors. In fact, we combine the main features of static and dynamic approaches; we use the estimation obtained during analysis to map some of the main computational tasks; the other tasks are dynamically scheduled at execution time. The main data structures (the original matrix and the factors) are similarly partially mapped according to the analysis phase. Part of the initial matrix is replicated to enable rapid task migration without data redistribution.

4.1 Mapping

A mapping of the assembly tree to the processors is performed statically as part of the analysis phase. The main objectives of this phase are to control the communication costs, and to balance the memory used and the computation done by each processor. The computational cost will be approximated by the number of floating-point operations, and only the matrix of the factors will be taken into account when balancing the memory used by the processors.

In this section, we describe the algorithms used to map the assembly tree onto the processors and show how we have combined memory and work balancing criteria.

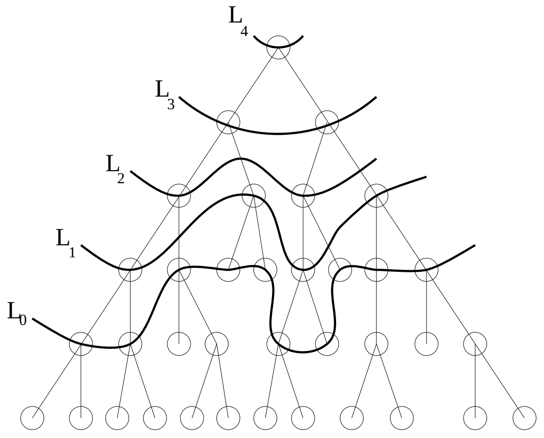


Figure 1: Decomposition of the assembly tree into levels.

The tree is processed from the bottom to the top, level by level (see Figure 1). Level L_0 is determined using the Algorithm 1 [34] and is illustrated in Figure 2. Then for $i > 0$, a node belongs to L_i if all its children belong to L_j, $j < i$. First, nodes of level L_0 (and the subtrees for which they are the root) are mapped. This first step is designed to balance the work in the subtrees and to reduce communication since all nodes in a subtree are mapped onto the same processor. Normally to get a good load balance it is necessary to have many more nodes in level L_0 than there are processors. Thus L_0 depends on the number of processors and a higher number of processors will lead to smaller subtrees.

Algorithm 1 – Construction and mapping of the initial level L_0

```
  Let $L_0 \leftarrow$ Roots of the assembly tree
  Repeat
    Find the node q in $L_0$ whose subtree has largest computational cost
    Set $L_0 \leftarrow (L_0 \backslash \{q\}) \cup \{children\ of\ q\}$ (See Figure 2)
    Cyclic mapping of the nodes of $L_0$ onto the processors.
    Estimate the load imbalance
  Until load imbalance < threshold
```

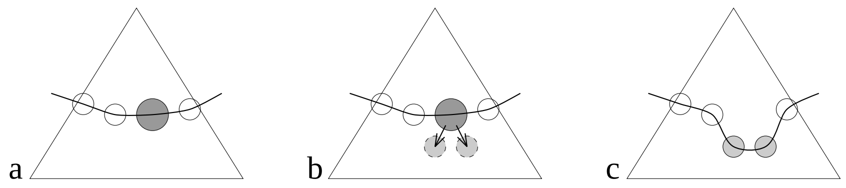


Figure 2: One step in the construction of the first level L_0.

The mapping of higher levels in the tree takes into account only memory balancing issues. For each processor, the memory load (total size of its factors) is first computed for the nodes at level L_0. For each level L_i, $i > 0$, each unmapped node of L_i is mapped to the processor with the smallest memory load and its memory load is revised.

The mapping is then used to explicitly distribute the permuted initial matrix onto the processors and to estimate the amount of work and memory required on each processor.

4.2 Sources of parallelism

We consider the condensed assembly tree of Figure 3, where the leaves are L_0 subtrees of the assembly tree.

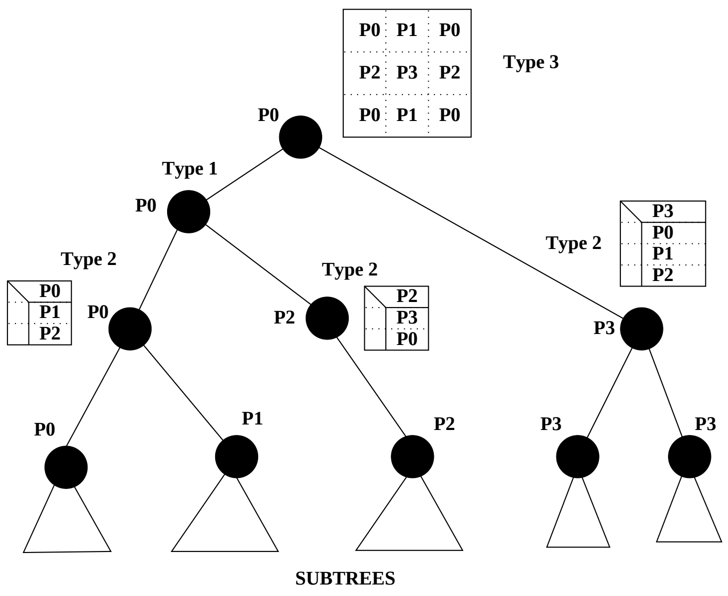


Figure 3: Distribution of the computations of a multifrontal assembly tree.

There will be in general more leaf subtrees than processors, and therefore we can expect a good overall load balance of the computation at the bottom of the tree. However, if we only exploit the tree parallelism, the speed-up is very disappointing. The actual speed-up from this parallelism depends on the problem but is typically only 2 to 4 irrespective of the number of processors. 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 versions of the blocked algorithms used during the factorization of the frontal matrices.

Nodes of the tree processed by only one processor 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 doing a 1D block partitioning of the rows of the frontal matrix for nodes with a large contribution block. Such nodes will be referred to as nodes of *type 2* and the corresponding parallelism as *type 2 parallelism.* Finally, if the root node is large enough, then 2D block cyclic partitioning of the root frontal matrix is performed. The parallel root node will be referred to as a node of *type 3* and the corresponding parallelism as *type 3 parallelism.*

4.2.1 Description of type 2 parallelism

If a node is of type 2, one processor (called the master of the node) holds all the fully summed rows and performs the pivoting and the factorization on this block while other processors (so called slaves) perform the updates on the contribution rows (see Figure 4).

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. During the analysis phase, based on the structure of the assembly tree, a node is determined to be of type 2 if its frontal matrix is sufficiently large. In terms of memory, the mapping algorithm assumes that the master processor holds the fully-summed rows and that any other processors might be selected as slave processes. As a consequence, part of the initial matrix is duplicated onto all the processors to enable efficient dynamic scheduling of computational tasks. At execution time, the master then first receives symbolic information describing the structure of the contribution blocks

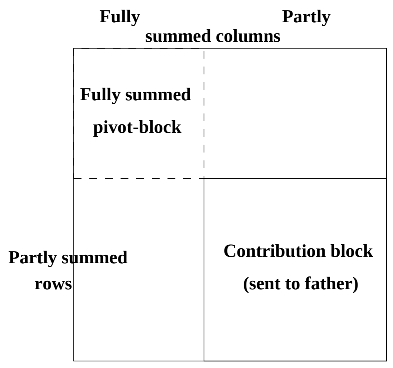


Figure 4: Type 2 nodes: partitioning of frontal matrix.

sent by its children. Based on this information, the master determines the exact structure of its frontal matrix and decides which slave processors will participate in the factorization of the node.

Further details on the implementation of type 2 nodes depends on whether the initial matrix is symmetric or not and will be given in Section 4.4.3.

4.2.2 Description of type 3 parallelism

In order to have good scalability, we perform a 2D block cyclic distribution of the root node. We use ScaLAPACK [15] or the vendor equivalent implementation (PDGETRF for unsymmetric matrices and PDPOTRF for symmetric matrices).

Currently, a maximum of one root node, chosen during the analysis, is processed in parallel. This node is of type 3. The node chosen will be the largest root provided its size is larger than a computer dependent parameter. One processor, the so-called master of the root, holds all indices describing the root frontal matrix.

We define the root node as determined by the analysis phase, the *estimated* root node. Before factorization, the estimated root node frontal matrix is statically mapped onto a 2D grid of processors. We use a static distribution and mapping for those variables known by the analysis to be in the root node so that, for an entry in the estimated root node, we know where to send it and assemble it using functions involving integer divisions, moduli, ...

In the factorization phase, the original matrix entries and the part of the contribution blocks from the children 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 uneliminated variables of its children and builds the structure of the frontal matrix. This symbolic information is broadcast to all participating processors. The contributions corresponding to uneliminated variables can then be sent by the children to the appropriate processors in the 2D grid for assembly, or directly assembled locally if the destination is the same processor. 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 uneliminated variables.

4.3 Parallel implementation issues

To enable automatic overlapping between computation and communication, we have chosen to use fully asynchronous communications. For flexibility and efficiency, explicit buffering in the user space has been implemented. We have developed a Fortran 90 module to send asynchronous messages, based on immediate sends. We define a send buffer for each processor based on information from the analysis phase. When we try to send contribution blocks, factorized blocks, ... we first check to see if there is room in the send buffer. Our module provides an equivalent of MPI_BSEND [24] with the advantage that messages are directly packed in the buffer and problems occurring when the buffer is full are overcome. Note that messages are never sent when the destination is identical to the source; in that case the associated action is performed directly locally, instead of the send.

Estimates of the minimum sizes needed for the send and receive buffers are computed by each processor prior to factorization. This estimation is based on the static mapping of the assembly tree and takes into account the three types of parallelism used during factorization. Note that, for example, using type 2 parallelism will significantly reduce the size of the contribution blocks sent between processors, and thus of the required buffers, as shown in Figure 5. Buffers are allocated on each processor at the beginning of the factorization.

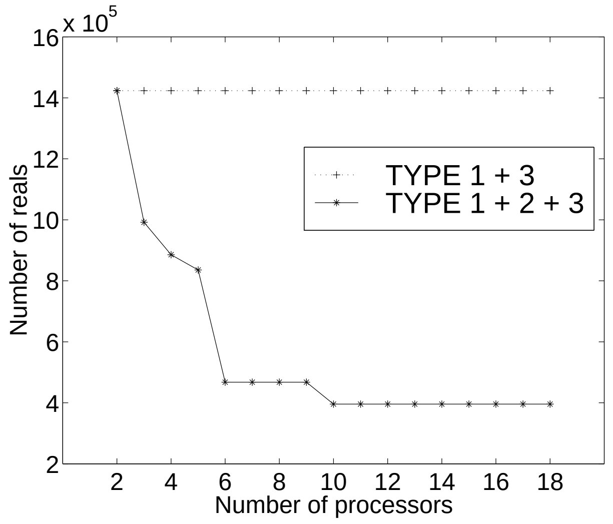


Figure 5: Impact of type 2 parallelism on the size (in number of 64-bit reals) of the send buffer. Test matrix is WANG3.

If there is not enough space to put the message in the buffer, the procedure requesting the send returns with an error code. In such cases, to avoid deadlock, the processor will try to receive messages until space becomes available in its local send buffer. Let us take a simple illustrative example. Processor A has filled-up its buffer doing an asynchronous send of a large message to processor B. Processor B has done the same to processor A. The next messages sent by both processors A and B will then be blocked until the other processor has received the first message. More complicated situations involving more processors can occur, but in all cases the key issue for avoiding deadlock is that each processor tries not to be the blocking processor.

MPI only guarantees that messages are non-overtaking, that is if a processor sends two messages to the same destination, then the receiver will receive them in the same order. For synchronous algorithms the non-overtaking property is often enough to ensure that messages are received in the correct order. With a fully asynchronous algorithm, based on dynamic scheduling of the computational tasks, it can happen that messages arrive "too early". In this case, it is crucial to be sure that the "missing" messages have already been sent so that blocking receives can be performed to process all messages that should have already been processed at this stage of the computation. As a consequence, the order used for sending messages is important. The impact on the algorithm design will be illustrated in Sections 4.4.1 and 4.4.3 during the detailed description of type 2 parallelism for LDL^T factorization.

4.4 LU versus LDL^T approaches

In this section, we describe the main differences between the symmetric and the unsymmetric algorithms. The symmetric code currently solves symmetric positive-definite systems, but it has been designed so that future developments like fully distributed LDL^T factorization with numerical pivoting and the detection of the null spaces, remain possible.

Taking into account the symmetry of the input matrix leads to a reduction in both the memory requirements (smaller input matrix, matrix of factors and frontal matrices) and the computational cost. Only the lower part of the original matrix is accessed and the LDL^T factorization is computed. Even if a significant part of the implementation issues are shared by the LU and LDL^T factorizations, taking into account the symmetry implies major modifications in the assembly process, in the blocked factorization of nodes of type 1 and 2, and in the type 2 and 3 parallel algorithms.

Taking into account the symmetry for a node of type 3 was rather straightforward because our implementation is based on the use of ScaLAPACK [15] routines (PDGETRF for the LU factorization and PDPOTRF for the LL^T factorization). Note that a parallel version of the LDL^T factorization for dense matrices does not exist in ScaLAPACK and that this issue will have to be addressed in a future release of the code that includes numerical pivoting for symmetric matrices.

4.4.1 Assembly process

An estimation of the frontal matrix structure (size, number of fully-summed variables) is computed during the analysis phase. The final structure and the list of indices in the front is however only computed during the assembly process of the factorization phase. The list of indices of a front is the result of a merge of the index lists of the contribution blocks of the children with the list of indices in the arrowheads associated with all the fully-summed variables of the front. Once the index list of the front is computed, the assembly of numerical values can be performed efficiently.

Let *inode* be a node of type 2. The master of *inode* defines the partition of rows of the frontal matrix into blocks, and chooses a set of slave processors that will participate in the parallel assembly and factorization of *inode*. It sends a message (identified by the tag DESC_STRIP) describing the work to be done on each slave processor. It also sends a message (with tag MAPROW) to all type 1 nodes and slave processors of type 2 nodes for the children of *inode*, giving them information on where to send their contribution blocks for the assembly process.

As already mentioned in Section 4.3, the order in which messages are sent is important. For example, a slave of *inode* may receive a contribution block before receiving the message of tag DESC_STRIP from its master. To allow this slave processor to safely perform a blocking receive on the missing DESC_STRIP message, we must ensure that the master of the node has sent DESC_STRIP before sending MAPROW. Otherwise we cannot guarantee that DESC_STRIP will actually be sent (for example, the send buffer might be full).

The main difference between the symmetric and the unsymmetric case is that a global ordering of the indices in the frontal matrices is necessary in the symmetric case to guarantee that all lower triangular entries in a contribution row of a child are in the lower triangular part of the corresponding row in the parent. We use the global ordering obtained during analysis, that is, the order in which variables would be eliminated if no numerical pivoting occurs.

Moreover, it is quite easy to perform a merge of sorted lists efficiently. If we assume that the list of indices of the contribution block of each child is sorted then the sorted merge algorithm will be efficient if the indices associated with the arrowheads are also sorted. Unfortunately, sorting all the arrowheads can be costly. Furthermore, the number of fully-summed variables (or number of arrowheads) in a front might be quite large and the efficiency of the merging algorithm might be affected by the large number of sorted lists to merge. Based on experimental results, we have observed that it is enough to sort only the arrowhead associated with the first fully-summed variable of each frontal matrix. The assembly process for the list of indices of the node is thus described in Algorithm 2.

Algorithm 2 Assembly of indices in a parent node

```
1. Sorted merge of the sorted lists of the indices of the children and of the first arrowhead.
2. Build and sort variables belonging only to the other arrowheads (and not found at step 1)
3. Merge the sorted list built at step 2 with the sorted list obtained at step 1.
```

The key issue for efficiency of Algorithm 2 is the fact that only a small number of variables are found at step 2. This has been experimentally validated. For example, on matrix WANG3, the average number of indices found at step 2 was 0.3. The numerical assembly can then be performed, row by row.

4.4.2 Factorization of type 1 nodes

Blocked algorithms are used during the factorization of type 1 nodes and, for both the LU and the LDL^T factorization algorithms, we want to keep the possibility of postponing the elimination of fully-summed variables. Note that classical blocked algorithms for the LU and LL^T factorizations of full matrices [11] are quite efficient, but it is not the case for the LDL^T factorization.

We will briefly compare kernels involved in the blocked algorithms. We then show how we have exploited the frontal matrix structure to design an efficient blocked algorithm for the LDL^T factorization.

Let us suppose that the frontal matrix has the structure of Figure 6, where A is the block of fully summed variables available for elimination. Note that, in the code, the frontal matrix is stored by rows.

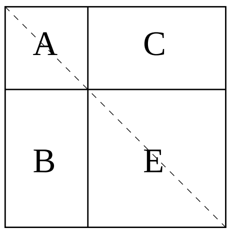


Figure 6: Structure of a type 1 node.

During LU factorization, a KJI-SAXPY blocked algorithm [3, 21] is used to compute the LU factor associated with the block of fully summed rows (matrices A and C). The Level 3 BLAS kernel DTRSM is used to compute the off-diagonal block of L (overwriting matrix B). Updating the matrix E is then a simple call to the Level 3 BLAS kernel, DGEMM.

During LDL^T factorization, a right-looking blocked algorithm is first used to factor the block column of the fully summed variables. Let L_{off} be the off diagonal block of L stored in place of the matrix B and D_A be the diagonal matrix associated with the LDL^T factorization of the matrix A. The updating operation of the matrix E is then of the form $E \leftarrow E - L_{off} D_A L_{off}^T$ where only the lower triangular part of E needs to be computed. No Level 3 BLAS kernel is available to perform this type of operation which corresponds to a generalized DSYRK kernel.

Note that, when we know that no pivoting will occur (symmetric positive definite matrices), L_{off} is computed in one step using the Level 3 BLAS kernel DTRSM. Otherwise, the trailing part of L_{off} has to be updated after each step of the blocked factorization, to allow for a stability test for choosing the pivot.

To update the matrix E, we have applied the ideas used by [22] to design efficient and portable Level 3 BLAS kernels. Blocking of the updating is done in the following way. At each step, a block of columns of E (E_k in Figure 7) is updated. In our first implementation of the algorithm, we stored

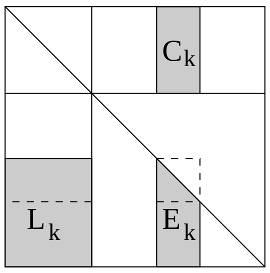


Figure 7: Blocks used for updates of the contribution part of a type 1 node.

the scaled matrix $D_A L_{off}^T$ in matrix C, used here as workspace. Because of cache locality issues, the Megaflop rate was still much lower than that of the LU or Cholesky factorizations. In the current version of the algorithm, we compute the block of columns of $D_A L_{off}^T$ (C_k in Figure 7) only when it will be used to update E_k. Furthermore, to increase cache locality, the same working area is used to store all C_k matrices. This was possible because C_k matrices are never reused in the algorithm. Finally, the Level 3 BLAS kernel DGEMM is used to update the rectangular matrix E_k. This implies more operations but is more efficient on the IBM SP2 than the updates of the shaded trapezoidal submatrix of E_k using a combination of DGEMV and DGEMM kernels. Our final blocked algorithm is summarized in Algorithm 3.

Algorithm 3 LDL^T **factorization of type 1 nodes**

```
  Blocked factorization of the fully summed columns
  do k = 1, nb_blocks
     Compute $C_k$ (block of columns of $D_A L_{off}^T$)
     $E_k \leftarrow E_k - L_k C_k$
  end do
```

4.4.3 Parallel factorization of type 2 nodes

The differences between the symmetric and the unsymmetric case come from a modification of both the frontal matrix structure and the parallel algorithm. The modification of the matrix structure is illustrated in Figure 8. In both algorithms, the master processor is in charge of all the fully summed rows and the blocked algorithms used to factor the block of fully-summed rows are the ones described in the previous subsection.

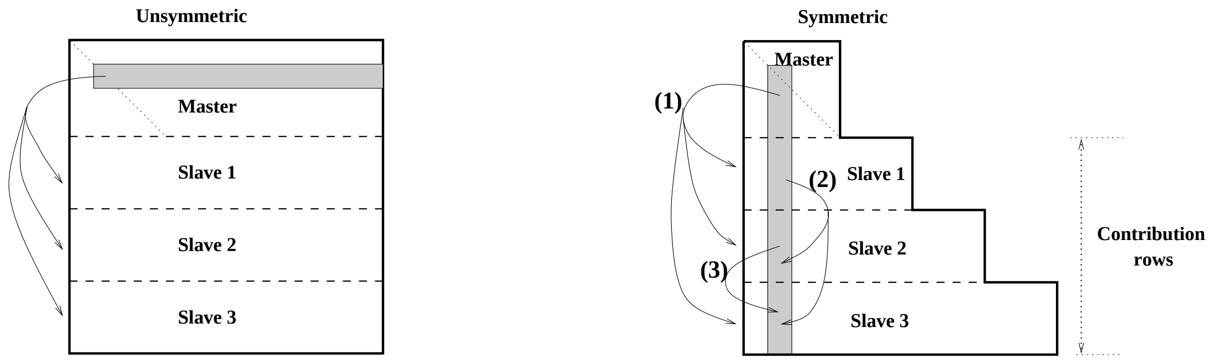


Figure 8: Structure of a type 2 node.

In the unsymmetric case, at each block step, the master processor sends the factorized block of rows to its slave processors and then updates its trailing submatrix. The behaviour of the algorithm is illustrated in Figure 9, where program activity is represented in black, inactivity in grey, and messages by lines between processes. The figure is a trace record generated by the VAMPIR package [43] from PALLAS. We see that, on this example, the master processor is relatively more loaded than the slaves.

In the symmetric case, a different parallel algorithm has been implemented. The master of the node performs a blocked factorization of only the diagonal block of fully-summed rows. At each block step, its part of the factored block of columns is broadcast to all slaves ((1) in Figure 8). Each slave can then use this information to compute its part of the block column of L and to update part of the trailing matrix. Each slave, apart from the last one, then broadcasts its just computed part of the block of column of L to the following slaves (illustrated by messages (2) and (3) in Figure 8). Note that, to process messages (2) or (3) at step k of the blocked factorization, the corresponding message (1) at step k must have been received and processed.

We have chosen a fully asynchronous approach to implement the algorithm. Messages (1) and (2) might thus arrive in any order. The only property that MPI guarantees is that messages of type (1) will be received in the correct order because they come from the same source processor. When a message (2) at step k arrives too early, we have then to force the reception of all the pending

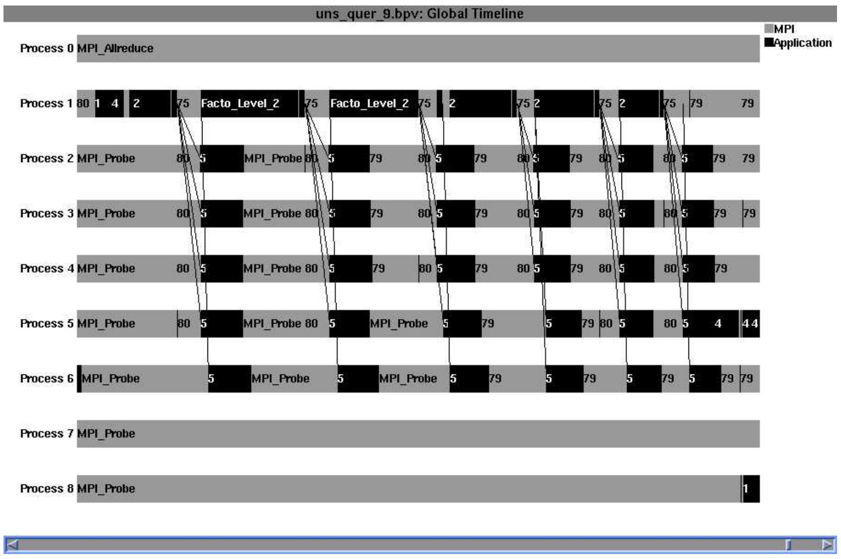


Figure 9: VAMPIR trace of an isolated type 2 unsymmetric factorization (Master is Process 1).

messages of type (1) for steps smaller than or equal to k. This induces a necessary property in the broadcast process of messages (1): if at step k, message (1) is sent to slave 1, we must be sure that it will also be sent to other slaves. In our implementation of the broadcast, we first check availability of memory in the send buffer (with no duplication of data to be sent) before starting effective send operations. Thus, if the asynchronous broadcast starts, it will complete.

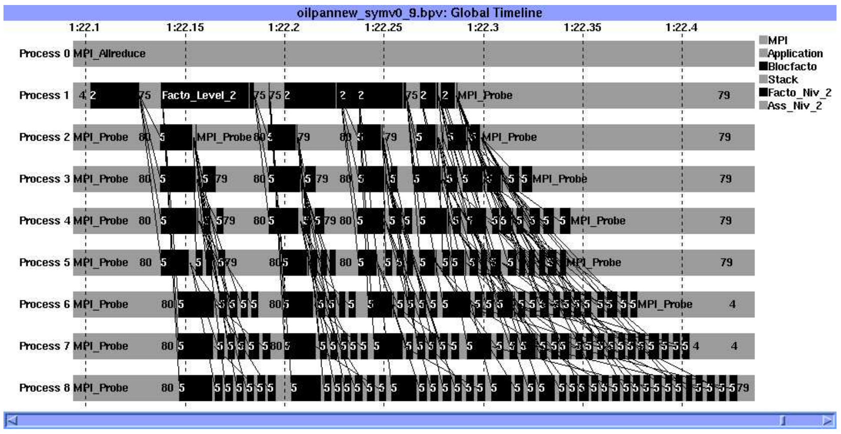


Figure 10: VAMPIR trace of an isolated type 2 symmetric factorization; constant row block sizes. (Master is Process 1).

Similarly to the unsymmetric case, our first implementation of the algorithm is based on constant row block size. We can clearly observe from the corresponding execution trace in Figure 10 that the later slaves have much more work to perform than the others. To balance work between slaves, later slaves should hold less rows. This has been implemented using a heuristic that aims at balancing the total number of floating-point operations involved in the type 2 node factorization on each slave. As a consequence, the number of rows treated varies from slave to slave. The corresponding execution trace is shown in Figure 11. We can observe that work on the slaves is much better balanced and both the difference between the termination times of the slaves and the elapsed time for factorization are reduced.

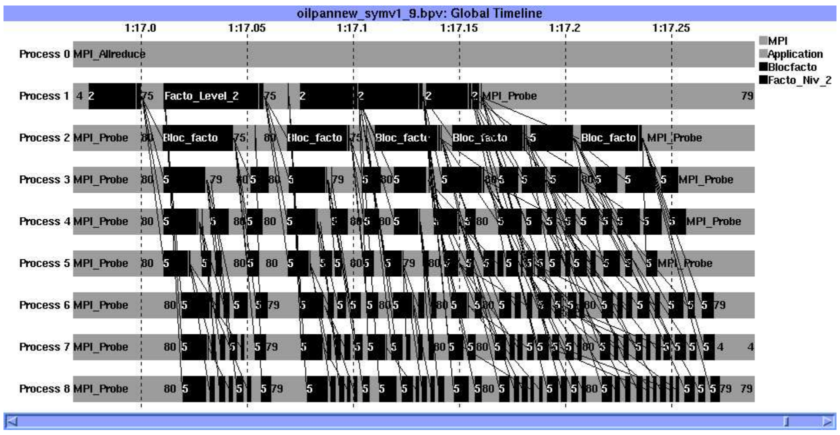


Figure 11: VAMPIR trace of an isolated type 2 symmetric factorization; variable row block sizes. (Master is Process 1).

However, the comparison of Figures 9 and 11 shows that firstly the number of messages involved in the symmetric algorithm is much larger than in the unsymmetric case; secondly, that the master processor performs relatively less work than in the parallel algorithm for unsymmetric matrices.

5 Some other functionalities of MUMPS

5.1 Multiple instances

MUMPS has been integrated in the DDM solver that is being developed at the University of Bergen. For the DDM solver, it is very important that MUMPS is able to factorize and solve in separate steps. It is equally important that MUMPS can handle several systems of equations simultaneously. This allows the DDM solver to first factor a set of matrices, and then use the factors to solve the associated systems in a cg-iteration. Since the MUMPS software defines each system of equations as an *instance*, MUMPS must therefore allow several instances to coexist. This feature was not planned initially, but has been implemented on request from the DDM developers.

5.2 Pre-processing and post-processing facilities

MUMPS offers pre-processing and post-processing facilities. Permutations for a zero-free diagonal [29] can be applied to very unsymmetric matrices and can help reduce fill-in and arithmetic. Prescaling of the input matrix can help reduce fill-in during factorization and can improve the numerical accuracy. A range of classical scalings are provided for the user and can be automatically performed before numerical factorization. Iterative refinement can be optionally performed after the solution step. Arioli, Demmel, and Duff [12] have shown that with only two to three steps of iterative refinement the solution can often be significantly improved. Finally, MUMPS also enables the user to perform classical error analysis based on the residuals.

MUMPS returns an estimate of the sparse backward error using the theory and metrics developed in [12]. We use the notation $\bar{\mathbf{x}}$ for the computed solution and a modulus sign on a vector or a matrix to indicate the vector or matrix obtained by replacing all entries by their moduli. The scaled residual

$$\omega_1 = \frac{|\mathbf{b} - \mathbf{A}\bar{\mathbf{x}}|_i}{(|\mathbf{b}| + |\mathbf{A}||\bar{\mathbf{x}}|)_i} \tag{1}$$

is computed for all equations except those for which the numerator is nonzero and the denominator is small. For all the exceptional equations,

$$\omega_2 = \frac{|\mathbf{b} - \mathbf{A}\bar{\mathbf{x}}|_i}{(|\mathbf{A}|\ \ |\bar{\mathbf{x}}|)_i + \|\mathbf{A}_i\|_\infty \|\bar{\mathbf{x}}\|_\infty} \qquad (2)$$

is used instead, where A_i is row i of A. The largest scaled residual (1) and the largest scaled residual (2) are both returned. If all equations are in category (1), ω_2 is set to zero. The computed solution $\bar{\mathbf{x}}$ is the exact solution of the equation

$$(\mathbf{A} + \delta\mathbf{A})\mathbf{x} = (\mathbf{b} + \delta\mathbf{b}),$$

where

$$\delta\mathbf{A}_{ij} \leq \max(\omega_1, \omega_2)|\mathbf{A}|_{ij},$$

and $\delta\mathbf{b}_i \leq \max(\omega_1|\mathbf{b}|_i, \omega_2\|\mathbf{A}_i\|_\infty\|\bar{\mathbf{x}}\|_\infty)$. Note that $\delta\mathbf{A}$ respects the sparsity of $\mathbf{A}$. An upper bound for the error in the solution is also returned.

5.3 Input of matrices in elemental format

MUMPS allows the user to input the matrix in elemental format. The matrix is not assembled by the solver but the elemental matrices are preserved and directly operated on. The main modifications that were necessary to the implementation lie in the analysis, the distribution of the matrix (especially the root matrix), and the assembly process. The corresponding right-hand side must still be provided in assembled format.

5.4 Rank revealing and null space basis determination

MUMPS provides options for rank detection and computation of the null space basis. The dynamic pivoting strategy available in both the symmetric and unsymmetric version of MUMPS postpones all the singularities to the root node of the elimination tree. Therefore, the problem of rank detection of the original matrix is reduced to the problem of rank detection of the root matrix. At this root, rank revealing algorithms are applied. The null space basis for the original matrix is computed from the null space basis for the root matrix by a backsubstitution with the computed sparse factors.

Strategies based on rank revealing QR and rank revealing LU are provided. The strategies assume that the rank deficiency of the matrix is small (which is the case in the applications of the PARASOL project). Details will be given in a forthcoming technical report [9].

5.5 Distributed assembled matrix

If the assembled matrix is initially held centrally on the host, the time to distribute the real entries of the original matrix can be comparable to the time to perform the actual factorization. For example, on matrix OILPAN, the time to distribute the input matrix is on average 6 seconds whereas the time to factorize the matrix on 16 processors of the IBM SP2 is 6.8 seconds. The distribution of the input matrix 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. Clearly, increasing the size of the matrix will make the time for the factorization phase dominant although a centralized matrix will limit the size of the problem that can be solved on a distributed memory computer. With a distributed input matrix format, a performance improvement can be expected because we parallelize the reformatting and sorting tasks using asynchronous all-to-all communications during the redistribution phase. To improve both the memory and the time scalability of our approach, the input matrix should thus be distributed. Note that, based on the static mapping of the tasks to processes, one can determine an *a priori* distribution of the input data so that no further remapping will be required. This approach, 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 6.3). In this case, the distribution depends on the static mapping which itself depends on the number of processors used.

In Figure 12, we compare three strategies for providing the input matrix:

1. Centralized matrix: the input matrix is held on one processor (the host).
2. MUMPS mapping: the input matrix is distributed over the processors according to a mapping array that is computed during the analysis phase.
3. Random mapping: the input matrix is distributed over the processors in a random but even manner that has no correlation to the mapping computed during the analysis phase.

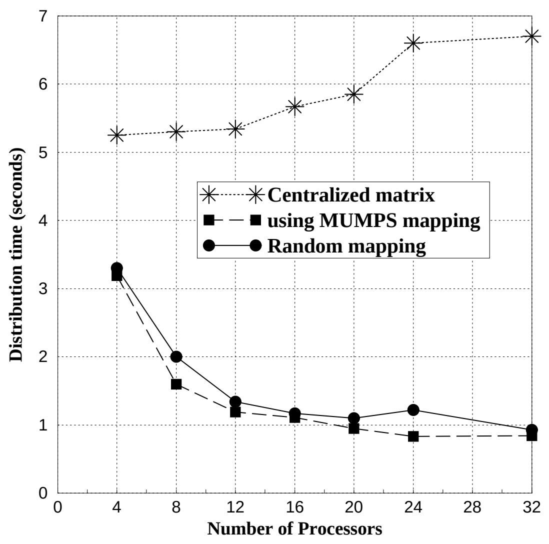


Figure 12: Impact of the input format for matrix OILPAN on the time for distribution on the IBM SP2.

We clearly see, in Figure 12, the benefit of using asynchronous all-to-all communications (MUMPS or random mapping options) with respect to using one-to-all communication patterns (centralized matrix option). Furthermore, it is even more interesting to observe that exploiting the mapping provided during analysis does not significantly reduce the time because of the good overlapping between communication and computation (here mainly data reformatting and sorting) in our redistribution algorithm.

5.6 Return a specified Schur complement

A Schur complement matrix can be returned to the user. The user must specify the list of indices of the Schur matrix. MUMPS then provides both a partial factorization of the complete matrix and access to the assembled Schur matrix. The Schur matrix is considered as a full matrix. The partial factorization that builds the Schur matrix can also be used to solve linear systems associated to the "interior" variables.

For example, consider the partitioned matrix

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \tag{3}$$

where the variables of $\mathbf{A}_{2,2}$ are those specified by the user. Then the Schur complement, as returned by MUMPS, is $\mathbf{A}_{2,2} - \mathbf{A}_{2,1}\mathbf{A}_{1,1}^{-1}\mathbf{A}_{1,2}$, and the solve is performed on $\mathbf{A}_{1,1}$ only.

6 Performance analysis and code tuning

6.1 Basic performance and influence of ordering

From earlier studies (for example [40]), 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. That is, we only consider tree parallelism and exclude any parallelism from within the nodes of the assembly tree (node parallelism). The results show that tree parallelism only does not produce very good speedups but they also show (see column "Speedup") that we usually get better parallelism from the tree with nested dissection based orderings than with minimum degree based orderings. We thus gain by using nested dissection because of a reduction in the number of floating-point operations (see Tables 3 and 4) and a better balanced assembly tree.

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.

We now discuss the performance obtained with MUMPS that will be used as a reference for this report. The performance obtained on matrices provided in elemental format (RSE matrices) will be shown in Section 6.2. In Tables 7 and 8, we show the performance of 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 when using a small number of processors. Performance gains when using nested dissection orderings on a small number of processors of the IBM SP2, are thus due also to the reduction in the memory required by each processor to factorize the matrix. Therefore, instead of reporting elapsed time on 1 processor, the uniprocessor CPU time has been reported. When the memory was not large enough to run on one processor, an estimation of the MFlops rate has been used to compute a uniprocessor time. This estimate was also used, when necessary, to compute the corresponding speedups in Table 6. Note that on a small number of processors there can still be a memory effect so that there appears to be no speedup relative to the CPU uniprocessor time although the speedup over the elapsed time on one processor can be considerable.

In Table 8, we also show the time for the solution phase; we observe that, relatively to what could be expected on this phase, the speedups are quite good.

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

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
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
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
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
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
MIXING-TANK	AMD	495	0	288.5	70.7	64.5	61.3
	ND	104	32.80	26.1	17.4	14.4	14.8

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.

Factorization phase							
Matrix	Ordering	Number of processors					
		1	2	4	8	16	32
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
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
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
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

Solution phase							
Matrix	Ordering	Number of processors					
		1	2	4	8	16	32
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
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
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
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

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

6.2 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 which correspond to variables in the ith element. It is usually held as a small square matrix that is dense and possibly unsymmetric. 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 simply due to the fact that the frontal matrix structure contains, by definition, all the variables adjacent to all the fully summed variables 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 [13], 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 implementation for assembled matrices lie in the analysis, the distribution of the matrix, and the assembly process.

During the analysis phase, we can exploit the elemental format of the matrix to detect supervariables. A **supervariable** is a set of variables having the same list of adjacent elements. This is illustrated in Figure 13 where the matrix is composed of two overlapping elements and three supervariables can easily be detected.

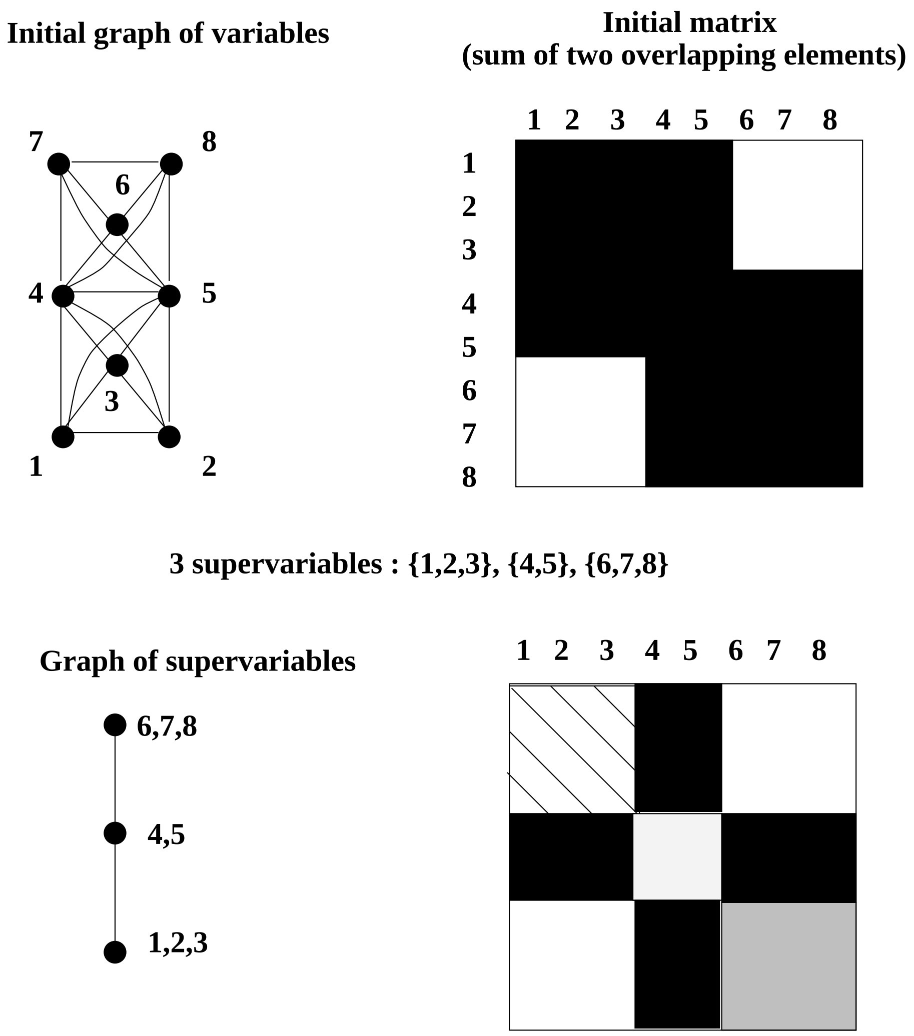
Supervariables have been used, in a similar context, to compress graphs associated with assembled matrices from structural engineering prior to a multiple minimum degree ordering [14]. For assembled matrices, however, it has been observed that the use of supervariables, in combination with an Approximate Minimum Degree ordering, is not more efficient, (see [1]).

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 size of the input graph *Graph_size* (length of adjacency lists) given to the ordering phase.

Tables 9 and 10 show the impact on both the size of the graph processed by the ordering phase (AMD ordering) and on the time for the complete analysis phase (including graph compression and ordering). *Graph_size* corresponds to the size of the graph (length of adjacency lists of variables/supervariables) given as input to the ordering phase. Table 2 shows that, when supervariable detection is off, *Graph_size* is twice the number of entries (diagonal entries excluded) in the symmetric 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. If 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. Finally, we see in Table 10 that, using supervariables, the reduction in time is not only due to the reduced time for ordering, but



Initial graph of variables
Initial matrix
(sum of two overlapping elements)
1 2 3 4 5 6 7 8
7
8
6
4
5
3
1
2
3 supervariables : {1,2,3}, {4,5}, {6,7,8}
Graph of supervariables
6,7,8
4,5
1,2,3

Figure 13: Supervariable detection for matrices in elemental format.

also to a significant reduction in time 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 the 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 unassembled 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 might 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 more input data will be assembled earlier in the processing of the assembly tree. This has two consequences. The assemblies are performed in a more distributed way and fewer element duplications might occur for nodes of type 2 (since we do not split the elements, we duplicate the complete elements belonging to a node of type 2). A more detailed analysis of the duplication issues linked to matrices in elemental format will be addressed in Section 6.3.

The experimental results in Tables 11 and 12, obtained on the SGI Origin 2000, show the good behaviour of the code on a limited number of processors for both the factorization and the solution phases.

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.

We have observed that the performance of MUMPS for assembled and unassembled problems is very similar, provided the same ordering is used. The reason for that is that the extra amount of assemblies of original data for unassembled problems is relatively small compared to the total number of flops.

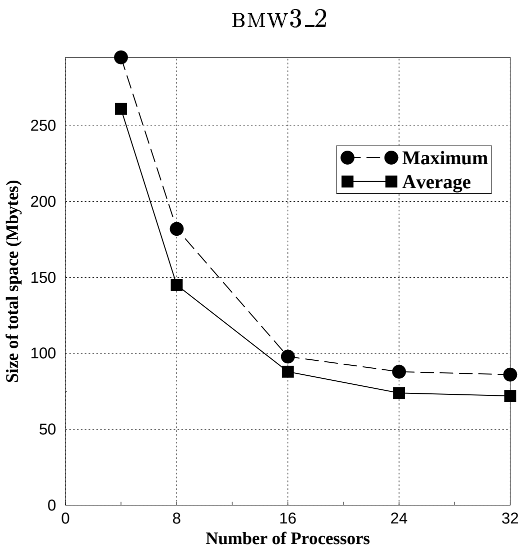
Matrix	Number of processors				
	1	2	4	8	16
M_T1.RSE	3.5	2.1	1.1	1.2	0.8
SHIP_003.RSE	6.9	3.6	3.3	2.5	2.0
SHIPSEC1.RSE	3.8	3.1	2.1	1.6	1.5
SHIPSEC5.RSE	5.5	4.2	2.9	2.2	1.9
SHIPSEC8.RSE	3.8	3.1	2.0	1.4	1.3
THREAD.RSE	2.3	1.9	1.3	1.0	0.8
X104.RSE	2.6	1.9	1.4	1.0	1.1

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

6.3 Memory scalability issues

On distributed memory computers, one of the main properties of a parallel algorithm is its capacity to efficiently exploit the increase in memory available when increasing the number of processors. This issue is often as critical as the reduction in execution time.

We show, in Figure 14, the maximum and average memory required per processor as a function of the number of processors. We see that these values are quite similar and this shows that the memory load is well balanced between the processors.



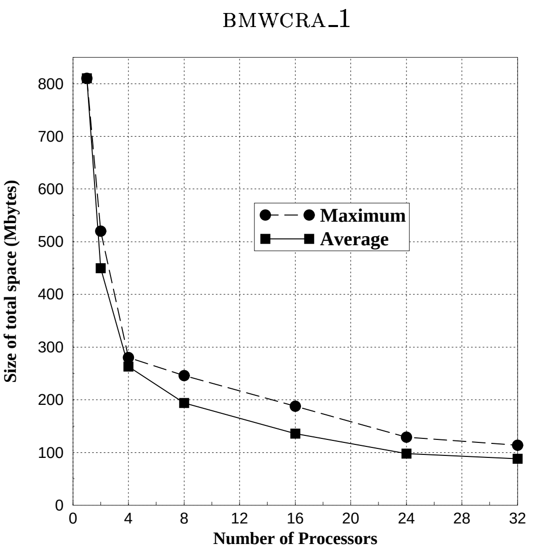


Figure 14: Total memory (maximum and average) requirement per processor (ND ordering).

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 size 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 size of the space used for stacking both the contribution blocks and the factors.
- INITIAL MATRIX: the size required to store the initial matrix in arrowhead format.
- COMMUNICATION BUFFERS: the space allocated for both send and receive buffers.

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.

- OTHER: the size of all the remaining working space 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. This allows us to check how the memory requirements of `MUMPS` scale over the processors.

We see that, even if the total memory (sum of all the local working spaces) increases, the average memory required per processor significantly decreases up to 16 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. Let N denotes the matrix order. We see, as expected, that the line OTHER does not scale at all since it corresponds to data arrays of size in $O(N)$ that need to be allocated on each contributing process. We see that although this space is $O(N)$, it still significantly affects the difference between TOTAL and *ideal*. 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 is explained by the fact that, to migrate tasks corresponding to type 2 nodes efficiently (see Section 6.4), part of the data associated with the original matrix is duplicated on all processors. We will study this feature 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 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 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 have sufficient node parallelism near the root of the assembly tree, MUMPS uses a heuristic that relates the number of type 2 nodes to the number of processors used. The influence of the number of working processors on the duplication of matrix data 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 entries from the original matrix. If there is only one processor, no type 2 nodes are introduced and no data is duplicated.

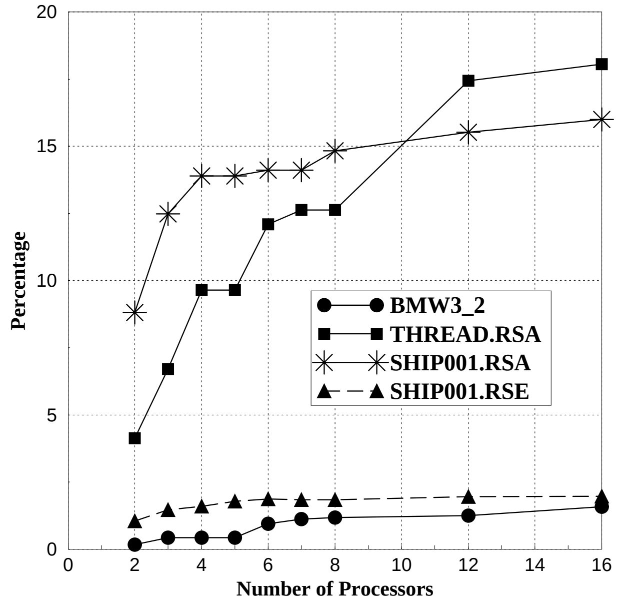


Figure 15: Study of the duplication of the input matrix due to type 2 nodes. The percentage of duplicate entries is relative to the number of entries in the original matrix. A value of k means that k percent of the original matrix is duplicated.

Typical effects are summarized in Figure 15 where we show the percentage of duplicate entries per process. On a globally addressable memory computer, such as the SGI Origin 2000, Table 14 reflects more the main property of the algorithm whereas on a purely distributed memory computer, such as the IBM SP2, the most relevant information is shown in Figure 15.

It is quite interesting to note that, since the original data for unassembled matrices are in general assembled earlier in the assembly tree than 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 case 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 from a memory point of view even if, on shared memory non-uniform memory access computers, our overestimation of the total stack area is a bottleneck that needs to be addressed. Limiting the dynamic scheduling to a subset of processors when performing the static allocation might be a simple solution to control the increase in the total memory required.

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
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
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

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

6.4 Dynamic scheduling strategies

To avoid the drawback of centralized scheduling on distributed memory computers, we have chosen to implement 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 6.3).

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 priority is given 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 that the sending process might be blocked because its send buffer is full.

Two scheduling strategies have been implemented. 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 processes. The load of a processor is here defined 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 routine (one-sided communication **MPI_GET**), each process could have access to the load of all other processes when necessary. This feature, included in MPI-2, is not available on our target computers. To overcome this problem of portability, we have designed a module based only on symmetric communication tools (MPI asynchronous send and receive). Each process is in charge of both updating its local load and broadcasting the information. To control the amount of extra data sent, an updated load is broadcast only if it represents a significant relative variation with respect to the last load value broadcast.

Matrix &	Number of processors				
scheduling	16	20	24	28	32
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
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

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

Matrix &	Number of processors		
scheduling	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).

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 flop-based scheduling. On a large number of processors (> 24), the gain is less significant because our test problems are too small to keep all 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. The results reported in Table 16 were obtained when the SGI Origin 2000 was not very loaded. Results on a loaded machine vary considerably and are not reported here because of the difficulty of interpreting them. An investigation on a computer providing non-symmetric communication routines might be of interest.

Another possible use of dynamic scheduling is to improve the memory usage. We have seen, in the previous section, that the size of the working space required for the stack area is overestimated and based on a very slack and unobtainable upper bound. Dynamic scheduling based on memory 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.

6.5 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. Considering the parallel factorization of the complete matrix, the elimination of the first fully summed rows can thus represent a potential bottleneck for scalability, especially for frontal matrices with a large fully summed block near the root of the tree, where parallelism arising from the tree is limited. To overcome this problem, we postprocess the tree to subdivide nodes with large fully summed blocks, as shown in Figure 16.

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 $\text{NFRONT} - \text{NPIV}_{son}$, with $\text{NPIV}_{father} = \text{NPIV} - \text{NPIV}_{son}$ pivots. This allows us to decrease the longest path in the graph of tasks at the cost of extra assembly operations and extra communications. Note that, in practice, we might divide the intial node into more than two new nodes.

We experimented with a simple algorithm that modifies the assembly tree to see how this feature could improve the parallelism. The algorithm is applied after the symbolic factorization. We applied the algorithm only to nodes near the root of the tree, that is only up to a certain maximum distance from the root. We chose this distance (that is the number of edges between the root and the node) to be $d_{max} = \log_2(\text{NPROCS} - 1)$. We do this because splitting large nodes far from the root of the tree where sufficient tree parallelism can already be exploited would only lead to additional assembly and communication costs.

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, controlled by a parameter p, $p > 0$.

Algorithm 2 *Splitting of a node*

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

Note that Algorithm 2 is applied only when NFRONT – NPIV/2 is large enough because we want to guarantee that the son of the split node will be of type 2. $\text{NFRONT} - \text{NPIV}_{son}$ is the size of the contribution block of the son and during Algorithm 2 NPIV_{son} is set to NPIV/2. The choice of p allows us to control the general amount of split and is a machine dependent parameter. At step 3 of the algorithm, we multiply p by $d(Inode) - 1$ to limit the number of split nodes as tree parallelism increases.

Splitting is analysed in Table 17 on both a symmetric test problem CRANKSEG_2 and an unsymmetric test problem INV-EXTRUSION-1. *Ncut* corresponds to the number of type 2 nodes cut to increase the parallelism. A value $p = 0$ is used as a flag to indicate no splitting, while more splitting occurs as p decreases. 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

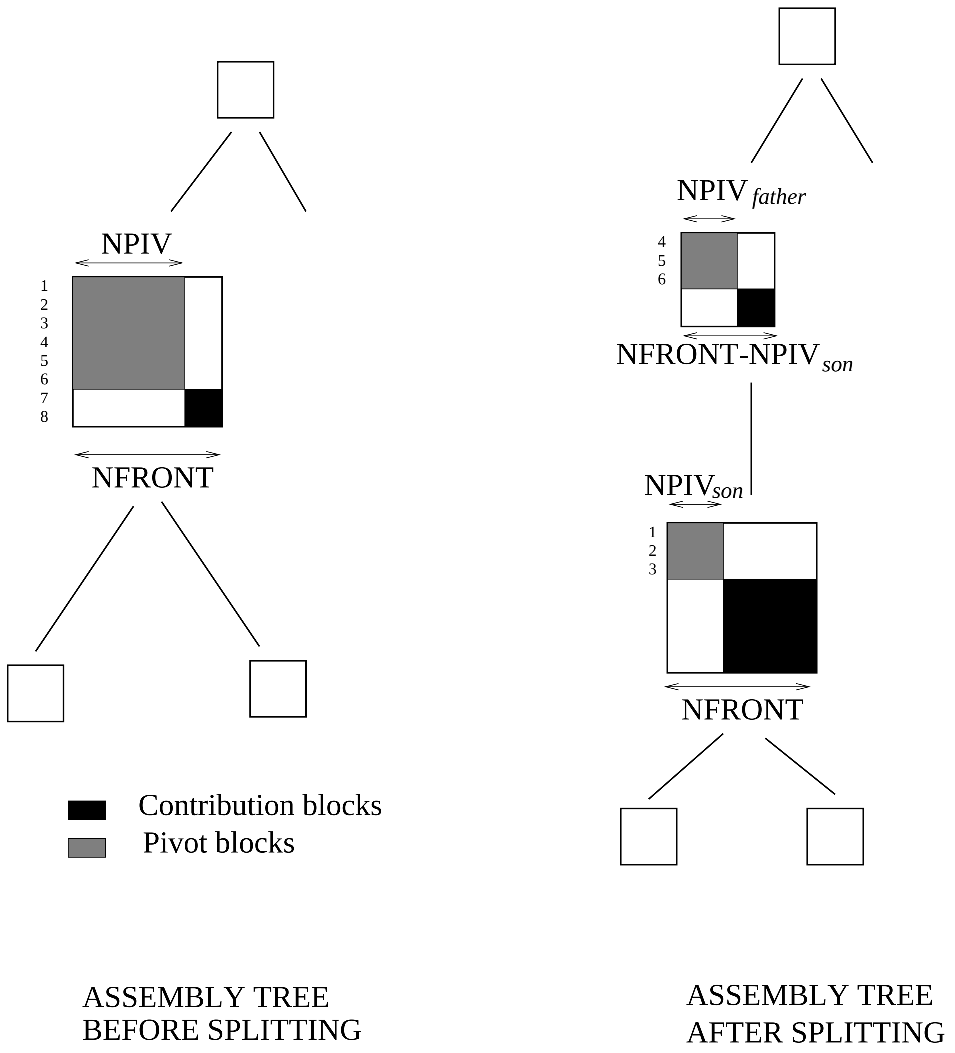


Figure 16: Tree before and after the subdivision of a frontal matrix with a large pivot block.

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.

node splitting. For small values of p, the number of nodes split does not increase too much and that even if the best timings are generally obtained for relatively large values of p, the time does not increase too much for small values of p.

7 Dissemination: the use of MUMPS in a joint project CERFACS-INRIA

In the framework of a joint research project between INRIA (A. Marrocco, P. Le Tallec) and CERFACS (L. Giraud, J.C. Rioual) the `MUMPS` package has been used within a parallel domain decomposition code for the solution of the drift diffusion equation involved in semiconductor device modeling [39].

The model problem describes the stationary state of a transistor (for $t \to \infty$) when a tension is applied on its bounds. The model is a system of six completely coupled nonlinear partial differential equations. The system is decoupled and discretized in time by an implicit nonlinear scheme. At each time step, three systems of two nonlinear partial differential equations have to be solved. The first system is associated with the electrostatic potential, the second with the negative charges (electrons), and the third with the positive charges (holes).

Each of these systems is discretized in space by a mixed finite element method defined on 2D unstructured meshes and solved by a Newton-Raphson method. At each step of this method, a linear system of equations has to be solved. For this, a domain decomposition technique without overlapping is used. Preliminary experimental results have shown the relevance of a two level preconditioner for the Schur complement. The coarse space component consists in associating one degree of freedom with each edge of the domain decomposition, referred to as "edge-based" preconditioner in [19]. Among the local components investigated, the most robust is the one that consists in assembling the local Schur complement associated with each subdomain. This results in a block diagonal preconditioner with overlap between the blocks [17, 18]. While this preconditioner is numerically relevant, we need an efficient sparse software to make it computationally competitive. In this context `MUMPS` and its Schur complement functionality are used. The list of unknowns of the Schur matrix (i.e. interface nodes of the Neumann local matrix in a domain decomposition framework) is given to `MUMPS` which provides the assembled Schur complement matrix and the possibility to use the factorization done on the interior nodes. The advantage of using `MUMPS` is twofold. First, it computes the local Schur complement matrix at only about twice the price of the factorization of the local Dirichlet problem. Second, it significantly reduces the computational time in the Krylov iteration as we only need to perform a DGEMV (compared to a sparse forward/backward substitution in a classical approach).

To validate the above domain decomposition approach, a comparison with a complete parallel direct technique has been made. For this, the distributed input matrix functionality of `MUMPS` has been used. The matrix associated with each subdomain is provided to `MUMPS` in a fully distributed way. Overlapping between the the subdomains is handled automatically, which solves the complete problem (with no centralized assembly) in parallel.

In conclusion, the ability to compute the local Schur complement at a low cost is a crucial feature for the design of an efficient preconditioner.

The present study is ongoing research that is part of the PhD work of Jean-Christophe Rioual.

8 Performance summary of MUMPS4.0 and comparison with PSPASES v1.0.2

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 6.4 and 6.5.

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
BMW7ST_1	104	-	29.8	13.7	11.7	11.3
BMW3_2	246	-	-	24.1	24.0	20.4
CRANKSEG_1	270	185.3	92.4	27.3	25.6	20.9
CRANKSEG_2	378	-	-	41.8	31.0	27.2
BBMAT	198	255.4	85.2	34.8	32.8	30.9
INV-EXTRUSION-1	70	24.9	16.3	13.5	13.4	12.4
MIXING-TANK	104	30.8	21.6	16.4	14.7	14.8

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
sc mh	37	21	13	7	5
BMW7ST_1	62	36	25	12	10
BMW3_2	151	96	53	33	18
BMWCRA_1	307	178	82	58	36
CRANKSEG_2	217	112	66	46	29
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.

To further analyse the performance obtained with `MUMPS`, we compare the bahaviour of the code with PSPASES v1.0.2 [37, 36]. We would like to emphasize that PSPASES is a software package for the parallel Cholesky (LL^T) factorization of symmetric positive definite matrices only; numerical pivoting is not possible and this allows for the use of much simpler static data distribution, data structures, and task mapping.

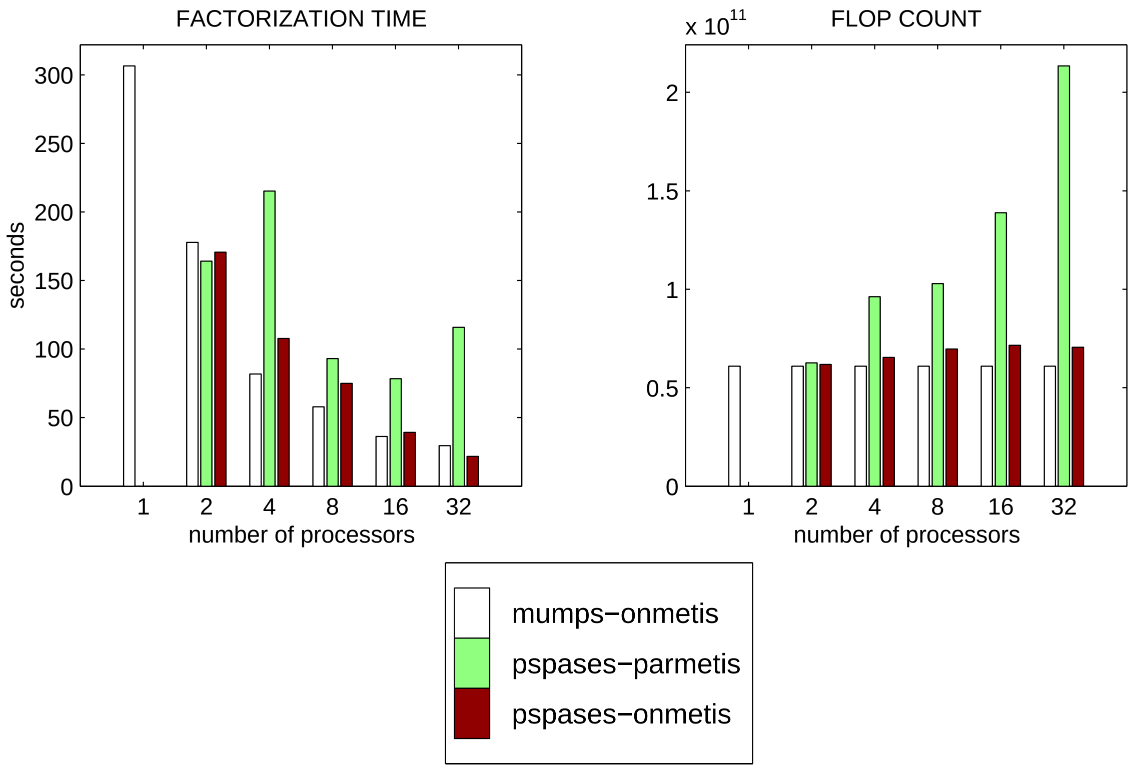


Figure 17: SGI Origin 2000, matrix BMWCRA_1

In Figure 17, we compare the performance obtained on matrix BMWCRA_1 with `MUMPS 4.0` and PSPASES v1.0.2. To have a fair comparison, nested dissection based orderings based on ONMETIS [41] were used for both methods. We observe that for the current version of PSPASES, the default ordering option (PARMETIS) should not be used for larger numbers of processors. We also see that, even if `MUMPS` performs a more costly LDL^T factorization of the matrix, the factorization times for the two packages are almost identical. Futhermore, we also notice the good parallel behaviour of the general purpose solver `MUMPS`, since timings are very comparable to those for PSPASES, even on 32 processors.

A PARASOL interface to MUMPS

The `MUMPS` solver has been fully integrated within the PARASOL Ttest Driver, referred to as `PTD`. In the following, we briefly describe how `MUMPS` can be called through the `PTD` and the PARASOL interface.

A.1 Configuring MUMPS through the PTD

Currently, `MUMPS` can be invoked through the `PTD` as follows:

```
"loader' ptd -uM [-"mode"] -d"path" "mat" "rhs" "sol" [-mi="list"]
```

Here, arguments between brackets `[]` are optional and

`"loader"` is a machine dependent system executable for loading and executing parallel programs on remote processors. For the IBM SP2 machine, this is normally `(/usr/bin/)poe`. For the SGI/CRAY Origin 2000 at Bergen, this is `mpirun -np "p"` where `"p"` is the number of processors.

`ptd` is the executable of the `PTD`.

`-uM` invokes `MUMPS` through the `PTD`.

`"mode"` specifies the parallel execution model that the `PTD` will use (y is hybrid-host mode. Y is hybrid-node mode). If this option is omitted, the PTD will run in host-node mode (which requires at least two processors).

`"path"` is the full path to the data files.

`"matrix"` is the name of the file containing the matrix in Rutherford-Boeing format.

`"rhs"` is the name of the file containing the right-hand side. If `"rhs"` is substituted by a `+`, the `PTD` will generate a random right-hand side and the user need not provide a file name.

`"sol"` is the name of the file that will hold the computed the solution vector (provided the `PTD` terminates successfully).

`"list"` is a comma-separated list of integers that are used to set up an instance for `MUMPS`. Currently, only two integers are accepted. The first integer specifies which parallel mode `MUMPS` will use (`0` for host-node mode, `1` for hybrid-host mode). The second integer specifies the type of symmetry of the input matrix (`0` symmetric positive definite, `1` general symmetric). This integer is ignored if the matrix is unsymmetric. If the integer list option is omitted, `MUMPS` will run in host-node mode and a symmetric matrix is considered positive definite.

Note that by setting the options `-y` and `-mi=1,0` (or `-mi=1,1`), the `PTD` will run on one processor. Running both the `PTD` and `MUMPS` in host-node mode, requires at least three processors.

Not all possible combinations of command line options are allowed. For example, if the PTD is invoked in hybrid-node mode (option -Y), MUMPS must be called in hybrid-host mode (option -mi=1,0 or -mi=1,1). If incompatible options are given, the PTD may modify some options (and issue an appropriate warning) and continue, or it may return with an error.

We note that some of the functionalities that are available within `MUMPS` cannot be invoked by the command-line options of the `PTD`.

A.2 Passing data to MUMPS

The PARASOL Interface Specification [32] defines four phases. Before being used, an instance of the package must always be initialized by the PARASOL initialization routine `psl_init`. The PARASOL mapping routine `psl_map` should then be called to analyse the data and map it onto the processors. After the mapping, the PARASOL solution phase, performed by routine `psl_solve`, solves the system of equations. The solution phase may be called multiple times so that a series of problems wit similar structure can be solved without recomputing the mapping. When the PARASOL instance is no longer needed, its allocated resources can be freed by the `psl_end` routine.

An instance of the PARASOL package operates on its own set of private data. Each instance has a descriptor that encodes several items, like for example an identification number, an MPI communicator, the selected solver, and the phase it has reached.

For `MUMPS`, the following actions are taken by the PARASOL interface at each phase:

initialization: The interface checks the PARASOL instance supplied by the user. If the instance is OK and specifies that `MUMPS` is to be used as solver, the interface extracts the data from this instance that is necessary to initialize a `MUMPS` instance (for example the MPI communicator and type of symmetry of the problem). Optionally, control parameters are read from a configuration file and the `MUMPS` instance is modified accordingly. The PARASOL instance and `MUMPS` instance have the same identification number.

mapping: The interface evaluates the mapping contract supplied by the user. If the contract is accepted, the interface passes the matrix pattern provided by the user to `MUMPS`. Optionally, the user can also provide an ordering of the variables *or* tell the interface to compute an ordering (see Section A.8). Parameters that control this are read from a second configuration file. After the (optional) ordering, `MUMPS` performs the analysis. `MUMPS` computed an ordering itself if it was not given an ordering. Optionally, auxiliary information output by `MUMPS` is returned to the user.

solve: The interface evaluates the solve contract supplied by the user. If the contract is accepted, depending on what is specified in the contract, the interface passes the matrix values and/or right-hand side to `MUMPS`. Optionally, the user may provide a scaling that will be used by `MUMPS`. `MUMPS` performs the factorization and/or the solve phase. Optionally, the solution vector, the deficiency of the matrix, a basis for the null space (if the matrix is deficient), and/or auxiliary information output by `MUMPS` is returned to the user.

end: The PARASOL instance and the corresponding `MUMPS` instance are terminated.

A.3 Instance descriptor

The PARASOL instance descriptor `id` is an integer array whose fields are assigned a prescribed meaning. It must be set by the user before the initialization phase of the PARASOL instance is called. The descriptor contains permanent fields and package- and user-controlled fields. We refer to the PARASOL Interface Specification [32] for a complete list of these fields, their meaning, and their possible values. Not all the descriptor fields need be relevant to a particular PARASOL solver and the range of values for a (relevant) field may be a subset of the range defined in [32]. Here, we list the permanent fields and values that are relevant to `MUMPS`. The package- and user-controlled fields must be used as described in [32].

Once set, the permanent fields of the instance descriptor must not be changed. The required PARASOL solver is held by the configuration field `id(PSL_CONF)`. To invoke `MUMPS`, this field must be set to `PSL_MUMPS`. The model of parallel operation of the PARASOL instance is held by the field `id(PSL_CONF)`. Currently, the values `PSL_HOSTNODE`, `PSL_HYBRIDHOST`, and `PSL_HYBRIDNODE` are allowed. The descriptor field `id(PSL_COMM)` must contain a valid MPI communicator. At the initialization phase, this communicator is duplicated (in hybrid-host mode) or a 'smaller' communicator is derived from it (in host-node mode). This new communicator is used by `MUMPS` for message passing. The field `id(PSL_DATA)` must contain a (unique) identifier for this PARASOL instance.

To fine tune the PARASOL instance for `MUMPS`, we currently use two additional permanent fields `id(PSL_CONF+1)` and `id(PSL_CONF+2)`. The field `id(PSL_CONF+1)` must be set to the type of symmetry of the input matrix (0 is unsymmetric, 1 is symmetric positive definite, 2 is general symmetric). The field `id(PSL_CONF+2)` must be set to the parallel execution model for `MUMPS` (0 is host-node, 1 is hybrid-host).

A.4 Data transfer descriptors

For each PARASOL mapping and solution phase, the data exchange between the user and the package (in our case `MUMPS`) is divided into data exchange sessions. Each data exchange session negiotates and fullfills data transfer from the user to the package and vice versa. At the start of each data exchange session, the user specifies to the package what (user-generated) data the package can get in order to compute the (packet-generated) data the user expects to receive upon return. This data is described by means of data transfer descriptors. A data transfer descriptor is an array of integers. Associated with each descriptor are up to three other arrays, the contents of which are interpreted according to the contents of the descriptor. If the package agrees with the description of the data offered by the user, a contract is established and for each data transfer descriptor the associated data is exchanged. The data exchange protocol is described in detail in [32].

In the following three sections, we list the data transfer descriptors that are used in the interface to MUMPS. For the meaning of the individual fields of these descriptors, we refer to [32].

A.5 Descriptors for mapping and solution phase

The parameters that control the actions taken by MUMPS during the mapping and solution phase can be provided by the user. To do this, the user may specify two descriptors, one for integer control parameters and one for double precision control parameters:

Integer control parameters (user-data descriptor) [optional]

```
ud(PSL_TYPE) = MPI_INTEGER
ud(PSL_NAME) = PSL_PARAM
ud(PSL_NROW) = 20
ud(PSL_NCOL) = 1
ud(PSL_LDIM) = ud(PSL_NROW)
ud(PSL_FORM) = PSL_TABLE
```

Double precision control parameters (user-data descriptor) [optional]

```
ud(PSL_TYPE) = MPI_DOUBLE_PRECISION
ud(PSL_NAME) = PSL_PARAM
ud(PSL_NROW) = 5
ud(PSL_NCOL) = 1
ud(PSL_LDIM) = ud(PSL_NROW)
ud(PSL_FORM) = PSL_TABLE
```

The data associated with the integer control parameter descriptor is an integer array of length 20. The data associated with the double precision control parameter descriptor is a double precision array of length 5. Their contents are described in Section B.1.5.

The user may ask MUMPS to return auxiliary integer and/or double precision information about the execution of MUMPS. The integer information (an array of length 20 and described in Section B.1.7) can be requested by specifying the descriptor

Auxiliary integer information (package-data descriptor) [optional]

```
pd(PSL_TYPE) = MPI_INTEGER
pd(PSL_NAME) = PSL_PARAM
pd(PSL_NROW) = 20
pd(PSL_NCOL) = 1
pd(PSL_LDIM) = pd(PSL_NROW)
pd(PSL_FORM) = PSL_TABLE
```

The double precision information descriptor is the same except that pd(PSL_TYPE) must be MPI_DOUBLE_PRECISION.

A.6 Descriptors for the mapping phase

When MUMPS is invoked, the PARASOL mapping phase must be given the pattern of a sparse matrix. The PARASOL interface accepts input data in sparse matrix format or in elemental format as defined in [32]. Therefore, one of the two following matrix data descriptors must *always* be specified by the user. If a matrix descriptor is not given, a contract will not be established.

Sparse matrix format (user-data descriptor) [required]

```
ud(PSL_NAME) = PSL_MATRIX
ud(PSL_ATTR) = PSL_SYMMETRIC ! or PSL_UNSYMMETRIC
ud(PSL_FORM) = PSL_SPARSEMAT
ud(PSL_NROW) = "order of matrix"
ud(PSL_NCOL) = ud(PSL_NROW)
ud(PSL_NVAL) = "number of matrix entries"
ud(PSL_ROW)  = 0
ud(PSL_COL)  = 0
ud(PSL_VAL)  = PSL_NOVAL
```

If ud(PSL_ATTR) = PSL_SYMMETRIC, the matrix supplied by the user is symmetric (in value and structure). If ud(PSL_ATTR) = PSL_UNSYMMETRIC, the matrix is unsymmetric. The data that is associated with this descriptor must be available in compressed column major order as specified in [32].

Note that, although the interface to MUMPS supports the sparse matrix format, the actual data passed to MUMPS is in coordinate form (see Section B.1.1). The interface performs this conversion.

Elemental matrix format (user-data descriptor) [required]

```
ud(PSL_NAME) = PSL_MATRIX
ud(PSL_ATTR) = PSL_SYMMETRIC  ! or PSL_UNSYMMETRIC
ud(PSL_FORM) = PSL_ELEMENTMAT
ud(PSL_NROW) = "order of matrix"
ud(PSL_NCOL) = ud(PSL_NROW)
ud(PSL_NELT) = "number of elemental matrices"
ud(PSL_NVAR) = "sum of dimensions of individual elements"
ud(PSL_NVAL) = "number of values in elements"
ud(PSL_ROW)  = 0
ud(PSL_COL)  = 0
ud(PSL_VAL)  = PSL_NOVAL
```

The elemental data that is associated with this descriptor must be available in dense column major order as specified in [32]. If the matrix is symmetric (ud(PSL_ATTR) = PSL_SYMMETRIC), only the lower triangular part of the elements (including the diagonal) must be given.

The fields PSL_ROW and PSL_COL specify that the pattern of the matrix must be passed to MUMPS. The field PSL_VAL specifies that values are not passed to MUMPS (even if they are provided by the user).

For each matrix descriptor, the field PSL_ATTR must be compatible with the instance descriptor field PSL_CONF+1.

If the user wants that MUMPS uses a pre-calculated ordering of the variables during the analysis, the following table descriptor must be specified:

User-defined ordering (user-data descriptor) [optional]

```
ud(PSL_TYPE) = PSL_INTEGER
ud(PSL_NAME) = PSL_VAR2NEW
ud(PSL_NROW) = "order of matrix"
ud(PSL_NCOL) = 1
ud(PSL_LDIM) = ud(PSL_NROW)
ud(PSL_FORM) = PSL_TABLE
```

The data associated with this descriptor is an integer array of length the order of the matrix. This array must specify for each variable its rank in the pivot sequence. If the PSL_NROW field of this descriptor is not equal to the same field of the matrix descriptor, a contract will not be established.

List of variables in the Schur complement matrix (user-data descriptor) [optional]

```
    ud(PSL_TYPE) = MPI_INTEGER
    ud(PSL_NAME) = PSL_MUMPS_VARSCHUR
    ud(PSL_NROW) = "order of Schur complement matrix"
    ud(PSL_NCOL) = 1
    ud(PSL_LDIM) = ud(PSL_NROW)
    ud(PSL_FORM) = PSL_TABLE)
```

The data associated with this descriptor is an integer array of dimension the order of the desired Schur complement matrix. The i-th component of this array must contain the i-th variable in the Schur complement matrix.

A.7 Descriptors for the solution phase

The matrix descriptor that was used during the mapping phase, must again be provided by the user for the solution phase, but now:

```
  ! new field
    ud(PSL_TYPE) = MPI_DOUBLE_PRECISION
  ! modified fields
    ud(PSL_ROW)  = PSL_KNOWN
    ud(PSL_COL)  = PSL_KNOWN
    ud(PSL_VAL)  = 0          ! or PSL_KNOWN
```

The field PSL_TYPE specifies that the data is provided in double precision format. The fields PSL_ROW and PSL_COL specify that the pattern of the matrix is already known by MUMPS and is not passed at the solution phase. If ud(PSL_VAL) = 0, the matrix values are passed to MUMPS. If ud(PSL_VAL) = PSL_KNOWN, the values must have been passed to MUMPS in a previous solution phase with the same instance, and now no values are passed (even if they are provided by the user).

If, besides factorization of the matrix, the user also wishes to solve a system of equations, the following right-hand side descriptor must be supplied by the user:

Right-hand side (user-data descriptor) [optional]

```
    ud(PSL_TYPE) = MPI_DOUBLE_PRECISION
    ud(PSL_NAME) = PSL_RHSIDE
    ud(PSL_ATTR) = PSL_RIGHT
    ud(PSL_NROW) = "order of matrix"
    ud(PSL_NVEC) = 1
    ud(PSL_LDIM) = ud(PSL_NROW)
    ud(PSL_FORM) = PSL_DENSEVEC
```

The data associated with this descriptor is a double precision array of length the order of the matrix that contains the right-hand side vector. If the PSL_NROW field of this descriptor is not equal to the same field of the matrix descriptor, a contract will not be established.

The solution to the system of equations is returned through the interface only when the following solution descriptor is specified:

Solution vector (package-data descriptor) [optional]

```
pd(PSL_TYPE) = MPI_DOUBLE_PRECISION
pd(PSL_NAME) = PSL_SOLUTION
pd(PSL_ATTR) = PSL_RIGHT
ud(PSL_NROW) = "order of matrix"
pd(PSL_NVEC) = 1
pd(PSL_LDIM) = pd(PSL_NROW)
pd(PSL_FORM) = PSL_DENSEVEC
```

The data associated with this descriptor is a double precision array of length the order of the matrix in which the solution vector will be returned. If the `PSL_NROW` field of this descriptor is not equal to the same field of the matrix descriptor, a contract will not be established. Furthermore, if the solution descriptor is provided by the user, the right-hand side descriptor must also be provided.

If the user wants `MUMPS` to return the deficiency (an integer) of the matrix, the following descriptor must be provided:

Matrix deficiency (package-data descriptor) [optional]

```
pd(PSL_TYPE) = MPI_INTEGER
pd(PSL_NAME) = PSL_MUMPS_DEFICIENCY
pd(PSL_NROW) = 1
pd(PSL_NCOL) = 1
pd(PSL_LDIM) = pd(PSL_NROW)
pd(PSL_FORM) = PSL_TABLE)
```

If the user also wants a basis for the null space of the (rank-deficient) matrix, the following descriptor must be provided:

Null space basis (package-data descriptor) [optional]

```
pd(PSL_TYPE) = MPI_DOUBLE_PRECISION
pd(PSL_NAME) = PSL_MUMPS_NULLSPACE
pd(PSL_NROW) = "order of matrix"
pd(PSL_NCOL) = "deficiency of matrix"
pd(PSL_LDIM) = pd(PSL_NROW)
pd(PSL_FORM) = PSL_TABLE
```

The data associated with this descriptor is a double precision array of dimension the order of the matrix times the deficiency. The null space descriptor can only be used when the deficiency of the matrix is known. Therefore, the null space descriptor must always be accompanied by the deficiency descriptor. If the null space descriptor is provided by the user but not the deficiency descriptor, a contract will not be established. The `PSL_NROW` field of the null space descriptor must be equal to the same field of the matrix descriptor.

Optionally, the user may provide a row and column scaling vector that `MUMPS` will use to scale the matrix values. For the row scaling, the following descriptor must be given:

Scaling vector (user-data descriptor) [optional]

```
ud(PSL_TYPE) = MPI_DOUBLE_PRECISION
ud(PSL_NAME) = PSL_MUMPS_ROWSCALE
ud(PSL_NROW) = "order of matrix"
ud(PSL_NCOL) = 1
ud(PSL_LDIM) = ud(PSL_NROW)
ud(PSL_FORM) = PSL_TABLE
```

The data associated with this descriptor is a double precision array of dimension the order of the matrix. The i-th component of the array must contain the scaling factor for row i of the matrix.

The descriptor for the column scaling is the same except that `ud(PSL_NAME) = PSL_MUMPS_COLSCALE`. The j-th component of the associated data array must contain the scaling factor for column j of the matrix.

Either no or both scaling descriptors must be present, otherwise a contract will not be established. Furthermore, the `PSL_NROW` fields of the scaling descriptors must be equal to the same field of the matrix descriptor.

Schur complement matrix (package-data descriptor) [optional]

```
pd(PSL_TYPE) = MPI_DOUBLE_PRECISION
pd(PSL_NAME) = PSL_MUMPS_MATSCHUR
pd(PSL_NROW) = "order of Schur complement matrix"
pd(PSL_NCOL) = pd(PSL_NROW)
pd(PSL_LDIM) = pd(PSL_NROW)
pd(PSL_FORM) = PSL_TABLE
```

The data associated with this descriptor is a two-dimensional double precision array that contains the Schur complement matrix. The Schur complement matrix descriptor can only be used when the Schur complement variables were given to the mapping phase. The `PSL_NROW` field of the Schur complement matrix descriptor must be equal to the same field of the Schur variables descriptor.

A.8 Additional functionality provided through the PARASOL interface

Besides the approximate minimum ordering algorithm built in `MUMPS`, the `PTD` currently implements 4 external ordering strategies for the `MUMPS` solver based on graph partitioning package METIS [42]. These orderings are computed by the PARASOL interface during the mapping phase and entered to the analysis phase of `MUMPS`. The seven orderings options currently available are:

- AMD - Approximate Minimum Degree ordering. This is the default (internal) ordering generated by `MUMPS` if no user-defined ordering was specified by the user or if an error occurred when an external ordering was computed.
- METIS based orderings. Multilevel spectral bisection from the METIS library is combined with minimum degree heuristics on the subgraphs. The user can define the maximum size of the subgraphs and the balancing factor for the size of the subgraphs at each bisection step. This allows the user to control the size/depth of the bisection tree. The following options are available:

  `OE0` - Multilevel bisection and Multiple Minimum Degree (MMD) on the subgraphs. It uses the OEMETIS code.

  `OE1` - Same as the `OE0` option, but AMD is used on the subgraphs.

  `OE2` - Same as the `OE0` option, but HALO-AMD is used on the subgraphs. Since the bisection is implemented in a recursive manner, only partial information on the boundaries of the subgraphs is available. Separators constructed at the last level of the bisection tree are used for the definition of the boundary in HALO-AMD. The complete boundary information is used in algorithm `RL0` (described below).

  `ON0` - Multilevel bisection based on the ONMETIS package. It is identical to the ONMETIS code except that AMD rather than MMD is used on the subgraphs.
- User-defined ordering. The user specifies an integer array that specifies for each variable in the problem its rank in the pivot sequence.

Based on our experience during the project, we recommend the use of `ON0` (ONMETIS). If the user has computed a good ordering, the user-defined ordering option is recommended.

B Fortran 90 interface to MUMPS

In the Fortran 90 interface, there is a single user callable subroutine called MUMPS that has a single parameter mumps_par of Fortran 90 derived datatype STRUC_MUMPS. MPI must be initialized by the user before the first call to MUMPS. The calling sequence looks as follows:

```
INCLUDE 'mpif.h'
INCLUDE 'mumps_struc.h'
...
INTEGER IERR
TYPE (STRUC_MUMPS) :: mumps_par
...
CALL MPI_INIT(IERR)
...
CALL MUMPS( mumps_par )
...
CALL MPI_FINALIZE(IERR)
```

The datatype STRUC_MUMPS holds all the data for the problem. It has many components, only some of which are of interest to the user. The other components are internal to the package (and could be declared private). Some of the components must only be defined on the host. Others must be defined on all processors. The file mumps_struc.h defines the derived datatype and must always be included in the program that calls MUMPS. The file mumps_root.h, which is included in mumps_struc.h, defines the datatype for an internal component root, and must also be available at compilation time. Components of the structure STRUC_MUMPS that are of interest to the user are shown in Figure 18.

The interface to MUMPS consists in calling the subroutine MUMPS with the appropriate parameters set in mumps_par.

B.1 Input and output parameters

Components of the structure that must be set by the user are:

mumps_par%JOB (integer) must be initialized by the user on all processors before a call to MUMPS. It controls the main action taken by MUMPS. It is not altered.

JOB=–1 initializes an instance of the package. This must be called before any other call to the package concerning that instance. It sets default values for other components of STRUC_MUMPS, which may then be altered before subsequent calls to MUMPS. Note that three components of the structure must always be set by the user (on all processors) before a call with JOB=–1. These are

- mumps_par%COMM,
- mumps_par%SYM, and
- mumps_par%PAR.

JOB=–2 destroys an instance of the package. All data structures associated with the instance, except those provided by the user in mumps_par, are deallocated. It should be called by the user only when no further calls to MUMPS with this instance are required. It should be called before a further JOB=–1 call on the same instance.

JOB=1 performs the analysis. It uses the pattern of the matrix **A** input by the user. The following components of the structure define the matrix pattern and must be set by the user (on the host only) before a call with JOB=1:

- mumps_par%N, mumps_par%NZ, mumps_par%IRN, and mumps_par%JCN if the user wishes to input the structure of the matrix in assembled format (ICNTL(5)=0, and ICNTL(18) $\neq$ 3),
- mumps_par%N, mumps_par%NELT, mumps_par%ELTPTR, and mumps_par%ELTVAR if the user wishes to input the matrix in elemental format (ICNTL(5)=1).

```
      INCLUDE 'mumps_root.h'
      TYPE STRUC_MUMPS
        SEQUENCE
C This structure contains all parameters for the
C interface to the user, plus internal information
C *****************
C INPUT PARAMETERS
C *****************
C    ------------------
C    Problem definition
C    ------------------
C    Solver (SYM=0 Unsymmetric, SYM=1 Sym. Positive Definite, SYM=2 General Symmetric)
C    Type of parallelism (PAR=1 host working, PAR=0 host not working)
         INTEGER SYM, PAR, JOB
C    ------------------
C    Control parameters
C    ------------------
         INTEGER ICNTL(20)
         DOUBLE PRECISION CNTL(5)
C    --------------------
C    Order of Input matrix
C    --------------------
         INTEGER N
C    ----------------------------------------
C    Assembled input matrix : User interface
C    ----------------------------------------
         INTEGER NZ
         DOUBLE PRECISION, DIMENSION(:), POINTER :: A
         INTEGER, DIMENSION(:), POINTER :: IRN, JCN
C        --------------------------------
C        Case of distributed matrix entry
C        --------------------------------
         INTEGER NZ_loc
         INTEGER, DIMENSION(:), POINTER :: IRN_loc, JCN_loc
         DOUBLE PRECISION, DIMENSION(:), POINTER :: A_loc
C    ----------------------------------------
C    Unassembled input matrix: User interface
C    ----------------------------------------
         INTEGER NELT
         INTEGER, DIMENSION(:), POINTER :: ELTPTR, ELTVAR
         DOUBLE PRECISION, DIMENSION(:), POINTER :: A_ELT
C    -----------------
C    MPI Communicator
C    -----------------
         INTEGER COMM
C    -------------------------------------------------
C    Ordering and scaling, if given by user (optional)
C    -------------------------------------------------
         INTEGER, DIMENSION(:), POINTER :: PERM_IN
         DOUBLE PRECISION, DIMENSION(:), POINTER :: COLSCA, ROWSCA
C ******************
C INPUT/OUTPUT data
C ******************
C    -------------------------------------------------------
C    RHS : on input it holds the right hand side
C          on output it always holds the assembled solution
C    -------------------------------------------------------
         DOUBLE PRECISION, DIMENSION(:), POINTER :: RHS
C **************************
C OUTPUT data and Statistics
C **************************
         INTEGER INFO(20)
         DOUBLE PRECISION RINFO(20)
C        Global information -- host only
         DOUBLE PRECISION RINFOG(20)
         INTEGER INFOG(20)
C    -----------------------------------------
C    Deficiency and null space basis (optional
C    -----------------------------------------
         INTEGER Deficiency
         DOUBLE PRECISION, DIMENSION(:,:), POINTER :: NULL_SPACE
C    -----------------
C    Schur
C    ------
         INTEGER SIZE_SCHUR
         INTEGER, DIMENSION(:), POINTER :: LISTVAR_SCHUR
         DOUBLE PRECISION, DIMENSION(:), POINTER :: SCHUR
C    -------------------------------------
C    Mapping potentially provided by MUMPS
C    -------------------------------------
         INTEGER, DIMENSION(:), POINTER :: MAPPING
      END TYPE STRUC_MUMPS
```

Figure 18: The components of the structure STRUC_MUMPS defined in mumps_struc.h that are of interest to the user.

(See Sections B.1.1-B.1.2.) These components should be passed unchanged when later calling the factorization (JOB=2) and solve (JOB=3) phases.
In the case of distributed assembled matrix,

- If ICNTL(18) = 1 or 2, the previous requirements hold except that IRN and JCN need not be passed unchanged to factorization phase.
- If ICNTL(18) = 3, the user should provide
  - N on the host
  - mumps_par%NZ_loc, mumps_par%IRN_loc and mumps_par%JCN_loc on all slave processors. Those should be passed unchanged to the factorization (JOB=2) and solve (JOB=3) phases.

See Section B.1.3 for more details and options for the distributed matrix entry.
In the analysis, MUMPS chooses pivots from the diagonal using a selection criterion to preserve sparsity. It uses the pattern of $\mathbf{A} + \mathbf{A}^T$ but ignores numerical values. It subsequently constructs subsidiary information for the numerical factorization (a JOB=2 call).
An option exists for the user to input the pivotal sequence (in array PERM_IN, ICNTL(7)=1, see below) in which case only the necessary information for a JOB=2 call will be generated.
For a call with JOB=1 on an assembled matrix, an integer array of size 2*NZ + 3*N + 1 is used as a temporary workspace for the analysis on the host. (For an elemental matrix, the size of this array is not known a priori, nut is generally smaller.) An integer component array IS1, of size 12*N, is allocated dynamically. It is transmitted to the factorization and solution phases (JOB=2 and JOB=3, respectively), and deallocated with JOB=–2.
A call to MUMPS with JOB=1 must be preceded by a call with JOB=–1 on the same instance.

JOB=2 performs the factorization. It uses the numerical values of the matrix **A** provided by the user and the information from the analysis phase (JOB=1) to factorize the matrix **A**.
If the matrix is centralized on the host (ICNTL(18)=0), the pattern of the matrix should be passed unchanged since the last call to the analysis phase (see JOB=1); the following components of the structure define the numerical values and must be set by the user (on the host only) before a call with JOB=2:

- mumps_par%A if the matrix is in assembled format (ICNTL(5)=0), or
- mumps_par%A_ELT if the matrix is in elemental format (ICNTL(5)=1).

If the initial matrix is distributed (ICNTL(5)=0 and ICNTL(18) $\neq$ 0), then the following components of the structure must be set by the user on all slave processors before a call with JOB=2:

- mumps_par%A_loc on all slave processors, and
- mumps_par%NZ_loc, mumps_par%IRN_loc and mumps_par%JCN_loc if ICNTL(18)=1 or 2. (For ICNTL(18)=3, NZ_loc, IRN_loc and JCN_loc have already been passed to the analysis step and must be passed unchanged.)

(See Sections B.1.1–B.1.2–B.1.3.) The actual pivot sequence used during the factorization may differ slightly from the sequence returned by the analysis if the matrix **A** is not diagonally dominant.
An option exists for the user to input scaling vectors or let MUMPS compute such vectors automatically (in arrays COLSCA/ROWSCA, ICNTL(8) $\neq$ 0, see below).
A call to MUMPS with JOB=2 must be preceded by a call with JOB=1 on the same instance.

JOB=3 performs the solution. It uses the right-hand side **x** provided by the user and the factors generated by the factorization (JOB=2) to solve a system of equations $\mathbf{Ax} = \mathbf{b}$ or $\mathbf{A}^T\mathbf{x} = \mathbf{b}$. The pattern and values of the matrix should be passed unchanged since the last call to the factorization phase (see JOB=2). The structure component mumps_par%RHS must be set by the user (on the host only) before a call with JOB=3. (See Section B.1.4.)
A call to MUMPS with JOB=3 must be preceded by a call with JOB=2 (or JOB=4) on the same instance.

JOB=4 combines the actions of JOB=1 with those of JOB=2. It must be preceded by a call to MUMPS with JOB=–1 on the same instance.

JOB=5 combines the actions of JOB=2 and JOB=3. It must be preceded by a call to `MUMPS` with JOB=1 on the same instance.

JOB=6 combines the actions of calls with JOB=1, 2, and 3. It must be preceded by a call to `MUMPS` with JOB=–1 on the same instance.

Consecutive calls with JOB=2 and consecutive calls with JOB=3 on the same instance are possible.

mumps_par%COMM (integer) must be set by the user on all processors before the initialization phase (JOB=–1) and must not be changed. It must be set to a valid MPI communicator that will be used for message passing inside `MUMPS`. It is not altered by `MUMPS`. The processor with rank 0 in this communicator is used by `MUMPS` as the **host** processor.

mumps_par%SYM (integer) must be initialized by the user on all processors and is accessed by `MUMPS` only during the initialization phase (JOB=–1). It is not altered by `MUMPS`. Possible values for SYM are:

0 **A** is unsymmetric

1 **A** is symmetric positive definite

2 **A** is general symmetric

mumps_par%PAR (integer) must be initialized by the user on all processors and is accessed by `MUMPS` only during the initialization phase (JOB=–1). It is not altered by `MUMPS`. Possible values for PAR are:

0 host is not involved in factorization/solve phases

1 host is involved in factorization/solve phases

If set to 0, the host will only hold the initial problem, perform symbolic computations during the analysis phase, distribute data, and collect results from other processors. If set to 1, the host will also participate in the factorization and solve phases. If the initial problem is large and memory is an issue, PAR = 1 is not recommended if the matrix is centralized on processor 0 because this can lead to memory imbalance, with processor 0 having a larger memory load than the other processors. Note that setting PAR to 1, and using only 1 processor, leads to a sequential code.

B.1.1 Centralized assembled matrix input

mumps_par%N (integer), mumps_par%NZ (integer), mumps_par%IRN (integer array pointer, dimension NZ), mumps_par%JCN (integer array pointer, dimension NZ), and mumps_par%A (double precision array pointer, dimension NZ) hold the matrix in assembled format. These components should be set by the user only on the host and only when ICNTL(5)=0:

- N is the order of the matrix **A**, N > 0. It is not altered by `MUMPS`.
- NZ is the number of entries being input, NZ > 0. It is not altered by `MUMPS`.
- IRN, JCN are integer arrays of length NZ containing the row and column indices, respectively, for the matrix entries. IRN and JCN are unchanged unless ICNTL(6)=1, in which case the original matrix is permuted to have a zero-free diagonal.
- A is a double precision array of length NZ. The user must set A(k) to the value of the entry in row IRN(k) and column JCN(k) of the matrix. A is not accessed when JOB=1. Duplicate entries are summed and any with IRN(k) or JCN(k) out-of-range are ignored. Note that, in the case of the symmetric solver, a diagonal nonzero a_{ii} is held as A(k)=a_{ii}, IRN(k)=JCN(k)=i, and a pair of off-diagonal nonzeros $a_{ij} = a_{ji}$ is held as A(k)=a_{ij} and IRN(k)=i, JCN(k)=j or vice-versa. Again, duplicate entries are summed and entries with IRN(k) or JCN(k) out-of-range are ignored.

The components N, NZ, IRN, and JCN describe the pattern of the matrix and must be set by the user before the analysis phase (JOB=1). Component A must be set before the factorization phase (JOB=2).

B.1.2 Element matrix input

mumps_par%N (integer), mumps_par%NELT (integer), mumps_par%ELTPTR (integer array pointer, dimension NELT+1), mumps_par%ELTVAR (integer array pointer, dimension ELTPTR(NELT+1)–1), and mumps_par%A_ELT (double precision array pointer) hold the matrix in elemental format. These components should be set by the user only on the host and only when ICNTL(5)=1:

- N is the order of the matrix **A**, N > 0. It is not altered by `MUMPS`.
- NELT is the number of elements being input, NELT > 0. It is not altered by `MUMPS`.
- ELTPTR is an integer array of length NELT+1. ELTPTR(j) points to the position in ELTVAR of the first variable in element j, and ELTPTR(NELT+1) must be set to the position after the last variable of the last element. It is not altered by `MUMPS`.
- ELTVAR is an integer array of length ELTPTR(NELT+1)–1 and must be set to the lists of variables of the elements. It is not altered by `MUMPS`. Those for element j are stored in positions ELTPTR(j), ..., ELTPTR(j+1)–1. Out-of-range variables are ignored.
- A_ELT is a double precision array. If N_p denotes ELTPTR(p+1)–ELTPTR(p), then the values for element j are stored in positions $K_\mathrm{j}+1, \ldots, K_\mathrm{j}+L_\mathrm{j}$, where
  - $K_\mathrm{j} = \sum_{\mathrm{p}=1}^{\mathrm{j}-1} {N_\mathrm{p}}^2$, and $L_\mathrm{j} = {N_\mathrm{j}}^2$ in the unsymmetric case (SYM = 0)
  - $K_\mathrm{j} = \sum_{\mathrm{p}=1}^{\mathrm{j}-1} (N_\mathrm{p}\cdot(N_\mathrm{p}+1))/2$, and $L_\mathrm{j} = (N_\mathrm{j}\cdot(N_\mathrm{j}+1))/2$ in the symmetric case (SYM $\neq 0$). Only the lower triangular part is stored.

  Values within each element are stored column-wise. Values corresponding to out-of-range variables are ignored and values corresponding to duplicate variables within an element are summed. A_ELT is not accessed when JOB = 1. Note that, although the elemental matrix may be symmetric or unsymmetric in value, its structure is always symmetric.

The components N, NELT, ELTPTR, and ELTVAR describe the pattern of the matrix and must be set by the user before the analysis phase (JOB=1). Component A_ELT must be set before the factorization phase (JOB=2).

B.1.3 Distributed assembled matrix input

We offer several options, defined by the control parameter ICNTL(18) described in Section B.1.5. The following components of the structure define the distributed assembled matrix input. They are valid for nonzero values of ICNTL(18), otherwise the user should refer to Section B.1.1.

mumps_par%N (integer), mumps_par%NZ (integer), mumps_par%IRN (integer array pointer, dimension NZ), mumps_par%JCN (integer array pointer, dimension NZ), mumps_par%IRN_loc (integer array pointer, dimension NZ_loc), mumps_par%JCN_loc (integer array pointer, dimension NZ_loc), mumps_par%A_loc (double precision array pointer, dimension NZ_loc), and mumps_par%MAPPING (integer array, dimension NZ).

- N is the order of the matrix **A**, N > 0. It must be set on the host before analysis. It is not altered by `MUMPS`.
- NZ is the number of entries being input in the definition of **A**, NZ > 0. It must be defined on the host before analysis if ICNTL(18) = 1, or 2.
- IRN, JCN are integer arrays of length NZ containing the row and column indices, respectively, for the matrix entries. They must be defined on the host before analysis if ICNTL(18) = 1, or 2. They can be deallocated by the user just after the analysis.
- NZ_loc is the number of entries local to a processor. It must be defined on all processors in the case of the working host model of parallelism (PAR=1), and on all processors except the host in the case of the non-working host model of parallelism (PAR=0), before analysis if ICNTL(18) = 3, and before factorization if ICNTL(18) = 1 or 2.
- IRN_loc, JCN_loc are integer arrays of length NZ_loc containing the row and column indices, respectively, for the matrix entries. They must be defined on all processors if PAR=1, and on all processors except the host if PAR=0, before analysis if ICNTL(18) = 3, and before factorization if ICNTL(18) = 1 or 2.

- A_loc is a double precision array of dimension NZ_loc that must be defined before the factorization phase (JOB=2) on all processors if PAR = 1, and on all processors except the host if PAR = 0. The user must set A_loc(k) to the value in row IRN_loc(k) and column JCN_loc(k).
- MAPPING is an integer array of size NZ which is returned by `MUMPS` on the host after the analysis phase as an indication of a preferred mapping if ICNTL(18) = 1. In that case, MAPPING(i) = IPROC means that entry IRN(i), JCN(i) should be provided on processor with rank IPROC in the `MUMPS` communicator.

We recommend the use of option ICNTL(18)=3 because it is the simplest and most flexible option and because it is in general almost as efficient as the more sophisticated (but more complicated for the user) option ICNTL(18)=1.

B.1.4 Right-hand side and solution vector

mumps_par%RHS (double precision array pointer, dimension N) is a double precision array that must be set by the user on the host only, before a call to `MUMPS` with JOB = 3, 5, or 6. On entry, RHS(i) must hold the i-th component of the right-hand side of the equations being solved. On exit, RHS(i) will hold the i-th component of the solution vector.

B.1.5 Control parameters

On exit from the initialization call (JOB=-1), the control parameters are set to default values. If the user wishes to use values other than the defaults, the corresponding entries in mumps_par%ICNTL and mumps_par%CNTL should be reset after this initial call and before the call in which they are used.

mumps_par%ICNTL is an integer array of dimension 20.

ICNTL(1) is the output stream for error messages. If it is negative or zero, these messages will be suppressed. Default value is 6.

ICNTL(2) is the output stream for diagnostic printing, statistics, and warning messages. If it is negative or zero, these messages will be suppressed. Default value is 0.

ICNTL(3) is the output stream for global information, collected on the host. If it is negative or zero, these messages will be suppressed. Default value is 6.

ICNTL(4) is the level of printing for error, warning, and diagnostic messages. Maximum value is 4 and default value is 2 (errors and warnings printed).

ICNTL(5) has default value 0 and is only accessed by the host and only during the analysis phase. If ICNTL(5) = 0, the input matrix must be given in assembled format in the structure components N, NZ, IRN, JCN, and A (or NZ_loc, IRN_loc, JCN_loc, A_loc, see Section B.1.3). If ICNTL(5) = 1, the input matrix must be given in elemental format in the structure components N, NELT, ELTPTR, ELTVAR, and A_ELT.

ICNTL(6) has default value 0 and is only accessed by the host and only during the analysis phase. If ICNTL(6) = 1, a maximum transversal algorithm is performed. Column permutations are then applied to the original matrix to get a zero-free diagonal. The user is advised to set ICNTL(6)=1 only when the matrix is very unsymmetric. If the input matrix is symmetric (SYM $\neq$ 0), or in elemental format (ICNTL(5)=1), or distributed (ICNTL(18) $\neq$ 0), or if the ordering is provided by the user (ICNTL(7)=1), then the value of ICNTL(6) is ignored.

ICNTL(7) has default value 0 and is only accessed by the host and only during the analysis phase. If ICNTL(7) = 1, the pivot order in PERM_IN (set by the user) is used. Otherwise, the pivot order will be chosen automatically.

ICNTL(8) has default value 0 and is only accessed by the host and only during the factorization phase. It is used to describe the scaling strategy. If the initial matrix is distributed (ICNTL(18) $\neq$ 0 and ICNTL(5) = 0), then the value of ICNTL(8) is ignored (no scaling). If ICNTL(8) = –1, the user must provide scaling vectors in the arrays COLSCA and ROWSCA. If ICNTL(8) = 0, no scaling is performed, and arrays COLSCA/ROWSCA are not used. If ICNTL(8) $\neq$ 0, the package allocates the arrays COLSCA/ROWSCA and computes one of the following scalings:

- ICNTL(8)=1: Diagonal scaling,
- ICNTL(8)=2: Scaling based on Harwell Subroutine Library code MC29,
- ICNTL(8)=3: Column scaling,
- ICNTL(8)=4: Row and column scaling,
- ICNTL(8)=5: Scaling based on MC29 followed by column scaling,
- ICNTL(8)=6: Scaling based on MC29 followed by row and column scaling.

If the input matrix is symmetric (SYM $\neq$ 0), then only options –1, 0, and 1 are allowed and other options are treated as 0; if ICNTL(8)=–1, the user should ensure that the array ROWSCA is equal to the array COLSCA. If the input matrix is in elemental format (ICNTL(5) = 1), then only option –1 is allowed and other options are treated as 0.

ICNTL(9) has default value 1 and is used only by the host during the solve phase. If ICNTL(9) = 1, $\mathbf{Ax} = \mathbf{b}$ is solved, otherwise, $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ is solved.

ICNTL(10) has default value 0 and is only accessed by the host and only during the solve phase. It corresponds to the maximum number of steps of iterative refinement. If ICNTL(10) = 0, iterative refinement is not performed.

ICNTL(11) has default value 0 and is only accessed by the host and only during the solve phase. A positive value will return (on the host) the infinite norm of the input matrix, the computed solution, and the scaled residual in RINFOG(4) to RINFOG(6), respectively, a backward error estimate in RINFOG(7) and RINFOG(8), and an estimate for the error in the solution in RINFOG(9).

Note that, although the following ICNTL entries (12 to 17) control the efficiency of the factorization and solve phases, they involve preprocessing work performed during analysis and must thus be set at the analysis phase.

ICNTL(12) has default value 0 and is only accessed by the host and only during the analysis phase. If ICNTL(12) = 0, node level parallelism is switched on, otherwise only tree parallelism will be used during factorization/solve phases.

ICNTL(13) has default value 0 and is only accessed by the host and only during the analysis phase. If ICNTL(13) = 0, use of ScaLAPACK will be made for the root node if the size of the root node of the assembly tree is larger than a machine-dependent minimum size. Otherwise, the root node of the tree will be processed sequentially.

ICNTL(14) is accessed by the host both during the analysis and the factorization phases. It corresponds to the percentage increase in the estimated working space. When significant extra fill-in is caused by numerical pivoting, larger values of ICNTL(14) may help use the real working space more efficiently. Default value is 20 % except for symmetric positive definite matrices (SYM=1) where default value is 10 %.

ICNTL(15) has default value 0 and is only accessed by the host during the analysis phase. If ICNTL(15) = 0, the criterion for mapping the top of the tree to the processors is based on memory balance only. Otherwise, mapping is based on the number of flops.

ICNTL(16) has default value 0 and is only accessed by the host. During the analysis phase, a valid positive value prepares the data for later use of the null space functionality. If ICNTL(16) is negative or zero, the null space feature will be disabled during the factorization phase.

During the factorization phase, if ICNTL(16) was positive for analysis, values of ICNTL(16) have the following meaning.

0 : no null space analysis is performed

1,3,5,7,9 : rank detection only

2,4,6,8,10 : rank detection and null space basis.

The deficiency of the matrix is returned in mumps_par%Deficiency (on all processors) after the factorization phase. If a null space basis was required, it is returned on the host in mumps_par%NULL_SPACE, a double precision array pointer of size N $\times$ Deficiency.

The following strategies have been implemented (see [9]):

1,2 : QR with partial pivoting,

3,4 : QR with partial pivoting improved by Chan algorithm,

5,6 : LU with partial pivoting,

7,8 : an improved strategy based on LU with partial pivoting.

9,10 : ICNTL(17) is used as the exact size of the (pseudo-)null space.

Options 3 to 8 although implemented and ready for experimentation are not currently available to the user, and are treated as 0.

ICNTL(17) has default value 0 and is only accessed by the host during the factorization phase if rank detection is effective (ICNTL(16) $\neq$ 0). In such cases,

- if $0 <$ ICNTL(16) ≤ 8, ICNTL(17) should hold an estimate of the maximum size of the null space. If ICNTL(17) is negative or zero, MUMPS assumes that the user has no information about the null space size.
- if ICNTL(16) is 9 or 10, ICNTL(17) should hold the exact size of the null space (or pseudo-null space).

ICNTL(18) has default value 0 and is only accessed by the host during the analysis phase, if the matrix format is assembled (ICNTL(5) = 0). ICNTL(18) defines the strategy for the distributed input matrix. Possible values are:

- 0: input matrix is centralized on the host. This is the default, see Section B.1.1.
- 1: user provides the structure of the matrix on the host at analysis, MUMPS returns a mapping and user should provide the matrix distributed according to the mapping.
- 2: user provides the structure of the matrix on the host at analysis, and the distributed matrix on all slave processors at factorization. Any distribution is allowed.
- 3: user directly provides the distributed matrix input both for analysis and factorization.

For options 1, 2, 3, see Section B.1.3 for more details on the input/output parameters to MUMPS. For flexibility and performance issues, option 3 is recommended.

ICNTL(19) has default value 0 and is only accessed by the host during the analysis phase. If ICNTL(19) $\neq$ 0 then the Schur matrix will be returned to the user. The user must set on entry on the host node (before analysis):

- the integer variable SIZE_SCHUR to the size of the Schur matrix,
- the integer array pointer LISTVAR_SCHUR to the list of indices of the Schur matrix.

On output to the factorization phase, and on the host node, the 1-dimensional pointer array SCHUR, of length SIZE_SCHUR^2, holds the (dense) Schur matrix of order SIZE_SCHUR. Note that the order of the indices in the Schur matrix is identical to the order provided by the user in LISTVAR_SCHUR and that the Schur matrix is stored **by rows**. It the matrix is symmetric then the lower triangular part of the Schur matrix is provided (**by rows**).

The partial factorization of the interior variables can then be exploited to perform a solve phase (transposed matrix or not). Note that the right-hand side (RHS) provided on input must still be of size N even if only the N-SIZE_SCHUR indices will be considered and if only N-SIZE_SCHUR indices of the solution will be relevant to the user.

Finally note that since the Schur complement can be viewed as a partial factorization of the global matrix (with partial ordering of the variables provided by the user) the following options of MUMPS are incompatible with the Schur option: null space, maximum transversal, ordering given, scaling, iterative refinement, error analysis.

ICNTL(20) is not used in the current version.

mumps_par%CNTL is a double precision array of dimension 5.

CNTL(1) is the relative threshold for numerical pivoting. It forms a trade-off between preserving sparsity and ensuring numerical stability during the factorization. In general, a larger value of CNTL(1) increases fill-in but leads to a more accurate factorization. If CNTL(1) is nonzero, numerical pivoting will be performed. If CNTL(1) is zero, no such pivoting will be performed and the subroutine will fail if a zero pivot is encountered. If the matrix is diagonally dominant, then setting CNTL(1) to zero will decrease the factorization time while still providing a stable decomposition. If the code is called for unsymmetric or general symmetric matrices, CNTL(1) has default value 0.01. For symmetric positive definite matrices, numerical pivoting is suppressed and the default value is 0.0. Values less than 0.0 are treated as 0.0, values greater than 1.0 are treated as 1.0.

CNTL(2) - CNTL(5) are not used in the current version.

B.1.6 Optional input parameters

mumps_par%COLSCA, mumps_par%ROWSCA (double precision array pointers, dimension N) are optional scaling arrays required only by the host. If a scaling is provided by the user (ICNTL(8)=–1), it must be allocated and initialized by the user on the host, before a call to the factorization phase (JOB=2). It should be passed unchanged to the solve phase (JOB=3). If the initial matrix is symmetric in value, ROWSCA must be equal to COLSCA to preserve the symmetry.

mumps_par%PERM_IN (integer array pointer, dimension N) must be allocated and initialized by the user on the host if ICNTL(7)=1. It is accessed during the analysis (JOB=1) and PERM_IN(i), i=1, ..., N must hold the position of variable i in the pivot order. Note that, even when the ordering is provided by the user, the analysis must still be performed before numerical factorization.

mumps_par%LISTVAR_SCHUR (integer array pointer, dimension mumps_par%SIZE_SCHUR must be allocated and initialized by the user on the host if ICNTL(19)$\neq$ 0. It is not altered by `MUMPS`. It is accessed during analysis (JOB=1) and LISTVAR_SCHUR(i), i=1, ..., SIZE_SCHUR must hold the i^{th} index of the Schur matrix.

mumps_par%MAXIS and mumps_par%MAXS (integers) are defined, for each processor, as the size of the integer and the real workspaces respectively required for factorization and/or solve. On return from analysis (JOB = 1), INFO(7) (resp. INFO(8)) returns the minimum value for MAXIS to the user. If the user has reason to believe that significant numerical pivoting will be required, it may be desirable to choose a higher value for MAXIS (resp. MAXS) than output from the analysis. At the beginning of the factorization, MAXIS (resp. MAXS) is set to the maximum of the estimate computed by the analysis and the value supplied by the user. An integer array IS of size MAXIS and a double precision array S of size MAXS are then dynamically allocated and used during the factorization and solve phases to hold the factors and various contribution blocks.

B.1.7 Information parameters

The parameters described in this section are returned by `MUMPS` and hold information that may be of interest to the user. Some of the information is available on each processor and some only on the host. If an error is detected (see Section B.2), the information may be incomplete.

Information available on each processor

The arrays mumps_par%RINFO and mumps_par%INFO are available on each process.

mumps_par%RINFO is a double precision array of dimension 20. It contains the following local information on the execution of `MUMPS`:

RINFO(1) - after analysis: The estimated number of floating-point operations on the processor for the elimination process.

RINFO(2) - after factorization: The number of floating-point operations on the processor for the assembly process.

RINFO(3) - after factorization: The number of floating-point operations on the processor for the elimination process.

RINFO(4) - RINFO(20) are not used in the current version.

mumps_par%INFO is an integer array of dimension 20. It contains the following local information on the execution of `MUMPS`:

INFO(1) is 0 if the call to `MUMPS` was successful, negative if an error occurred (see Section B.2).

INFO(2) holds additional information about the error. If INFO(1)=–1, INFO(2) is the processor number (in communicator mumps_par%COMM) on which the error was detected.

INFO(3) - after analysis: Estimated real space needed on the processor for factors.

INFO(4) - after analysis: Estimated integer space needed on the processor for factors.

INFO(5) - after analysis: Estimated maximum front size on the processor.

INFO(6) - after analysis: Number of nodes in the complete tree. The same value is returned on all processors.

INFO(7) - after analysis: Minimum value of MAXIS estimated by the analysis phase to run the numerical factorization successfully.

INFO(8) - after analysis: Minimum value of MAXS estimated by the analysis phase to run the numerical factorization successfully.

INFO(9) - after factorization: Size of the real space used on the processor to store the LU factors.

INFO(10) - after factorization: Size of the integer space used on the processor to store the LU factors.

INFO(11) - after factorization: Order of the largest frontal matrix processed on the processor.

INFO(12) - after factorization: Number of off-diagonal pivots encountered on the processor.

INFO(13) - after factorization: The number of uneliminated variables, corresponding to delayed pivots, sent to the father. If a delayed pivot is subsequently passed to the father of the father, it is counted a second time.

INFO(14) - after factorization: Number of memory compresses on the processor.

INFO(15) - after analysis: estimated total size (in millions of bytes) of all `MUMPS` internal data for running numerical factorization.

INFO(16) - after factorization: total size (in millions of bytes) of all `MUMPS` internal data used during numerical factorization.

INFO(17) - INFO(20) are not used in the current version.

Information available on the host

The arrays mumps_par%RINFOG and mumps_par%INFOG are significant only on the host.

mumps_par%RINFOG is a double precision array of dimension 20. It contains the following global information on the execution of `MUMPS`:

RINFOG(1) - after analysis: The estimated number of floating-point operations (on all processors) for the elimination process.

RINFOG(2) - after factorization: The total number of floating-point operations (on all processors) for the assembly process.

RINFOG(3) - after factorization: The total number of floating-point operations (on all processors) for the elimination process.

RINFOG(4) to RINFOG(9) - after solve with error analysis: Only returned on the host process if ICNTL(11) $\neq 0$. See description of ICNTL(11).

RINFOG(10) - RINFOG(20) are not used in the current version.

mumps_par%INFOG is an integer array of dimension 20. It contains the following global information on the execution of MUMPS:

INFOG(1) is 0 if the call to MUMPS was successful, negative if an error occurred (see Section B.2).

INFOG(2) holds additional information about the error.

The difference between INFOG(1:2) and INFO(1:2) is that INFOG(1:2) is the same on all processors. It has the value of INFO(1:2) of the processor which returned with smallest INFO(1) value. For example, if processor p returns with INFO(1)=-13, and INFO(2)=10000, then all other processors will return with INFOG(1)=-13 and INFOG(2)=10000, but still INFO(1)=-1 and INFO(2)=p.

INFOG(3) - after analysis: Total estimated real workspace for factors on all processors.

INFOG(4) - after analysis: Total estimated integer workspace for factors on all processors.

INFOG(5) - after analysis: Estimated maximum front size in the complete tree.

INFOG(6) - after analysis: Number of nodes in the complete tree.

INFOG(7:9) : not significant.

INFOG(10) - after factorization: Total integer space to store LU factors.

INFOG(11) - after factorization: Order of largest frontal matrix.

INFOG(12) - after factorization: Total number of off-diagonal pivots.

INFOG(13) - after factorization: Total number of delayed pivots.

INFOG(14) - after factorization: Total number of memory compresses.

INFOG(15) - after solution: Number of steps of iterative refinement.

INFOG(16) - after analysis: Estimated size (in million of bytes) of all MUMPS internal data for running factorization: value on the most memory consuming processor.

INFOG(17) - after analysis: Estimated size (in millions of bytes) of all MUMPS internal data for running factorization: sum over all processors.

INFOG(18) - after factorization: Size in millions of bytes of all MUMPS internal data during factorization: value on the most memory consuming processor.

INFOG(19) - after factorization: Size in millions of bytes of all MUMPS internal data during factorization: sum over all processors.

INFOG(20) - after analysis: Estimated number of entries in the factors.

B.2 Error diagnostics

MUMPS uses the following mechanism to process errors that may occur during the parallel execution of the code. If, during a call to MUMPS, an error occurred on a processor, this processor informs all the other processors before they return from the call. In parts of the code where messages are sent asynchronously (for example factorization and solve phases), the processor on which the error occurred sends a message to the other processors with a specific error tag. On the other hand, if the error occurs in a subroutine that does not use asynchronous communication, the processor propagates the error to the other processors after the subroutine call via the subroutine

```
MUMPS_PROPINFO( ICNTL, INFO, COMM, MYID ).
```

This routine is called in a SPMD way by all processors.

On successful completion, a call to MUMPS will exit with the parameter mumps_par%INFO(1) set to zero. A negative value for mumps_par%INFO(1) indicates that an error has been detected on one of the processors. For example, if processor s returns with INFO(1)=–8 and INFO(2)=1000, then processor s ran out of integer workspace during the factorization and the size of the workspace

MAXIS should be increased by 1000 at least. The other processors are informed about this error and return with INFO(1) = –1 (i.e., an error occurred on another processor) and INFO(2)=s (i.e., the error occurred on processor s). Processors that detected a local error, do not overwrite INFO(1), i.e., only processors that did not produce an error will set INFO(1) to –1 and INFO(2) to the processor having the smallest error code.

The behaviour is slightly different for INFOG(1) and INFOG(2): in the previous example, all processors would return with INFOG(1)=–8 and INFOG(2)=1000.

The possible error codes returned in INFO(1) (and INFOG(1)) have the following meaning:

–1 An error occurred on processor INFO(2).

–2 NZ is out of range. INFO(2)=NZ.

–3 `MUMPS` was called with an invalid value for JOB. This may happen for example if the analysis (JOB=1) was not performed before the factorization (JOB=2), or the factorization was not performed before the solve (JOB=3). See item for JOB in Section B. This error also occurs if JOB does not contain the same value on all processes on entry to `MUMPS`.

–4 Error in user-provided permutation array PERM_IN in position INFO(2). This error occurs on the host only.

–5 Not enough real space (MAXS) to preprocess the matrix (for scaling or arrowhead calculation).

–6 Matrix is singular in structure.

–7 Problem of workspace allocation during analysis.

–8 MAXIS too small for factorization. This may happen, for example, if numerical pivoting leads to significantly more fill-in than was predicted by the analysis. The user should increase the value of ICNTL(14) or the value of MAXIS before entering the factorization (JOB=2).

–9 MAXS too small for factorization. The user should increase the value of ICNTL(14) or MAXS before entering the factorization (JOB=2).

–10 Numerically singular matrix.

–11 MAXS too small for solution. See error INFO(1)=–9.

–12 MAXS too small for iterative refinement. See error INFO(1)=–9.

–13 Error in a Fortran ALLOCATE statement. INFO(2) contains the size that was asked for.

–14 MAXIS too small for solution. See error INFO(1)=–8.

–15 MAXIS too small for iterative refinement and/or error analysis. See error INFO(1)=–8.

–16 N is out of range. INFO(2)=N.

–17 The internal send buffer that was allocated dynamically by `MUMPS` on the processor is too small. The user should increase the value of ICNTL(14) before entering the analysis (JOB=1).

–18 MAXIS too small to process root node. See error INFO(1)=–8.

–19 MAXS too small to process root node. See error INFO(1)=–9.

–20 The internal reception buffer that was allocated dynamically by `MUMPS` on the processor is too small. INFO(2) holds the minimum size of the reception buffer required (in bytes). The user should increase the value of ICNTL(14) before entering the analysis (JOB=1).

–21 Incompatible values of PAR=0 and NPROCS=1. INFO(2)=NPROCS. Running `MUMPS` in host-node mode (the host is not a slave processor itself) requires at least two processors. The user should either set PAR to 1 or increase the number of processors.

–22 A pointer array is provided by the user that is not associated or that has insufficient size. INFO(2) points to the pointer array having the wrong format:

INFO(2)	array
1	IRN or ELTPTR
2	JCN or ELTVAR
3	PERM_IN
4	A or A_ELT
5	ROWSCA
6	COLSCA
7	RHS

−23 MPI was not initialized by the user prior to a call to `MUMPS` with JOB=–1.

−24 NELT is out of range. INFO(2)=NELT.

B.3 Examples of use of `MUMPS`

B.3.1 An assembled problem

An example program to use `MUMPS` on assembled problems is given in Figure 19. Two files must be included in the program: `mpif.h` for MPI and `mumps_struc.h` for `MUMPS`. The file `mumps_root.h` must also be available because it is included in `mumps_struc.h`. The initialization and termination of MPI are performed in the user program via the calls to MPI_INIT and MPI_FINALIZE.

The `MUMPS` package is initialized by calling `MUMPS` with JOB=–1, the problem is read in by the host (in the components N, NZ, IRN, JCN, A, and RHS), and the solution is computed in RHS with a call on all processors to `MUMPS` with JOB=6. Finally, a call to `MUMPS` with JOB=–2 is performed to deallocate the data structures used by the instance of the package.

Thus for the assembled 5 × 5 matrix and right-hand side

$$\begin{pmatrix} 2 & 3 & 4 & & \\ 3 & & -3 & & 6 \\ & -1 & 1 & 2 & \\ & & 2 & & \\ & 4 & & & 1 \end{pmatrix}, \qquad \begin{pmatrix} 20 \\ 24 \\ 9 \\ 6 \\ 13 \end{pmatrix}$$

we could have as input

```
5
12
1 2 4 5 2 1 5 3 2 3 1 3
2 3 3 5 1 1 2 4 5 2 3 3
3.0 -3.0 2.0 1.0 3.0 2.0 4.0 2.0 6.0 -1.0 4.0 1.0
20.0 24.0 9.0 6.0 13.0
```

and we obtain the solution RHS(i) = i, i = 1, ..., 5.

B.3.2 An elemental problem

An example of the use of `MUMPS` for element problems is given in Figure 20. The calling sequence is similar to that for the assembled problem in Section B.3.1 but now the host reads the problem in components N, NELT, ELTPTR, ELTVAR, A_ELT, and RHS. Note also that for elemental problems ICNTL(5) must be set to 1. For the two-element matrix and right hand side

$$\begin{array}{c} 1 \\ 2 \\ 3 \end{array} \begin{pmatrix} -1 & 2 & 3 \\ 2 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \qquad \begin{array}{c} 3 \\ 4 \\ 5 \end{array} \begin{pmatrix} 2 & -1 & 3 \\ 1 & 2 & -1 \\ 3 & 2 & 1 \end{pmatrix}, \qquad \begin{pmatrix} 12 \\ 7 \\ 23 \\ 6 \\ 22 \end{pmatrix}$$

we could have as input

```
5
2
6
18
1 4 7
1 2 3 3 4 5
-1.0 2.0 1.0 2.0 1.0 1.0 3.0 1.0 1.0 2.0 1.0 3.0 -1.0 2.0 2.0 3.0 -1.0 1.0
12.0 7.0 23.0 6.0 22.0
```

and we obtain the solution RHS(i) = i, i = 1, ..., 5.

```
      PROGRAM MUMPS
      INCLUDE 'mpif.h'
      INCLUDE 'mumps_struc.h'
      TYPE (STRUC_MUMPS) mumps_par
      INTEGER IERR
      CALL MPI_INIT(IERR)
C  Define a communicator for the package
      mumps_par%COMM = MPI_COMM_WORLD
C  Ask for unsymmetric code
      mumps_par%SYM = 0
C  Host working
      mumps_par%PAR = 1
C  Initialize an instance of the package
      mumps_par%JOB = -1
      CALL MUMPS(mumps_par)
C  Define problem on the host (processor 0)
      IF ( mumps_par%MYID .eq. 0 ) THEN
        READ(5,*) mumps_par%N
        READ(5,*) mumps_par%NZ
        ALLOCATE( mumps_par%IRN ( mumps_par%NZ ) )
        ALLOCATE( mumps_par%JCN ( mumps_par%NZ ) )
        ALLOCATE( mumps_par%A( mumps_par%NZ ) )
        ALLOCATE( mumps_par%RHS ( mumps_par%N  ) )
        READ(5,*) ( mumps_par%IRN(I) ,I=1, mumps_par%NZ )
        READ(5,*) ( mumps_par%JCN(I) ,I=1, mumps_par%NZ )
        READ(5,*) ( mumps_par%A(I),I=1, mumps_par%NZ )
        READ(5,*) ( mumps_par%RHS(I) ,I=1, mumps_par%N  )
      END IF
C  Call package for solution
      mumps_par%JOB = 6
      CALL MUMPS(mumps_par)
C  Solution has been assembled on the host
      IF ( mumps_par%MYID .eq. 0 ) THEN
        WRITE( 6, * ) ' Solution is ',(mumps_par%RHS(I),I=1,mumps_par%N)
      END IF
C  Deallocate user data
      IF ( mumps_par%MYID .eq. 0 )THEN
        DEALLOCATE( mumps_par%IRN )
        DEALLOCATE( mumps_par%JCN )
        DEALLOCATE( mumps_par%A   )
        DEALLOCATE( mumps_par%RHS )
      END IF
C  Destroy the instance (deallocate internal data structures)
      mumps_par%JOB = -2
      CALL MUMPS(mumps_par)
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

Figure 19: Example program using MUMPS on an assembled problem

```
      PROGRAM MUMPS
      INCLUDE 'mpif.h'
      INCLUDE 'mumps_struc.h'
      TYPE (STRUC_MUMPS) mumps_par
      INTEGER IERR, LELTVAR, NA_ELT
      CALL MPI_INIT(IERR)
C  Define a communicator for the package
      mumps_par%COMM = MPI_COMM_WORLD
C  Ask for unsymmetric code
      mumps_par%SYM = 0
C  Host working
      mumps_par%PAR = 1
C  Initialize an instance of the package
      mumps_par%JOB = -1
      CALL MUMPS(mumps_par)
C  Define problem on the host (processor 0)
      IF ( mumps_par%MYID .eq. 0 ) THEN
        READ(5,*) mumps_par%N
        READ(5,*) mumps_par%NELT
        READ(5,*) LELTVAR
        READ(5,*) NA_ELT
        ALLOCATE( mumps_par%ELTPTR ( mumps_par%NELT+1 ) )
        ALLOCATE( mumps_par%ELTVAR ( LELTVAR ) )
        ALLOCATE( mumps_par%A_ELT( NA_ELT ) )
        ALLOCATE( mumps_par%RHS ( mumps_par%N  ) )
        READ(5,*) ( mumps_par%ELTPTR(I) ,I=1, mumps_par%NELT+1 )
        READ(5,*) ( mumps_par%ELTVAR(I) ,I=1, LELTVAR )
        READ(5,*) ( mumps_par%A_ELT(I),I=1, NA_ELT )
        READ(5,*) ( mumps_par%RHS(I) ,I=1, mumps_par%N  )
      END IF
C  Specify element entry
      mumps_par%ICNTL(5) = 1
C  Call package for solution
      mumps_par%JOB = 6
      CALL MUMPS(mumps_par)
C  Solution has been assembled on the host
      IF ( mumps_par%MYID .eq. 0 ) THEN
        WRITE( 6, * ) ' Solution is ',(mumps_par%RHS(I),I=1,mumps_par%N)
      END IF
C  Deallocate user data
      DEALLOCATE( mumps_par%ELTPTR )
      DEALLOCATE( mumps_par%ELTVAR )
      DEALLOCATE( mumps_par%A_ELT )
      DEALLOCATE( mumps_par%RHS )
C  Destroy the instance (deallocate internal data structures)
      mumps_par%JOB = -2
      CALL MUMPS(mumps_par)
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

Figure 20: Example program using MUMPS on an element problem

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, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. Technical Report RT/APO/99/2, ENSEEIHT-IRIT, 1999. (submited to SIAM Journal on Matrix Analysis and Application).

[6] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal solvers within the PARASOL environment. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pages 7–11, Berlin, 1998. Springer-Verlag.

[7] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Parallélisation d'un solveur direct creux pour architectures à mémoire distribuée. In *Proceedings de la 3e Ecole d'Informatique des Systèmes Parallèles et Répartis, ISYPAR 98, IRIT, Toulouse*, 1998.

[8] 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.

[9] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and M. Tuma. Rank detection strategies in MUMPS. Technical Report TR/PA/98/57, CERFACS, Toulouse, France, 1998.

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

[11] 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.

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

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

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

[15] 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.

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

[17] L.M. Carvalho. *Preconditioned Schur complement methods in distributed memory environments*. PhD thesis, INPT/CERFACS, France, october 1997. TH/PA/97/41, CERFACS.

[18] L.M. Carvalho, L. Giraud, and G. Meurant. Local preconditioners for two-level non-overlapping domain decomposition methods. Tech. Rep. in preparation, CERFACS, France, 1999.

[19] L.M. Carvalho, L. Giraud, and P. Le Tallec. Algebraic two-level preconditioners for the schur complement method. Tech. Rep. TR/PA/98/18, CERFACS, France, 1998. submitted to SIAM SISC.

[20] S. Chandrasekaran and I. Ipsen. On rank-revealing factorizations. *SIAM J. Matrix Anal. Appl.*, 15:592–622, 1994.

[21] M. J. Daydé and I. S. Duff. Use of level 3 BLAS in LU factorization in a multiprocessing environment on three vector multiprocessors, the ALLIANT FX/80, the CRAY-2, and the IBM 3090/VF. *Int. J. of Supercomputer Applics.*, 5:92–110, 1991.

[22] M. J. Daydé and I. S. Duff. A block implementation of level 3 BLAS for RISC processors. Technical Report RT/APO/96/1, ENSEEIHT-IRIT, 1996.

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

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

[25] J. Dongarra and R. C. Whaley. A users' guide to the blacs. Technical Report CS-95-281, University of Tennessee, Knoxville, Tennessee, USA, 1995.

[26] 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.

[27] 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.

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

[29] I. S. Duff. Algorithm 575. Permutations for a zero-free diagonal. *ACM Transactions on Mathematical Software*, 7:387–390, 1981.

[30] 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.

[31] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

[32] A. Supalov (ed). PARASOL Interface Specification, version 2.1. Technical report, GMD SCAI, Sankt Augustin, Germany, January 1998.

[33] 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.

[34] A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *Int J. Parallel Programming*, 18:291–314, 1989.

[35] 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.

[36] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. Pspases: An efficient and scalable parallel sparse direct solver. Technical Report (to appear), Department of Computer Science, University of Minnesota and IBM T.J. Watson Research center, 1999.

[37] 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.

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

[39] F. Hecht and A. Marrocco. Mixed finite element simulation of heterojonction structures including a boundary layer model for the quasi-fermi levels. *COMPEL*, 13:757–770, 1995.

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

[41] G. Karypis and V. Kumar. MeTiS – *Unstructured Graph Partitioning and Sparse Matrix Ordering System – Version 2.0.* University of Minnesota, June 1995.

[42] 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.

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

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

[45] 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.