Deliverable D2.1c : MUMPS Version 3.1
A MUltifrontal Massively Parallel Solver

RAL, CERFACS, ENSEEIHT-IRIT

February 13, 1999

Abstract

This document describes work under ESPRIT project 20160 (PARASOL) carried out at CERFACS (France), Rutherford Appleton Laboratory (England), and ENSEEIHT-IRIT (France) in 1998.

1 Roadmap for parallel direct solver MUMPS

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	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	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

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 [1, 3, 16, 17] for full details of this technique.

In both the symmetric and unsymmetric case, the structure of the matrix is first *analysed* to determine an ordering that, in the absence of any numerical pivoting, will preserve sparsity

in the factors. An approximate minimum degree ordering strategy is used on the symmetrized pattern $\mathbf{A}+\mathbf{A}^T$, and this analysis phase produces both an ordering and an assembly tree. The assembly tree is then used to drive the subsequent numerical factorization and solution phases. At each node of the tree, a dense submatrix (called a *frontal matrix*) is assembled using data from the original matrix and from the children of the node. Pivots can be chosen from within a submatrix of the frontal matrix (called the *pivot block*) and eliminations performed. The rows and columns of the pivot block are fully summed, meaning that no further contributions to them will come from rows or columns later in the pivotal sequence. The resulting factors are stored for use in the solution phase, and the Schur complement (the *contribution block*) is passed to the parent node for assembly at that node. In the numerical factorization phase, the tree is processed from the leaf nodes to the root (if the matrix is reducible, we have a forest, and each component tree of the forest will be treated similarly and independently). The subsequent forward and backward substitutions during the solution phase process the tree from the leaves to the root and from the root to the leaves, respectively. A crucial aspect of the assembly tree is that it defines only a partial order for the factorization since the only requirement is that a child must complete its elimination operations before the parent can be fully processed. It is this freedom that enables us to exploit parallelism in the tree (*tree parallelism*).

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.

In both the unsymmetric and symmetric cases, data is first assembled at a node combining the Schur complements from the children with data from the original matrix. The original matrix data comprises rows and columns corresponding to variables that the analysis forecasts should be eliminated at this node. This data is supplied in so-called arrowhead format for assembled matrices and in elemental format for elemental matrices, with the matrix ordered according to the permutation from the analysis phase and row 1 preceding column 1 followed by row 2 (from the diagonal) and column 2 and so on, where the columns are not supplied in the symmetric case because they are identical to the rows. This data and the contribution blocks from the children are assembled (or summed) into a frontal matrix using indirect addressing (sometimes called an extended add operation).

Eliminations are then performed on the assembled frontal matrix. A right-looking factorization can be used and is blocked so that cache effects can be reduced and use can be made of the Level 3 BLAS [12, 13]. This can be done by eliminating a fixed number of pivots (`nb`, say). When numerical pivoting is required, the fully summed rows must be updated during these eliminations but the major part of the frontal matrix is not updated until the computations on the fully summed rows are completed whence the remaining rows can be updated using Level 3 BLAS kernels. It is possible either to use parallel versions of the Level 3 BLAS or to update the rows in independent strips. This gives rise to so-called *node parallelism*.

The size of the root matrix (the matrix corresponding to the root node of the assembly tree) may be large, and sometimes significantly larger than was predicted during the analysis phase. If the size of the root node is small, it is factorized using the BLAS3-based LAPACK sequential software. Otherwise, a parallel processing is used. In order to keep the good scalability, `MUMPS` switches to a 2D cyclic distribution of the root matrix and uses ScaLAPACK software (preferably the vendor optimized version) to perform the eliminations.

3 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 [13, 14], LAPACK, BLACS, and ScaLAPACK [7] 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.

3.1 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.

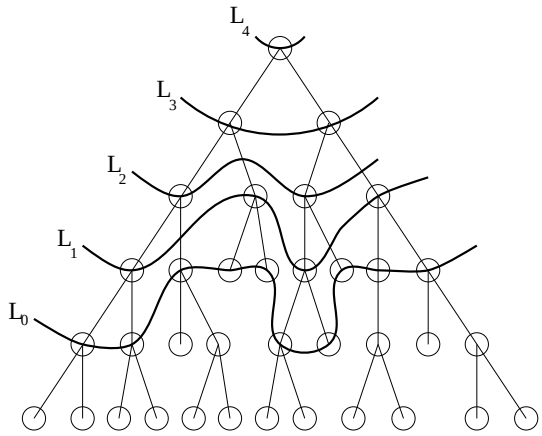

Figure 1: Decomposition of the assembly tree into levels.

The tree is processed from the bottom to the top, level by level (see Figure 1). Level L_0 is determined using the Algorithm 3.1 [18] 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.

– **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

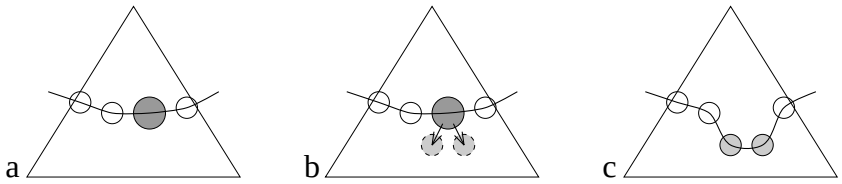

Figure 2: One step in the construction of the first level L_0.

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.

3.2 Sources of parallelism

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

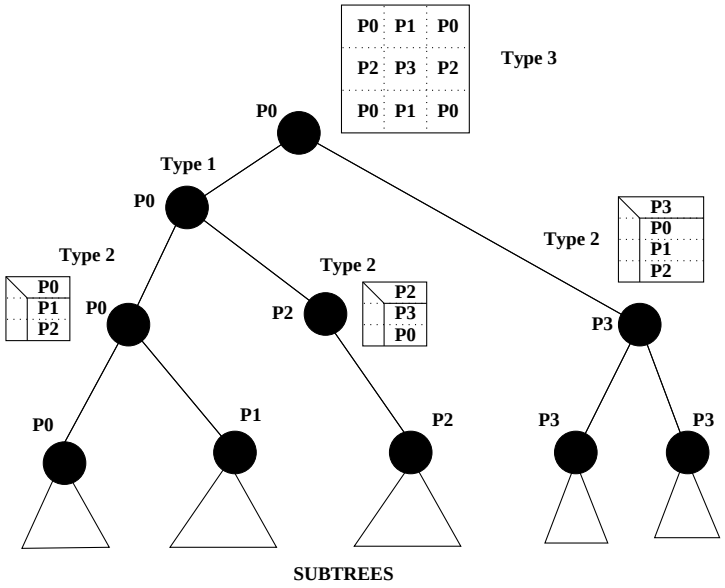

Figure 3: Distribution of the computations of a multifrontal assembly tree.

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 [2]) 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*.

3.2.1 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

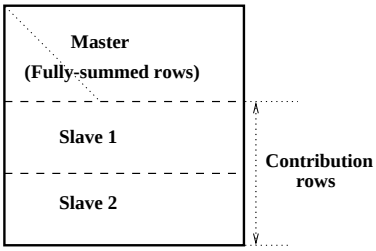

Figure 4: Type 2 nodes: partitioning of frontal matrix.

information describing the structure of the contribution blocks 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 3.4.3.

3.2.2 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 [7] 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.

3.2.3 Impact of parallelism on memory and work balance

Processor number	1	2	3	4	5
Original matrix ($\times 10^3$)	1920	2904	2475	2571	2059
LU factors ($\times 10^3$)	15927	15982	15993	16149	16117
Flop count ($\times 10^9$)	18.2	21.5	18.6	22.6	19.5

Table 1: Study of memory and work balancing on matrix CRANKSEG_1 using 5 working processors (that is, we exclude the host processor) and all levels of parallelism of the method. All sizes are in number of 64-bit reals per processor.

We show, in Table 1, the distribution of both the input matrix and the LU factors during the factorization of matrix CRANKSEG_1 on five working processors. The matrix is considered unsymmetric and has 10.6×10^6 nonzeros with 80.6×10^6 nonzeros in the LU factors. We see, in Table 1, how well the mapping algorithm balances the storage of the LU factors between the processors. Concerning the original matrix, we observe that the extra space due to duplication for type 2 node parallelization only represents around 10% of the size of the original matrix. Finally we see that, even if the algorithm does not aim to balance the work near the top of the tree, balancing the memory used for the factors also leads to a good balance for the floating-point operations.

3.3 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 [11] 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.

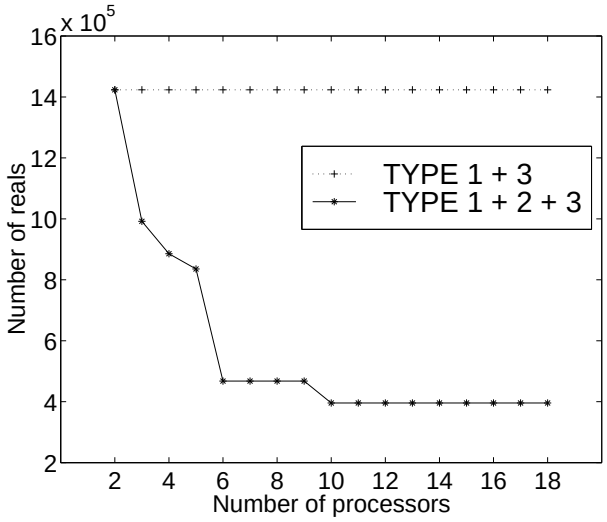

Figure 5: Impact of type 2 parallelism on the size (in number of 64-bit reals) of the send buffer. Test matrix is WANG3.

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 3.4.1 and 3.4.3 during the detailed description of type 2 parallelism for LDL^T factorization.

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

while (all nodes not processed)
  if local pool empty **then**
    blocking wait 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 processor might be blocked because its send buffer is full.

3.4 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 [7] 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.

3.4.1 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 3.3, 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 3.4.1.

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 3.4.1 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.

3.4.2 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 [5] 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.

During LU factorization, a KJI-SAXPY blocked algorithm [1, 9] is used to compute the LU factor associated with the block of fully summed rows (matrices A and C). The Level 3

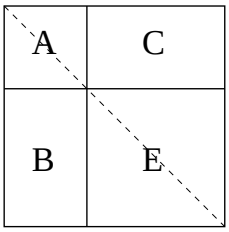

Figure 6: Structure of a type 1 node.

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 [10] 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

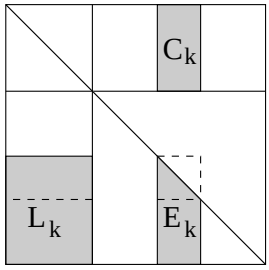

Figure 7: Blocks used for updates of the contribution part of a type 1 node.

of the algorithm, we stored 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 3.4.2.

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

3.4.3 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.

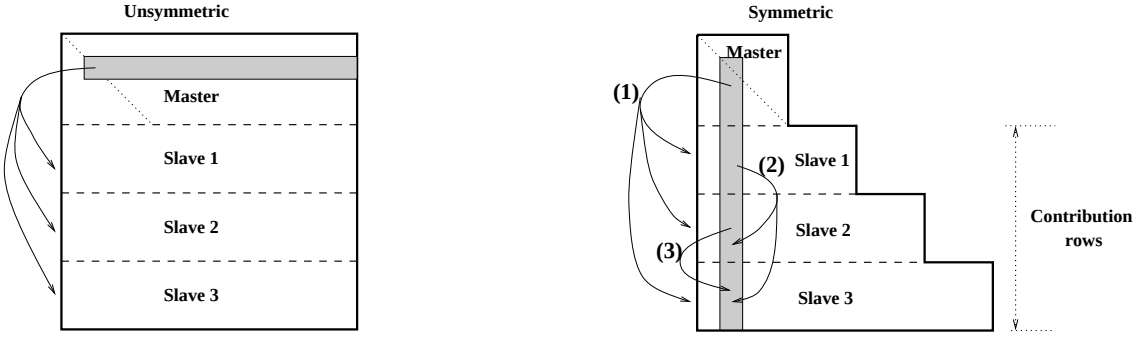

Figure 8: Structure of a type 2 node.

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 [19] from PALLAS. We see that, on this example, the master processor is relatively more loaded than the slaves.

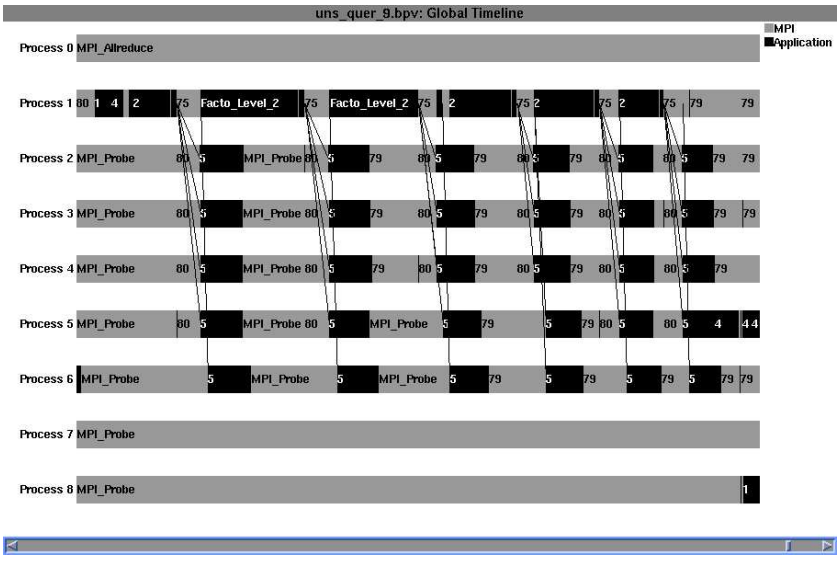

Figure 9: VAMPIR trace of an isolated type 2 unsymmetric factorization (Master is Process 1).

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 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.

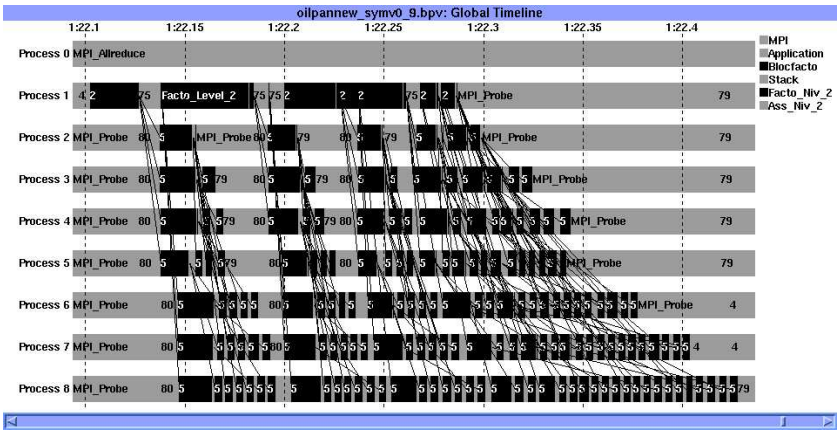

Figure 10: VAMPIR trace of an isolated type 2 symmetric factorization; constant row block sizes. (Master is Process 1).

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.

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.

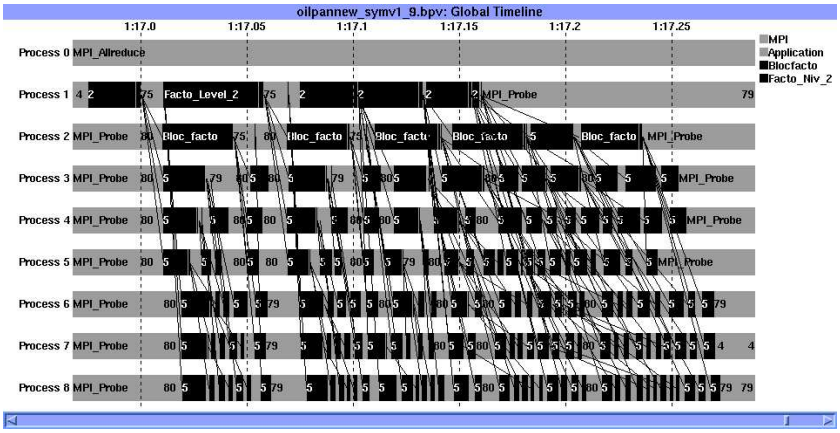

Figure 11: VAMPIR trace of an isolated type 2 symmetric factorization; variable row block sizes. (Master is Process 1).

4 Some other functionalities of MUMPS 3.1

4.1 Pre-processing and post-processing facilities

MUMPS offers pre-processing and post-processing facilities. Permutations for a zero-free diagonal [15] 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 [6] 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.

We calculate an estimate of the sparse backward error using the theory and metrics developed in [6]. 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

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

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

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

is used instead, where A_i is row i of A. The largest scaled residual (1) is returned in RINFOG(7) and the largest scaled residual (2) is returned in RINFOG(8). If all equations are in category (1), zero is returned in RINFOG(8). 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

$$\delta\mathbf{A}_{ij} \leq \max(\text{RINFOG}(7), \text{RINFOG}(8))|\mathbf{A}|_{ij},$$

and $\delta\mathbf{b}_i \leq \max(\text{RINFOG}(7)|\mathbf{b}|_i, \text{RINFOG}(8)\|\mathbf{A}_i\|_\infty \|\bar{\mathbf{x}}\|_\infty)$. Note that $\delta\mathbf{A}$ respects the sparsity of $\mathbf{A}$. An upper bound for the error in the solution is returned in RINFOG(9).

4.2 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. 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 is then computed using backward transformations.

Strategies based on rank revealing LU and rank revealing QR are provided. Details will be given in a forthcoming technical report [4].

5 PARASOL Interface

Besides the approximate minimum ordering algorithm built in MUMPS, the PARASOL test driver (from now on called PTD-MUMPS) currently implements 5 other ordering strategies for the MUMPS solver based on graph partitioning packages METIS and RALPAR. The orderings currently available are:

AMD Approximate minimum degree. This is the default (internal) ordering generated by MUMPS if no ordering is specified or in case an error occurred when another ordering was generated.

- Multilevel bisection and Minimum Degree on subgraphs ML spectral bisection from METIS library is combined with minimum degree heuristics on subgraphs in the OEx family of algorithms. The user can the define maximal size of the subgraphs and the balancing factor for the size of subgraphs at each bisection step. This allows the user to control the size/depth of the bisection tree.

  OE0 Multilevel Bisection and Multiple Minimum Degree on subgraphs. This is the simplest ordering generated with the use of multilevel spectral bisection library contained in METIS. It uses MMD ordering heuristics on subgraphs. It should generate the same results as a code which is a part of METIS - OEMETIS.

  OE1 Multilevel Bisection and AMD on subgraphs. Same as the OE0 option, but AMD is used on subgraphs.

  OE2 Multilevel Bisection and HALO-AMD on subgraphs. Same as the OE0 option, but HALO-AMD is used on subgraphs. Since the bisection is implemented in a recursive manner only partial information about boundaries of subgraphs is known. The last separators are used for definition of boundary in HALO-AMD. The complete boundary information is used in algorithms RL0 and RL1.

  ON0 Multilevel Bisection, Vertex separator and AMD on subgraphs. Same as OE0 but the vertex separator is computed and AMD used on subgraphs. It is identical to the METIS code ONMETIS except that AMD rather than MMD is used on subgraphs.

- Multilevel bisection, Multisector and HALO-AMD on subgraphs Using ML spectral bisection from RALPAR library is combined with minimum degree heuristics on subgraphs in the OEx family of algorithms. The user can define the maximal size of the subgraphs and/or the maximal number of subgraphs and the balancing factor for the size of subgraphs at each bisection step. This allows the user to control the size/depth of the bisection tree. The graph of the boundary (multisector) for each subgraph is computed and used for ordering by HALO-AMD.

  RL0 Multilevel Bisection and HALO-AMD on subgraphs. The complete boundary information for each subgraphs is used in HALO-AMD.

6 Fortran 90 Interface

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 the 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 12.

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

6.1 Input parameters

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

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 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.

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.

```
      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 symmetric Positive Definite,
C         SYM=2 general symmetric)
C    Type of parallelism (PAR=1 host working, PAR=0 host not working)
         INTEGER SYM, PAR
         INTEGER 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    Unassembled input matrix: User interface
C    -------------------------------------
         INTEGER NELT
         INTEGER, DIMENSION(:), POINTER :: ELTPTR
         INTEGER, DIMENSION(:), POINTER :: 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)
C        Cost (flops) of subtrees on local process
         DOUBLE PRECISION COST_SUBTREES
         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   Root structure(internal)
C   ------------------------
         TYPE (TYPE_ROOT_STRUC) :: root
       END TYPE STRUC_MUMPS
```

Figure 12: Definition of the components of the structure defined in `mumps_struc.h` that are of interest to the user.

- 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 are deallocated. It should be called by the user only when no further call to MUMPS with this instance is 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 must always 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 matrix in assembled format (ICNTL(5)=0), or
  - 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).

  (See sections 6.2-6.3.) 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.) A 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**. The following components of the structure must always 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).

  (See Sections 6.2-6.3.) 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 solve. 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 $\mathbf{Ax} = \mathbf{b}$. The structure component mumps_par%RHS must always be set by the user (on the host only) before a call with JOB=3. (See Section 6.4.) It must be preceded by a call to MUMPS 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.

6.2 Assembled matrix format

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 non-zero a_{ii} is held as A(K)=a_{ii}, IRN(K)=JCN(K)=i, and a pair of off-diagonal non-zeros $a_{ij} = a_{ji}$ 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).

6.3 Element matrix format

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 $N_{\rm p}$ denotes ELTPTR(p+1)–ELTPTR(p), then the values for element J are stored in positions $K_{\rm J}+1, \ldots, K_{\rm J}+L_{\rm J}$, where
  - $K_{\rm J} = \sum_{p=1}^{\rm J-1} {N_p}^2$, and $L_{\rm J} = {N_{\rm J}}^2$ in the unsymmetric case (SYM = 0)
  - $K_{\rm J} = \sum_{p=1}^{\rm J-1} (N_p * (N_p+1))/2$, and $L_{\rm J} = (N_{\rm J} * (N_{\rm J}+1))/2$ 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).

6.4 Right-hand side and solution vector

mumps_par%RHS (double precision array pointer, dimension N) is a double precision array that is only accessed when 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.

6.5 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 only, before a call to the factorization phase (JOB=2). It should still be available and not be modified for the solve phase. 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 then accessed only by the host, during the analysis phase. In this case, 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%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.

6.6 Control arrays

mumps_par%ICNTL is an integer array of dimension 20, containing:

- 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. Default value is 2.

- 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. 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 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 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 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) has default value 20 and is only accessed by the host during the analysis phase. When significant extra fill-in is caused by numerical pivoting, larger values of ICNTL(14) may help use the real working space more efficiently.
- 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 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 are provided:
  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. Currently only available if MUMPS is run on one processor or if ICNLT(13) $\neq$ 0.
  On numerically easy problems, we advise options 5/6. On more difficult problems, options 3/4 are recommended.
- 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) $\leq$ 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) - ICNTL(20) are not used in the current version.

mumps_par%CNTL is a double precision array of dimension 5, containing:

- 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.

6.7 Information arrays

mumps_par%RINFO is a double precision array of dimension 20. This array is available on each processor and contains the following local information on the execution of MUMPS:

- RINFO(1) *(produced during analysis)*: The estimated number of floating-point operations on the processor for the elimination process.
- RINFO(2) *(produced during factorization)*: The number of floating-point operations on the processor for the assembly process.
- RINFO(3) *(produced during 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%RINFOG is a double precision array of dimension 20. This array is only significant on the host and contains the following global information on the execution of MUMPS:

- RINFOG(1) *(produced during analysis)*: The estimated number of floating-point operations (on all processors) for the elimination process.
- RINFOG(2) *(produced during factorization)*: The total number of floating-point operations (on all processors) for the assembly process.
- RINFOG(3) *(produced during factorization)*: The total number of floating-point operations (on all processors) for the elimination process.
- RINFOG(4) to RINFOG(9) *(produced 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%INFO is an integer array of dimension 20. This array is available on each processor and contains the following local information on the execution of MUMPS:

- INFO(1) is 0 if the routine is successful, negative if an error occurred (see Section 7).
- INFO(2) holds additional information about the error.
- INFO(3) *(produced after analysis)*: Estimated real space needed on the processor for factors.
- INFO(4) *(produced after analysis)*: Estimated integer space needed on the processor for factors.
- INFO(5) *(produced after analysis)*: Estimated maximum front size on the processor.
- INFO(6) *(produced after analysis)*: Number of nodes in the complete tree. The same value is returned on all processors.
- INFO(7) *(produced after analysis)*: Minimum value of MAXIS estimated by the analysis phase to run the numerical factorization successfully.
- INFO(8) *(produced after analysis)*: Minimum value of MAXS estimated by the analysis phase to run the numerical factorization successfully.
- INFO(9) *(produced after factorization)*: Size of the real space used on the processor to store the LU factors.
- INFO(10) *(produced after factorization)*: Size of the integer space used on the processor to store the LU factors.
- INFO(11) *(produced after factorization)*: Order of the largest frontal matrix processed on the processor.
- INFO(12) *(produced after factorization)*: Number of off-diagonal pivots encountered on the processor.
- INFO(13) *(produced 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) *(produced after factorization)*: Number of memory compresses on the processor.

- INFO(15) - INFO(20) are not used in the current version.

mumps_par%INFOG is an integer array of dimension 20. This array is only significant on the host and contains the following global information on the execution of MUMPS:

- INFOG(1:2) : not significant.
- INFOG(3) *(produced after analysis)*: Total estimated real workspace for factors on all processors.
- INFOG(4) *(produced after analysis)*: Total estimated integer workspace for factors on all processors.
- INFOG(5) *(produced after analysis)*: Estimated maximum front size in the complete tree.
- INFOG(6) *(produced after analysis)*: Number of nodes in the complete tree.
- INFOG(7:9) : not significant.
- INFOG(10) *(produced after factorization)*: Total integer space to store LU factors.
- INFOG(11) *(produced after factorization)*: Order of largest frontal matrix.
- INFOG(12) *(produced after factorization)*: Total number of off-diagonal pivots.
- INFOG(13) *(produced after factorization)*: Total number of delayed pivots.
- INFOG(14) *(produced after factorization)*: Total number of memory compresses.
- INFOG(15) *(produced after solution)*: Number of steps of iterative refinement.
- INFOG(16) - INFOG(20) are not used in the current version.

7 Treatment of errors

An error may occur during the execution of MUMPS. Since the code is parallel, the processing of errors is complicated. If an error occurs on one processor, it must inform all the other processors before it returns.

Therefore, a mechanism to inform other processors of an error. During factorization and solve phases, where all messages are asynchronous, a process sends a message with a specific tag to the other processes to inform them about the error so that they can return.

Sometimes, an error may occur on a processor in a routine that does not use asynchronous communication. In such cases, the error is propagated 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 return, INFO(1) will be smaller than 0 if an error occurred on one of the processors.

Suppose, for example, that processor s returns from a subroutine with INFO(1)=–7, INFO(2)=1000, which means that processor s ran out of integer workspace and that the size of the integer workspace should be increased by 1000 at least. Then after MUMPS_PROPINFO, other processors will have INFO(1) = –1 (i.e., an error occurred on another processor) and INFO(2)=s (i.e., the processor on which the error occurred is s in the communicator mumps_par%COMM). If more than processor returns from a subroutine with an error, local error codes are not overwritten by -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 possible error codes after a call to MUMPS are currently the following:

- INFO(1) = –1: An error occurred on processor INFO(2).
- INFO(1) = –2: NZ is out of range. INFO(2)=NZ.
- INFO(1) = –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 6. This error also occurs if JOB does not contain the same value on all processes on entry to MUMPS.

- INFO(1) = –4: Error in user-provided permutation array PERM_IN in position INFO(2). This error occurs on the host only.
- INFO(1) = –5: Not enough real space (MAXS) to preprocess the matrix (for scaling or arrowhead calculation).
- INFO(1) = –6: Matrix is singular in structure.
- INFO(1) = –7: MAXIS too small for analysis.
- INFO(1) = –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 MAXIS before entering the factorization (JOB=2) or increase the value of ICNTL(14) before the analysis (JOB=1).
- INFO(1) = –9: MAXS too small for factorization. See error INFO(1) = –8.
- INFO(1) = –10: Numerically singular matrix.
- INFO(1) = –11: MAXS too small for solution.
- INFO(1) = –12: MAXS too small for iterative refinement.
- INFO(1) = –13: Error in a Fortran ALLOCATE statement. INFO(2) contains the size that was asked for.
- INFO(1) = –14: MAXIS too small for solution.
- INFO(1) = –15: MAXIS too small for iterative refinement and/or error analysis.
- INFO(1) = –16: N is out of range. INFO(2)=N.
- INFO(1) = –17: The send buffer on the processor is too small. The user should increase the value of ICNTL(14) before entering the analysis (JOB=1).
- INFO(1) = –18: MAXIS too small to process root node.
- INFO(1) = –19: MAXS too small to process root node.
- INFO(1) = –20: The reception buffer 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).
- INFO(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.
- INFO(1) = –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

- INFO(1) = –23: MPI was not initialized by the user prior to a call to `MUMPS` with JOB=–1.
- INFO(1) = –24: NELT is out of range. INFO(2)=NELT.

8 Examples of use of `MUMPS`

8.1 An assembled problem

An example program to use `MUMPS` on assembled problems is given in Figure 13. 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 package MUMPS 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.

```
      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  Destroy the instance (deallocate data structures)
      mumps_par%JOB = -2
      CALL MUMPS(mumps_par)
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

Figure 13: Example program using MUMPS on an assembled problem

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.

8.2 An elemental problem

An example of the use of `MUMPS` for element problems is given in Figure 14. The calling sequence is similar to that for the assembled problem in Section 8.1 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

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

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.

9 Concluding remarks

We report in Table 2 results for the matrix QUER of the symmetric and unsymmetric code on an IBM SP2 located at GMD (Bonn, Germany), where each node is a 66 MHertz processor with 128 MBytes of physical memory and 512 MBytes of virtual memory.

We observe a problem of memory paging due to the use of the virtual memory on the SP2. However speedups are good when comparing times with uni-processor CPU times. Furthermore, comparison of symmetric and unsymmetric performance shows a good performance of the symmetric code, which is almost twice as fast as the unsymmetric version.

Finally, Table 3 presents results for the larger test problem CRANKSEG2 both on the SP2 and an Origin 2000 located at Parallab (Bergen).

The speedups on the SP2 are due to both algorithmic and memory effects. This is not the case on the Origin 2000 where the speedups only comes from the algorithm.

Working processors	Time for factorization unsymmetric	symmetric
1 (CPU)	64.9	41.1
1 (elapsed)	299.2	150.4
2 (elapsed)	109.1	21.0
4 (elapsed)	19.1	12.9
8 (elapsed)	15.2	9.3
16 (elapsed)	11.5	6.4
24 (elapsed)	10.1	6.6
32 (elapsed)	10.5	5.8

Table 2: . Performance of the unsymmetric and symmetric codes on SP2

Machine	Working processors	Time for numerical factorization
SP2	16	1045.34
	24	457.26
	32	139.66
Origin	1	635.4
	2	411.0
	3	275.7
	4	220.4
	5	175.1
	6	158.3
	7	142.9
	8	135.7

Table 3: Results for the symmetric version of the code on matrix CRANKSEG2.

References

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

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

[3] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. Technical Report RAL-TR-98-051, Rutherford Appleton Laboratory, 1998.

[4] 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.

[5] 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.

[6] M. Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.*, 10:165–190, 1989.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] I. S. Duff. Algorithm 575. Permutations for a zero-free diagonal. *ACM Trans. Math. Softw.*, 7:387–390, 1981.

[16] 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.

[17] 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.

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

[19] 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.

```
      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  Destroy the instance (deallocate data structures)
      mumps_par%JOB = -2
      CALL MUMPS(mumps_par)
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

Figure 14: Example program using MUMPS on an element problem