MUMPS
MUltifrontal Massively Parallel Solver
Version 2.0[1]

P.R. Amestoy[2], I.S. Duff[3], and J.-Y. L'Excellent[4]

Technical Report TR/PA/98/02

February 6, 1998

CERFACS
42 Ave G. Coriolis
31057 Toulouse Cedex
France

ABSTRACT
We describe aspects of the interface and design of Version 2.0 of the MUltifrontal Massively Parallel Solver MUMPS. This code solves sets of sparse linear equations $Ax = b$, where the matrix A is unsymmetric. It is written in Fortran 90 and uses MPI for message passing. It also calls the ScaLAPACK code which in turn uses the BLACS. Level 3 BLAS are also used by the code. MUMPS is the direct solver in the PARASOL project, an EU LTR Project with twelve partners from five countries. The main aim of PARASOL is to develop a public domain library of sparse codes for distributed memory parallel computers.
This report describes the interface to the MUMPS code and the message passing mechanisms that are used in the package.

Keywords: Multifrontal, sparse solver, distributed memory parallelism, MPI, BLAS, BLACS, ScaLAPACK, PARASOL.
AMS(MOS) subject classifications: 65F05, 65F50.

---

[1] Current reports available at http://www.cerfacs.fr/algor/algo_reports.html.

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

[3] duff@cerfacs.fr. Also at Atlas Centre, RAL, Oxon OX11 0QX, England.

[4] excelle@cerfacs.fr.

Contents

1 Introduction

This document describes aspects of the interface and design of Version 2.0 of the MUltifrontal Massively Parallel Solver MUMPS.

MUMPS is a code for distributed memory parallel computers that solves sparse sets of linear equations using a multifrontal method, which is a direct method based on the LU factorization of the coefficient matrix. We refer the reader to our earlier papers [1, 3, 4] for full details of this technique. Version 2 of MUMPS solves the system

$$Ax = b,$$

where A is unsymmetric.

Although we do not wish to describe the multifrontal method in any detail in this document, we will now briefly examine some of the features that are important to our subsequent discussion of the MUMPS code.

The structure of the coefficient matrix is first *analysed* to determine an ordering that, in the absence of any numerical pivoting, will preserve sparsity in the factors. In Version 2.0 of MUMPS, a minimum degree ordering strategy is used on the symmetric pattern of $A+A^T$, and this analysis phase produces both an ordering and an assembly tree. The assembly tree is then used to drive subsequent factorization and solution. At each node of the tree, a dense submatrix (called a *frontal matrix*) is assembled using data from the original matrix and from the sons of the node. Note that, throughout this document, the term *node* refers to a tree node and not to a processor. Pivots can be chosen from within a submatrix of the frontal matrix (called the *fully summed block*) and eliminations performed. The resulting factors are stored for use in the solution phase and the Schur complement (the *contribution block*) is passed to the father 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 son must complete its elimination operations before the father can be fully processed. It is this freedom that enables us to exploit parallelism in the tree (*tree parallelism*).

Because the matrices are unsymmetric, threshold pivoting is used in the numerical factorization phase to maintain numerical stability so that it is possible that the pivots selected by the analysis phase are unsuitable. We are at liberty to choose pivots from anywhere within the fully summed block (including off-diagonal pivots) but it still may not be possible to eliminate all variables from this block. The result is that the Schur complement that is passed to the father node may be larger than anticipated by the analysis phase and so our data structures may be different

from those forecast by the analysis phase. This implies that we need to allow dynamic scheduling during numerical factorization in contrast to the symmetric positive definite case where only static scheduling is required.

We still, however, perform a static allocation of processors to the nodes of the tree generated by the analysis phase. Thereafter, since we don't know the size of each frontal matrix until the numerical factorization, we partition nodes with large frontal matrices over several processors dynamically. Also, since we do not know in advance the length of messages communicating data between a son node and its father we must handle this message passing dynamically also. That is to say, storage management is dynamic.

A version of the multifrontal code for shared memory computers was developed by Amestoy and Duff [1] and was included in Release 12 of the Harwell Subroutine Library [8] as code `MA41`. In this version, control and synchronization were enabled through a centralized pool of work, initialized to the leaf nodes. A new node was added to the pool when all its sons had been processed and a node was removed when it had been processed. Because of the reduction in tree parallelism towards the root of the tree, it is also necessary to parallelize the computations within a node (*node parallelism*) and this was also accommodated in the shared memory code by having two types of task in the work pool identified by a simple flag. Amestoy and Espirat [6] developed a distributed version of the multifrontal code using PVM but did not include node parallelism. This was the basis for Version 1.0 of MUMPS that was released in May 1997.

In this current version (Version 2.0) of MUMPS, we distribute the pool among the processors, but our model still requires an identified host processor to perform the analysis phase, distribute the incoming matrix, collect the solution, and generally oversee the computation. In the context of PARASOL, we thus support either the host-node model or the hybrid host model.

All routines called by the user for the different steps are SPMD (Single Processor Multiple Data), and the distinction between the host and the other processors is made by the MUMPS code.

In Section 2, we discuss the Fortran 90 interface, describing the data structures and calling sequence for the code. In Section 3, we describe how we handle the nodes of the tree and how we accommodate both tree and node parallelism. We discuss, in some detail, the messages and communications used in the parallel implementation of the factorization phase in Section 4, and in Section 5, we consider the solution phase. Finally, in Section 6, we describe some mechanisms implemented in the code, such as the mapping, the distribution of the original matrix, the processing of errors and the management of asynchronous messages.

2 Fortran 90 Interface

This section describes the Fortran 90 interface to MUMPS. In the context of the PARASOL project, the PARASOL Interface (see [5]) will be used. The PARASOL interface to MUMPS will consist of two basic calls. The first, `psl_map`, determines an ordering of the sparse matrix and a static mapping to processors. In the second, `psl_solve`, the user provides a matrix and right-hand side, and receives the solution. An option exists for the user to give a subsequent right-hand side to `psl_solve` and receive a corresponding solution. This interface is discussed in a separate document [9] of which this document is an appendix.

In the Fortran 90 interface, there is a single user callable subroutine called `PSL_MUMPS` that has a single parameter `mumps_par` of Fortran 90 derived datatype `STRUC_MUMPS`, viz.

```
TYPE (STRUC_MUMPS) :: mumps_par
CALL PSL_MUMPS( mumps_par )
```

This derived datatype, `STRUC_MUMPS`, 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 them must only be defined on the host. Others must be defined on all processors. The complete structure is shown in Figure 1. Note that the SEQUENCE statement is needed because we use INCLUDE statements instead of modules and we want the compiler to organize the datatype in the same way in the user program and in the library.

The Fortran 90 interface to MUMPS consists in calling the subroutine `PSL_MUMPS` with the appropriate parameter settings. The `psl_mumps_struc.h` file, shown in Figure 1, should be included in the program to define the derived data type. The file `psl_mumps_root.h`, which is included, defines the datatype for the component `root`. A description of the parameters is given Section 2.1. An example of how to use the interface is given in Section 2.2.

2.1 Parameters from the structure available to the user

`mumps_par%JOB` (integer): must be initialized on all processors before a call to MUMPS. It is not altered by MUMPS.

- JOB=–1 initializes an instance of the package. This must be called before any other call to the package and will set default values for other components of `STRUC_MUMPS`, which may then be altered before subsequent calls to `PSL_MUMPS`.
- JOB=–2 deallocates all data structures associated with an instance of the package. This should be called by the user when no further call to `PSL_MUMPS` with this instance is required. It should be called before a further JOB=–1 call for another instance.

```
      INCLUDE 'psl_mumps_root.h'
      TYPE STRUC_MUMPS
        SEQUENCE
C
C  Problem definition
         INTEGER JOB, N, NZ
         INTEGER, DIMENSION(:), POINTER :: IRN, JCN
         DOUBLE PRECISION, DIMENSION(:), POINTER :: A
         DOUBLE PRECISION, DIMENSION(:), POINTER :: RHS
         DOUBLE PRECISION, DIMENSION(:), POINTER :: COLSCA, ROWSCA
C
C        Ordering, if given by user
         INTEGER, DIMENSION(:), POINTER :: PERM_IN
C
C        For MPI
         INTEGER  COMM, MYID, NPROCS, NSLAVES
         INTEGER, DIMENSION(:), POINTER :: POIDS
C   BUFR  --> Buffer for received messages
C   Size is estimated dynamically. There is one buffer per processor.
C   Note that there is also a separate buffer for sending asynchronously
C   non-contiguous data (defined in the module BUFFER).
        INTEGER LBUFR
        INTEGER, DIMENSION(:), POINTER ::  BUFR
C
C  Work arrays:
C    IS  used for the factors + contribution blocks only
C    IS1 computed during analysis and used for factorization.
C    IS, S and PTLUST are computed during factorization and used
C        during the solve phase.
         INTEGER MAXIS, MAXS
         INTEGER MAXIS1
         INTEGER, DIMENSION(:), POINTER :: IS
         INTEGER, DIMENSION(:), POINTER :: PTLUST
         INTEGER, DIMENSION(:), POINTER :: IS1
         DOUBLE PRECISION, DIMENSION(:), POINTER :: S
C    Storage for the mapping
         INTEGER, DIMENSION(:), POINTER :: PROCNODE
         INTEGER nbsa
C    Distributed original matrix in arrowhead format
         INTEGER, DIMENSION(:), POINTER :: INTARR
         DOUBLE PRECISION, DIMENSION(:), POINTER :: DBLARR
C Control parameters (external and internal), statistics
         INTEGER ICNTL(20)
         INTEGER INFO(20), KEEP(50)
         DOUBLE PRECISION RINFO(20), CNTL(2)
C Next field is added for parasol interface
C It is used to tell the index in ud of data to send
C (see psl_what2send)
         INTEGER pass
C     Internal datatype for parallel root
         TYPE (TYPE_ROOT_STRUC) :: root
      END TYPE STRUC_MUMPS
```

Figure 1: Definition of the structure for psl_mumps_struc.h

- JOB=1 accepts the pattern of A and chooses pivots from the diagonal using a selection criterion to preserve sparsity. It uses the pattern of $A + A^T$ but ignores numerical values. It subsequently constructs subsidiary information for the actual factorization (a JOB=2 call). An option exists for the user to input the pivot 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, an integer array of size 2*NZ + 3*N + 1 is used as a temporary workspace for the analysis on the host. The component array IS1, of size 12*N, is dynamically allocated. It is transmitted to the factorization and solution phases (JOB=2 and JOB=3, respectively), and deallocated with JOB=–2.
- JOB=2 factorizes a matrix A using the information from a previous call with JOB=1. The actual pivot sequence used may differ slightly from that of this earlier call if A is not diagonally dominant.
- JOB=3 uses the factors generated by the factorization (a JOB=2 call) to solve a system of equations $Ax = b$.
- JOB=4 combines the actions of JOB=1 with those of JOB=2.
- JOB=5 combines the actions of JOB=2 and JOB=3. It must be preceded by a call with JOB=1.
- JOB=6 combines the actions of calls with JOB=1, 2, and 3.

A call with JOB=3 must be preceded by a call with JOB=2, which in turn must be preceded by a call with JOB=1. Several calls with JOB=2 and several calls with JOB=3 are possible.

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%A (double precision array pointer, dimension NZ), mumps_par%RHS (double precision array pointer, dimension N): Description of the test problem — should be set on the host only:

- N is the order of the matrix A. Not altered by MUMPS.
- NZ is the number of entries being input. 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 pointer 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 only accessed when JOB = 2, 4, or 6.

- RHS is a double precision array pointer of length N 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.

mumps_par%COLSCA, mumps_par%ROWSCA (double precision array pointers, dimension N): Optional scaling arrays required only by the host. If scalings are provided by the user (ICNTL(8)=−1), they should be allocated and initialized on the host only.

mumps_par%PERM_IN (integer array pointer, dimension N): Should be allocated and initialized by the user on the host if ICNTL(7)=1 (ordering given by the user).

mumps_par%MAXIS: Is defined, for each processor, as the size of the integer workspace required for factorization and/or solve. On return from analysis (JOB = 1), INFO(7) 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 than output from the analysis. At the beginning of factorization, MAXIS is set to the maximum of the estimate computed by the analysis and the value supplied by the user. An array IS of size MAXIS is then dynamically allocated, and is used until the end of the solve phase to hold the factors and various contribution blocks.

mumps_par%MAXS: Is defined, for each processor, as the length of the double precision array S required for factorization/solve. A value for MAXS is set in a similar way to that for MAXIS.

mumps_par%ICNTL (integer array, dimension 20): Control array, containing

- ICNTL(1), ICNTL(2), ICNTL(3), ICNTL(4): used to control level/unit numbers for output as used in the HSL package `MA41`.
- ICNTL(5): unused in current version.
- 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 applied.
- 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 should provide scaling arrays in COLSCA

and ROWSCA. If ICNTL(8) = 0, no scaling is performed, and arrays COLSCA/ROWSCA are not used. If ICNTL(8) = 1, .. 6, the package will allocate and compute the scaling arrays COLSCA/ROWSCA:

  - 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.

- ICNTL(9): not used in the current version.
- 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): not used in the current version.
- 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 is applied.
- 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 large enough. 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. It corresponds to the percentage increase allowed to the estimated size of the LU area and of the buffers used for communication. 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) to ICNTL(20) are currently not used.

mumps_par%CNTL (double precision array, dimension 5): Control array, containing

- CNTL(1) has default value 0.01 and is the threshold for pivoting.
- CNTL(2) to CNTL(5) are not used in the current version.

mumps_par%RINFO (integer array, dimension 20): Information array, containing

- RINFO(1) *(produced during analysis)*: The estimated number of floating-point operations on the processor to perform factorization. The value of RINFO(1) on the host is the sum of the estimated number of floating-point operations on all processors.
- RINFO(2) *(produced during factorization)*: The number of floating-point operations on the processor for the assembly process. The host holds the total number of floating-point operations for assembly on all processors.
- RINFO(3) *(produced during factorization)*: The number of floating-point operations on the processor for the elimination process. The host holds the total number of floating-point operations on all processors.
- RINFO(4) to RINFO(20) are currently not used.

mumps_par%INFO(integer array, dimension 20): Information array, containing

- INFO(1) is 0 if the routine is successful, negative if an error occurred (see Section 6.5).
- INFO(2) holds additional information about the error.
- INFO(3) (*produced after analysis*): Estimated real space needed on the processor for factors. On the host: total estimated real space for factors on all processors.
- INFO(4) (*produced after analysis*): Estimated integer space needed on the processor for factors. On the host: total estimated integer workspace for factors on all processors.
- INFO(5) (*produced after analysis*): Estimated maximum front size on the processor. On the host: estimated maximum front size in the complete tree.
- 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. On the host, 0 is returned since the host does not explicitly participate in the computations.
- INFO(8) (*produced after analysis*): Minimum value of MAXS estimated by the analysis phase to run numerical factorization. On the host, 0 is returned.

- INFO(9) (*produced after factorization)*: Size of the real space used on the processor to store the LU factors. On the host: total real space to store LU factors.
- INFO(10) (*produced after factorization)*: Size of the integer space used on the processor to store the LU factors. On the host: total integer space to store LU factors.
- INFO(11) (*produced after factorization)*: Order of the largest frontal matrix processed on the processor. On the host: order of largest frontal matrix.
- INFO(12) (*produced after factorization)*: Number of off-diagonal pivots encountered on the processor. On the host: total number of off-diagonal pivots.
- INFO(13) (*produced after factorization)*: On a processor: 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. On the host: total number of uneliminated variables sent to the father.
- INFO(14) (*produced after factorization)*: On a processor: number of memory compresses on the processor. On the host: total number of memory compresses.
- INFO(15) (*produced after solution*): Number of steps of iterative refinement (on the host only).
- INFO(16) to INFO(20) are currently not used.

2.2 Example of use of MUMPS

An example of the use of MUMPS is given Figure 2. Two files have to be included: mpif.h for MPI and `psl_mumps_struc.h` for MUMPS. 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 `PSL_MUMPS` with JOB=−1, the problem is read in by the host, and the solution is computed with a call on all processors to `PSL_MUMPS` with JOB=6. Finally, a call to `PSL_MUMPS` with JOB=−2 is performed to deallocate the data structures used by the instance of the package.

2.3 Description of the KEEP array (not needed by the user)

The component KEEP is an array that is used by the package for internal communication and to hold machine dependent parameters that are initialized in the call with JOB=−1. It is not altered or referenced by the user but is included here to enhance the description of the package and to record our use of this component.

```
      PROGRAM MUMPS
      INCLUDE 'mpif.h'
      INCLUDE 'psl\_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  Initialize an instance of the package.
      mumps_par%JOB = -1
      CALL PSL_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 PSL_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 PSL_MUMPS(mumps_par)
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

Figure 2: Example program using MUMPS

mumps_par%KEEP(integer array, dimension 50):

- KEEP(1) is used to control the node amalgamation process. Negative or zero values are treated as 1. Increasing the value of KEEP(1) has the effect of decreasing the amount of indirect addressing at the cost of more arithmetic.
- KEEP(2): is the maximum size of a contribution block of a frontal matrix as determined during the analysis phase. It is used to estimate the maximum size of the send and receive buffers.
- KEEP(3) to KEEP(10) control blocking and node parallelism.
  KEEP(3), KEEP(4), KEEP(5), KEEP(6) are used to determine the block size to use in the KJI scheme (Row Gauss elimination). Let NASS be the number of fully assembled variables at a node and NFRONT the order of the corresponding frontal matrix.

  For local blocking to be used in the factorization, it is necessary that NASS > KEEP(4).
  If NASS > KEEP(3), then
  block size for KJI scheme is KEEP(6)
  else
  block size for KJI scheme is KEEP(5)
  endif
  The following should always be satisfied:
  KEEP(5) $\leq$ KEEP(4) $\leq$ KEEP(3) and KEEP(6) $\leq$ KEEP(3)
  For node parallelism to be activated, it is necessary that
  NFRONT–NASS > KEEP(9)
  The contribution block of size NFRONT–NASS is then subdivided into strips of size KEEP(10) (KEEP(10) $\leq$ KEEP(9)).
  KEEP(7) and KEEP(8) are not used in the current version.
  For example, the following values could be used; optimal values are machine-dependent.

```
KEEP(3)  = 96
KEEP(4)  = 32
KEEP(5)  = 16
KEEP(6)  = 32
KEEP(9)  = 800
KEEP(10) = 200
```

- KEEP(11) is unused in current version.
- KEEP(12) corresponds to the percentage increase allowed to the estimated size of the LU area and of the buffers used for communication. It holds a copy of the value set in ICNTL(14).

- KEEP(13) and KEEP(14) hold the sizes of the arrowheads held on the processor (see Section 6.2). KEEP(13) corresponds to the real entries. KEEP(14) to the integer entries.
- KEEP(15) is set during analysis to an estimate of the integer LU area size. The integer LU area is defined to be the integer space needed to store the integer information associated with the factors and the contribution blocks during the factorization phase.
- KEEP(16) is set during analysis to an estimate of the real LU area size.
- KEEP(17) is currently used to increase workspace estimates for factorization. This allows better performance by avoiding compressing the large arrays IS/S. This is different from KEEP(12) that is used as an estimate of the increase of memory for both the buffers and IS/S.
- KEEP(18) is set during factorization phase to the size in real words allocated to the LU area.
- KEEP(19) to KEEP(22) are unused in current version.
- KEEP(23) is set to 1 if during the analysis phase, a maximum transversal was found (ICNTL(6)=1) and if this led to a column permutation different from the identity. Otherwise, it is set to 0.
- KEEP(24) is not used in the current version.
- KEEP(25) to KEEP(33) are defined for each processor.
- KEEP(25) Estimated real space needed for factors.
- KEEP(26) Estimated integer space needed for factors.
- KEEP(27) Estimated maximum front size.
- KEEP(28) Number of nodes in the tree.
- KEEP(29) Minimum value of MAXIS estimated by the analysis phase to run the numerical factorization.
- KEEP(30) Minimum value of MAXS estimated by the analysis phase to run the numerical factorization.
- KEEP(31) Size of the real space used to store the LU factors.
- KEEP(32) Size of the integer space used to store the LU factors.
- KEEP(33) Order of the largest frontal matrix.
- KEEP(34) Size of an integer in bytes.
- KEEP(35) Size of a real in bytes.
- KEEP(36) Used to detect termination of factorization in current version.
- KEEP(37) Minimum size of a root node to be factorized using ScaLAPACK.

- KEEP(38) is 0 if no root node has been selected for parallelization, or is INODE if INODE is the selected parallel root node.
- KEEP(41) holds the total number of contributions to be received by a process involved in the root (dynamic).
- KEEP(42) holds the total number of delayed pivots in the root node.
- KEEP(43-44) are set during analysis to an estimate of the minimum sizes of the buffers for receiving and sending, respectively. These buffers are allocated during analysis.
- KEEP(45) is a copy of ICNTL(15) which determines the strategy for mapping the top levels of the tree.
- KEEP(46) to KEEP(50) are unused in the current version of the package.

3 Parallelism of factorization — 3 Types of nodes

We consider the assembly tree of Figure 3 where, instead of single nodes as the leaves, there are subtrees whose constituent nodes have frontal matrices of small order. Each leaf subtree is processed by a single processor to avoid communication at that stage. This mapping of leaf subtrees to processors is performed by the analysis phase. For large problems, there will be more leaf subtrees than processors which will aid in the overall load balancing of the computation.

Above the leaf subtrees, there can still be some nodes processed by only one processor. These nodes (as well as nodes inside the leaf subtrees) are called nodes of Type 1. The parallel root node (if one exists) is of Type 3. The other nodes, in which we exploit node level parallelism, are called nodes of Type 2. We describe these more in the following subsections.

3.1 Nodes of Type 2

Let INODE be a node, with NASS variables to eliminate, and NCB rows in the contribution block. NFRONT = NASS + NCB is the order of the frontal matrix associated with the node. A node is of Type 2 if it is not a root and if the following condition is satisfied:

1. NCB > KEEP(9),

where KEEP(9) depends on the target architecture.

If a node is of Type 2, one processor (called the master of the node) holds all of the rows in the fully summed block and performs the pivoting and the factorization on this block while other processors (slaves) perform the updates on the contribution block. In Figure 4, P0 performs a blocked factorization, and sends its blocks (of size KEEP(5) $\leq$ KEEP(4)) to P1, P2, P3 to perform the updates. The distribution

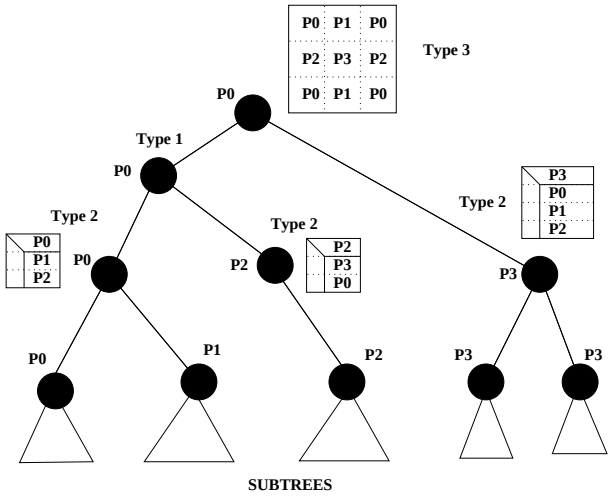

Figure 3: Distribution of the computations of a multifrontal tree

of the contribution block is decided in the analysis phase. It is distributed to NCB/KEEP(10) processors, different from P0. This strategy avoids any need for communication when choosing pivots.

This strategy for nodes of Type 2 can be justified by the fact that sufficiently near the root of the tree, there is not enough tree parallelism to keep all the processors busy. Macro-pipelining is used to overlap communication with computation. KEEP(10), the target number of rows in a block, should be set large enough to limit overheads due to start-up in the communications but small enough to provide sufficient parallelism.

3.2 Root node — or Type 3 node

In order to have good scalability for the root node, we use a 2-D cyclic distribution. We use ScaLAPACK [2] or the vendor equivalent implementation in the current version of the code.

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 some constant, KEEP(37). One processor is said to be the master of the root, and holds the indices.

Before factorization, the parallel root node is assembled in a 2D grid of NPROW by NPCOL processors, with block sizes MBLOCK and NBLOCK.

We use a static distribution and mapping for variables known by the analysis to be in the root node so that for an entry (i, j) in the root node, we know where to send it and assemble it using functions involving integer divisions, moduli, ... However, because of possible delayed pivots, we then need to modify the structure of the frontal matrix at the root node (the *root matrix*) to include the delayed pivots.

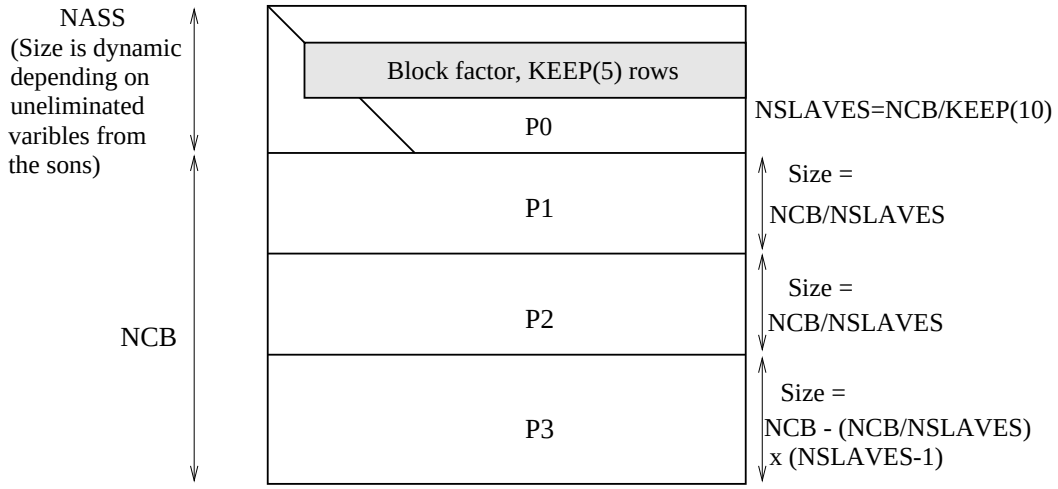

Figure 4: Structure of a node of Type 2

In detail, this is done in the following way:

- A grid of processors is defined statically for the root (NPROW by NPCOL, depending on the number of processors available), without taking into account the uneliminated variables of the son nodes. Each son node can then know to which processor it should send its contributions and the positions of its contributions in the root node.

- All the static contributions can be assembled at the root. This is all asynchronous.

- The master of the root receives a list of delayed pivots from each master of its sons. A local index is assigned to each delayed pivot in the root and is sent to the son processors.

- The structure of the root node is modified because of delayed pivots (the leading dimension of the root matrix might change).

- According to the indices associated with the delayed pivots, the sons of the root send their uneliminated blocks of contributions to the appropriate processor of the root node.

- Reception and assembly of uneliminated contributions from the sons is performed.

- The ScaLAPACK LU factorization is called.

The assembly for a cyclic 2-dimensional grid is based on the algorithm described in Section 6.3.

Also the mechanism for stacking is different for the Type 1 and Type 2 sons of the root in order to send the static and dynamic contributions in two different steps.

More details concerning the messages involved in the assembly of the root node and actions performed on reception are given in Section 4.5.

3.3 Estimated speed-ups

In Table 2, we report estimates of the theoretical maximum speed-up[1] with different strategies, based on the number of flops for the factorization. Corresponding test problems are described in Table 1.

Matrix	N	NZ	Tree depth	Total flops (10^6)	Number of nodes	Number of leaves	Size of root
goodwin	7320	324784	20	194	837	360	272
k15	3948	117816	15	327	588	295	409
wang3	26064	177168	22	10476	12661	10614	1601
wang4	26068	177196	20	10523	12175	10079	1716

Table 1: Statistics for test problems

	Complete tree			Tree without root	
Matrix	Level 1	Level 2	Level 3	Level 1	Level 2
goodwin	1.73	1.88	2.09	1.83	2.01
k15	1.98	4.58	10.0	2.35	11.1
wang3	1.38	3.08	13.8	1.60	11.9
wang4	1.67	2.78	20.3	2.45	18.2

Table 2: Estimated speed-up. Level 1 = Tree parallelism only,; Level 2 = 1D partitioning, only for matrices with front size > 200; Level 3 = Level 2 + 2D partitioning of root (block-size 64).

3.4 Driver for factorization

An SPMD procedure is applied, with the following algorithm.

If a node is mapped onto one processor but is of Type 2, other processors participate in the assembly/factorization. The processor on which the node is mapped is called the master of the node and is the one that initiates the assembly/factorization phase of that node.

[1] An infinite number of processes is assumed to be available. The theoretical maximum speed-up is computed to be the ratio of the total amount of work divided by the amount of work along the longest path. Operation counts for the update of contribution blocks by slaves, for nodes of Type 2 with front size larger than 200, are not included in the counts for determining the longest path.

For each processor, a pool of tasks is used. Initially only the leaf nodes are in the pool. When the master of a node detects that a new node can be assembled, it is put in the pool.

It is necessary to count the number of processed root nodes to detect termination of the algorithm.

The following algorithm should be executed on each processor.

```
Put the leaf subtrees allocated to the current processor into the pool.
NBRTOT ← Global number of root nodes
NBFIN ← NBRTOT ! Will hold total number of root nodes left to process
NBROOT_TRAITEES ← 0 ! Current number of local root nodes processed
NBROOT ← Number of root nodes on this processor.
Begin Main loop
  if pool is empty, then
    perform a blocking receive ! Wait for a message
  else
    try to receive a message ! Priority given to receiving messages
  end if
  if a message has been received, then
    process the message ! See Sections 4.4 and 4.5
    ! this can assemble a contribution, update NBFIN, ...
    if error or NBFIN = 0 Exit Main Loop
    ! In case of error, error will have already been posted to other processors
  else ! No message arrived
    Extract a node INODE from the pool
    if INODE > N then ! factorization should be performed
      INODE ← INODE - N
      if INODE is parallel root then
        Call Update_root
        if NBFIN = 0 then
          Exit Main Loop
        else
          Cycle Main Loop
        endif
      endif
      if INODE is of Type 1 then
        Call Facto_niv1
      else ! INODE is of Type 2
        Call Facto_niv2
      endif
      Call Stack
```

```
        if INODE is a Root then
          Call Update_root
          if NBFIN = 0 then
            Exit Main Loop
          else
            Cycle Main Loop
          endif
        else if Father(INODE) is on my processor and
            is ready for assembly then
            Add Father(INODE) to the pool
        endif
      else ! INODE ≤ N, assembly operation
        if INODE is of Type 1 then
          Call assembly_niv1(INODE)
          if error Exit Main Loop
          if INODE has sons of Type 2, then
            Cycle Main Loop
            ! INODE will be inserted in the pool for factorization
            ! only once assembly is finished
          else
            Put INODE + N in the pool
            ! In the actual code, factorization is started
          endif
        else ! INODE is of Type 2
          Call assembly_niv2(INODE)
          if error Exit Main Loop
        end if ! INODE is of Type 1
      end if ! Test of INODE compared to N
    end if ! A message has been received
End Main Loop
Return from procedure.

Procedure Update_root
    NBROOT_TRAITEES ← NBROOT_TRAITEES + 1
    if NBROOT_TRAITEES = NBROOT then
      NBFIN ← NBFIN - NBROOT
      Send a message with TAG=RACINE containing NBROOT
      to other processors
    end if
end Procedure Update_root
```

If there is a parallel root, the assembly has to be performed by each processor in the grid for the root. This is done while receiving messages and INODE + N is inserted in the pool when assembly is finished. The initialization of the NBRTOT and NBROOT is slightly changed for termination detection:

1. NBRTOT is the sum of the total number of Type 1 root nodes and the number of processors participating in the root node.
2. NBROOT is the number of root nodes to be treated on the local processor, including the parallel root.

4 Communications and algorithms for the factorization phase

In this section, we first describe the code where tree parallelism only is used, then we give an example of how a parallel node is assembled and factorized. We then describe some of the data structures used for the factors and contribution blocks. Finally, we describe in more detail the actions performed depending on the messages received.

4.1 Multifrontal distributed factorization, tree parallelism only

In the previous version of the code, only tree parallelism was implemented. A PVM version of this code has been developed by Amestoy and Espirat, and is described in [6]. This was announced in January 1997 [10]. The MPI version, referred to as MUMPS Version 1.0, was available from May 1997.

4.2 Example of assembly/factorization of two Type 2 nodes of the tree (Figures 5 and 6)

An example seems to be the best way to explain how assembly/factorization is performed. Let us consider two Type 2 nodes in the multifrontal tree: a son and its father.

In the analysis phase, the son node was mapped to processor P0, and was determined to be of Type 2. The rows of the contribution block were then subdivided into strips of size KEEP(10). For the sake of simplicity, we assume there is only one strip in this subdivision and we assume that this is allocated to processor P1. In the current code, we choose the processors to allocate dynamically in a round robin manner. However, this choice could be statically computed by the analysis, or processors could be chosen dynamically depending on the loading. P0 is called the master of the son, P1 is the first (and, in this case, only) slave of the son.

Similarly, the father node has processor P2 for its master, and its contributions are divided between two slaves (slave 1 and slave 2) on processors P3 and P4.

We suppose that the son node has finished its factorization phase, and has $NASS$ fully summed variables, $NPIV$ eliminated pivots, $NELIM$ delayed pivots ($NPIV + NELIM = NASS$) and that the size of the contribution block (including delayed pivots) is $NCB \times NCB$, with $NCB + NPIV = NFRONT$. In our case, the only slave (slave 1) holds $NCB - NELIM = NFRONT - NASS$ rows.

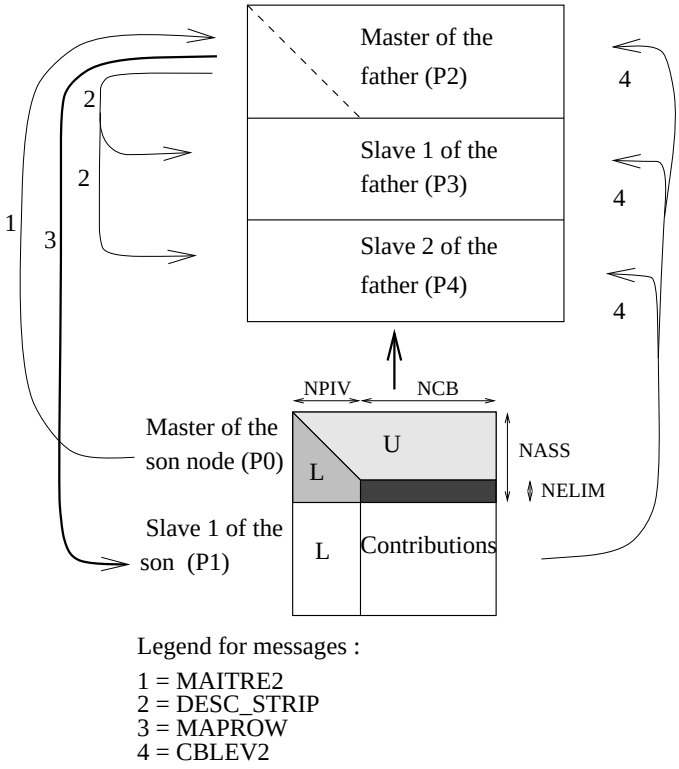

Figure 5: Messages involved in the assembly of a node of Type 2

The following messages are necessary :

Assembly

- First, the master of the son node (P0) sends the structure of the contribution of the son ($NCB \times NCB$), and the numerical values for the block of delayed pivots ($NELIM \times NCB$) to the master of the father (P2). This is a message with TAG=MAITRE2.

- Once P2 has received a message MAITRE2 from all its sons, the assembly of its indices can be performed: this task is put in the pool and a task is extracted from the pool as soon as P2 has no more messages to receive. When the assembly is started, P2 computes the structure of the father node (assembly of indices), and assembles entries from the original matrix in arrowhead format, and delayed pivots received from the masters of its sons.

- P2 sends a message to P3 and P4 defining the structure of the strip: message DESC_STRIP. On reception, P3 or P4, respectively, allocate space for the strip.

- P2 sends a message with TAG=MAPROW to all slaves of its sons, in our case to P1. This message is an integer array of size $NCB - NELIM$ containing the position in the father node of the variables of the slave of the son.

- On reception of MAPROW, P1 sends its contributions to P2, P3, and P4 (TAG=CBLEV2). P1 knows where to send the contributions because of information contained in the message MAPROW.

- On reception of CBLEV2, P2, P3, and P4 assemble the contributions into the structure. When all contributions are received by P2, the father node is placed in the pool of P2 which means that the factorization can be started.

Assembly should be quite efficient since it is performed in parallel and because the assembly of the fully summed sons is done as fast as in the sequential code. Shifts for rows and columns are stored and not recomputed for optimization.

Factorization of the father node

Factorization of a node is started by the master of the node. A block factorization is used with pivoting on rows and columns.

The details of the factorization are given in Figure 6.

$NPIV$ is a running count of the number of eliminated pivots. Initially $NPIV$ is zero. After factorization, $NPIV$ is the number of eliminated pivots. The master of the node tries to factorize the next block of size $NBLPIV \times NBLPIV$. $NBLPIV$ is initially taken equal to KEEP(5), but can dynamically increase if there are numerical problems. (If only a few pivots in a block can be eliminated, it helps to use larger blocks.)

Let $NPIV1$ be the number of pivots eliminated in the current block of size $NBLPIV$.

- The master of the node (P2) sends a block of rows of size $NPIV1 \times LROW$, where $LROW = NFRONT - NPIV$ to P3 and P4, and an integer array of size $NPIV1$ defining the column pivot sequence.

- $NPIV \leftarrow NPIV + NPIV1$.
- On reception P3 and P4 perform the same column interchanges as was performed on P2, on the block of rows that they own. Then they update their block of rows using the BLAS routines TRSM and GEMM.

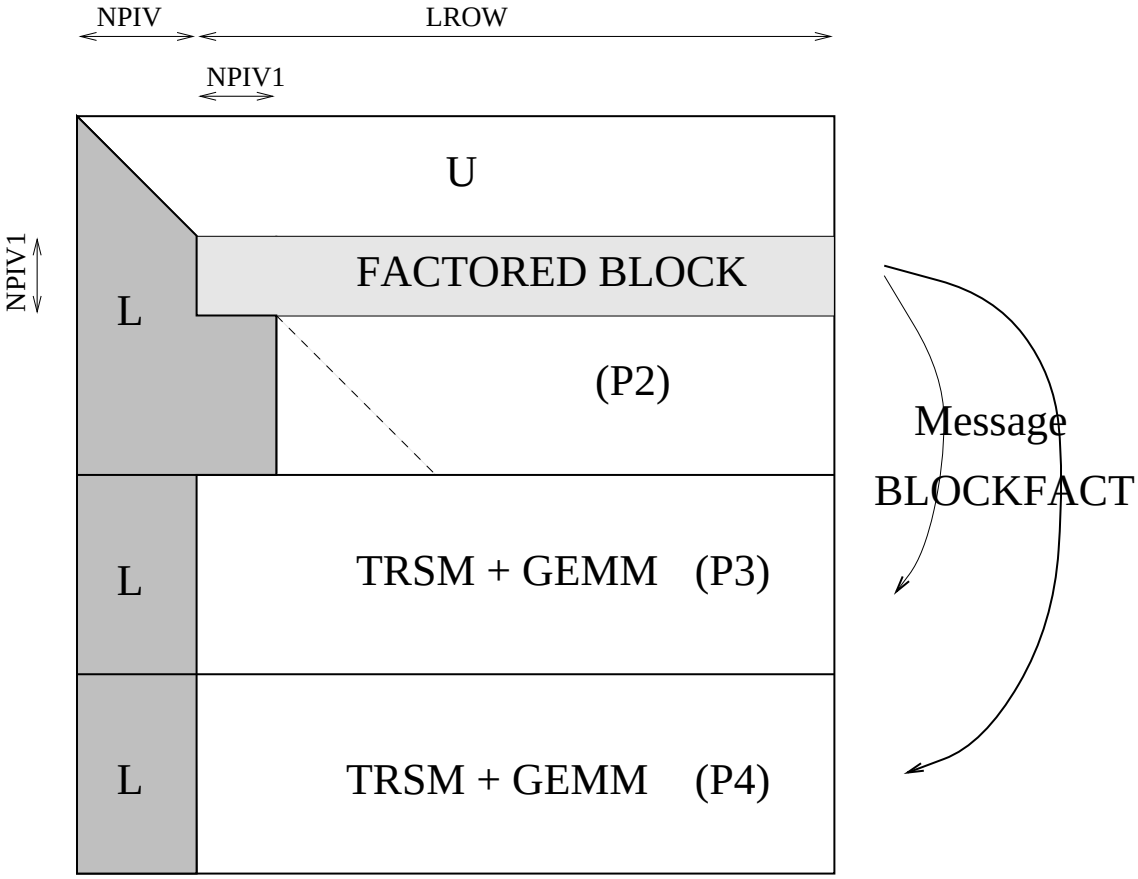

Figure 6: Factorization of a node of Type 2

It can happen that the master of a son node is on the same processor as a slave of the father node. If this occurs, in order to avoid sending a message to itself, we need to perform the action that would be done at reception instead of sending the message.

Also, notice that all messages are asynchronous. This means that processors should, as much as possible, not be blocked on reception of a message.

When, for example, processor P3 has received DESC_STRIP and can start assembling the contributions from the son, it can receive other messages concerning other nodes or perform some work available in its pool while contributions are sent to it. Then when it actually receives a message CBLEV2 concerning this father node, assembly is performed on the fly.

4.3 Data structures for factors and contribution blocks

During factorization, an integer array IS of size MAXIS and a double precision array S of size MAXS are used. Basically the beginning of these arrays is used to store factors, and the end to store contribution blocks before they are assembled. In practice, the subdivision is slightly more complicated.

If ISON and IFATH are a son node and its father in the tree, then the following arrays of pointers are held on each processor:

- PIMASTER(ISON) and PAMASTER(ISON) point to the integer and real description, respectively, of the contribution block from the master of ISON to be assembled in IFATH.
- PTRIST(IFATH) and PTRAST(IFATH) point to the assembled father node. This might be the complete father node or just a strip.
- PTLUST(IFATH) points in IS to the integer description of the factors of IFATH (left part of the array).

Apart from the factors, most records concerning a node in IS have a similar structure: a header of size 6, followed by a list of slave nodes, followed by row indices and column indices. The header contains some information about what is stored, the size, and the number of the associated reals.

Some records in IS have only size 3, but have in that case a negative first entry.

Some records are allocated/deallocated, and there can be holes in the arrays, but the strategy used allows us to compress the memory when workspace for contribution blocks, factors, and temporary space for messages is not directly available.

4.4 Message types and associated actions

In order to perform parallel assembly and parallel factorization of a node of Type 2 in a distributed environment, we use tagged messages. Usually, a processor probes for the arrival of messages and, if a message is received, applies the action corresponding to the tag of the message, otherwise, it continues with work available in its pool. Message types, defined by their tags, and actions to perform on reception, are described in the following subsections.

4.4.1 TERREUR

A message with TAG=TERREUR is sent to all processors when an error is detected.

On reception of a message with this tag, the receiver sets INFO(1) to –1, and INFO(2) to the sending processor, then returns (see Section 6.5). Subsequent checkings on INFO(1) < 0 will finally cause a return from MUMPS.

4.4.2 RACINE

A message with tag = RACINE is used to detect termination of the factorization. It is sent by a processor that has finished all the root nodes it owns and contains an integer NBROOT equal to the number of roots the processor has processed. On each processor, a variable NBFIN is initialized to the total number of root nodes in the tree, and is decremented by the value of NBROOT when a message with tag RACINE is sent or received. If NBFIN is 0, the factorization is finished and the processor can return.

4.4.3 NOEUD

A message with tag = NOEUD contains a contribution block from a son node ISON of Type 1 to a father node IFATH of Type 1.

Content:

- ISON
- IFATH
- LCONT, the size of the contribution block
- Header:

Position	1	LCONT
Position	2	NASS–NPIV
Position	3	LCONT
Position	4	0
Position	5	1
Position	6	0

- IROW(LCONT)
- ICOL(LCONT)
- VAL(LCONT*LCONT)

On reception of a message with tag = NOEUD, the header and what follows is stored in the CB area of arrays IS and A in locations pointed to by PIMASTER(ISON) and PAMASTER(ISON).

If this was the last contribution concerning IFATH to be received, the assembly action of IFATH is put in the pool.

4.4.4 MAITRE2

A message with tag = MAITRE2 contains a contribution

- from the master of node ISON of Type 2 to the master of node IFATH of Type 2, or
- from the master of node ISON of Type 2 to the processor owning node IFATH of Type 1, or
- from the processor owning node ISON of Type 1 to the master of node IFATH of Type 2.

The message contains the uneliminated fully summed variables of the son node and has the following structure:

- IFATH
- ISON
- NSLAVES
- ISLAVES(NSLAVES)
- NROW
- IROW(NROW)
- NCOL
- ICOL(NCOL)
- VAL(NROW*NCOL)

On reception, the message is stacked in the CB area and pointed to by PIMASTER(ISON) and PAMASTER(ISON) with the following header:

Position	1	NCOL
Position	2	NROW
Position	3	NROW
Position	4	0
Position	5	1
Position	6	NSLAVES

The header is followed by the list of slaves, the indices of the rows and the indices of the columns. Also, uneliminated entries from the son node are stacked in the double precision array A, pointed to by PAMASTER(ISON).

Finally, if the block is the last contribution received, the assembly action for IFATH is put in the pool.

4.4.5 CBLEV2

A message with tag = CBLEV2 contains a contribution

- from the slave of a node ISON of Type 2 to the master of a node IFATH of Type 2, or
- from the slave of a node ISON of Type 2 to a slave of a node IFATH of Type 2, or
- from the processor owning a node ISON of Type 1 to a master/slave of a node IFATH of Type 2, or
- from the slave of a node ISON of Type 2 to the processor owning a node IFATH of Type 1.

The structure of the message is the following

- IFATH,
- ISON,
- NBROW,
- LROW,
- if the destination is a slave, COLIND(LROW), the columns indices. (if the destination is a master, the column indices have already been sent by the master of ISON to the master of IFATH).
- for i=1 to NBROW
  - the position in the father node of the row sent, and
  - ROW(LROW) the ith row.

On reception, the message is processed "on the fly" and does not need to be stacked in the CB area. If the destination is the master then the structure of the node is known and already allocated. If the destination is a slave, we may have to wait for the reception of a message DESC_STRIP in order to allocate the block of rows and assemble into it. In practice, this message has been sent before and has almost always been received.

4.4.6 DESC_STRIP

A message with TAG = DESC_STRIP contains the description of a block of rows. It is sent from a master to a slave of the same (Type 2) node INODE.

It contains :

- INODE: the concerned node,
- NBPROCFILS: the number of processors involved in the sons of node INODE,
- NROW: the number of rows that should be processed by the slave,
- NCOL: the number of columns in the frontal matrix,
- NASS: the number of assembled variables,
- IROW(NROW): the indices of the rows,
- ICOL(NCOL): the indices of the columns.

On reception by a slave processor of node INODE, space is allocated in the CB area. PTRIST(INODE) points in IS to a zone containing 6 + NCOL + NROW integers, and PTRAST points in A to a zone containing NCOL * NROW reals:

- Header:

Position	1	NCOL
Position	2	–NASS
Position	3	NROW
Position	4	0
Position	5	PTRAST(INODE)
Position	6	0

- IROW(NROW),
- ICOL(NCOL),
- NCOL * NROW reals.

Position 2 in the header contains –NASS as a flag to indicate that the arrowhead has still to be assembled. Once this has been done, during the reception of a contribution block of a son, this entry is set positive so that the arrowhead is not assembled again.

During factorization, when a block is received, position 1 in the header is decreased by NPIV, position 4 is increased by NPIV.

4.4.7 BLOCKFACT

A message with TAG = BLOCKFACT contains :

- INODE,
- NPIV1, size of the square pivot block factorized by the master,
- IPIV(NPIV1), column pivoting array (row pivoting is not performed on slave processors),
- VAL(NPIV1*LROW), the factorized block, where LROW is the front size less the number of already received pivots for this processor and this node.

On reception of BLOCKFACT, pivoting and updates are performed on the contribution block of rows of INODE owned by the current process. If all the contributions have not yet been received yet, the processor waits for them to finish the assembly first, but this almost never happens.

4.5 Messages and actions performed for the parallel root node

In our discussion of messages and actions performed for the parallel root node we will use the diagram in Figure 7. The mechanism implemented for the root node ensures full parallel assembly of the contributions. Also, the contributions, at least the static ones, can be assembled as soon as they are available.

Messages with the following tags are used:

4.5.1 Root_Cont_Static

First, static contributions, whose structure is already known from the analysis phase, are sent to the root node (they are sent immediately to the appropriate local position on each root processor). A message with the tag Root_Cont_Static is sent from a processor involved in a son of the root node to a processor involved in the root node and contains:

- ISON: the son node,
- NROW: the number of rows sent,
- NCOL: the number of columns sent,
- POSROW(1:NROW): the row positions in the root matrix on this processor where assembly should be performed,

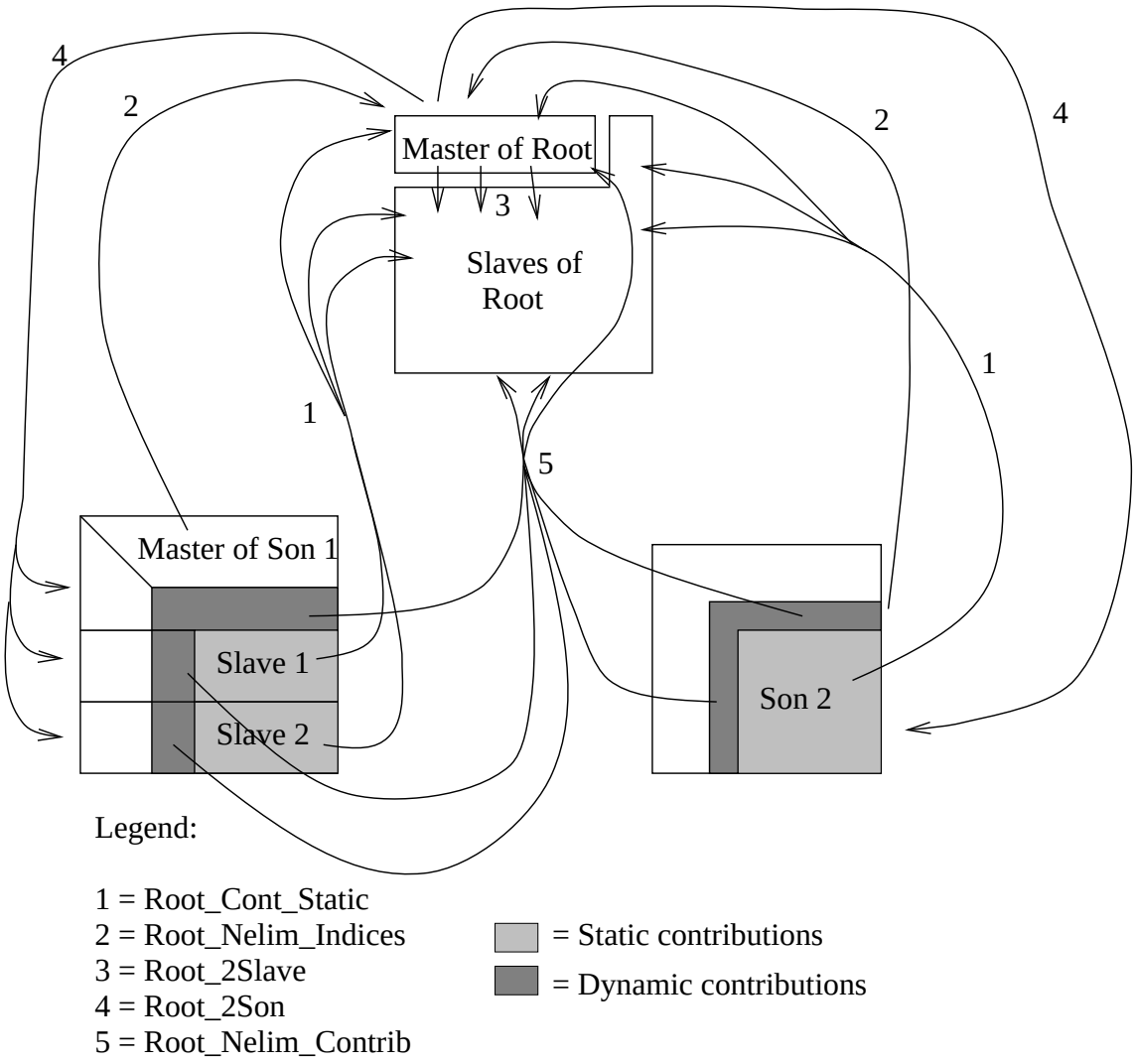

Figure 7: Messages involved in the assembly of the parallel root node

- POSCOL(1:NCOL): the column positions in the root matrix on this processor where assembly should be performed,
- VAL_SON(1:NROW,1:NCOL): the numerical values of the contribution block.

On reception, the contributions are assembled in a simple loop for which the code is of the form

```
VAL_ROOT(POSROW(i),POSCOL(j)) =
VAL_ROOT(POSROW(i),POSCOL(j)) + VAL_SON(i,j)
```

When the last contribution is received, the root node is put in the pool, so that the main algorithm (Section 3.4) can detect that it is ready for factorization and can perform adequate termination detection.

4.5.2 Root_Nelim_Indices

This message is sent from a master of a son of the root to the master of the root. It contains:

- ISON: the son node,
- NELIM: the number of uneliminated variables of ISON,
- NSLAVES: the number of slaves of ISON,
- NELIM_ROW(1:NELIM): the list of uneliminated rows of ISON,
- NELIM_COL(1:NELIM): the list of uneliminated columns of ISON,
- SLAVE_LIST(1:NSLAVES): the list of slaves of ISON.

On reception, this information is stacked in the CB area, and is pointed to by PIMASTER(ISON) with a header of size 6 as follows:

Position	1	2*NELIM
Position	2	NELIM
Position	3	0
Position	4	0
Position	5	1
Position	6	NSLAVES

Also, a count of the total number of contributions that will be received and the total number of delayed pivots are stored in KEEP(41) and KEEP(42), respectively.

When the message Root_Nelim_Indices from the last son is received, the master of the root builds the complete structure of the root (including delayed pivots) in the LU area; it is pointed to by PTLUST(IROOT) and PTRAST(IROOT). The messages Root_2Slave and Root_2Son are sent:

- Root_2Slave tells other processes involved in the root to build their new structure for the root with delayed pivots included.
- Root_2Son gives the necessary information to son processors so that they can start sending their dynamic contributions.

Indeed, in the actual code, when the last message with tag = Root_Nelim_Indices is received, the tasks just described are put in the pool so that priority is given to receiving other messages rather than to sending the messages Root_2Slave and Root_2Son.

4.5.3 Root_2Slave

This message is sent from the master of the root to other processors involved in the root. It contains

- TOT_ROOT_SIZE: the total size of the root including the delayed pivots.
- TOT_CONT_TO_RECV: the total number of contributions that the processors of the root should wait for.

TOT_CONT_TO_RECV is computed by the master of the root when receiving messages with tag = Root_Nelim_Indices and is the sum for all sons ISON of the root of:

- NSLAVES if ISON is of Type 2 with NELIM = 0,
- 2 * NSLAVES + 1 if ISON is of Type 2 with NELIM $\neq$ 0,
- 1 if ISON is of Type 1 with NELIM = 0,
- 3 if ISON is of Type 1 with NELIM $\neq$ 0.[2]

On reception, a processor involved in the root builds the complete structure of its part of the root (including delayed pivots) in the LU area, pointed to by PTLUST(IROOT) and PTRAST(IROOT).

TOT_CONT_TO_RECV is then used to detect termination.

4.5.4 Root_2Son

This message is sent from the master of the root to a processor involved in the treatment of a son of the root. It is sent only to sons for which there are some delayed pivots (NELIM $\neq$ 0) and contains:

- ISON: the son concerned.

[2] In the latter case, there is one static contribution and the dynamic part is divided into two rectangular blocks. See Figure 8.

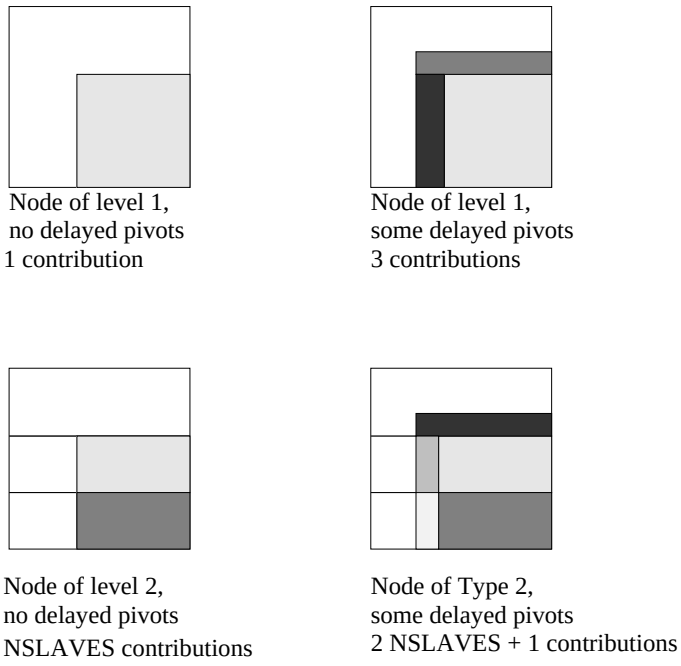

Figure 8: Number of contributions to send depending on the type of the node

- NELIM_ROOT: the position in the root node chosen by the master of the root for the first delayed pivot of the son.

This message allows each processor to send its uneliminated contributions to the correct processor in the root and to the appropriate position. The two arrays RG2L_ROW and RG2L_COL are used to map a global variable to its position in the root node, and the algorithm of Section 6.3 is used to determine where the contributions should go.

4.5.5 Root_Nelim_Contrib

This message contains a dynamic contribution to the root node. The contribution is assembled in the same way as if it were a static contribution (see Section 4.5.1).

5 Solve phase

An initial version of the solve reassembled the factors for all nodes (it acted as if all nodes were of Type 1). Now, we use the distribution of the factors obtained from the factorization.

The solve is based on the same elimination tree, which is processed

- from the leaves to the roots, for the forward elimination, and

- from the root to the leaves, for the backward substitution.

Between forward and backward eliminations, the possible parallel root node (normally the largest root node) is solved using the ScaLAPACK routine PDGETRS, that should be called in a SPMD way.

If there is a parallel root, the algorithms in 5.1 and 5.2 are modified slightly so that the forward elimination returns a dense right-hand side vector corresponding to the root (called RHS_ROOT), and the backward substitution accepts on entry a corresponding already computed solution.

Between these two steps, RHS_ROOT is redistributed between the processors, PDGETRS is called, and the solution is reassembled and used as an input parameter to the backward substitution. The details of interfacing the root with forward and backward substitution is not mentioned in the algorithms to avoid overcomplicating their description. There is a simple test in these routines that avoids computing the solution for the root because either it will be, or it has already been, computed by ScaLAPACK.

5.1 Forward elimination

For the forward elimination, the tree is processed from the leaves to the roots. We describe, in this section, the algorithm used to perform the forward elimination, except the parallel root. This algorithm should be executed by all processors.

5.1.1 Main algorithm

Initialize the pool with local leaf nodes
Begin Main Loop
  if *the pool is empty,* **then**
    Perform a blocking receive
  else
    Try to receive a message
  end if
  if *a message has been received,* **then**
    Treat the message
    ! this can store a contribution, detect termination, insert a node in the pool,
    ! or perform an update, see Section 5.1.3
  else *! No message*
    Extract a node INODE from the pool
    Call **forw_solve_node***(INODE)*
  end if
  if *error* **or** *Termination detected* **Exit Main Loop**
End Main Loop
Return

5.1.2 Details of forw_solve_node

The subroutine **forw_solve_node** is called by the processor holding the node INODE. In the case where INODE is of Type 2, the lower triangular factors of INODE have the structure described in Figure 9.

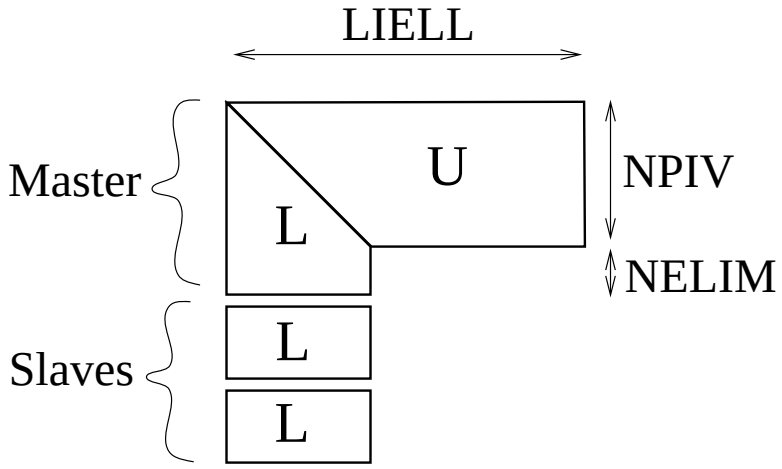

Figure 9: Factors of a node INODE

Let INODE be a node with frontal matrix of order LIELL, with NPIV eliminated pivots, and NELIM delayed pivots from the initial fully summed block used in the factorization.

RHS is a vector of size N that initially holds the right-hand side vector, and is corrupted.

W is a working array that has at least the size of the maximum front.

The subroutine **forw_solve_node**(INODE) performs the following steps:

Assemble the contribution vectors for INODE into $\mathbf{W}_{1:LIELL}$
Add contributions from right-hand side into $\mathbf{W}_{1:NPIV}$
$\mathbf{W}_{1:NPIV} \leftarrow \mathbf{L}^{-1}_{1:NPIV,1:NPIV}\mathbf{W}_{1:NPIV}$. *! Solve for the pivot block (DTRSV)*
Scatter $\mathbf{W}_{1:NPIV}$ *back into* **RHS**.
if *INODE is of Type 2* **then**
  $N_{UPD} \leftarrow NELIM$
else
  $N_{UPD} \leftarrow LIELL - NPIV$ *! all contribution block should be updated*
end if
if $N_{UPD} \neq 0$ **then**
  ! Update contributions local to master of the node
  $\mathbf{W}_{NPIV+1:NPIV+N_{UPD}} \leftarrow \mathbf{W}_{NPIV+1:NPIV+N_{UPD}} -$
  $\mathbf{L}_{NPIV+1:NPIV+N_{UPD},1:NPIV} \times \mathbf{W}_{1:NPIV}$

if *Father of INODE is on the same processor* **then**
Store locally $\mathbf{W}_{NPIV+1:NPIV+N_{UPD}}$ *and*
the corresponding global indices in the CB area
else
Send $\mathbf{W}_{NPIV+1:NPIV+N_{UPD}}$ *and the*
corresponding integer indices to the master of
the father of node INODE
end if
end if
if *INODE is of Type 2,* **then**
! Strips of L are on remote processors
for *each slave of INODE*
Send to current slave a message with Tag = Master2Slave containing
- INODE
- $W_{1:NPIV}$ *! the partial solution*
- appropriate components of $W_{NPIV+1:LIELL}$
end for
end if

5.1.3 Actions performed on reception

During the forward elimination, the messages with the following tags can be sent and received:

- TAG=RACINE_SOLVE:

  Same mechanism as for factorization. Used to detect termination.

- TAG=TERREUR:

  Same behaviour as for factorization.

- TAG=ContVec:

  On reception of a contribution vector, the contribution is stored in the contribution area with appropriate pointers. If a node can then be activated (all contributions received), it is put in the pool for further execution of forw_solve_node.

  More precisely, this mechanism works as follows:

  The message (tag=ContVec) was sent from a processor of son node ISON to the master of its father node INODE. It contains:

  -INODE,
  -ISON,
  -NCB, which is the size of the contribution block for the son,

-LONG, the size of the contribution sent,
-IW(1:LONG), the global indices,
-W(1:LONG), the real contributions

An array NSTK(INODE) has initially been initialized with the number of sons of INODE. Another array PTRICB has been initialized to 0 at the beginning of the solve phase.

if *PTRICB(ISON) = 0* **then**

! this is the first contribution received from ISON
Allocate a contribution vector consisting of NCB + 2 integers
and NCB reals in the contribution area, and
set pointers PTRICB(ISON) and PTRACB(ISON)
! The integers contain a header of size 2: $\mathbf{H}(2)$
! and will be followed by global indices
! The real part will contain numerical values of contribution
! This contribution vector is accessed using
! the pointers PTRICB(ISON) and PTRACB(ISON)
$\mathbf{H}(1) \leftarrow$ *NCB + 1*
$\mathbf{H}(2) \leftarrow$ *1*
end if

Fill the integer and real contribution with IW and W respectively,

in position $\mathbf{H}(2)$.

$\mathbf{H}(2) \leftarrow \mathbf{H}(2)+$ *LONG*

if $\mathbf{H}(1) = \mathbf{H}(2)$ **then**

! All contributions for ISON have been received

NSTK(INODE) $\leftarrow$ *NSTK(INODE) - 1*

end if

if *NSTK(INODE) = 0* **then**

Insert INODE in the pool

end if

- TAG=Master2Slave:

  The message contains:

  - INODE
  - x (1: NPIV) = solution
  - y (1: NROWS) = vector to update

On reception, a processor holding a strip of the lower triangular factor for node INODE, performs the update $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{Lx}$ and sends the contribution vector $\mathbf{y}$ to the father node.

5.2 Backward substitution

For the backward substitution, the factors inside one node are not distributed. The algorithm used for Type 1 can still be applied.

We use the following definition: a node is a root of a subtree if it is of Type 1 and all the subtree is mapped on the same processor.

Initialize a pool **P** *with local root nodes*
Initialize a pool $\mathbf{P}_{\mathbf{subtree}}$ *to 0*
Initialize a vector RHSSOL to 0
NBFINF $\leftarrow$ *Number of processors holding at least one leaf*
Begin Main Loop
  if *the* **P** *and* $\mathbf{P}_{\mathbf{subtree}}$ *are both empty* **then**
    Perform a blocking receive
  else
    Try to receive a message
  end if
  if *a message has been received* **then**
    ! Treat the message
    if *message tag is FEUILLE* **then**
      ! message means that a processor has finished all its leaves
      NBFINF $\leftarrow$ *NBFINF – 1*
    else if *message tag is ContVec*
      Store the contribution vector
      Put the corresponding nodes in the pool
      ! in **P_subtree** *for roots of subtrees,* **P** *otherwise*
    else if *message tag is TERREUR* **then**
      Set appropriate error code and exit
    end if
  else
    if P *is empty* **then**
      if $\mathbf{P}_{\mathbf{subtree}}$ *is not empty* **then**
        Extract the root of a subtree from $\mathbf{P}_{\mathbf{subtree}}$
        Solve locally for the complete subtree
        Update RHSSOL
        if *I have finished all my leaves* **then**
          send a message with TAG=FEUILLE to other processors

```
        end if
      end if
    else
      Extract a node INODE from the pool P
      ! Treat the node INODE (Order LIELL, NPIV pivots)
      Load appropriate components of RHS into x(1 : NPIV)
      Load already computed solution (sent by father node)
      into y(1 : LIELL − NPIV)
      Deallocate already computed solution from CB area if no longer
      required
      x ← x − U_{1:NPIV,NPIV+1:LIELL}  y
      x ← U^{-1}_{1:NPIV,1:NPIV}  x
      for each son ISON of INODE
        if ISON is on the same processor then
          if ISON is the root of a subtree then
            Put ISON in the pool P_subtree
          else
            Put ISON in the pool P
          end if
          Store x in the contribution area if not already done
          (for a previous son on same processor)
        else
          Send x to the master of ISON if not already sent
          to that processor
        end if
      end for
    end if
  end if
  if NBFINF = 0 or an error occurred, Exit Main Loop
End of Main Loop
Finally perform a REDUCE operation in order to merge all
the local solutions RHSSOL into the global solution.
Return
```

5.3 Transposed system

Solution of the transposed system $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ can be of interest to the user. It is also used to get an estimate of the condition number of the matrix if required.

This is not implemented yet. The algorithms are slightly different from those described before, in particular the solve with triangular factor $\mathbf{L}^T$ is non trivial.

6 Other mechanisms in the code

6.1 Mapping of the tree to the processors

A mapping of the tree to the processors is performed statically as part of the analysis phase. The objective of the mapping is to provide a partition of the multifrontal tree to the processors, with a good balance of work on the processors. If memory is an issue, the mapping of higher levels in the tree only takes into account memory balance. This is the default and is controlled by ICNTL(15).

6.1.1 Description of the algorithm

The tree is processed from the bottom to the top, level by level (see Figure 10).

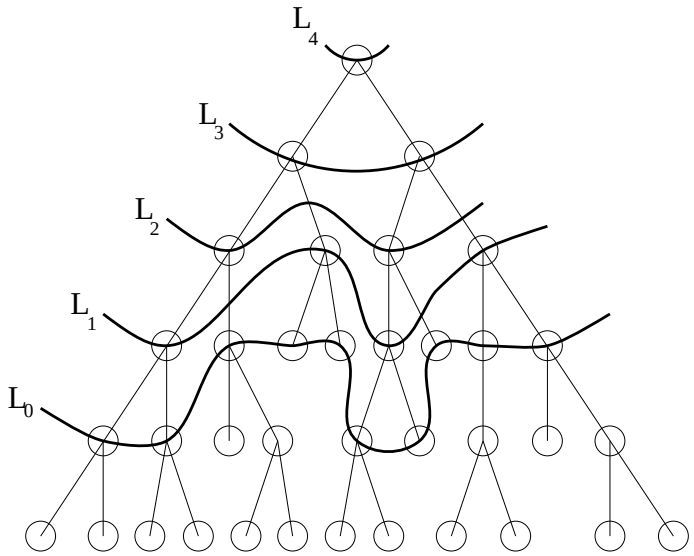

Figure 10: Decomposition of the elimination tree into levels

A level in the tree is defined to be a set of nodes, cutting the tree horizontally. L_0 is determined using the algorithm of Section 6.1.2, then for $i > 0$, a node belongs to L_{i+1} if all its son nodes belong to L_j, $j \leq i$. First, nodes of level L_0 (and associated subtrees) are mapped. Then for each level L_i, $i > 0$, each non-mapped node of L_i is mapped to the processor with smallest load, based on the number of flops. If memory is an issue, the mapping of higher levels in the tree only takes into account memory balance.

6.1.2 Construction and mapping of the initial level L_0

The construction of the initial level L_0 aims to find a good balance for the computations in the subtrees defined by the nodes of L_0 and is based on the work of

Geist and Ng [7]. We start from the set of roots of the tree, and apply the following algorithm :

Let $L_0 \leftarrow$ *Roots of the elimination tree*
Repeat
Find the node n *in* L_0 *with largest cost in the subtree*
Set $L_0 \leftarrow L_0 \backslash \{n\} \cup \{sons\ of\ n\}$ *(See Figure 11)*
Map the nodes of L_0 *on to the processors in a cyclic fashion (starting with the largest node)*
Estimate the load unbalance
Until *load unbalance* $<$ *threshold*

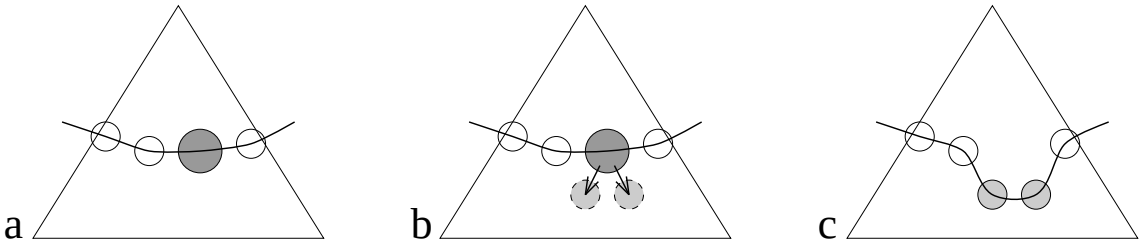

Figure 11: Construction of the first level L_0

6.2 Distribution of the original matrix

As required by the interface, the original matrix (arrays IRN, JCN, A) is given from the user to the package on the host processor. During factorization, MUMPS requires a reordered matrix in a special format, referred to as *arrowhead format*. In the arrowhead format, the reordered matrix is held in a compressed sparse data structure with row 1 of the upper triangle, preceding column 1 of the strictly lower triangle, preceding row 2 of the upper triangle and so on. This format facilitates assembly of the entries from the original matrix into the frontal matrices.

In a previous version of the code, the host reordered the initial matrix and the whole matrix in arrowhead format was broadcast to all processors.

Now, only the host holds the original matrix and it distributes the matrix in arrowhead form at the beginning of the factorization phase. Indeed, local sizes and arrays of pointers are computed during the analysis step, and the distribution of the numerical values from the host to the other processors is done at the beginning of the factorization. We use a blocking factor both on the host and on the processors to avoid too small messages; the host processes the entries in IRN, JCN, A and decides where to send them.

As we have 3 types of nodes, this is done as follows:

1. For an arrowhead that belongs to a node of Type 1 : the values are sent to the processor on which the node is mapped.

2. For an arrowhead for a node of Type 2 : the values from the rows and the columns are sent to the master of the node, but the columns are duplicated on all the processors. This avoids introducing more communications during the factorization phase for a Type 2 node.

3. For an arrowhead for a node of Type 3: based on the static grid allocated to the root node, the values are sent by the host to the appropriate processor, and are assembled on reception into the static root. This means that the root is allocated at the beginning of the factorization.

6.3 Mechanisms for assembly in a 2D grid

We describe here the mechanism used to send a rectangular contribution to the distributed root for further assembly by the receiving processor and use by ScaLAPACK. For simplicity, we suppose that a send is always performed but, in the actual code, some of the assemblies are performed locally. Also, the asynchronous mechanisms are still in effect and, in the case where a send does not succeed, a reception can be performed first.

The algorithm is detailed in Section 6.3.1. An illustrative example is given in Section 6.3.2.

6.3.1 Algorithm

On entry, we have

- RG2L_ROW, RG2L_COL : 2 arrays whose components give the position in the root matrix of each global variable. For a static variable, RG2L_ROW and RG2L_COL have the same value; for delayed pivots, it is possible that they are not the same. Entries in these arrays are set at the beginning of the factorization for static variables and after reception of messages with Root_2Son for delayed pivots.

- $CB_{1:NBROW,1:NBCOL}$: is a rectangular matrix from the contribution block of one of the son's of the root with NBROW rows and NBCOL columns, corresponding to the global variables $INDROW_{1:NBROW}$ and $INDCOL_{1:NBCOL}$ (see Figure14).

- NPROW, NPCOL are the number of processors in each direction in a two-dimensional grid of processors (there are NPROW*NPCOL processors on which the root matrix is distributed). This grid is defined in the analysis phase. It will normally depend on the number of processors available and the estimated size of the root. Currently for our runs on the SP2, if SLAVEF is the number of processors, we use a flat grid of NPROW = 1 row processors and

NPCOL = SLAVEF column processors, although these can easily be changed for other environments.

- The root matrix is partitioned into blocks of size MBLOCK by NBLOCK. These blocks are the unit that is sent to each processor according to the processor grid (see Figure 12). The best block size will depend on the computing platform. For the IBM SP2, we choose MBLOCK = NBLOCK = 70.

Note that MBLOCK, NBLOCK, NPROW, NPCOL are known by all processors.

The following integer arrays are used to facilitate assembly of the contribution blocks in the root matrix.

- $ROW\_PERM_{1:NBROW}$: Permutation for row indices from CB,
- $COL\_PERM_{1:NBCOL}$: Permutation for column indices from CB,
- $PTRROW_{0:NPROW}$: Pointers into ROW_PERM to indicate the rows from CB to be sent to each processor row,
- $PTRCOL_{0:NPCOL}$: Pointers into COL_PERM to indicate the columns from CB to be sent to each processor column.

The algorithm is then as follows:

Initialize $PTRROW_{0:NPROW} = 0$
Do *i = 1, NBROW ! Count number of blocks for each processor row :*
  $POS\_IN\_ROOT \leftarrow RG2L\_ROW(INDROW_i)$
  $irow \leftarrow (POS\_IN\_ROOT - 1)/MBLOCK \ modulo \ NPROW$
  $PTRROW_{irow} \leftarrow PTRROW_{irow} + 1$
End Do
$PTRROW_0 \leftarrow PTRROW_0 + 1$
Do *irow = 1, NPROW - 1 ! Build pointers to ends instead of counts*
  $PTRROW_{irow} \leftarrow PTRROW_{irow} + PTRROW_{irow-1}$
End Do
$PTRROW_{NPROW} \leftarrow PTRROW_{NPROW-1}$
Do *i = 1, NBROW ! Fill ROW_PERM with the processor rows for each variable*
  $POS\_IN\_ROOT \leftarrow RG2L\_ROW(INDROW_i)$
  $irow \leftarrow (POS\_IN\_ROOT - 1)/MBLOCK \ modulo \ NPROW$
  $PTRROW_{irow} \leftarrow PTRROW_{irow} - 1$
  $ROW\_PERM(PTRROW_{irow}) \leftarrow i$
End Do
! The same steps should be performed for the column indices

! Send the contributions to each processor
Do *irow = 0, NPROW - 1*
  $Nrow2Send \leftarrow PTRROW_{irow+1} - PTRROW_{irow}$
  Do *jcol = 0, NPCOL - 1*
    $Ncol2Send \leftarrow PTRCOL_{jcol+1} - PTRCOL_{jcol}$
    Call Send_Contrib($Nrow2send, Ncol2send, irow, jcol,$
    $ROW\_PERM(PTRROW_{irow}),$
    $COL\_PERM(PTRCOL_{jcol}),$ *CB,...*)
  End do
End do
Return

The previous algorithm uses the following sending routine:

Procedure Send_Contrib(*Nrow2send, Ncol2send, irow, jcol, SubsetRow, SubsetCol, CB, ...*)
Pack *the son node number, Nrow2Send, Ncol2Send into a new buffer*
Do *isub = 1, Nrow2send*
  $i \leftarrow SubsetRow_{isub}$
  $POS\_IN\_ROOT \leftarrow RG2L\_ROW(INDROW_i)$
  ! Find position for assembly in distributed root array
  $ILOC\_ROOT \leftarrow 1 + MBLOCK * \frac{POS\_IN\_ROOT-1}{MBLOCK*NPROW}$
    $+(POS\_IN\_ROOT - 1)$ *modulo* $MBLOCK$
  Pack $ILOC\_ROOT$
End do
Do *jsub = 1, Ncol2send*
  Do the same for the columns and pack $JLOC\_ROOT$
End do
Do *isub = 1, Nrow2send*
  $i \leftarrow SubsetRow_{isub}$
  Do *jsub = 1, Ncol2send*
    $j \leftarrow SubsetCol_{jsub}$
    Pack $CB_{i,j}$
  End do
End do
Send *the packed message to processor irow, jcol in the grid*
End procedure Send_Contrib

Note that we are packing the message with tag Root_Cont_Static as discussed in Section 4.5.1. `ILOC_ROOT` entries comprise the array POSROW in that message.

6.3.2 Example

We suppose that we have four processors available for the root node, organized as a grid of NPROW = 2 rows by NPCOL = 2 columns.

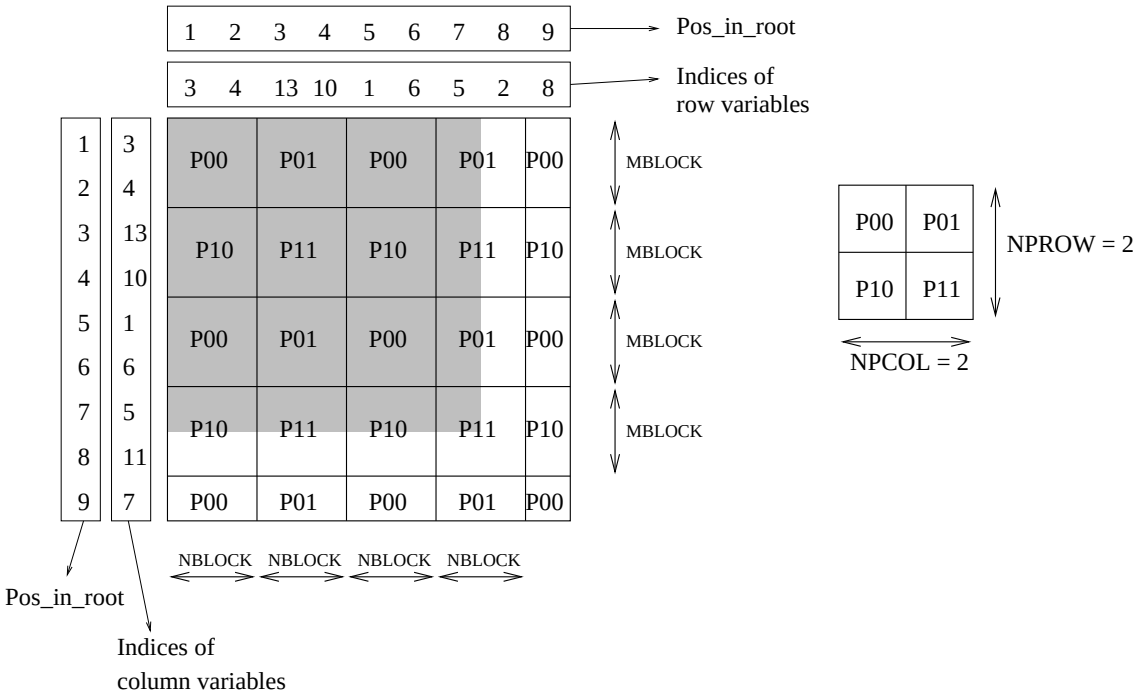

Figure 12: Example of distribution of the root node and associated ScaLAPACK grid

The static root size is 7 and has static global variables 3, 4, 13, 10, 1, 6, 5. In Figures 12 and 13, the static root is in grey, and has symmetric variable indices. The dynamic part, corresponding to delayed pivots, is in white.

According to their definitions, RG2L_ROW and RG2L_COL will contain the values in Table 3. That is, for each global variable, (RG2L_ROW, RG2L_COL) gives the position of the variable in the root matrix.

Variable	1	2	3	4	5	6	7	8	9	10	11	12	13
RG2L_ROW	5	-	1	2	7	6	9	-	-	4	8	-	3
RG2L_COL	5	8	1	2	7	6	-	9	-	4	-	-	3

Table 3: Contents of RG2L_ROW and RG2L_COL arrays. In practice, the indices of variables in the root matrix will usually be scattered over the range of possible values.

Now we suppose that a processor of the son needs to send the contribution defined in Figure 14 to the root.

The positions of the variables in the root matrix and their calculated location in the grid of processors are:

- RG2L_ROW(13) = 3 : Second row of the grid of processors,

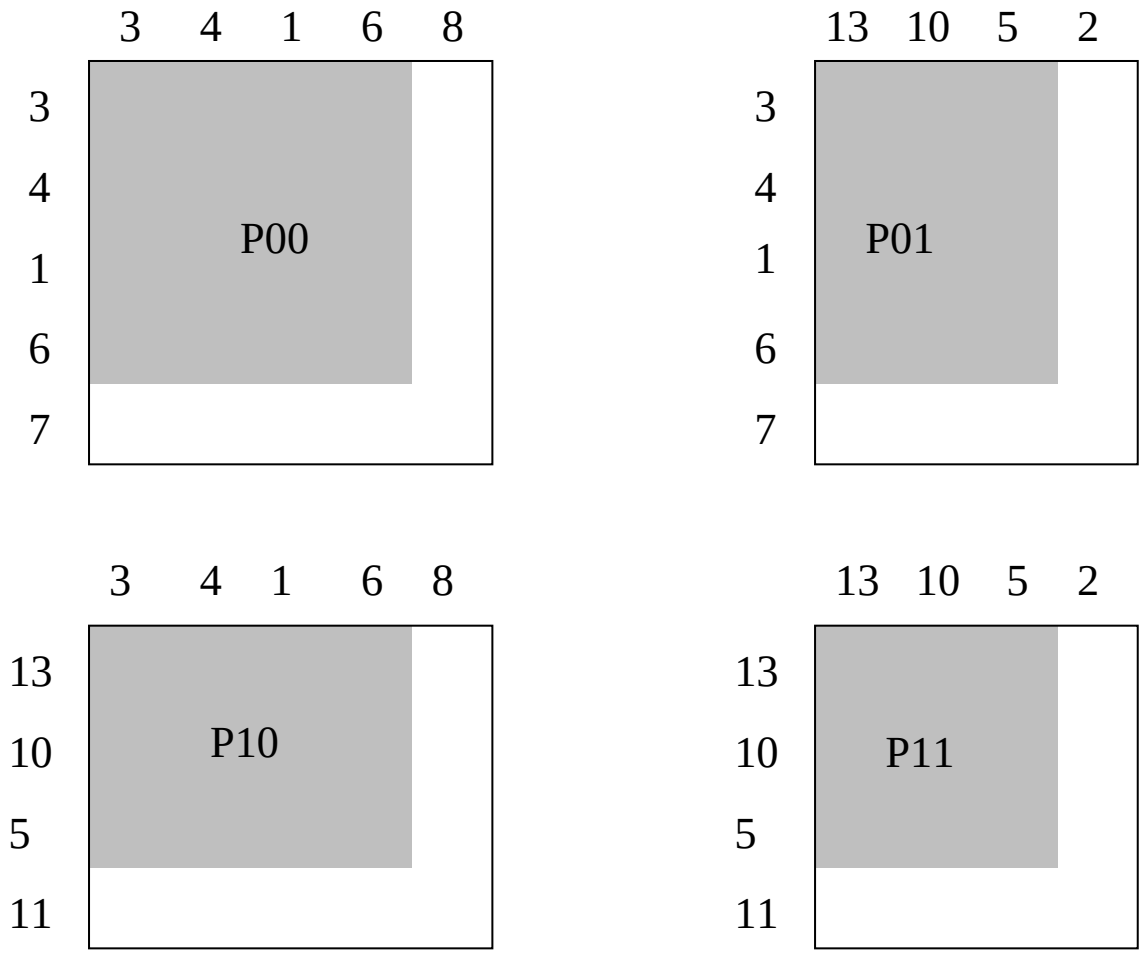

Figure 13: Structure of the arrays used by ScaLAPACK to hold the root node

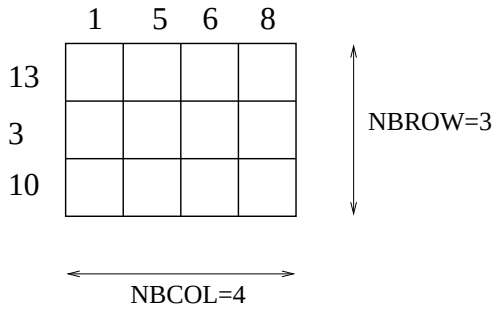

Figure 14: Example of a rectangular contribution block to be sent to the root node

- RG2L_ROW(3) = 1 : First row of the grid of processors,
- RG2L_ROW(10) = 4: Second row of the grid of processors.
- RG2L_COL(1) = 5 : First column of the grid of processors,
- RG2L_COL(5) = 7 : Second column of the grid of processors,
- RG2L_COL(6) = 6 : First column of the grid of processors,
- RG2L_COL(8) = 9 : First column of the grid of processors.

Then, the arrays ROW_PERM, COL_PERM will be:

Position	1	2	3
ROW_PERM	2	1	3

and

Position	1	2	3	4
COL_PERM	1	3	4	2

with the pointer arrays PTRROW = 1 2 4, and PTRCOL = 1 4 5 for ROW_PERM and COL_PERM, respectively.

Now for $(irow, jrow) \in \{(0,0)(0,1)(1,0)(1,1)\}$, this information allows us to identify that the block to be sent to $(irow, jrow)$ in the grid of processors for the root is

$$CB_{ROW\_PERM(PTRROW_{irow}:PTRROW_{irow+1}-1),\ COL\_PERM(PTRCOL_{icol}:PTRCOL_{icol+1}-1)}.$$

To facilitate the assembly at the root level, the positions in the root arrays of Figure 13 are computed before sending the message. This is done in the procedure Send_Contrib and yields the Row and Column indices shown in that table.

The actual message sent to (irow, jcol) is thus defined by Table 4.

6.4 Asynchronous messages, use of buffers

We use the same reception buffer throughout an instance of the package. The size of this reception buffer is computed by the analysis and is allocated on all processors at the beginning of the factorization.

Asynchronous messages have to be sent and received all the time. There are two possibilities in MPI to send a message and return from the send call even if reception has not been posted on the other side:

- buffered send, and

	jcol = 0	jcol = 1
irow = 0	NRow2Send = 1 NCol2Send = 3 Row indices = 1 Column indices = 3 4 5 $CB_{2,1}, CB_{2,3}$	NRow2Send = 1 NCol2Send = 1 Row indices = 1 Column indices = 3 $CB_{2,2}$
irow = 1	NRow2Send = 2 NCol2Send = 3 Row indices = 1 2 Column indices = 3 4 5 $CB_{1,1}, CB_{1,3}, CB_{1,4}, CB_{3,1}, CB_{3,3}, CB_{3,4}$	NRow2Send = 2 NCol2Send = 1 Row indices = 1 2 Column indices = 3 $CB_{1,2}, CB_{3,2}$

Table 4: Content of the messages sent to the processors of the root node

- immediate send.

A first version of our code used buffered send only, but there were several inconveniences:

- A buffer had to be attached to MPI with MPI_BUFFER_ATTACH. This could be a problem inside a library because of possible interference with the use of an attached buffer outside MUMPS.
- If the buffer is full, a call to MPI_BSEND causes an error. Even if a handler is available to process errors, it is stated in the MPI reference guide that "After an error is detected, the state of MPI is undefined. That is, using a user-defined handler, or MPI_ERRORS_RETURN, does not necessarily allow the user to continue to use MPI after an error is detected." In fact this is what happened when MPI_BSEND was used on an already full buffer: there was no way to continue.
- Also, we were using one buffer in which to pack the message, then a buffered send, which was an unnecessary copy.

Therefore, we implemented a Fortran 90 module to send asynchronous messages, based on immediate sends. There are controls on the buffer (allocation, deallocation, size available), routines to *try to send* contribution blocks, factorized blocks, ... (all kinds of messages required).

The buffer is allocated/deallocated by the routines PSL_BUF_ALLOC and PSL_BUF_DEALLOC. Each routine of the form PSL_BUF_TRY_SEND_XXXX, where XXXX represents the message to be sent, performs the following steps.

1. Compute an upper bound for the message size.
2. Find space in the buffer.

- Free messages that are safely received.
- See if the new message can be stored somewhere.

3. Pack the message in the buffer.
4. Call MPI_ISEND to start an asynchronous send.

The buffer itself is a structure defined by

```
TYPE PSL_BUFFER_TYPE
  INTEGER LBUF, LBUF_INT, HEAD, TAIL, ILASTMSG
  INTEGER, DIMENSION(:),POINTER :: CONTENT
END TYPE PSL_BUFFER_TYPE
```

where

- LBUF is the size of the buffer in number of bytes,
- LBUF_INT is the size of the buffer (in integer words),
- HEAD is a pointer to the first message in the buffer,
- TAIL is a pointer to the first free position in the buffer,
- ILASTMSG is a pointer to the last message in the buffer (required to set a pointer on a possible next message),
- CONTENT is the buffer itself.

A message in the buffer contains a header of size 2: a pointer to the next message, and the request (for example, req1 or req2) associated with the asynchronous MPI_ISEND call (see Figure 15).

The buffer is cyclic. Let B be the buffer.

If B%TAIL > B%HEAD, we first try to put the message between B%TAIL and B%LBUF_INT. If there is not enough space, then we try to put the message between position 1, and B%HEAD.

Now if B%TAIL < B%HEAD, we try to put the message between B%TAIL and B%HEAD – 1.

If there is not enough space to put the message in the buffer, the error code –1 is returned and the procedure requesting the send will try again later. In fact, to avoid deadlock, the procedure must try to receive messages first and try to send again afterwards, until the buffer is sufficiently empty.

Before putting a message in the buffer, we try to free space in the buffer corresponding to safely received messages, using the function MPI_TEST. MPI_TEST needs a request number as argument and returns a logical indicating if the message associated with the request has been safely sent and can be freed.

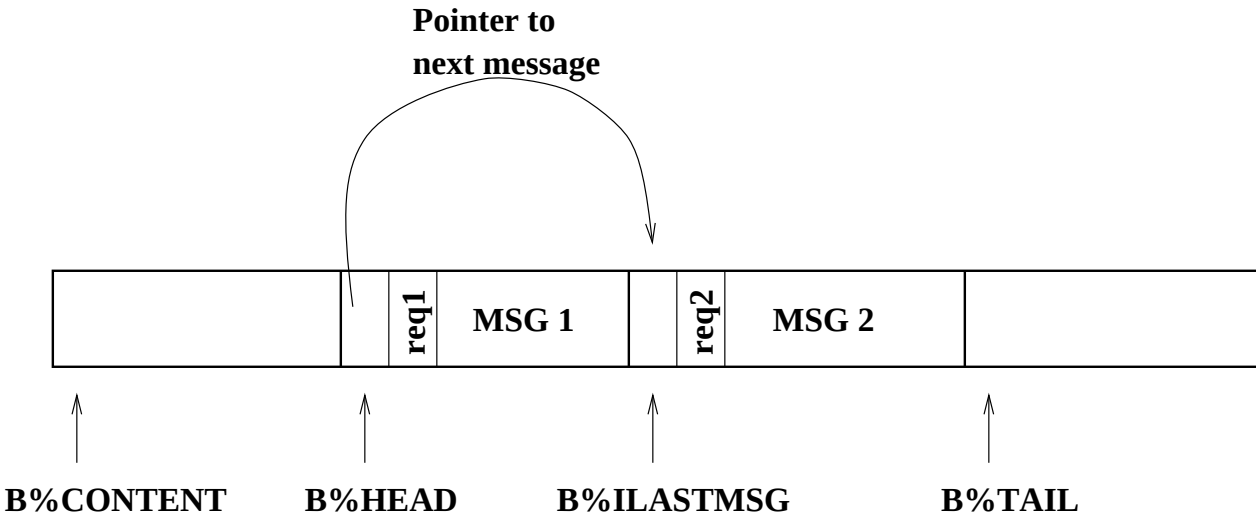

Figure 15: Structure of buffer B used for asynchronous sends

We first try to deallocate the message in position B%HEAD (request req1 in our example), then the following messages, using the pointer in the header of each message to go from one message to the next.

This mechanism provides an equivalent of MPI_BSEND with the advantage that messages are directly packed in the buffer, and problems occurring when the buffer is full are overcome.

6.5 Treatment of errors

An error may occur during the execution of MUMPS. Since the code is parallel, the processing of errors is a little complicated. If an error occurs on one processor, it should not return without informing all the other processors.

Thus, there is 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 tell the others to return.

In some cases, an error may occur on one processor in a routine where there is no asynchronous communication. In that case, the error is propagated after the subroutine call via the subroutine
`PSL_MUMPS_PROPINFO( ICNTL, INFO, COMM, ID )`,
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 `PSL_MUMPS_PROPINFO`, other processors will have INFO(1) = –1 (which means: *an error occurred on another processor*) and INFO(2)=s (*and the*

processor on which the error occurred is s in the communicator `mumps_par%COMM`).

The possible error codes after a call to `PSL_MUMPS` are the following.

- mumps_par%INFO(1)=–1: An error occurred on processor INFO(2).
- mumps_par%INFO(1)=–2: NE is out of range. INFO(2)=NE.
- mumps_par%INFO(1) = –3: JOB has a wrong value or analysis was not performed before factorization. This error also occurs if JOB does not contain the same value on all processes on entry to `PSL_MUMPS`.
- mumps_par%INFO(1) = –4: Error in permutation array. This error occurs on the host only.
- mumps_par%INFO(1) = –5: Not enough real space (MAXS) to preprocess the matrix (scaling or arrowhead calculation).
- mumps_par%INFO(1) = –6: Matrix is singular in structure.
- mumps_par%INFO(1) = –7: MAXIS too small for analysis... should never happen since dynamic allocation of enough space is now used for analysis.
- mumps_par%INFO(1) = –8: MAXIS too small for factorization... can still happen if too much pivoting occurs compared to what was predicted. The user should then provide a larger value for MAXIS.
- mumps_par%INFO(1) = –9: MAXS too small for factorization.
- mumps_par%INFO(1) = –10: Numerically singular matrix.
- mumps_par%INFO(1) = –11: MAXS too small for solution.
- mumps_par%INFO(1) = –12: MAXS too small for iterative refinement.
- mumps_par%INFO(1) = –13: Error in an ALLOCATE statement; INFO(2) contains the size that was asked for.
- mumps_par%INFO(1) = –14: MAXIS too small for solution
- mumps_par%INFO(1) = –15: MAXIS too small for iterative refinement.
- mumps_par%INFO(1) = –16: N is out of range. INFO(2)=N
- mumps_par%INFO(1) = –17: Send buffer too small.
- mumps_par%INFO(1) = –18: MAXIS too small to process root node.
- mumps_par%INFO(1) = –19: MAXS too small to process root node.
- mumps_par%INFO(1) = –20: Reception buffer too small. INFO(2) holds the minimum size of reception buffer required (in bytes).

References

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

[2] 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.

[3] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 9:302–325, 1983.

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

[5] A. Supalov (editor). PARASOL Interface Specification. Version 2.1. January 9, 1998.

[6] V. Espirat. Développement d'une approche multifrontale pour machines à mémoire distribuée et réseau hétérogène de stations de travail. Rapport de stage de 3ieme année, ENSEEIHT-IRIT, Toulouse, France, 1996.

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

[8] HSL. *Harwell Subroutine Library. A Catalogue of Subroutines (Release 12).* AEA Technology, Harwell Laboratory, Oxfordshire, England, 1996. For information concerning HSL contact: Dr Scott Roberts, AEA Technology, 552 Harwell, Didcot, Oxon OX11 0RA, England (tel: +44-1235-434988, fax: +44-1235-434136, email: Scott.Roberts@aeat.co.uk).

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

[10] PARASOL. Deliverable D2.1a: distributed multifrontal code. January 25, 1997.