Analysis of the Out-of-Core Solution Phase of a Parallel Multifrontal Approach*

P. Amestoy † I.S. Duff ‡ A. Guermouche § Tz. Slavova ¶

April 25, 2007

Abstract

We consider the parallel solution of sparse linear systems of equations in a limited memory environment. A preliminary out-of core version of a sparse multifrontal code called MUMPS (MUltifrontal Massively Parallel Solver) has been developed as part of a collaboration with members of the INRIA project GRAAL.

In this context, we assume that the factors have been written on the hard disk during the factorization phase, and we discuss the design of an efficient solution phase. Two different approaches are presented to read data from the disk, with a discussion on the advantages and the drawbacks of each one.

Our work differs and extends the work of [10] and [11] because firstly we consider a parallel out-of-core context, and secondly we focus on the performance of the solve phase.

1 Introduction

We are interested in solving large sparse linear systems $Ax = b$ with direct methods [6], in a parallel limited memory environment. Indeed one main limitation in the use of sparse direct methods comes from the need to store the matrix factors that has often many more entries (10 to 100 times) than the original matrix.

In this context, an out-of-core **(OOC)** multifrontal [7, 8] approach is considered. Here the complete matrix of factors is written to disk during the factorization phase, as a sequence of blocks (so called **factor blocks**). Overlapping communications and I/O with computations during the factorization phase is an important issue (see [1]), but is not the scope of this work . During the subsequent phase (forward and backward solutions, the so called **solve phase**) we have to load the factor blocks from the local disks of the computer to the main memory. In this context, the cost of the solution phase can become the dominant phase of the complete solution process. When the solution phase has to be performed for many right-hand sides (simultaneously or not) then it is even more critical.

In this paper, we focus on the performance of the solution phase. We first recall in Section 2 the main features of our target solver (MUMPS [2],[3],[4]) and fully describe the in-core distributed memory solution phase (never done in previous publications related to MUMPS). Then we explain how it has been adapted to the out-of-core context

*This work is partially supported by ANR project SOLSTICE, ANR-06-CIS6-010.

†ENSEEIHT-IRIT (Toulouse), Patrick.Amestoy@enseeiht.fr

‡CERFACS (Toulouse), Iain.Duff@cerfacs.fr

§LaBRI, Univ. Bordeaux 1 / INRIA Futurs, Abdou.Guermouche@labri.fr

¶CERFACS (Toulouse), Mila.Slavova@cerfacs.fr

(Section 3). We describe in Section 4 our testing environment - the hardware we use and our choice of matrices for the tests. We show in Section 5 the limitations of a simple demand driven approach, that we call a **System based approach**, based on automatic system I/O caching mechanisms. In Section 6 we show how user buffers can be introduced to improve the behaviour of the solve phase and then describe an approach where the memory used is completely controlled, which we call the **Direct IO based approach**. We show that a naive implementation of the Direct IO based approach is not suitable for parallel implementation and introduce a new scheduling that constrains the ordering of the tasks. We first prove that the new algorithm is correct. We then illustrate the gain in performance obtained on a large problem of order 8 million in a parallel environment.

2 Main In-core Features

Direct solvers try to preserve the zero pattern and to exploit the independence of some computations in parallel environments. So called three-phase approaches have become very popular:

- The **analysis** phase considers only the pattern of the matrix and builds the necessary data structures for numerical computations.
- The **factorization** phase tries to follow the decision of the analysis and builds the sparse factors (LU for unsymmetric case, or LDL^T for the symmetric case).
- The **solution** phase performs a forward and backward substitution and, optionally, performs iterative refinement to improve the solution.

Multifrontal methods use an *elimination tree* [9] to represent the dependencies of the computations. Each node of this tree is associated with a frontal matrix that is assembled (summed) by contributions from the children and the original matrix. In practice, nodes of the elimination tree are amalgamated so that more that one variable can be eliminated at each node of the tree. The resulting amalgamated tree is referred to as the *assembly tree*.

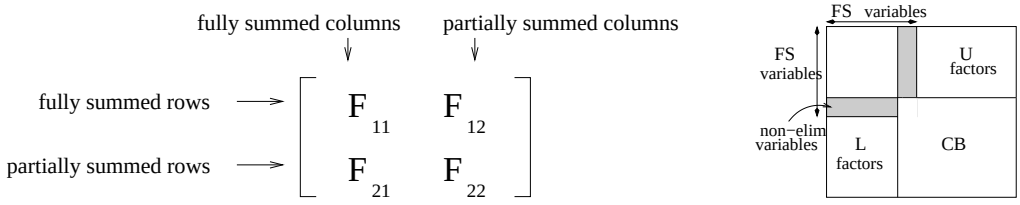


Figure 1: General structure of the frontal matrix

The work associated with an individual node of the assembly tree corresponds to the factorization of a so called *frontal matrix*. Frontal matrices are always considered as dense matrices (see Figure 1). Once all *eliminations* have been performed, the Schur complement matrix $F_{22} - F_{21}F_{11}^{-1}F_{12}$ is computed. It is used to update later rows and

columns of the overall matrix which are associated with the parent nodes. We call this Schur complement matrix the *contribution block* (*CB*) of the node, see Figure 1.

If some variables are not eliminated because of numerical issues, they are included in the contribution block and their elimination is postponed to the parent node or later. These *non-eliminated variables* (delayed pivots) increase the fill-in the factors, the number of the operations and the factorization time, but allow the factorization to compute accurate factors.

2.1 Parallelism during the factorization phase

The MUMPS solver [2, 3, 4] provides three different types of parallelism for both factorization and solve phases. The reason for the three types is to balance the total work and the memory on each processor.

We use the *assembly tree*, representing the order in which the matrix will be factorized, to distribute the nodes over the processors. Depending on the size of the node and on which level of the assembly tree the node is situated, we have :

Type1 node: **sequential processing of a node** — essentially for the low levels of the tree (near the leaves), where the tree parallelism is sufficient.

Type2 node: **irregular 1D decomposition of the node** — for the intermediate levels when the node is large enough: the contribution blocks are partitioned and each partition assigned to a different processor. The master is in charge of factorizing the block of fully summed variables and of deciding how many slave processes will be used to process this node.

Type3 node: **block cyclic 2D distribution of the frontal matrix** — reserved only for the root node, if it is large enough. In this case, ScaLAPACK [5] is used on the node.

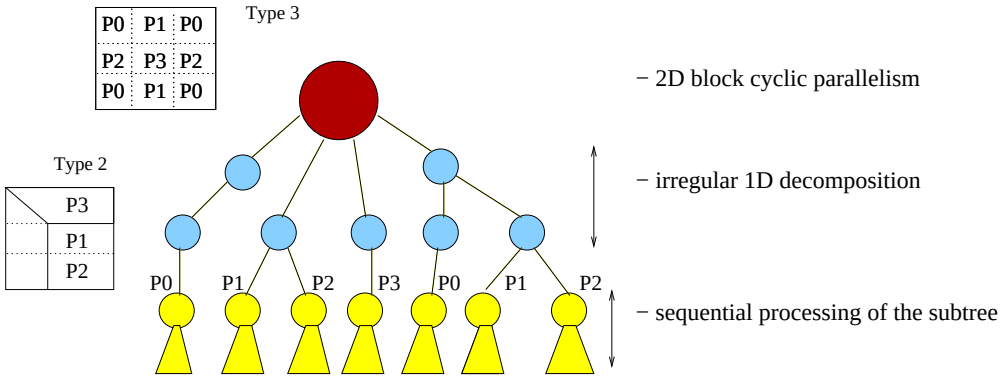


Figure 2: Different types of parallelism in MUMPS.

Note that, in a sequential environment, we choose to process the nodes of the elimination tree, using a post-ordering. In a parallel environment, the tree nodes are distributed onto processors and only a topological ordering is followed.

In Figure 3 we show the distribution of the L factors of a frontal matrix depending on the type of the node. $Elim$ corresponds to a block of eliminated fully summed variables and *NE* to the block of non-eliminated fully summed variables of the frontal matrix. In Figure 3.a the whole frontal matrix is mapped to one processor. In Figure 3.b the fully summed variables (*FS*) are on the master processor. The contributions blocks (*CB* in Figure 1) can be on several processors.

a) Type 1 Node *b) Type 2 Node*

Figure 3: Example of mapping of the L factors of a frontal matrix. *FS*, *elim*, *NE*, *NCB* hold respectively the fully summed, eliminated, non-eliminated and contribution variables

2.2 In-core solution phase

The solution phase is divided in two steps: forward (**Fwd**) and backward (**Bwd**) substitution. We can present the two steps in several ways. Mathematically, they replace the original equation (using the whole matrix), by a system of two equations (using only triangular matrices). So, for example, if the matrix is symmetric, we will replace:

$$Ax = b \quad by \quad \begin{array}{ll} LDy = b & (forward\ \ step) \\ L^T x = y & (backward\ \ step) \end{array}$$

where L is a lower triangular matrix, and D is a diagonal matrix (or a matrix with 2x2 blocks on the diagonal in the case of numerical pivoting for indefinite systems).

We can also present the two steps graphically using the elimination tree. The forward step is a bottom-top traversal of the tree (post-ordering for the sequential case and topological ordering for the parallel case). The backward step is in the reverse order. We first describe how a pool of tasks, on which the forward and the backward algorithms will be based, is used to schedule both of the steps.

2.2.1 Pool of tasks

To handle the task dependency, we use a distributed pool of tasks. It contains a list of all tasks ready to be executed. The pool is used to schedule work in both the sequential and the parallel cases.

Figure 4 illustrates the pool of tasks. At the beginning of the solve part, using the assembly tree, we first add to the pool all tasks ready to be processed. For the forward step, this corresponds to the leaf nodes. As in the factorization, a node will be placed at the end of the pool as soon as all of its children are processed. A node can be split over more than one processor (Type2 nodes) and, in this case, only the master tasks are added to the pool

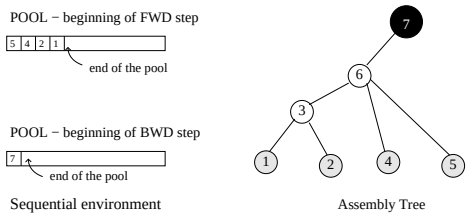


Figure 4: The POOL of tasks at the beginning of each step. Leaf nodes are coloured grey. Root node is coloured black.

for scheduling. The slave tasks are processed on the fly. At the beginning of the backward step the pool is initialized only with the root nodes. At the end of a node process, we add to the end of the pool all of its children. The general algorithm for extracting nodes from the pool is described in Algorithm 1. It is very similar for the forward and for the backward step.

Algorithm 1 : General Algorithm for extracting a node from the POOL

```
Step = Fwd or Bwd
if ( Fwd ) then
   Initialise POOL with the leaf nodes mapped on Myid
else if (Bwd) then
   Initialise POOL with root nodes mapped on Myid
end if
while (Not finished) do
   if (POOL is not empty) then
      Try to receive a message and then Process_Message(message)      [See Algorithms 2 and 3]
   else
      Wait to receive a message and then Process_Message(message)      [See Algorithms 2 and 3]
   end if
   if (POOL is not empty) then
      Extract node Inode from the end of the POOL
      if (Fwd) then
         Fwd_Process_node(Inode)      [See Algorithm 2]
      else if (Bwd) then
         Bwd_Process_node(Inode) [See Algorithm 3]
      end if
   end if
end while
```

Note that priority is given to the reception of messages - to a blocking or non-blocking receive. We look at the pool for work only when no messages need to be processed. The algorithm for the forward case finishes when all root nodes have been treated. The backward algorithm finishes when all leaf nodes have been processed.

Without loss of generality we will assume in the remainder of this paper that we have only one right-hand side and thus one solution to compute although the extension to multiple right-hand sides is straightforward.

For the sake of completeness references to BLAS (Basic Linear Algebra Subroutines) kernels (`GEMM/V` and `TRSM/V`) have been added to the description of the algorithms.

2.2.2 Parallel forward and backward substitution

We start by presenting the general algorithm for the forward substitution ($Ly = b$). We then show details of the algorithm used to process a node and add comments on how messages are processed.

Algorithm 2 : General algorithm for the forward step

Myid - process number; $Inode$ - the current node mapped on process *Myid*;
$Nb\_children$ - the number of children of $Inode$ and *Pfather* - the process on which the master of the father of $Inode$ is mapped. Nb_slaves - the number of slaves; and $Nb\_ContVect$ - the number of contribution vectors, initialized to $Nb\_children + \sum(Nb\_slaves$ involved in each child). Wb - a local working array, initialized to 0 and designed to accumulate modifications of the right-hand side b; $temp\_Wb$ - a small working array of maximum size the size of the largest frontal matrix, initialized to 0.

Fwd_Process_node($Inode$**)** {I am the master of node $Inode$}

Update b with entries of Wb corresponding to *elim* variables
`Use factors` to compute solution corresponding to *elim* variables (`TRSM/V`)
if ($Inode$ is of Type2) **then**
  for i = 1, Nb_slaves
    send to slave i the computed solution and entries of Wb corresponding to rows mapped on slave i (message `MASTER2SLAVE`)
  end for
  Update entries of Wb related to *NE* variables (`GEMM/V`) and send them to *Pfather*
  Reset to zero all entries of Wb sent to slaves and to *Pfather*
else if ($Inode$ is of Type1) **then**
  Update Wb for *NCB+NE* rows (`GEMM/V`)
  if (*Myid* $\neq$ *Pfather*) **then**
    Send Wb to *Pfather* (message `ContVec`)
    Reset the sent entries of Wb to zero
  else
    Decrement $Nb\_ContVect$
    if ($Nb\_ContVect = 0$) **then** {last modification}
      Add $father(Inode)$ to the end of the pool (since $father(Inode)$ is ready to be processed)
    end if
  end if
else {Parallel Root Node}
  ScaLAPACK will be used to perform both forward and backward steps on all processes
end if

Process_Message(Message) {I am updating $Inode$}

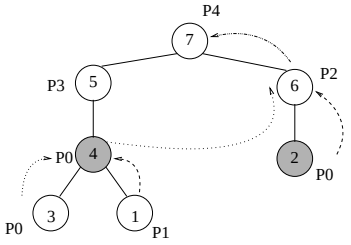
if (Message = `ContVect`) **then**
  Update Wb with contribution received (scatter and add)
  Decrement $Nb\_ContVect(Inode)$
  if ($Nb\_ContVect(Inode) = 0$) Add $Inode$ to the end of the pool
else if (Message = `MASTER2SLAVE`) **then**
  Initialize $temp\_Wb$ with the part of Wb just received
  `Use factors` and the solution sent by the master to update $temp\_Wb$ (`GEMM/V`)
  if (*Myid*=*Pfather*) **then**
    Update Wb with $temp\_Wb$ (scatter and add $temp\_Wb$ in Wb)
    Decrement $Nb\_ContVect(father(Inode))$
    if ($Nb\_ContVect(father(Inode) = 0$) Add $father(Inode)$ to the end of the pool
  else
    Send $temp\_Wb$ to *Pfather* (message `ContVec`)
  end if
end if

Note that in our algorithm (and in practice) the same working space can be used to store both y and b. We will keep two separate vectors in our algorithm only for the sake of simplicity. The general algorithm for the forward substitution is described in Algorithm 2.

To better understand the distributed memory version of our algorithms, we first introduce a few properties related to the use of the elimination tree. For properties 1 and 2 please note that the terms eliminated and non-eliminated variables were described in Figure 3.

Property 1 *All contributions to eliminated variables of a node, say $Inode$, come only from processes involved in the $children\ of\ Inode$ (both master or slave processes).*
Proof: This property is clearly preserved by the algorithm, since in our algorithm only processes involved in the children nodes send contributions to the master of the father - message `ContVec` or direct update of Wb either during $Fwd\_Process\_Node$ for type 1 nodes or at the reception of message `MASTER2SLAVE` for type 2 nodes. Furthermore contributions to the eliminated variables of a node can only come from nodes involved in the sub-tree rooted at that node (main property of the elimination tree). This proves our property. □

Property 2 *All contributions of descendants of a node $Inode$, to non fully summed and non-eliminated variables of $Inode$, are not always sent to processes in charge of $Inode$.*



Dotted arrow between nodes 4 and 6 indicates that part of the contributions from the sub-tree rooted at node 4 to node 7 are in fact sent by process P0 when sending contributions from node 2 to node 6.

Figure 5: Example used to prove Property 2: part of the contributions of node 1 are not sent to process P3, in charge of node 5.

Proof: All nodes in Figure 5 are Type 1 nodes. Node 1 (mapped on P1) sends to node 4 (mapped on P0) updates to Wb (corresponding to entries of Wb on P1) and resets those entries to zero. Node 2 (mapped on P0) updates Wb and sends its updates to P2 (corresponding to entries of Wb on P0) and resets those entries to zero. At this point, part of the contributions of the sub-tree rooted at node 5 will circulate through P2 on node 6. This is the case if both node 1 and node 2 have a common row in the frontal matrix of node 7. This update to Wb will then be sent to P4 by P2 during the processing of node 6.

As a consequence during the processing of node 4, process P0 will not send to its father (node 5) all contributions from node 1 to node 7. Instead Property 1 says that the common row updated by node 2 and node 4 could not be eliminated at node 5, but at node 7.

Note that, Property 1 is one of the main properties of the elimination tree, exploited by the multifrontal approach and preserved, on each process, by the algorithm for the eliminated variables. However, contrary to what is exploited during multifrontal factorization, this elimination tree property is no longer respected on each process for non-eliminated variables (Property 2). Property 2 also explains the importance of resetting of Wb to zero in our algorithm. □

Property 3 *At any time a computed contribution is stored in the Wb array of only one process.*

Proof: We recall that Wb is designed to sum contribution vectors. Wb is first initialized to zero on each process at the beginning of the forward step. It corresponds to updates to the right-hand side b due to solution terms already computed. Each time part of Wb is sent to a process (message `ContVec` or `MASTER2SLAVE`) then the corresponding entries are reset to zero in the procedure **Fwd_Process_node**.

Let us now check, that updates to Wb are never lost. First, during the function $Process\_Message$(`MASTER2SLAVE`), each slave updates in a local array $temp\_Wb$, contributions sent by its master. $Temp\_Wb$ is then either used to update Wb locally, if the $process\_id$ of the slave is equal to *Pfather*, or is forwarded (message `ContVec`) to process *Pfather* without updating Wb locally. □

Property 4 *When starting to process a node (first line of Algorithm 2, procedure* **Fwd_Process_node***($Inode$)), b holds all contributions needed to compute the solution corresponding to the eliminated variables on the node.*

Proof: Results from Property 1 and 2.

Property 4 recursively proves that Algorithm 2 computes the correct solution. □

The general algorithm for the backward substitution is described in Algorithm 3. The algorithm performs backward elimination operations using the factors U or L^T. To describe the algorithms, we consider the case of $L^T x = b$. As for the forward step, priority is given to message reception. If no message is received, a node from the pool is extracted.

In the backward step there are three types of messages: `Bwd_MASTER2SLAVE`, similar to the forward case, `UpdateRHS` which corresponds to `ContVec` in the forward step, and a new type of message: `Node`, used to activate the children nodes.

Algorithm 3 : General algorithm for the backward step

Myid - process number; $Inode$ - current node mapped on *Myid*; Nb_slaves and $Nb\_ContVect$ are respectively the number of slaves and the number of contribution vectors (initialized to 0) of $Inode$.

Bwd_Process_Node($Inode$**)**

if ($Inode$ is of Type2) **then**
  Master distributes already computed solution between slaves (message `Bwd_MASTER2SLAVE`).
else if ($Inode$ is of Type1) **then**
  `Use factors`
    associated with *NE+NCB* columns of L^T to update b (`GEMM/V`)
    associated with *elim* variables to compute solution (`TRSM/V`)
  for each child of $Inode$ **do**
    if (Child mapped on *Myid*) **then**
      Add child node to the end of the pool
    else
      Send the solution corresponding to all column indices of the frontal matrix of $Inode$ to
        all processes on which at least one master of a child node is mapped (message `NODE`)
    end if
  end for
end if

Process_Message(Message)

if (Message = `NODE`) **then**
  Update known solution
  Add $Inode$ and all the brothers of $Inode$ mapped on this process to the pool.
else if (Message = `Bwd_MASTER2SLAVE`) **then**
  `Use factors` mapped on this slave process together with the received solution to compute a contribution to b (`GEMM/V`)
  Send the contribution to b to the master (message `UpdateRHS`)
else if (Message = `UpdateRHS`) **then**
  Update b with the contribution received and increment $Nb\_ContVect$
  if ($Nb\_ContVect = Nb\_slaves$) **then** {last update}
    `Use factors`
      associated with *NE* columns of L^T to update b (`GEMM/V`)
      associated with *elim* variables to compute the solution (`TRSM/V`)
    for each child of $Inode$ **do**
      if (Child mapped on *Myid*) **then**
        Add child node to the end of the pool
      else
        Send the solution corresponding to all column indices of the frontal matrix of $Inode$ to
          all processes on which at least one master of a child node is mapped (message `NODE`)
      end if
    end for
  end if
end if

3 Out-of-Core (OOC) Main Features

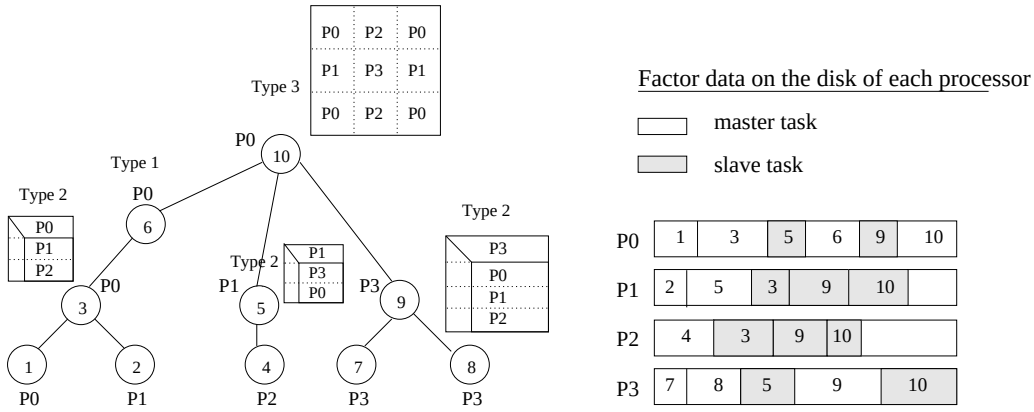
The out-of-core implementation of our algorithms is very critical for large matrices when we may have problems with a limited memory environment. Our objective is to reach good performance with respect to both run-time and memory in both sequential *and* in parallel cases. The OOC run time is strongly related to the hard disk time access. The latency, the number of disk accesses, and the regularity of the reading pattern are issues that will have to be taken into consideration.

In this section, we describe the main OOC features of our algorithms.

3.1 OOC factorization phase

During the OOC execution, the computed factors are stored on the hard disk and are written in the order in which they have been computed. Results obtained by [1] show that this can be obtained with limited overhead with respect to the in-core factorization.

In a sequential environment factors are written on the hard disk following a post-ordering traversal of the tree. For the parallel runs only a topological ordering, with unpredictable dynamic interleaving of slave and master tasks is followed (see Figure 6).



a) elimination tree mapped on 4 processors *b) Hard disk of each processor*

Figure 6: Example of interleaving of master and slave tasks during the factorization and influence on the disk usage on each processor.

Although one could clearly take advantage of keeping part of the factors in-core at the end of the factorization, for the sake of clarity, we will consider in the following that all factors have been written to the disk at the end of the factorization phase.

3.2 OOC Solution phase

We use the factorization write sequence in order or in reverse order, to prefetch factor blocks to the user buffers during the forward and the backward steps respectively. Looking at the hard disk storage area, these two steps can be represented as directions for reading data. The forward step needs factors from the disk in a left-right direction. That is why, for the forward step, we may want to prefetch data in the natural direction (the order in which data has been written) (see Figure 7). The backward step needs factors in the

reverse order: right-left direction on the disk. Here, the inverse of the natural reading direction is used, so that, one could expect the performance of the backward step to be worse than the forward step.

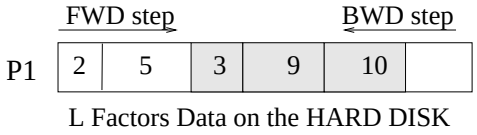


Figure 7: Reading direction on the disk in the solution step.

For this OOC method we use almost the same algorithms as for the in-core case. The only modification (see Algorithm 4) for the OOC execution is to load data from disk for each occurrence of the sequence '`use factors`' in the Algorithms 2 and 3.

Algorithm 4 : Modification of Fwd and Bwd algorithm for OOC execution

`Use factors` of $Inode$ ...	$\Longrightarrow$	**if** (OOC run) **then**
		Load data from disk($Inode$)
		end if
		`Use factors` ...

4 Testing environment

All our runs have been performed on the multiprocessor Cray XD1 located at CERFACS (58 nodes with 2 processors per node; and 4 GB per node). Each node is equipped with an `IDE` disk managed by the `reiserfs` file system of maximum bandwidth for a read operation close to 16 MB/sec with one MPI process per node.

Two matrices have been used for the experiments:

- The matrix Grid 300-100-10 corresponds to an 11-point discretization of the Laplacian operator on a three-dimensional grid problem of reasonably large size : $300 \times 100 \times 10$ (factor size: 748 MB, order : 300 000)
- Qimonda07 from Qimonda AG company is a large real symmetric matrix from circuit simulation of order 8,613,291 with factors of size 2534 MB and a total working space for incore factorization of 50084 MB.

We recall that during factorization **all** factors are written to the local disks. We have no factors kept in memory at the beginning of the solve part *and* between the forward and backward steps. So we have no intended reuse of data, which will help to better understand the behaviour of each step.

With these assumptions, we will thus have to load all of the factors during the solution phase. Furthermore, Qimonda07 is a large and very sparse matrix with more than 3 million nodes in the assembly tree. IO access might occur for each node of the elimination tree and thus it is an interesting matrix to illustrate the behaviour of our algorithms.

Two possibilities for accessing data on disk will be considered. In the first approach, we rely on system buffers (or page caching) to access the disk, referred to as the `System based method`. A second approach consists in a direct access to the disk and will be referred to as a `Direct IO based method`.

5 System based demand driven approach

A simple way to implement the OOC solution phase is to make a demand driven approach. We do not use any explicit prefetching. We let the operating system handle intermediate caches when loading data.

To illustrate the potential and the limitations of a demand driven approach we compare in Table 1 its behaviour on our two test matrices. We analyse the situation when the matrix fits in the main memory (the matrix Grid 300-100-10); and when the memory is critical (the larger matrix Qimonda07).

5.1 Performance

Strategy	Factor Time (sec)	Solve		
		Fwd (sec)	Bwd (sec)	Disk access (MB/s)
Grid 300-100-10				
in-core	34.91	0.39	0.37	-
OOC	34.90	1.26	1.17	616

Strategy	Factor Time (sec)	Solve		
		Fwd (sec)	Bwd (sec)	Disk access (MB/s)
Qimonda07				
in-core$^{(**)}$	40.4	0.9	0.9	-
OOC	88.8	161.6	221.6	13

Table 1: a) Demand driven approach. b) $^{(**)}$ In-core time obtained on 8 processors.

Let us first focus on the grid matrix with factors of size 748 MB. We compare the in-core and OOC time for the solution phase. The time for the factorization phase is included only for information. We see that the extra time required in both forward and backward phases for the OOC execution corresponds to copying the factor data at a rate of 616 MB/s so that the copy is not from the disk (bandwidth of 16 MB/s) but from the system buffers.

Indeed the system based demand driven approach unpredictably affects the behaviour of our processes in an intrusive way. Even if the factors were written to the disk during the factorization, a significant part of them still remains in the system caches, so that the cost of accessing them during the solution phase is the cost of a main-memory access.

On the larger problem (Qimonda07), the size of the total workspace for sequential incore factorization (5084 MB) is bigger than the available memory (4 GB). In out-of-core, a working space of size 2 GB is needed during factorization so that the system cannot keep all the factors in the system caches at the end of the factorization phase. Some factor blocks then must be loaded from the disk. In this case, increasing the number of disk accesses will decrease the time performance. Note that in Table 1-b) the backward step takes 37% more time for the same number of operations than the forward step. We compare the time for in-core, obtained on 8 processors, with the OOC sequential run. This time the disk access is more realistic - 13 MB/s. Note that, the peak speed of a memory read from the disk is 16 MB/s, so that the minimum time to only load all the factor blocks is 158 seconds.

We thus see that, even in a simple context (sequential reading of data from the disk in an identical order to that used when writing) the performance is far from the minimum. The reason is that even in this relatively simple case the system I/O mechanism is in conflict with the automatic system swapping mechanisms.

5.2 Limitations of the demand driven algorithm

As shown in Table 1-b), the System based method is inefficient on large matrices, when the volume of data on the disk is larger than the memory size. In this case, we observe the so called swapping effect: the system decides when and which data to swap to the disk. The decision is done by the system and is often based on a variant of a least recently used strategy. Note that, the system has no knowledge of the data access pattern of the algorithm.

Furthermore, even more critical is the fact that the system cache grows with each disk access (reading or writing data). It is impossible to control the memory effectively used: its size or effective bandwidth to access the disk. So we do not know how much real memory is used. Moreover the system cache management may lead to user space swaps - on our own or on other user's data. Thus, if we consider that OOC is requested when the memory is limited, this unpredictable behaviour is more likely to occur very often.

These drawbacks lead us to look for a new mechanism to load data from the hard disk.

6 Direct IO based method

In this section we present a new approach based on a direct access to the hard disk, that will be named `Direct IO based method`. Using the Direct IO access, the user has full knowledge and control of the memory used. This is a specific feature existing on many operating systems that can be specified while opening the files. Data must be aligned in memory when using direct I/O mechanisms: the address and the size of the buffer must be a multiple of the page size and/or of the cylinder size. The use of this kind of I/O operations ensures that a requested I/O operation is effectively performed and that no caching is done by the operating system. Strategies can then be used to prefetch data. The inconvenience of this method is that the System based cache mechanism is not available, it is more complex to implement and requires more algorithmic effort.

6.1 General presentation of the Direct IO based method

To solve large problems efficiently, which is the main target in designing an OOC solver, we propose to use small **user buffers** to explicitly control how data is prefetched from the disk.

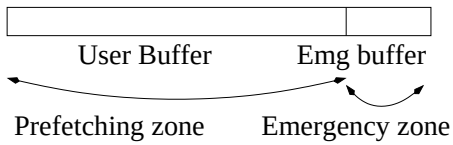


Figure 8: User defined buffers.

The memory is divided in two areas: a prefetching zone and an emergency one - Figure 8. In the prefetching zone, the whole available free space is used to load data. We prefetch each time a large enough contiguous block (1 MB in our experiments) is free. The emergency zone is used when a block factor is not prefetched or not 'on the way' (part of a prefetch request - see Algorithm 5). It has to be as large as the largest frontal

matrix. In this zone we load only one factor data at a time and it is used only in so called emergency cases.

The implemented algorithm reduces the disk access to the strict minimum - each data is loaded only once and kept in memory until it is used. To handle this, four states of the node are used to describe these transitions, (see Figure 9).
For every node the possible states are:
— **on disk only** - data is not available in the main memory
— **on the way** - data is not available, but it is being loaded
— **ready** - data is in the buffer and is ready to be processed
— **used** - data is in the buffer but has been already used. Corresponding space can be freed.

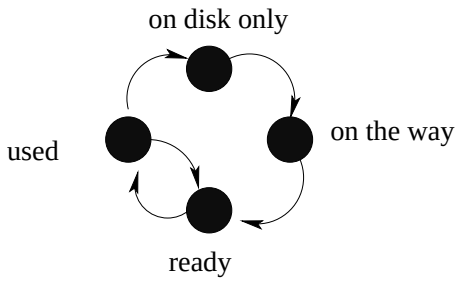


Figure 9: The 4 possible states of the node

The statement ‘on disk only’ means that the factors are not yet accessed. If we need to access data ‘on disk only’, we have to verify that there is enough free space in the buffer to load the data. The statement ‘on the way’ corresponds to data that is not yet in main memory, but we know that it is being loaded. So we may have to wait until the data is ‘ready’. After the prefetching process, all loaded data in the user buffers are in the state ‘ready’.

Here we use again the algorithms presented for the in-core execution (see Algorithms 2 and 3) with some additional functionalities (see Algorithm 5). Loading data is performed each time enough free space becomes available in the user buffer. Before processing a node, we check whether it is ‘ready’ or ‘on the way’, or whether we need to load it in the emergency buffer. The verification of data availability is done each time we have ‘`Use factors`’ in the algorithms. Prefetching data is done every time enough contiguous space is freed in the buffers.

Algorithm 5 : OOC functionalities for Direct IO based method

```
if (OOC run) then
  if (factors of Inode are 'on disk only') then
    Load data from disk      (emergency loading of Inode)
  else if ( the factors of Inode are 'on the way') then
    wait until the end of the prefetch
  end if
end if
Use factors to do ...
```

We compare the performance of the System based and the Direct IO based method on the large matrix Qimonda07 in a sequential environment and also analyse the behaviour

of our algorithm of using one or two buffers (emergency buffer and/or user buffer).

Methods	T_Fwd (sec)	T_Bwd (sec)	Nb_Req 1buffer Fwd	Nb_Req Emg zone Fwd	Nb_Req 1buffer Bwd	Nb_Req Emg zone Bwd
Direct IO (Emg)	1160.6	1295.8	0	3 083 998	0	3 083 998
Direct IO (Emg+1)	171.5	176.8	541	0	496	0
System based	161.6	221.6	—	—	—	—

Table 2: Influence of the number of buffers on the uni-processor performance on Qimonda07. Fwd=forward phase. Bwd=backward phase. Emg=emergency buffer:1 MB; User buffer:10MB.

When only the emergency buffer (Emg) is used, the total number of requests to the disks (Nb_Req Fwd and Nb_Req Bwd) is high and incurs a very significant time overhead (see Table 2). Using an additional single buffer (of small size 10 MB only), our prefetching mechanism can anticipate and in this case suppress the use of the emergency buffer.

In this case, the System based and the Direct IO approaches have similar execution times for the solution step. Even if the Direct IO based method is marginally better for the backward substitution (20% time reduction), it is not in this case the main advantage. Indeed, the memory effectively used for buffers in Direct IO is 10 MB whereas the cache for the System based method may be as large as 2.5 GB (the size of the factors). The performance of the solve is stabilized, while controlling the size of the buffers being effectively used.

6.2 Influence of scheduling on the sequential performance

The order in which nodes are extracted from the pool can be very critical for the execution time because this will influence the order in which data is read from the disk. Indeed solving a matrix using irregular access to the hard disk could slow down the time for both forward and backward steps by a factor of more than 10 (see Table 3). Therefore an efficient scheduler has to be implemented to reduce the number of disk accesses and to improve the regularity of accesses.

Scheduling the order of a node's processing is possible in the pool of tasks. We add nodes only at the end of the pool, but we can extract them in any order. We show the differences between two strategies - FIFO and LIFO, in terms of disk access (Figures 10 and 11 respectively). We describe how the factor data are stored on the hard disk and how, by using the assembly tree, we add into the pool all the ready tasks at each step.

We use three data structures: the assembly tree (tasks dependency), the pool of tasks (only for the ready tasks) and the user buffers (to load data from the disk). Two user buffers are used here: one with a prefetch mechanism and one for 'emergency' loading. We do not differentiate the states 'on the way' and 'ready'. All prefetched data are thus ready to be used. In our figures, the *arrows* point to the node to be processed. The numbers in grey with a diagonal line across represent already used data. Each time we have to process a node, that is not in memory, we load it to the emergency (Emg) buffer. Prefetching is performed, in this example, each time half of the user buffer is free, because the associated node factors are in the state 'used'.

In Figure 10, we present the optimal (for sequential execution) LIFO (Last In First Out) strategy for extracting a node from the pool. Hence we have no calls to the emergency zone during both forward and backward steps.

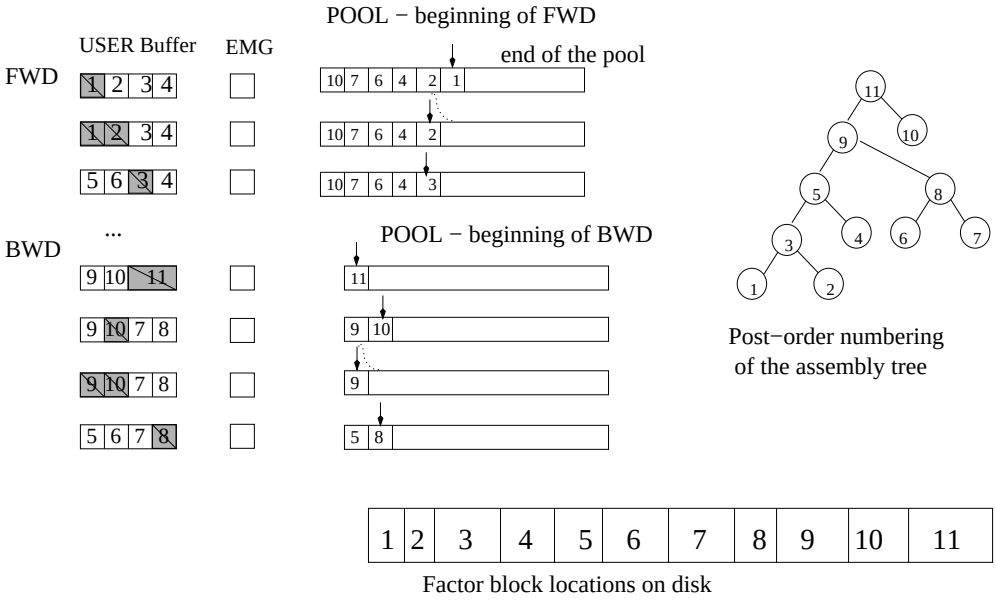


Figure 10: Algorithm with a LIFO processing of the tree in sequential mode

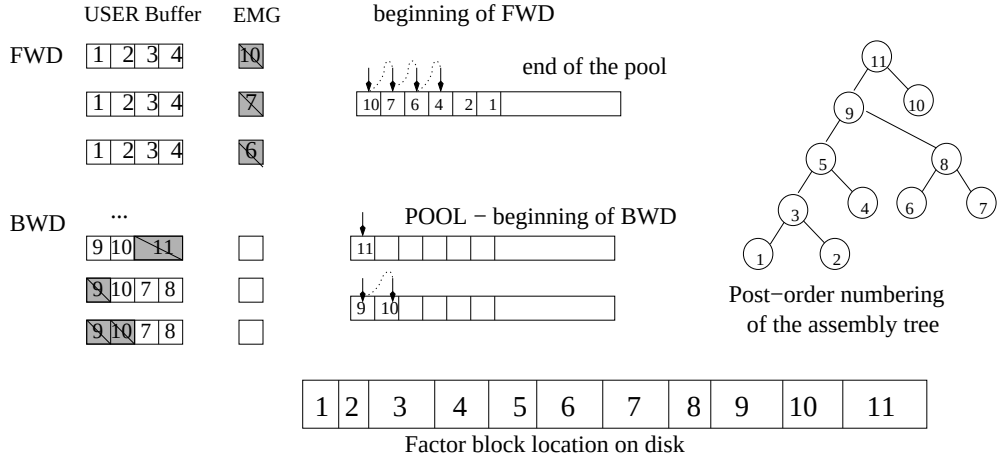


Figure 11: Algorithm with a FIFO processing of the tree in sequential mode

In Figure 11 we use a FIFO (First In First Out) strategy to extract the nodes from the pool. Starting with the forward step, the leaf nodes are added into the local pools so that the post-ordering is respected (from the end to the beginning of the pool). The user buffer prefetches data in the forward direction from the disk. Nodes 10, 7, and 6 are loaded through the emergency buffer. Loading data in the Emg buffer often leads to an irregular access (of relatively small size) to the data on the hard disk. This will influence the execution time of the whole phase.

For the backward step, the user buffer has prefetched in the reverse direction. Firstly, the root node is extracted from the pool and processed. Nodes 7 and 8 are, in our case, prefetched in place of the root factor blocks. This time we have less emergency calls and more regular access to the disk. Similar effects are observed on real matrices, which explains the relatively better behaviour of the backward step with the FIFO strategy (see Table 3).

We present the results of the two strategies in Table 3. We compare the time for the

forward and for the backward step with the minimum time needed if all factor data could be loaded at once (T_min). We compare, also, the number of requests per step for the user buffer and the emergency one (Nb_Req). Note that the FIFO strategy, which does not respect the node order on the disk, is significantly slower than the LIFO strategy. Furthermore, as expected, the forward step is even slower than the backward step in this case.

Strategy	Nb of Procs	T_min (sec)	T_Fwd (sec)	T_Bwd (sec)	Nb_Req 1buffer Fwd	Nb_Req Emg zone Fwd	Nb_Req 1buffer Bwd	Nb_Req Emg zone Bwd
LIFO	1	158.4	171.5	176.8	541	0	496	0
FIFO	1	158.4	2360.9	1480.1	338	3 054 580	30 053	2 877 695

Table 3: Influence of the scheduling of the tasks on Qimonda07. Size of Emg buffer is 1 MB; Size of the user buffer is 10 MB per processor

Running sequentially, the FIFO based extraction shows how critical the scheduling can be on the performance. In parallel, we cannot guarantee a post-ordering and a contiguous access to the factors and thus similar effects can be expected.

6.3 Influence of parallelism on the performance

In this section, we study the impact of the scheduling used on the performance of the parallel OOC solution step. We compare again the time for the forward and backward steps with the minimum time (T_min) to load factors from the disk and with the maximum bandwidth (16MB/s) on the most loaded processor. We also show the number of requests for the buffers per step.

Strategy	Nb of Procs	T_min (sec)	T_Fwd (sec)	T_Bwd (sec)	Nb_Req$^{(*)}$ 1buffer Fwd	Nb_Req$^{(*)}$ Emg zone Fwd	Nb_Req$^{(*)}$ 1buffer Bwd	Nb_Req$^{(*)}$ Emg zone Bwd
LIFO	1	158.4	171.5	176.8	541	0	496	0
LIFO	2	79.9	89.6	88.7	274	0	250	0
LIFO	3	57.9	64.9	262.1	190	3	169	422 497
LIFO	4	41.3	47.2	91.6	138	0	127	0
LIFO	6	31.5	38.0	186.7	102	6	86	422 498
LIFO	8	21.8	24.9	137.6	70	0	64	321 871
LIFO	16	11.9	13.2	94.4	39	2	32	214 245
LIFO	25	9.1	10.4	50.6	30	5	26	120 131

Table 4: Influence of the parallelism on Qimonda07. Size of Emg buffer is 1 MB; Size of the complementary buffer is 10 MB per processor; $^{(*)}$: Max per processor.

On one processor, a LIFO order to extract tasks from the pool leads to a contiguous access to the hard disk. In parallel, we cannot guarantee that the order of processing of the tasks (and the factor blocks) will correspond to the order used to write them to the disks. We thus see in Table 4 that work needs to be done on the scheduling to reduce the gap between the minimum time to load factors and the actual time, particularly for the backward substitution. In fact, this gap results from the large number of emergency calls during the backward step. Note that, in this example, we have almost no emergency requests during the forward step. One reason is that we use a relatively small number of processors. Another reason is that our large matrix has many frontal matrices (nodes in the tree) of relatively small size, so we have a relatively small number of type 2 tasks that could require the use of the Emg buffer.

On a large number of processors, we can expect the performance of both steps (Fwd and Bwd) to be even more sensitive to the order in which nodes are extracted from the pool. On a limited number of processors, however, one can expect the backward to be more sensitive than the forward step.

Indeed, at the beginning of the backward step, we have a small number of root-nodes, mapped onto a few processors. The other processors have no work and are waiting. During the backward step, the end of one task results in the freeing of multiple other tasks without prioritizing them. But choosing one instead of another task in the pool is critical for the performance. Furthermore, if we choose to process a node $Inode$, a LIFO strategy will induce the processing of all of its children before the brother of $Inode$. If the factors of this node, $Inode$, are not in memory then the factors of the children will not be in memory either. This will lead to emergency requests. We will further illustrate this in Figure 12. On the other hand, during the forward phase, where we exploit the large task independence of the leaves, all processors often have at least one node to process. In this case, all processors start working almost at the same moment. As the work is distributed regularly among the processors, they will progress in a synchronous way. The algorithm will more naturally process the complete tree respecting the post-ordering of the nodes in the tree.

For all these reasons and since we have seen in Table 4 that the performance of the backward phase is critical even on a limited number of processors, we discuss a modification of the scheduler and will illustrate it for the backward phase.

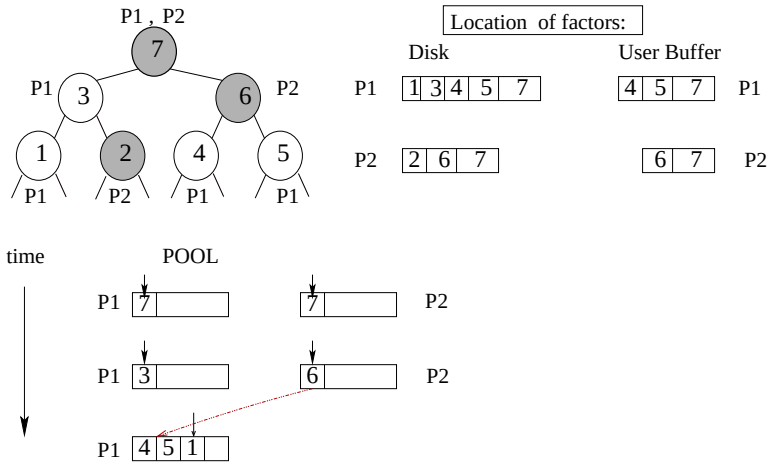


Figure 12: Parallel backward phase with LIFO extraction from the pool. The nodes with the same colouring are mapped on the same processor. The root node is of Type3 and is mapped on both processors.

On the given assembly tree, mapped on two processors (P1 and P2), see Figure 12,we represent the beginning of the backward step and the data in the user buffer and in the pool of tasks. To simplify the illustration of our algorithm, we assume that the root is mapped on both processors and that all other nodes are on only one processor (type1 nodes). We will comment on type 2 nodes in our algorithm later.

Some data are pre-loaded in the user buffer on both processors, respecting the backward step direction of needed data. After processing the root node, P1 continues with the only node in its POOL (node 3). This node is not 'in memory' and requires an emergency

access. If node 1 is added to the pool after the end of node 6 (that would add nodes 4 and 5 to the pool of processor P1) on processor P2, then accessing the factors of node 1 will lead to another emergency call.

The natural way to improve performance could be to strictly follow the write sequence of the factorization step. Doing so we would always get the node at the top of the memory. We hope that this algorithm will free more contiguous space in the user buffer, so that less emergency calls will be needed.

In our new algorithm (Algorithm 6), a new 'blocking receive' (at line β) has been introduced with respect to the Algorithm 1. The main difference between the blocking receive from the original algorithm (at line α of Algorithm 6) and the one introduced at line β is that, at line β, our blocking receive is performed while we have tasks ready to be activated in the pool. Since this is done separately on each processor (local pool) we have to prove that it does not introduce a deadlock between processes.

Changes made to our scheduling Algorithm 1 are written with larger font in Algorithm 6. All unchanged parts are written in tiny characters.

Algorithm 6 : Scheduling the POOL with next node in the sequence (NNS) strategy

```
step = Fwd_step or Bwd_step
if (step = Fwd_step ) then
   Initialise POOL with the leaf nodes mapped on Myid
   Initialise pointer to the first leaf node
else
   Initialise POOL with root nodes mapped on Myid
   Initialise pointer to the first root node
end if
while (Not finished) do
   if (POOL is not empty) then
      Try to receive a message and then Process_Message(message)
   else
      [α] Wait to receive a message and then Process_Message(message)
   end if
   if (POOL is not empty) then
      Try to extract node Next Node in the Sequence from POOL
      if ( Inode ≠ NNS) then
         [β] Wait to receive a message and Process_Message(message)
         cycle (to while)
      else
         Update position of the pointer to the next node master task in the sequence
      end if
      Step_Process_node(Inode)
   end if
end while
```

To prove the correctness of our new algorithm, we will formulate and demonstrate two more properties, based on the assembly tree and the task dependency.

Property 5 *Forcing the sequence to schedule nodes in Algorithm 6 does not introduce deadlock.*

Proof: First of all, as explained before, Type2 slave tasks do not go through the pool of tasks and are processed 'on the fly' (at the reception of a message MASTER2SLAVE for both forward and backward steps). Therefore, our blocking receive will not prevent us from treating such slaves tasks. Type3 tasks are only concerned with the largest root node of which only the master task will go through the pool. In our proof, we can thus focus

on the master tasks (of any type) since they are the only ones that might be blocked in the local pool.

Let us focus on the backward case. (The proof for the forward case is similar and can be easily deduced from the backward case.)

Let $\boxed{NBps}$ be the number of processes and let us suppose that we have a deadlock between r processes ($r \leq NBps$). On each process $\boxed{P_i}$ ($P_i \leq r$) let $\boxed{N_{P_i}}$ be the next node not processed in the sequence of processes P_i.

We first mention/prove a simple intermediate property between nodes ready to be activated in the local pools.

Property 6 *During the backward step, if node j is ready on process P_i, then j is not an ancestor of N_{P_i}.*

Proof: Thanks to the elimination main property, if j were an ancestor of N_{P_i} then it would be in the sequence of the backward step before N_{P_i}. This contradicts the definition of N_{P_i}. □

Proof of property 5 (continued)
Let N_{P_i} $i \in [0\,..\,r-1]$ be the nodes in the sequence that processes P_i are waiting for. If N_{P_0} is not ready (not in the pool), then it means that one of its ancestors (j_1) has not been processed. Because of Property 6, j_1 cannot be ready in the pool of P_0. Let us suppose, without loss of generality, that j_1 is in the pool of process P_1. Furthermore, on process P_1, N_{P_1} is not in the local pool. (Note that N_{P_1} might be equal to j_1). Therefore there exists an ancestor j_2 of N_{P_1}, ready to be activated on another process P_2. Either N_{P_2} is equal to N_{P_0} and we have a cycle of dependencies between processes, or we can continue and will end up with a cycle between r processes.

Let us suppose that we have reached a cycle of size r', $r \geq r' \geq 2$. Let

$$(N_{P_0}, j_1), (N_{P_1}, j_2), (N_{P_2}, j_3), ...(N_{P_{r'}}, j_0)$$

be such a cycle, where j_0 is ready on process P_0 and is an ancestor of $N_{P_{r'}}$. In each couple (N_{P_i}, j_{i+1}) j_{i+1} is an ancestor of N_{P_i} and is thus processed strictly before N_{P_i} in the backward sequence. Furthermore, by the definition of N_{P_i}, N_{P_i} is in the sequence before any node in the local pool of P_i. Let $\rightarrow$ denote the precedence in the backward sequence. $x \rightarrow y$ mean that x is before y in the backward sequence. $\stackrel{a}{\rightarrow}$ indicates an ancestor relation, $x \stackrel{a}{\rightarrow} y$ indicates that x is before y because x is an ancestor of y. (Note that $x \stackrel{a}{\rightarrow} y$ implies $x \rightarrow y$ and $x \neq y$). We thus have :

$$j_0 \stackrel{a}{\rightarrow} N_{P_{r'}} \rightarrow j_{r'} \stackrel{a}{\rightarrow} N_{P_{r'-1}} \ \ldots \ j_2 \stackrel{a}{\rightarrow} N_{P_1} \rightarrow j_1 \stackrel{a}{\rightarrow} N_{P_0} \ ,$$

which means that N_{P_0} is not the first ready node *in the sequence of process* P_0, since j_0 is ready and is before N_{P_0} in the sequence. Thus j_0 is equal to N_{P_0}. Furthermore, thanks to our cycle, j_0 is before j_1 in the sequence ($j_1 \neq j_0$), which contradicts the fact that j_1 is an ancestor of $N_{P_0}(= j_0)$ located on process P_1. We have thus proved that our algorithm does not introduce any deadlock. □

In our algorithm, we use a pointer to the next node in the sequence, that is why we call this strategy **Next Node in the Sequence** (NNS). Each time we have to extract a

node from the pool, we choose this particular node. The NNS strategy is illustrated in Figure 13. One can see that with a LIFO strategy node 3 is added to the pool of process P1 at the end of the process of the root node 7 mapped on both processes. Node 3 is then treated by P1 before nodes 4 and 5. On the other hand, with the NNS strategy, node 3 is not processed and P1 waits for node 5 to be added to the pool since it the next node in the sequence after node 7.

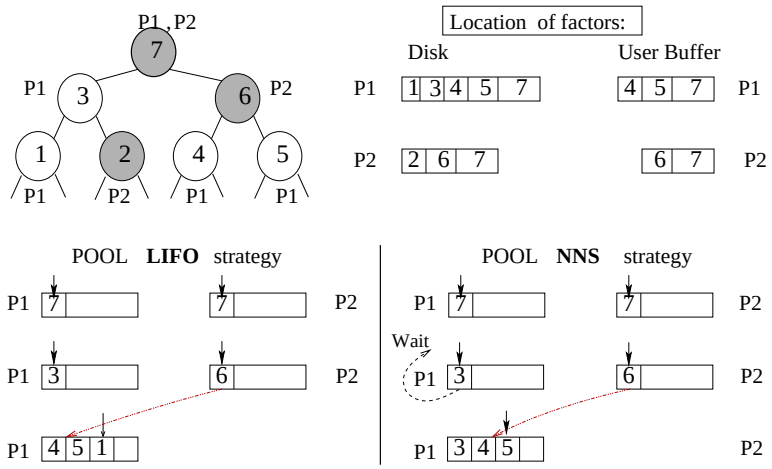


Figure 13: Comparison of LIFO and NNS extraction from the pool.

Note that slave tasks are not considered in this sequence. The slave tasks, for type 2 nodes, are processed on the fly (do not use the pool) and are driven by the order in which the messages are received. Our pointer thus only considers master tasks that are scheduled through the POOL. In our new algorithm the slave tasks of type 2 nodes might still involve requests to the Emg buffer and/or to be prefetched out of sequence. This will influence the performance mostly on a large number of processors.

Normally, the next node in the sequence is located at the end of the user buffer, and processing this node will free more contiguous space in the buffer. We hope that this will lead to more regular disk access and will improve the performance especially for the backward step in a parallel environment.

Strategy	Nb of Procs	T_min (sec)	T_Bwd (sec)	Nb_Req$^{(*)}$ 1buffer Bwd	Nb_Req$^{(*)}$ Emg zone Bwd
NNS	1	158.4	177.2	496	0
NNS	2	79.9	93.7	250	0
NNS	3	57.9	65.5	174	1
NNS	4	41.3	50.5	117	0
NNS	6	31.5	37.9	93	0
NNS	8	21.8	45.2	57	0
NNS	16	11.9	13.8	36	0
NNS	25	9.1	10.7	25	0

Table 5: Influence of the scheduling NNS of the tasks on Qimonda07. Emg= 1 MB; User buffer = 10 MB per processor; $^{(*)}$: Max per processor.

The results, presented in Table 5 show that using the NNS strategy helps to reach good performance in the backward step on parallel runs. Qimonda07 has a large number of relatively small nodes, with a relatively small number of Type2 nodes. This explains

why with our NNS algorithm in general we have no emergency calls in both steps of the solution phase. The time for the backward step has a more realistic behaviour and is reduced by a factor of 5 (see 6 processors: using LIFO strategy — 186.7 sec and NNS strategy — 37.9 sec). As shown, the NNS strategy is much closer to the minimum time for loading factors from disk.

7 Concluding Remarks

We have described, in this paper, the main steps of a multifrontal algorithm for distributed forward and backward substitutions. We have shown that our original algorithms can be easily adapted for OOC execution. We have then compared two different approaches to read factors from the hard disk. In this context a 'naive' System based OOC approach is not suitable mostly because of its unpredictable memory use.

A direct IO access to the disk with small user buffers has thus been introduced to control the memory effectively used. In a sequential environment we have first shown how critical the task scheduling can be. We have observed that one important issue is to control the number of hard disk accesses. Another issue is to obtain 'regular' disk accesses. While controlling memory effectively used, we then studied the parallel behaviour of our solver. We have shown that the optimal sequential task scheduling is not efficient in a parallel context. To obtain more regular disk access, especially for the backward step, we have constrained the scheduler to follow the factorization write sequence of factor matrices during the factorization. We have proved the correctness of the algorithm and have shown that we can significantly reduce the time for solution.

A possible extension to this work would be to experiment on a larger number of processors to find the limitation of our constrained scheduling approach. In this context, we may then have to exploit also the write sequence of the slave tasks of type 2 nodes. Another extension to this work could be to study the overlapping between computations and disk access in the context of multiple right-hand sides. Finally we may try to influence the scheduler of the factorization to improve the solve (future collaboration with Agullo and L'Excellent, INRIA-LIP-ENS Lyon).

8 Acknowledgement

We are very grateful to J.-Y. L'Excellent for many useful discussions and for comments on the first draft of this work.

References

[1] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. A preliminary out-of-core extension of a parallel multifrontal solver. In *EuroPar'06 Parallel Processing*, pages 1053–1063, 2006.

[2] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[3] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.

[4] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[5] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, 97:1–15, 1996. (also as LAPACK Working Note #95).

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

[7] 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.

[8] 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.

[9] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[10] E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, 1999.

[11] Vladimir Rotkin and Sivan Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Trans. Math. Softw.*, 30(1):19–46, 2004.