

Analysis of the Solution Phase of a Parallel Multifrontal Approach

P. R. Amestoy¹, I.S. Duff², A. Guermouche³, Tz. Slavova²

Technical Report

August 25, 2008

ABSTRACT

We study the forward and backward substitution phases of a sparse multifrontal factorization. These phases are often neglected in papers on sparse direct factorization but, in many applications, they can be the bottleneck so it is crucial to implement them efficiently. In this work, we assume that the factors have been written on disk during the factorization phase, and we discuss the design of an efficient solution phase.

We will look at the issues involved when we are solving the sparse systems on parallel computers and will consider in particular their solution in a limited memory environment when out-of-core working is required. Two different approaches are presented to read data from the disk, with a discussion on the advantages and the drawbacks of each.

We present some experiments on realistic test problems using an out-of-core version of a sparse multifrontal code called MUMPS (MULTifrontal Massively Parallel Solver).

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

Keywords: direct methods, multifrontal solver, MUMPS, sparse matrices, out-of-core algorithms.

¹ ENSEEIHT-IRIT UMR 5505, Université de Toulouse, France.

Email: Patrick.Amestoy@enseeiht.fr

Current report also available at <http://apo.enseeiht.fr>.

² CERFACS, Toulouse, France

Email: Iain.Duff@cerfacs.fr & Mila.Slavova@cerfacs.fr

³ LaBRI, Univ. Bordeaux 1 / INRIA Futurs

Email: Abdou.Guermouche@labri.fr

1 Introduction

In the direct solution of a linear system, $Ax = b$, the matrix A is first factorized into the factors LDL^T (when A is symmetric) or LU (when A is unsymmetric), where L and U are triangular matrices and D is diagonal (or block diagonal with blocks of order 1 or 2 in the case of numerical pivoting for indefinite systems). Note that in our factorization expressions, we have omitted, for the sake of clarity, the permutations performed to preserve sparsity and to implement numerical pivoting. These factors are then used to solve the system through the forward and backward substitution steps

$$[LDy = b \text{ and } L^T x = y] \quad \text{or} \quad [Ly = b \text{ and } Ux = y], \quad (1)$$

depending on whether the matrix is symmetric or not. In this paper, we are concerned with the case when the matrix A is large and sparse [5]. One main limitation in the use of sparse direct methods comes from the need to store the matrix factors that have often many more entries (10 to 100 times) than the original matrix.

Usually the most time consuming part of the solution process is in the initial matrix factorization and it is this step that most previous work has addressed. However, in many applications, the substitution phases can be performed very many times for each factorization so that the accumulated time for these phases dominates. This is true, for example, in some algorithms for nonlinear optimization and for applications where solutions with many different right-hand sides are required (for example, in electromagnetic or seismic modelling). Furthermore, when solving systems in parallel or when working out-of-core, the substitution times can be greatly increased. We believe this is the first in depth study of the substitution phases in a parallel and out-of core environment. Our work differs and extends the work of [10, 11, 12, 9] because firstly we consider a parallel out-of-core context, and secondly we focus on the performance of the solve phase.

In this paper, we use an out-of-core (**OOC**) multifrontal [6, 7] approach. Here the matrix factors are written to disk during the factorization phase, as a sequence of blocks (that we call **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 forward and backward substitution phases, that we call the **solution 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.

We first discuss in Section 2 the main aspects of the in-core distributed memory solution phase. Although we have described details of our solver MUMPS in previous publications [2, 3, 4] this is the first time we have considered the solution phase in detail. We explain why our parallel solution phase does not follow the standard dependency structure of the factorization phase and prove the correctness of our approach. We then explain how our algorithms have been adapted to the out-of-core context (Section 2). We describe in Section 3 our testing environment - the hardware we use and our choice of matrices for the tests. We show in Section 4 the limitations of a simple demand driven approach, that we call `SYSTEM_BASED`, based on automatic system I/O caching mechanisms. In Section 5 we show how user buffers can be introduced to improve the behaviour of the solution phase and then describe an approach where the memory used is completely controlled, which we call `DIRECT_IO`. We show that a naive implementation of the `DIRECT_IO` approach is not suitable for parallel implementation and introduce a new scheduling scheme that constrains the ordering of the tasks. We first prove that the new algorithm is correct and then illustrate the gain in performance obtained on a set of large real problems.

2 Main parallel features of the solution phase

For the sake of clarity we will focus on symmetric matrices and will not consider pivoting for numerical stability in the description of our algorithms. Note that our algorithms handle both symmetric and unsymmetric matrices and incorporate numerical pivoting (threshold pivoting and two-by-two pivots). Our approach is a multifrontal one that uses an **elimination tree** [8] to represent the dependencies of the computations during the solution phase. In practice, nodes of the elimination tree are amalgamated so that more than one variable can be eliminated at each node of the tree. The resulting amalgamated tree is referred to as the **assembly tree**.

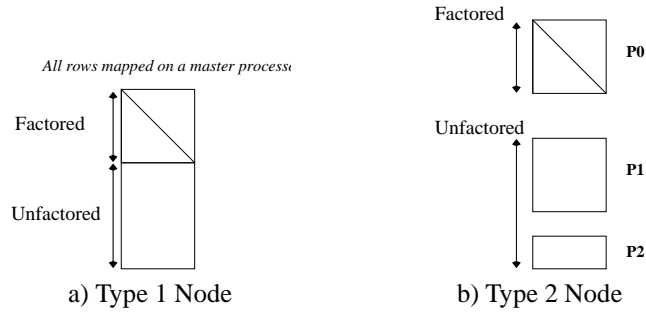


Figure 1: Partitioning and distribution of the factor blocks depends on the type of the node. On the Type 2 node, P0 is the master process in charge of factored row variables, and P1 and P2 are slave process in charge of a partition of the unfactored row variables.

During the solution phase, each node of this tree holds the L factor block computed during factorization. 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. The factor block can be partitioned into **factored** row variables and **unfactored** row variables as shown in Figure 1. In our parallel context, the distribution of the L factors onto the processes depends on how each node was factored. If a node is factored by a unique process, so called **master** process, the computed L factors are located on this process and the node will be referred to as a **Type 1 node**. If the node is factored on more than one process the node will be referred to as a **Type 2 node**. A so called master process holds the first diagonal block of the factored rows and the off-diagonal block is distributed on so called **slave** processes (see Figure 1-b). The terminology master-slave (as we use here) only reflects the fact that during factorization a process dynamically decides to distribute work to other processes that become slaves for this node process [2, 3, 4]. Finally for a root node, that we call a **Type 3 node**, a block cyclic 2D distribution of the factors is performed.

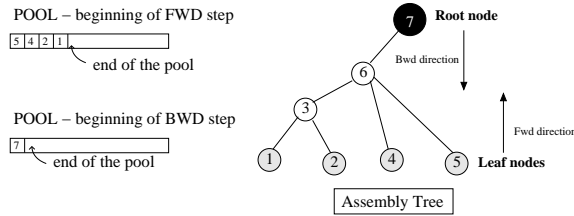


Figure 2: The POOL of tasks at the beginning of the forward and backward sequential solution steps.

The task dependency graph of both the forward and backward algorithms makes use of a distributed pool of tasks, that we call the **POOL**. This pool contains a list of all ready tasks to be executed and is used to schedule work in both the sequential and the parallel cases. At the beginning of each step, we initialize the distributed pool with all tasks ready on each process (see Figure 2 for a description of the situation on one process). Task are then extracted from the end of the pool (LIFO strategy). For the forward step, the pool is initialized with the leaf nodes of the assembly tree. A node will then be placed at the end of the pool as soon as all of its children are processed. 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. Furthermore, for both the forward and the backward steps, when a node is distributed over more than one process (Type 2 or Type 3 nodes) only the master task is added to its local pool. The slave tasks are processed on the fly. The algorithm for extracting nodes from the pool is described in Algorithm 1.

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 message need 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.

We first describe in Algorithm 2 the parallel forward substitution ($Ly = b$). Note that in Algorithm 2 (and in practice) the same working space can be used to store both y and b . We kept two separate vectors in our algorithm only for the sake of simplicity. To analyse the main features of our algorithm, we prove two

Algorithm 1 : Algorithm for extracting a node from the POOL (LIFO strategy)

```
Myid - process number; Inode - the current node mapped on process Myid;  
Step = Fwd or Bwd  
if ( Fwd ) Initialise POOL with the leaf nodes mapped on Myid  
if ( Bwd ) Initialise POOL with root nodes mapped on Myid  
while (Not finished) do  
  if (POOL is not empty) then  
    if a message is available Process_Message(message)    [See Algorithms 2 and 3]  
  else  
    Wait for a message and then Process_Message(message)  [See Algorithms 2 and 3]  
  end if  
  if (POOL is not empty and Process_Message not called) then  
    Extract node, say Inode, from the end of the POOL  
    if ( Fwd ) Fwd_Process_node(Inode)  [See Algorithm 2]  
    if ( Bwd ) Bwd_Process_node(Inode)  [See Algorithm 3]  
  end if  
end while
```

properties related to the use of the assembly tree. We show (Property 2) that our algorithm does not always follow the dependency paths of the assembly tree which explains why we must reset our working array Wb to zero.

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 Algorithm 2 and Algorithm 3. 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 since the extension to multiple right-hand sides is straightforward.

Property 1 *All updates to factored 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 updates 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 updates to the factored variables of a node can only come from nodes involved in the sub-tree rooted at that node (main property of the assembly tree). This proves our property. \square

Property 2 *All updates of descendants of a node *Inode*, to unfactored variables of a node are not always sent to processes in charge of that node.*

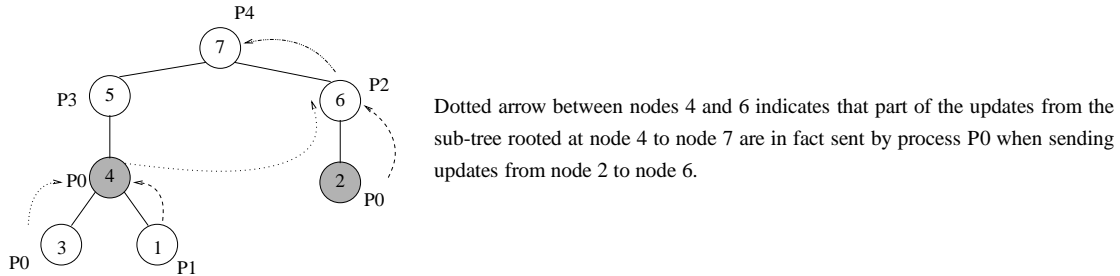


Figure 3: Example used to prove Property 2: part of the updates of node 1 are not sent to process P3, in charge of node 5.

Proof: Figure 3 will be used to prove our property. All nodes in Figure 3 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 updates of the sub-tree rooted at node 5 will circulate through node 6 on P2. This is the case if both node 1 and node 2 have a common row in the factor block 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 updates from node 1 to node 7. Instead Property 1 says that the common row updated by node 2 and node

Algorithm 2 : Algorithm for the forward step ($LDy = b$)

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 *father(Inode)* is mapped.
Wb - a local working array, initialized to 0 and designed to accumulate modifications of the right-hand side *b*;
`Use_factors` will be expanded in later discussion to cases when such use or access to them is non-trivial.
Fwd_Process_node(*Inode*) {I am the master of node *Inode* }
For factored rows, update *b* with entries of *Wb* and `Use_factors` to compute the partial solution (TRSM/V)
if (*Inode* is of Type 2) **then**
 Send to each slave of *Inode* the computed solution and entries of *Wb* corresponding to rows mapped on this slave (message MASTER2SLAVE) and reset these entries of *Wb* to zero (see Property 2)
else if (*Inode* is of Type 1) **then**
 Update *Wb* for unfactored rows (GEMM/V)
 if (*Myid* \neq *Pfather*) **then**
 Send updated entries of *Wb* to *Pfather* (message ContVec) and reset them to zero (see Property 2)
 else
 Increment updates for *Pfather* and if last update add *father(Inode)* to the end of the pool
 end if
else
 Type 3 root node process based on ScaLAPACK for both forward and backward steps on all processes
end if
Process_Message(Message) {I am updating *Inode*}
if (Message = ContVec) **then**
 Update *Wb* with contribution received; Increment number of updates
 if last update, add *Inode* to the end of the pool
else if (Message = MASTER2SLAVE) **then**
 Gather in a small *local array* entries of *Wb* just received
 `Use_factors` and the solution sent by the master to update the *local array* (GEMM/V)
 if (*Myid* = *Pfather*) **then**
 Scatter and add the *local array* in *Wb*
 Increment number of updates and if last update add *Inode* to the end of the pool
 else
 Send *local array* to *Pfather* (message ContVec)
 end if
end if
end if

4 could not be eliminated at node 5, but at node 7. \square

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 factored variables. However, contrary to what is exploited during multifrontal factorization, this elimination tree property is no longer respected on each process for unfactored variables (Property 2). Property 2 also explains the importance of resetting *Wb* to zero in Algorithm 2.

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

Proof: We recall that *Wb* is designed to sum update 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 gathers in a local array, contributions sent by its master. This local array is 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. \square

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 factored variables of the node.*

Proof: Results from Property 1 and 2.

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

The algorithm for the backward substitution ($L^T x = y$) is described in Algorithm 3. As for the forward step, priority is given to message reception. If no message is received, a node from the pool is extracted. The

backward step manages three types of messages: `Bwd_MASTER2SLAVE` and `Bwd_ContVec` are similar to `MASTER2SLAVE` and `ContVec` of the forward case respectively; a new type of message `Bwd_Node` is used to control the activation of the children.

Algorithm 3 : Algorithm for the backward step ($L^T x = y$)

```

Myid - process number; Inode - current node mapped on Myid;
Bwd_Process_Node(Inode)
if (Inode is of Type 2) then
    Master distributes already computed solution corresponding to factored variables between the slaves of Inode (message
    Bwd_MASTER2SLAVE)
else if (Inode is of Type 1) then
    Use_factors associated with unfactored variables to update y (GEMM/V) and with factored variables to compute solution
    (TRSM/V)
    for each child of Inode whose master process is mapped on Myid, add it to the end of the pool
    Send the solution corresponding to all variables of Inode to processes on which at least one master of a child node is mapped
    (message Bwd_Node)
end if
Process_Message(Message)
if (Message = Bwd_Node) then
    Update known solution and add to the pool Inode and all of its brothers whose master process is mapped on Myid
else if (Message = Bwd_MASTER2SLAVE) then
    Use_factors mapped on this slave process together with the received solution to compute a contribution to y (GEMM/V) and
    send it to the master of Inode (message Bwd_ContVec)
else if (Message = Bwd_ContVec) then
    Update y with message and increment number of updates received.
    if last update then
        Use_factors associated with factored variables to compute the solution (TRSM/V)
        for each child of Inode whose master process is mapped on Myid, add it to the end of the pool
        Send the solution corresponding to all variables of Inode to processes on which at least one master of a child node is mapped
        (message Bwd_Node)
    end if
end if

```

During the backward step, when a Type 2 node is processed the slave processes are first involved in the updating of the right-hand side *y* (after reception of message `Bwd_Master2Slave` from the master process of that node). Once the master process has received all updates to the right-hand side computed by the slaves (message `Bwd_ContVec`), the solution associated with the factored variables is then computed. A message `Bwd_Node` is then sent to each process on which at least one master node of the children is mapped. Note that even if several nodes are mapped on the same process, messages `Bwd_Node` will be sent only once to this process.

sectionOut-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 achieve 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 access time. 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.

2.1 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 4).

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 data has been written to the disk at the end of the factorization phase.

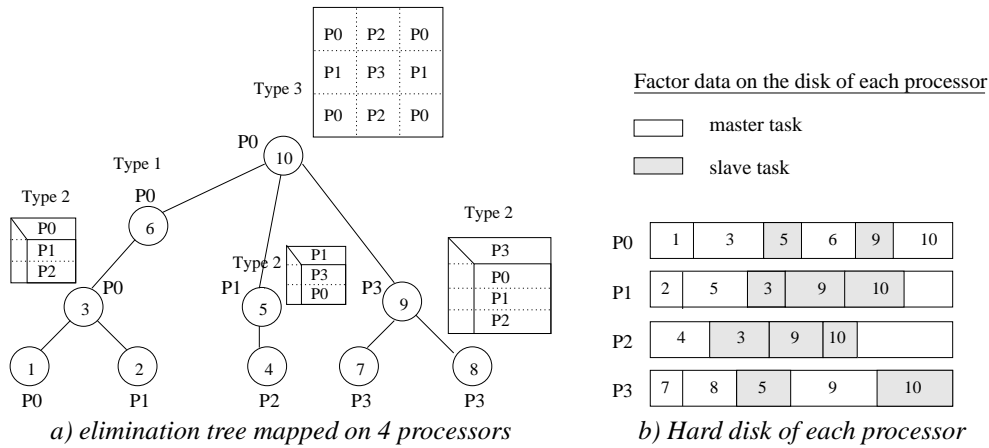


Figure 4: Example of interleaving of master and slave tasks during the factorization and influence on the disk usage on each processor. We note that the sequence is not unique because of the non-deterministic nature of our asynchronous algorithm.

2.2 OOC solution phase

We use the factorization write sequence in order or in reverse order, to prefetch factor blocks 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 prefetch data in the natural direction (the order in which data has been written) (see Figure 5). 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 slightly worse than the forward step.

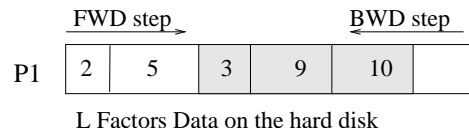


Figure 5: Reading direction on the disk in the solution step.

For this OOC implementation 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 Algorithms 2 and 3.

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

<code>Use_factors</code>	of <code>Inode</code> ...	⇒	<pre> if (OOC run) then Load data from disk(<code>Inode</code>) end if <code>Use_factors</code> ... </pre>
--------------------------	---------------------------	---	--

3 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, 2 GB per MPI process). 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.

Our set of test matrices ¹ used for the experiments is described in Table 1, sorted by factor size. We note

¹Publicly available matrices from the PARASOL collection (AUDI) or from our applications partners are available on the

the difficulty in getting very large problems from industry. It is also necessary that the integer description of the problem will fit on a single processor in order for us to complete the analysis and construct the data structures for subsequent numerical factorization and solution.

Matrix name	Order	Nb entries (Millions)	Factor size (MBytes)	Nb Nodes in the tree	Description (origin)
QIMONDA07	8 613 291	66.9	2 534	3 083 998	Circuit simulation (Qimonda AG company)
CAS4R-L15	2 423 135	195.8	4 832	864 447	3D Electromagnetism (EADS Innovation Works)
CONESHL	1 262 212	43.0	5 908	113 513	3D finite element from SAMCEF code (SAMTECH)
NICE20MC	715 923	28.1	9 263	68 134	Seismic processing (BRGM Lab.)
AUDI	943 695	39.3	12 202	113 119	Automotive crankshaft model (Parasol collection)
GRID3.5M	3 500 000	37.8	15 720	1 535 044	3D 11pt-discretization of Laplacian operator
COR5HZ	2 233 031	90.2	21 622	268 798	Seismic processing (BRGM Lab.)
AMANDE	6 994 683	584.8	55 295	871 621	3D Electromagnetism (CEA-CESTA)
NICE9HZ	5 140 838	215.5	64 848	603 495	Seismic processing (BRGM Lab.)
GRID5M	5 000 000	53.8	17 335	2 204 519	3D 11pt-discretization of Laplacian operator

Table 1: Test matrices (size and origin); Size of factors obtained with Metis reordering of the original matrix. All test matrices are real symmetric except CAS4R-L15 and AMANDE which are complex symmetric.

We recall that during factorization **all** factors are written to the local disks. We have no factors data kept in memory at the beginning of the solution phase *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. Note that QIMONDA07 is a large and very sparse matrix with more than 3 million nodes in the assembly tree. I/O access might occur for each node of the elimination tree and thus it is an interesting matrix to illustrate the behaviour of our algorithms. We thus use this example extensively in our detailed analysis but show relevant results on all our test problems later in the paper.

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, that we will refer to as SYSTEM_BASED. A second approach consists in a direct access to the disk and will be referred to as DIRECT_IO. With the DIRECT_IO approach we will explain that we have complete control of the memory used during the solution phase.

4 System based demand driven approach

A simple way to implement the OOC solution phase is to use 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 illustrate in Table 2 its behaviour on our test matrix QIMONDA07. We analyse the situation when the matrix fits in the main memory (parallel execution) and when the memory is critical (uniprocessor execution) and also report, as a reference, the in-core solution time on 8 processors.

Nprocs	Factor Size (per proc) MB	Solve		
		Fwd (sec)	Bwd (sec)	Disk access (MB/s)
In core				
8	317.5	0.9	0.9	—
OOO (Out-Of-Core)				
8	317.5	3.6	4.5	92.6
4	635.0	45.9	15.1	83.3
2	1 270.1	129.4	93.1	22.8
1	2 534.3	269.4	282.9	9.2

Table 2: Influence of memory used per node of the Cray XD1 on the performance of the solution phase on matrix QIMONDA07. The OOO is based on a simple SYSTEM_BASED approach.

On 8 processors, we see that the extra time required in both forward and backward phases for the OOO execution corresponds to copying the factor data at a rate of 92.6 MB/s so that the copy is not from the disk (bandwidth of 16 MB/s) but from the system cache. Indeed the SYSTEM_BASED demand driven approach unpredictably affects the behaviour 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

gridtlse.org web site; COR5HZ matrice corresponds to dynamic analysis of the Corniglio (Italy) earthquake (1994) with maximum signal frequency of 5 Hz. NICE20MC and NICE9HZ correspond to dynamic analysis of the Nice earthquake (2001) with maximum signal frequency of 1.5 Hz and 9 Hz respectively.

them during the solution phase is the cost of a main-memory access. The OOC execution allows us to decrease the number of processes used by increasing the local factor size per process. The fewer processes that are used, the fewer factors remain in the system caches and, as a consequence, the speed of access to the factors decreases. On QIMONDA07, the size of the total workspace for sequential in-core factorization (5 GB) is bigger than the available memory (4 GB). In OOC execution, a working space of size 2 GB is still needed during the factorization so that the system cannot keep all the factors in the system caches at the end of the factorization phase. Some factor blocks must then be loaded from the disk. In this case, increasing the number of disk accesses will increase the execution time. On one process, the disk access speed is really slow – 9.2 MB/s. Note that the peak speed of a memory read from the disk is 16 MB/s, so that the minimum time just to load all the factor blocks is 158 seconds.

We thus see that, when the memory is critical, the performance of the SYSTEM_BASED approach is far from the minimum. The reason is that the system I/O mechanism is in conflict with the automatic system swapping mechanisms.

As shown in Table 2, the SYSTEM_BASED approach 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, the fact that the system cache grows with each disk access (reading or writing data) is even more critical. It is impossible to control the actual memory used: either its size or the effective bandwidth for accessing 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, or even other system processes. 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.

5 Direct IO based method

In this section we present a new approach based on direct access to the hard disk, that will be named DIRECT_IO. 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_IO 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 operation 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 cache mechanism exploited by the SYSTEM_BASED approach is not available; it is thus more complex to implement and requires more algorithmic effort.

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.

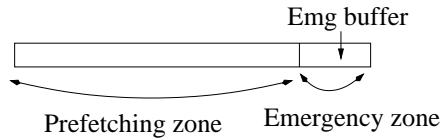


Figure 6: User defined buffers.

The buffer is divided into two areas: a prefetching zone and an emergency one - as we show in Figure 6. In the prefetching zone, all the space allocated to this is used to load data. We prefetch each time a large enough contiguous block in the prefetching zone is free (1 MB in our experiments). 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 factor block. In this zone we load only one factor block at a time and it is used only in so called emergency cases.

The size of the user buffers can influence the performance of the solution phase. The size of the emergency zone $EmgSize(p)$ is defined as the largest block factor mapped on processor p . Let $AvgEmgSize$

denote the average of $\text{EmgSize}(p)$ over the processors. Let $\text{FactorSize}(p)$ be the size of the L factors per processor and let AvgFactorSize be its average size. The prefetching buffer zone on each processor $\text{PrefetchBufferSize}(p)$ is then defined as

$$\text{PrefetchBufferSize}(p) = \max \left(\min \left(10 \times \text{AvgEmgSize}, \frac{\text{AvgFactorSize}}{4}, 500\text{MBytes} \right), \text{EmgSize}(p), 10 \text{ MBytes} \right) \quad (2)$$

The total size of buffers per processor is then

$$\text{Size of buffers}(p) = \text{PrefetchBufferSize}(p) + \text{EmgSize}(p) \quad (3)$$

In the context of our study we want to control the buffer size with respect to a fixed value (here 500 MBytes) and with respect to the volume of I/O per processor ($\text{AvgFactorSize}/4$); to reduce the buffer size when increasing the number of processors and limit the difference of the buffer sizes on the processors (upper bounds based on average distributions) and finally to enable some prefetching for our algorithms ($10 \times \text{AvgEmgSize}$). In the remainder of this paper, equation (3) will be used to define the size of buffer area for our experiments.

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 7).

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.

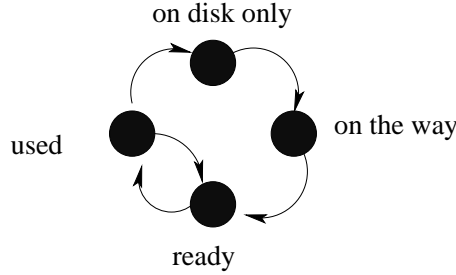


Figure 7: The 4 possible states of the node

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 contiguous free space becomes available in the prefetching zone.

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.

Algorithm 5 : OOC functionalities for the DIRECT_IO approach

```

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 first compare the performance of the `SYSTEM_BASED` and the `DIRECT_IO` approaches on the large matrix `QIMONDA07` in a sequential environment and also analyse the behaviour of our algorithm when using one or two buffers (emergency buffer and/or prefetch buffer). When only the emergency buffer (Emg) is used (PrefetchBufferSize set to zero in equation (3)), 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 3). Using a prefetch buffer of small size, our prefetching mechanism can anticipate and in this case suppress the use of the emergency buffer. As the total size of the factors is bigger than the available memory (2 GB), in this case, both the `SYSTEM_BASED` and the `DIRECT_IO` approaches *really* load factors from disk. Thus it becomes possible to compare their execution time on the solution phase. Even if the `DIRECT_IO` time is better for both forward and backward steps, there is a more major reason to favour this approach.

Methods	Fwd (sec)	Bwd (sec)	Nb_Req Fwd		Nb_Req Bwd	
			Prefetch	Emg zone	Prefetch	Emg zone
DIRECT_IO (Emg)	1160.6	1295.8	0	3 083 998	0	3 083 998
DIRECT_IO (Emg+Prefetch)	171.5	176.8	541	0	496	0
SYSTEM_BASED	269.4	282.9	—	—	—	—

Table 3: 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.

The main advantage is that the memory effectively used for buffers in the `DIRECT_IO` approach is 10 MB whereas the cache for the `SYSTEM_BASED` approach may be as large as 2.5 GB (the size of the factors). The performance of the solve is thus stabilized using the `DIRECT_IO` strategy, while controlling the size of the buffers being effectively used.

To illustrate the time limitations of the `SYSTEM_BASED` approach, we show in Table 4 the parallel behaviour of the solution phase on our complete set of test matrices. We are interested in the case where factors are written to disk during the factorization phase because memory was limited. For the sake of clarity we thus assume that before each step (forward or backward) the system cache is flushed so that we are sure that both the `SYSTEM_BASED` and the `DIRECT_IO` approaches will have to read the L factors from disk. For each matrix, the minimum number of processors required to run the factorization phase was chosen. We see in Table 4 that, in parallel, the `SYSTEM_BASED` approach does not efficiently prefetch the L factors from the disk.

Matrix name	L factor size		Pr	Method	Workspace per proc (MB)	Fwd	Bwd
	Avg (MB)	Max (MB)					
QIMONDA07	2534	2534	1	sb	(*)	269.4	282.9
				od	12	171.5	176.8
CAS4R-L15	2416	2547	2	sb	(*)	595.3	1061.2
				od	559	336.3	270.1
CONESHL	5908	5908	1	sb	(*)	446.1	448.1
				od	709	375.2	378.3
NICE20MC	1537	1689	6	sb	(*)	158.4	239.0
				od	491	148.7	225.2
AUDI	2741	2872	4	sb	(*)	298.6	573.5
				od	676	231.8	355.2
GRID3.5M	7860	7900	2	sb	(*)	680.2	808.9
				od	639	507.0	519.0
COR5HZ	2702	2970	8	sb	(*)	334.8	507.4
				od	660	397.1	476.5
AMANDE	1404	1625	20	sb	(*)	512.4	1291.8
				od	425	725.9	964.8
NICE9HZ	3208	3651	20	sb	(*)	596.8	1299.4
				od	893	685.9	1050.2
GRID5M	4259	4356	4	sb	(*)	439.8	614.5
				od	699	325.4	554.0

Table 4: Time performance of the `DIRECT_IO` (**od**) and the `SYSTEM_BASED` (**sb**) methods; Workspace holds the average working space used by the solution phase (including prefetching buffer defined in equation (3)). (*) It cannot be estimated in the `SYSTEM_BASED` approach because of the system cache).

5.1 Influence of parallelism on the performance

In the previous section, we have thus shown that the `SYSTEM_BASED` approach is not efficient in terms of both memory (no control of the effective memory used) and time (automatic system based prefetching not adapted to a parallel execution). In this section, we analyse in more detail the parallel behaviour of

the DIRECT_IO approach and focus on the influence of tasks scheduling on the performance. Scheduling the order the processing of a node 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. A LIFO (Last In First Out) strategy was used in the initial Algorithm 1 because it is an optimal strategy (in terms of regularity of disk access to block factors) for sequential execution.

We first compare in Table 5 the time for the forward and backward steps with the minimum time (T_{min}) to load factors from the disk on the QIMONDA07 matrix. We also report the maximum bandwidth (16MB/s) on the most loaded processor and the number and type of buffer requests per step. 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 see in Table 5 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.

Nb of Procs	Factor Size per proc (MB)	T_{min} (sec)	Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
					Fwd (*)		Bwd (*)	
					Prefetch	Emg zone	Prefetch	Emg zone
1	2 534	158.4	171.5	176.8	541	0	496	0
2	1 270	79.9	89.6	88.7	274	0	250	0
3	846	57.9	64.9	262.1	190	3	169	422 497
4	635	41.3	47.2	91.6	138	0	127	0
6	423	31.5	38.0	186.7	102	6	86	422 498
8	317	21.8	24.9	137.6	70	0	64	321 871
16	159	11.9	13.2	94.4	39	2	32	214 245
24	105	9.0	10.9	48.5	42	5	38	119 792
32	79	8.2	9.1	53.1	25	1	30	116 209

Table 5: Influence of the parallelism on QIMONDA07 using LIFO strategy. Emg=emergency buffer:1 MB; Prefetch buffer:10MB per processor; (*): Max per processor.

In fact, this gap is correlated with the large number of emergency calls during the backward step. Note that, in this example, we have relatively less emergency requests during the forward step than during the backward step. One reason is that the QIMONDA07 matrix has many nodes of relatively small size, so we have a relatively small number of Type 2 tasks that could require the use of the Emg buffer. Another reason, illustrated in the following discussion, is that one can expect the backward step to be more sensitive to scheduling than the forward step. Indeed, at the beginning of the backward step, we have in general a small number of root nodes, mapped onto few processors. The other processors have no work and are waiting. During the backward step, the end of one task results in the activation of multiple other tasks on other processors. 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.

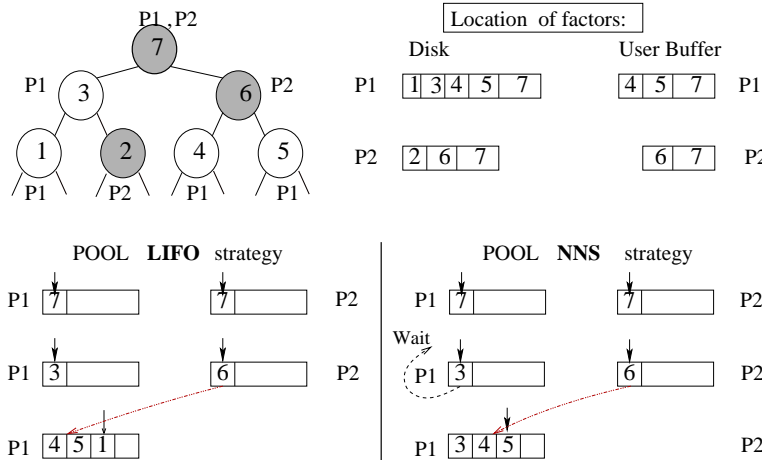


Figure 8: Comparison of LIFO and NNS extraction from the pool.

We illustrate this on the small example described in Figure 8. For the given assembly tree, mapped onto two processors (P1 and P2), we represent the beginning of the backward step and the data in the prefetch

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 (Type 1 nodes). We will comment on Type 2 nodes in our algorithm later. Some data are pre-loaded in the prefetch buffer on both processors, respecting the backward step direction of needed data. With a LIFO strategy, 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 on processor P2 (that would add nodes 4 and 5 to the pool of processor P1), then accessing the factors of node 1 will lead to another emergency call.

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 at almost 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 5 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 focus in this section on the backward phase.

One way to limit the number of disk accesses is to follow strictly the write sequence of the factorization step. By doing so we will always get the node at the top of the memory. We hope that this new algorithm will free more contiguous space in the prefetch buffer, so that less emergency calls will be needed. We define the **Next Node in the Sequence (NNS)** to be the next node to be processed with respect to the write sequence on the disk (dynamically decided during factorization). During the forward step it will be the next non-processed master-node whereas during the backward step it will correspond to the previous non-processed master-node. 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 new algorithm (fully described in Algorithm 6) thus consists in respecting the sequence order to process nodes on each processor. This so called NNS strategy is illustrated in Figure 8. One can see that with a LIFO strategy node 3 was added to the pool for P1 at the end of the process of the root node 7 mapped on both processes. Node 3 was 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 is the next node in the sequence after node 7. In the following, we first describe the new NNS strategy to extract work from the pool and we prove that we can safely wait for the NNS node.

In our NNS algorithm (Algorithm 6), a new ‘blocking receive’ (at line β) has been introduced with respect to 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 will 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.

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 as in Algorithm 6 does not introduce deadlock.*

Proof: First of all, as explained before, Type 2 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. Type 3 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 $\boxed{P_i}$ ($i \in [0 .. r - 1]$), 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} .*

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

```
Step = Fwd or Bwd
if (Fwd) then
  Initialise POOL with the leaf nodes mapped on Myid
  Initialise NNS to the first leaf node
else
  Initialise POOL with root nodes mapped on Myid
  Initialise NNS to the first root node
end if
while (Not finished) do
  if (POOL is not empty) then
    if a message is available Process_Message(message)    [See Algorithms 2 and 3]
  else
     $\alpha$  Wait for a message and then Process_Message(message)    [See Algorithms 2 and 3]
  end if
  if (POOL is not empty and Process_Message not called) then
    if (NNS in POOL) then
      Inode = NNS ; Update NNS
      if ( Fwd ) Fwd_Process_node(Inode)    [See Algorithm 2]
      if ( Bwd ) Bwd_Process_node(Inode)    [See Algorithm 3]
    else
       $\beta$  Wait for a message and then Process_Message(message)
    end if
  end if
end while
```

Proof: Thanks to the main elimination 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} . \square

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

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

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. $\overset{a}{\rightarrow}$ indicates an ancestor relation, $x \overset{a}{\rightarrow} y$ indicates that x is before y because x is an ancestor of y . (Note that $x \overset{a}{\rightarrow} y$ implies $x \rightarrow y$ and $x \neq y$). We thus have :

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

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. \square

Normally, the next node in the sequence is located at the end of the prefetch 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.

The results, presented in Table 6 show that using the NNS strategy on the QIMONDA07 matrix significantly improves the performance in the backward step on parallel runs. QIMONDA07 has a large number

Nb of Procs	T_min (sec)	Bwd (sec)	Nb_Req ^(*) in Bwd step	
			Prefetch	Emg
			1	158.4
2	79.9	93.7	250	0
3	57.9	65.5	174	1
4	41.3	50.5	117	0
6	31.5	37.9	93	0
8	21.8	45.2	57	0
16	11.9	13.8	36	0
24	9.0	13.2	38	0
32	8.2	10.7	34	0

Table 6: Influence of the scheduling NNS of the tasks on QIMONDA07. Emg=emergency buffer:1 MB; Prefetch buffer:10MB per processor; (*): Max per processor.

of relatively small nodes, with a relatively small number of Type 2 nodes. This explains why in general with our NNS algorithm 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.

6 Performance analysis

In this section the NNS and LIFO schedulings are compared on all our test matrices. One main difference with respect to the QIMONDA07 matrix used in the detailed analysis of the previous sections is that for the other matrices the factor block is on average much larger and thus results in a large number of type 2 nodes split over more than one processor. The parallel behaviour of each matrix is reported on the minimum number of processors required to run the out-of-core factorization phase with one MPI process per node of the CRAY XD1. For each matrix and each run with the same number of processes, the same physical processors are used with LIFO and NNS to guarantee similar experimental conditions. The workspace size for the solution phase is divided between two buffers – *Prefetch* and *Emg* (see Figure 6). The average (*Avg*) and the maximum factor size (*Max*) are included in our tables firstly to show that the factors are well equilibrated among the processors and secondly to compare the maximum factor size with the effective maximum workspace used during the solution phase. Indeed, one main property of the DIRECT_IO strategy is that we explicitly control the size of the working space used. It is thus critical to show that our runs are performed in a limited memory environment. For each test we report the performance (time and number of access to the buffers) obtained during forward (Fwd) and backward (Bwd) substitutions.

We first notice the effect of equation (2) on the size of the prefetch zone. On CONESHL with 1 processor, 709 MB of working space are used for 5.9 GB of factor data (209 MB for the emergency buffer and 500 MB for the prefetching zone). For larger number of processors, the increase in the number of Type 2 nodes leads to a decrease in the size of the factor blocks which results in a decrease in the size of the buffers. In Table 14, however, we see that with the matrix NICE9HZ when the size of the emergency buffer remains relatively large with respect to the maximum factor size then equation (2) limits the size of the prefetch zone to 500 MBytes which is only 1.27 times the size of the emergency buffer. This will limit the capacity of the algorithm to perform prefetching.

Furthermore, for a given matrix, the decrease in the size of the factors often leads to a decrease in the time for both the forward and the backward steps. As observed in the previous section, one can see a correlation between the performance and the number of accesses to the emergency buffer. However, although an access to the emergency buffer will always block the process during the time to load the corresponding block factor from the disk, its effect on the node tasks mapped on other processes will depend on the mapping of the tree to the processes. Therefore one should not expect that the smallest number of emergency calls will result in the best performance (see, for example, Table 8 on 8 processors with strategy NNS : 14 Emg calls and 64.3 sec during forward compared to 2 and 67.0 sec during backward). It is clear, however, that the backward step is more sensitive to the accumulation of those time delays even if with the NNS strategy this is significantly reduced with respect to the LIFO strategy. On all matrices, we see that the NNS strategy is better both during forward and backward steps in limiting such effects by forcing an order compatible with the order used to write the factor blocks during the factorization. For both phases, the NNS strategy also ensures more regular disk access and significantly improves the execution time for all our test matrices.

Strategy	Nb of Procs	Factor Size per proc (MB)		Workspace per proc (MB)		Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
		Avg	Max	Prefetch	Emg			Fwd		Bwd	
								Prefetch	Emg	Prefetch	Emg
LIFO	2	2416	2547	500	59	336.3	270.1	11	0	11	3279
NNS						334.0	269.7	11	0	10	0
LIFO	4	1200	1291	300	34	221.0	356.3	11	10	10	133594
NNS						220.0	190.6	12	1	14	1
LIFO	8	596	756	149	34	165.5	203.3	20	68	10	74582
NNS						117.7	99.9	14	8	10	1
LIFO	16	295	336	74	10	102.7	156.0	25	129	10	37861
NNS						63.9	84.1	21	28	10	4
LIFO	24	196	239	56	24	59.9	103.3	18	79	11	23850
NNS						52.1	74.5	17	17	10	2
LIFO	32	146	170	36	6	47.0	102.3	16	74	13	37055
NNS						44.5	69.8	15	10	10	2

Table 7: Parallelism on CAS4R-L15

Strategy	Nb of Procs	Factor Size per proc (MB)		Workspace per proc (MB)		Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
		Avg	Max	Prefetch	Emg			Fwd		Bwd	
								Prefetch	Emg	Prefetch	Emg
LIFO	1	5908	5908	500	209	375.2	378.3	27	3	26	5
NNS						374.7	378.3	27	3	26	5
LIFO	4	1465	1481	366	77	102.6	139.0	9	6	8	2
NNS						102.4	133.9	9	6	8	1
LIFO	8	726	987	181	52	63.9	95.7	15	14	13	12
NNS						64.3	67.0	13	14	12	2
LIFO	16	360	393	90	12	36.3	64.6	12	17	9	6488
NNS						33.6	48.2	10	10	9	2
LIFO	24	239	285	64	17	36.6	50.2	21	74	20	6161
NNS						31.4	45.8	22	28	18	17
LIFO	32	179	221	44	7	24.8	40.2	19	60	11	4040
NNS						22.9	33.6	20	89	20	21

Table 8: Parallelism on CONESHL

Strategy	Nb of Procs	Factor Size per proc (MB)		Workspace per proc (MB)		Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
		Avg	Max	Prefetch	Emg			Fwd		Bwd	
								Prefetch	Emg	Prefetch	Emg
LIFO	6	1537	1689	384	107	148.7	225.2	10	6	8	5602
NNS						134.5	158.6	10	0	9	0
LIFO	8	1147	1232	286	90	126.9	153.2	10	20	11	4570
NNS						120.7	135.3	14	9	9	2
LIFO	16	564	774	141	26	116.9	116.7	24	205	13	5042
NNS						92.7	80.1	18	42	13	27
LIFO	24	372	457	87	23	76.1	83.9	19	138	20	3248
NNS						68.4	66.7	22	69	27	40
LIFO	32	276	399	69	21	67.2	76.2	37	214	22	2578
NNS						57.5	57.1	40	114	21	607

Table 9: Parallelism on NICE20MC

Strategy	Nb of Procs	Factor Size per proc (MB)		Workspace per proc (MB)		Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
		Avg	Max	Prefetch	Emg			Fwd		Bwd	
								Prefetch	Emg	Prefetch	Emg
LIFO	4	2741	2872	500	176	231.8	355.2	3	1	4	7179
NNS						218.3	233.5	14	0	12	1
LIFO	8	1354	1480	338	216	152.5	215.5	15	45	13	12523
NNS						147.8	166.2	11	23	10	1
LIFO	16	664	955	166	81	144.8	159.0	29	65	16	7314
NNS						118.2	121.0	22	52	20	452
LIFO	24	437	498	102	33	78.4	131.0	25	83	22	6168
NNS						62.7	76.2	23	44	12	2
LIFO	32	325	573	81	20	73.7	101.4	29	86	27	4315
NNS						73.3	80.4	42	151	36	63

Table 10: Parallelism on AUDI

Strategy	Nb of Procs	Factor Size per proc (MB)		Workspace per proc (MB)		Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
		Avg	Max	Prefetch	Emg			Fwd		Bwd	
								Prefetch	Emg	Prefetch	Emg
LIFO	2	7860	7900	500	139	507.0	519.0	37	0	34	0
NNS						506.0	513.6	37	0	34	0
LIFO	4	3919	3951	500	139	273.6	383.3	20	0	17	191695
NNS						273.2	293.0	20	0	17	0
LIFO	8	1948	1994	487	139	174.1	289.9	14	24	9	96156
NNS						144.9	184.9	9	1	8	0
LIFO	16	963	1041	240	139	104.5	207.0	20	27	9	48039
NNS						87.3	125.6	21	22	9	1
LIFO	24	633	723	148	39	156.1	214.0	32	346	15	66477
NNS						79.4	93.9	21	35	11	2
LIFO	32	472	593	118	39	95.6	149.4	39	225	23	60630
NNS						74.2	83.3	53	100	38	39

Table 11: Parallelism on GRID3.5M

Strategy	Nb of Procs	Factor Size per proc (MB)		Workspace per proc (MB)		Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
		Avg	Max	Prefetch	Emg			Fwd		Bwd	
								Prefetch	Emg	Prefetch	Emg
LIFO	8	2702	2970	500	160	397.1	476.5	20	16	17	26981
NNS						298.6	351.3	18	12	18	10
LIFO	12	1793	2154	448	160	249.1	447.8	18	34	15	23368
NNS						230.1	325.3	16	11	13	1
LIFO	16	1340	1584	335	160	261.7	353.7	21	52	14	14278
NNS						220.1	303.8	19	28	23	3675
LIFO	24	886	1096	165	67	207.8	347.6	25	99	24	10868
NNS						195.3	256.1	21	49	23	18
LIFO	32	660	820	165	45	189.9	310.2	30	142	39	9090
NNS						185.9	218.1	22	47	19	10

Table 12: Parallelism on COR5HZ

Strategy	Nb of Procs	Factor Size per proc (MB)		Workspace per proc (MB)		Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
		Avg	Max	Prefetch	Emg			Fwd		Bwd	
								Prefetch	Emg	Prefetch	Emg
LIFO	20	1404	1625	351	74	725.9	964.8	31	114	23	40323
NNS						678.0	866.1	20	70	14	4
LIFO	24	1171	1364	292	74	679.8	1071.6	25	156	27	37950
NNS						475.5	629.5	19	37	16	8
LIFO	32	872	1028	218	43	358.9	814.6	19	37	28	28334
NNS						350.9	564.6	15	42	10	6

Table 13: Parallelism on AMANDE

Strategy	Nb of Procs	Factor Size per proc (MB)		Workspace per proc (MB)		Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
		Avg	Max	Prefetch	Emg			Fwd		Bwd	
								Prefetch	Emg	Prefetch	Emg
LIFO	20	3208	3651	500	393	685.9	1050.2	38	66	30	30735
NNS						651.8	696.7	45	78	28	35
LIFO	24	2661	3048	500	228	642.9	844.7	39	86	30	29098
NNS						571.6	684.8	32	25	32	8765
LIFO	32	1989	2454	497	228	559.8	734.0	52	59	51	21501
NNS						471.9	604.7	45	58	58	9293

Table 14: Parallelism on NICE9HZ

Strategy	Nb of Procs	Factor Size per proc (MB)		Workspace per proc (MB)		Fwd (sec)	Bwd (sec)	Max Nb Requests per step			
		Avg	Max	Prefetch	Emg			Fwd		Bwd	
								Prefetch	Emg	Prefetch	Emg
LIFO	4	4259	4356	500	199	325.4	554.0	19	1	19	413902
NNS						347.9	321.5	19	15	19	1
LIFO	8	2120	2583	500	136	247.8	368.1	15	56	10	138017
NNS						186.7	223.6	13	3	11	4
LIFO	16	1048	1113	262	66	122.7	236.0	33	116	9	71173
NNS						84.5	133.4	9	0	10	2
LIFO	24	695	795	188	49	114.5	189.2	21	264	14	105205
NNS						77.6	116.9	30	68	14	6
LIFO	32	519	567	129	30	62.6	171.6	20	221	9	34909
NNS						45.2	77.1	20	15	9	1

Table 15: Parallelism on GRID5M

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 large and unpredictable memory use.

A DIRECT_IO access to the disk with relatively small prefetch buffers has thus been introduced to control the use of memory. 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 the memory 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 blocks during the solution phase. We have proved the correctness of the algorithm and have shown that we perform consistently better and often significantly reduce the time for solution on a set of large real problems.

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. We want to thank anonymous referees of *Parallel Computing Journal* for their work and suggestion on this paper. They asked us to publish this complete version as a technical report to better focus on a journal shortened version.

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] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.
- [6] 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.
- [7] 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.
- [8] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.
- [9] J. K. Reid and J. A. Scott. An out-of-core sparse Cholesky solver. Technical Report RAL-TR-2006-013, Rutherford Appleton Laboratory, 2006. Revised March 2007.
- [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] V. Rotkin and S. Toledo. The design and implementation of a new out-of-Core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, 30(1):19–46, 2004.

- [12] S. Toledo and A. Uchitel. A supernodal out-of-core sparse gaussian elimination method. In *Proceedings of PPAM 2007*, 2007.