Introduction of shared-memory parallelism in a distributed-memory multifrontal solver

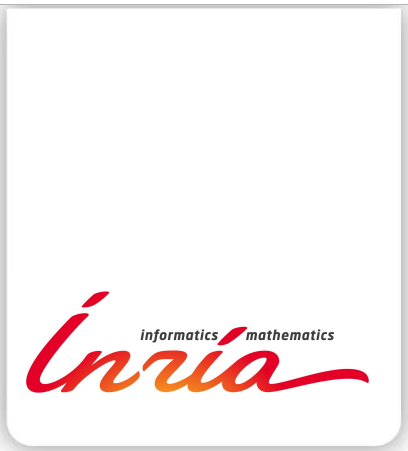
Jean-Yves L'Excellent, Mohamed W. Sid-Lakhdar

Introduction of shared-memory parallelism in a distributed-memory multifrontal solver

Jean-Yves L'Excellent, Mohamed Sid-Lakhdar

RESEARCH REPORT
N° 8227
February 2013
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8227--FR+ENG



Introduction of shared-memory parallelism in a distributed-memory multifrontal solver

Jean-Yves L'Excellent*, Mohamed Sid-Lakhdar†

Project-Team ROMA

Research Report n° 8227 — February 2013 — 35 pages

Abstract: We study the adaptation of a parallel distributed-memory solver towards a shared-memory code, targeting multi-core architectures. The advantage of adapting the code over a new design is to fully benefit from its numerical kernels, range of functionalities and internal features. Although the studied code is a direct solver for sparse systems of linear equations, the approaches described in this paper are general and could be useful to a wide range of applications. We show how existing parallel algorithms can be adapted to an OpenMP environment while, at the same time, also relying on third-party optimized multithreaded libraries. We propose simple approaches to take advantage of NUMA architectures, and original optimizations to limit thread synchronization costs. For each point, the performance gains are analyzed in detail on test problems from various application areas.

Key-words: shared-memory, multi-core, NUMA, LU factorization, sparse matrix, multifrontal method

* Inria, University of Lyon and LIP laboratory (UMR 5668 CNRS-ENS Lyon-INRIA-UCBL), Lyon, France (jean-yves.l.excellent@ens-lyon.fr)

† ENS Lyon, University of Lyon and LIP laboratory (UMR 5668 CNRS-ENS Lyon-INRIA-UCBL), Lyon, France (mohamed.sid_lakhdar@ens-lyon.fr)

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Adaptation sur machines multicœurs d'un solveur multifrontal à base de passage de messages

Résumé : Dans ce rapport, nous étudions l'adaptation d'un code parallèle à mémoire distribuée en un code visant les architectures à mémoire partagée de type multi-cœurs. L'intérêt d'adapter un code existant plutôt que d'en concevoir un nouveau est de pouvoir bénéficier directement de toute la richesse de ses fonctionnalités numériques ainsi que de ses caractéristiques internes. Même si le code sur lequel porte l'étude est un solveur direct multifrontale pour systèmes linéaires creux, les algorithmes et techniques discutés sont générales et peuvent s'appliquer à des domaines d'application plus généraux. Nous montrons comment des algorithmes parallèles existant peuvent être adaptés à un environnement OpenMP tout en exploitant au mieux des librairies existantes optimisées. Nous présentons des approches simples pour tirer parti des spécificités des architectures NUMA, ainsi que des optimisations originales permettant de limiter les coûts de synchronisation dans le modèle fork-join que l'on utilise. Pour chacun de ces points, les gains en performance sont analysés sur des cas tests provenant de domaines d'applications variés.

Mots-clés : mémoire partagée, multi-cœur, NUMA factorisation LU, méthode multifrontale, matrice creuse

Contents

1 Introduction

Since the arrival of multi-core systems, many efforts must be done to adapt existing software in order to take advantage of these new architectures. Whereas shared-memory machines were common before the mid-90's, since then, message-passing (and in particular `MPI` [13]) has been widely used to address distributed-memory clusters with large numbers of nodes. Although message-passing can be applied inside multi-core processors, the shared-memory paradigm is also a convenient way to program them. Unfortunately, codes that were designed a long time ago for SMP machines often do not exhibit good performance on today's multi-core architectures. The costs of synchronizations, the increasing gap between processor and memory speeds, the NUMA (Non-Uniform Memory Accesses) effects and the increasing complexity of modern cache hierarchies are among the main difficulties encountered.

In this study, we are interested in a parallel direct approach for the solution of sparse systems of linear equations of the form

$$Ax = b, \tag{1}$$

where A is a sparse matrix, b is the right-hand side vector (or matrix) and x is the unknown vector. Several parallel direct solvers, based on matrix factorizations and targeting distributed-memory computers have been developed over the years [18, 27, 19]. In our case, we consider that matrix A is unsymmetric but has a symmetric pattern, at the possible cost of adding explicit zeros when treating matrices with an unsymmetric pattern. We rely on the multifrontal method [15] to decompose A under the form $A = LU$. In this approach, the task graph is a tree called *elimination tree*, which must be processed from the leaves to the root following a topological order (i.e., children must be processed before their parent). At each node of the tree, a partial factorization of a small dense matrix is performed; nodes on distinct subtrees can be processed independently. Because sparse direct solvers using message-passing often have a long development cycle, sometimes with many functionalities and numerical features added over the years, it is not always feasible to redesign them from scratch, even when computer architectures evolve a lot.

The objective of this paper is thus to study how, starting from an existing parallel solver using `MPI`, it is possible to adapt it and improve its performance on multi-core architectures. Although the resulting code is able to run on hybrid-memory architectures, we consider here a pure shared-memory environment. We use the example of the MUMPS solver [3, 5], but the methodology and approaches described in this paper are more general. We study and combine the use (i) of multithreaded libraries (in particular `BLAS`– Basic Linear Algebra Subprograms [25]); (ii) of loop-based fine grain parallelism based on OpenMP [1] directives; and (iii) of coarse grain OpenMP parallelism between independent tasks. On NUMA architectures, we show that the memory allocation policy and resulting memory affinity has a strong impact on performance and should be chosen with care, depending on the set of threads working on each task. Furthermore, when treating independent tasks in a multithreaded environment, if no ready task is available, a thread that has finished its share of the work will wait for the other threads to finish and become idle. In an OpenMP environment, we show how, technically, it is in that case possible to re-assign idle threads to active tasks, increasing dynamically the amount of parallelism exploited and speeding up the corresponding computations.

In relation to this work, we note that multithreaded sparse direct solvers aiming at addressing multi-core machines have been the object of a lot of work [2, 7, 10, 12, 16, 23, 18, 22, 26, 33]. Our approach and contribution in this paper are different in several aspects. First, we start from an existing code, originally designed for distributed-memory architectures, with a wide range of specific functionalities and numerical features; our objective is to show how such a code can be modified without a strong redesign. Second, most solvers use serial `BLAS` libraries and manage all the parallelism themselves; on the contrary, we aim at taking advantage as much as possible of existing multithreaded `BLAS` libraries, that have been the object of a lot of tuning by specialists of dense linear algebra. Notice that in the so called DAG-based approaches [10, 21] (and in the code described in reference [2], much earlier), tree parallelism and node parallelism are not separated: each individual task can be either a node in the tree or a subtask inside a frontal matrix. This is also the case of the distributed-memory approach we start from, where a processor can for example start working on a parent node even when some work remains to be done at the child level [4]. In our case and in order to keep the management of threads simple, we study and push as far as possible the approach consisting in using tree parallelism up to a certain level, and then switching to node parallelism in a radical manner, at the cost of an acceptable synchronization. Some recent evolutions in dense linear algebra tend to use task dispatch engines [10, 21] or runtime systems like Dague [8] or StarPU [6], for example. Such

approaches start to be experimented in the context of sparse direct solvers. In particular, the Pastix [19] solver has an option to use such runtime systems, with the advantage of being able to use not only multi-core systems but also accelerators. However, numerical pivoting issues leading to dynamic task graphs, or specific numerical features not available in dense libraries relying on runtime systems, or application-specific approaches to scheduling, still make it hard to use such runtime systems in all cases. Remark that, even if we were using such runtime systems instead of OpenMP, most of the observations and contributions of this paper would still apply.

The paper is organized as follows. In Section 2, we present the multifrontal approach used and the software we rely on, together with our experimental setup. In Section 3, we assess the limits of the fork-join approach to multithreaded parallelism, when using multithreaded `BLAS` libraries and OpenMP directives. We also compare such an approach to the use of `MPI` and to different combinations of `MPI` processes and threads. After Section 3, we focus on a pure shared-memory environment, where only one `MPI` process is used. In Section 4, we propose and study an algorithm to go further in the parallelization thanks to a better granularity of parallelism in the part of our task graph (the elimination tree) where this is necessary. In Section 5, the case of NUMA architectures is studied, where we show how the memory accesses can dramatically impact performance. In Section 6, because the approach retained involves a synchronization when switching from one type of parallelism to the other, we study how OpenMP allows to reuse idle cores by assigning them to active tasks dynamically. Finally, we conclude by summarizing the lessons learned while adapting an existing code to multi-core architectures, and we give some perspectives to this work.

2 Context of the study

2.1 Multifrontal method and solver

We refer the reader to [15, 29] for an overview of the multifrontal method. In this section, we only provide the algorithmic details of our multifrontal solver that will be necessary in the next sections.

As said in the introduction, the task graph in the multifrontal method is a tree called *elimination tree*. At each node of this tree, a dense matrix called *front* or *frontal matrix* is first assembled, using contributions from its children and entries of the original matrix; some of its variables are then factorized (partial factorization), and the resulting Schur complement formed (updated but not yet factorized block) is copied for future use. The Schur complement is also called *contribution block*, because it will contribute to the assembly of the parent's frontal matrix. At the root node, a full factorization is performed.

The three steps above namely assembly, factorization and stacking, correspond to three computational kernels of our multifrontal solver, which we describe in Algorithms 1, 2, and 3. When assembling contributions in the frontal matrix of a parent (Algorithm 1), indirections are required because contiguous variables in the contribution block of a child are not necessarily contiguous in the parent. The corresponding summation is called an *extend-add* operation. Rows and columns in the parent that have received all their contributions are said to be *fully summed*. The method places them first in the front, as shown in Figure 1, and the corresponding variables can be factorized, as shown in Algorithm 2. In the factorization, the fully summed rows are factorized panel by panel, then the other rows are updated using `BLAS` calls of large size. Finally, the stacking operation consists in saving the Schur complement in a stack area, for future use as a contribution block at the parent level. We note that the memory accesses for contribution blocks follow a stack mechanism as long as nodes are processed using a postorder. Furthermore, in our software environment, a work array is preallocated, in which factors are stored on the

left, contribution blocks are stored on the right, and current frontal matrices are stored right on top of the factors area. This work array is typical in multifrontal codes as it allows for more control and optimizations in the memory management than dynamic allocation. Because frontal matrices are stored by rows, only the non fully summed rows of the L factors must be moved at line 5 of Algorithm 3. At that line, in the case of an out-of-core setting where factors have been written to disk panel by panel (or in the case they are not needed), factors are simply discarded.

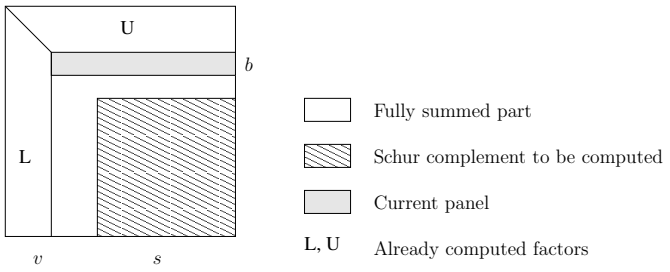


Figure 1: Structure of a frontal matrix.

Algorithm 1 Sketch of the assembly of a set of children into a parent node $\mathcal{N}$.

1: **1. Build row and column structures of frontal matrix associated to node $\mathcal{N}$:**
2: Merge lists of variables from children and from original matrix entries to be assembled in $\mathcal{N}$
3: Build indirections (overwriting index list IND_c of child c with the relative positions in the parent)
4: **2. Numerical assembly:**
5: **for all** children c of node $\mathcal{N}$ **do**
6: **for all** contribution rows i of child c **do**
7: **for all** contribution columns j of child c **do**
8: Assemble entry (i,j) at position $(IND_c(i), IND_c(j))$ of parent (*extend-add* operation)
9: **end for**
10: **end for**
11: Assemble entries from original matrix in fully summed rows and columns of $\mathcal{N}$
12: **end for**

There are typically two sources of parallelism in multifrontal methods. From a coarse-grain point of view, elimination trees are DAGs that define dependencies between their fronts. The structure of tree then offers an inner parallelism, which consists in factorizing different independent fronts at the same time. This is *tree parallelism*. From a fine-grain point of view, the partial factorization of a frontal matrix at a given node of the elimination tree (see Algorithm 2) can also be parallelized: this is called *node parallelism*. In a distributed-memory environment, the `MUMPS` solver we use to illustrate this study implements these two types of parallelism, which were called type 1 and type 2 parallelism [4], respectively. Tree parallelism decreases near the root, where node parallelism generally increases because frontal matrices tend to be bigger.

2.2 Experimental environment

The set of test problems used in our experiments is given in Table 1. Although some matrices are symmetric, we only consider the unsymmetric version of the solver. We use a nested dissection ordering (in our case, METIS [24]) to reorder the matrices. By default, we use double precision

Algorithm 2 `BLAS` calls during partial dense factorization of a frontal matrix F of order $v+s$ with v variables to eliminate and a Schur of order s. b is the block size for panels. We assume that all pivots are eliminated.

1: **for all** horizontal panels $P = F(k : k+b-1, k : v+s)$ in fully summed block **do**
2: `BLAS 2` factorization of the panel:
3: **while** A stable pivot can be found in columns $k : v$ of P **do**
4: Perform the associated row and/or column exchanges
5: Scale pivot column in panel (_SCAL)
6: Update panel (_GER)
7: **end while**
8: Update fully summed column block $F(k+b : v, k : k+b-1)$ (_TRSM)
9: Right-looking update of remaining fully summed part $F(k+b : v, k+b : v+s)$ (_GEMM)
10: **end for**
11: % All fully summed rows have been factorized
12: Update $F(v+1 : v+s, 1 : v)$ (_TRSM)
13: Update Schur complement $F(v+1 : v+s, v+1 : v+s)$ (_GEMM)

Algorithm 3 Stacking operation for a frontal matrix F of order $v+s$. Frontal matrices are stored by rows.

1: Reserve space in stack area
2: **for** $i = v+1$ **to** $v+s$ **do**
3: Copy $F(i, v+1 : v+s)$ to stack area
4: **end for**
5: Make L factors $F(v+1 : v+s, 1 : v)$ contiguous in memory (or free them)

arithmetic, real or complex. The horizontal lines in the table define five areas; the first one (at the top) corresponds to matrices for which there are very large fronts (e.g. 3D problems). The third one corresponds to matrices with many small fronts, including sometimes near the root (e.g. circuit simulation matrices). The second one corresponds to matrices intermediate between those two extremes. Finally, the fourth (resp. fifth) zone corresponds to 3D (resp. 2D) geophysics applications.

Matrix	Symmetry	Arithmetic	N	NZ	Application field
3Dspectralwave	Sym.	real	680943	30290827	Materials
AUDI	Sym.	real	943695	77651847	Structural
conv3D64 (*)	Uns.	real	836550	12548250	Fluid
Serena (*)	Sym.	real	1391349	64131971	Structural
sparsine	Sym.	real	50000	1548988	Structural
ultrasound	Uns.	real	531441	33076161	Magneto-Hydro-Dynamics
dielFilterV3real	Sym.	real	1102824	89306020	Electromagnetism
Haltere	Sym.	complex	1288825	10476775	Electromagnetism
ecl32	Uns.	real	51993	380415	Semiconductor device
G3_circuit	Sym.	real	1585478	7660826	Circuit simulation
QIMONDA07	Uns.	real	8613291	66900289	Circuit simulation
GeoAzur_3D_32_32_32	Uns.	complex	110592	2863288	Geo-Physics
GeoAzur_3D_48_48_48	Uns.	complex	262144	6859000	Geo-Physics
GeoAzur_3D_64_64_64	Uns.	complex	512000	13481272	Geo-Physics
GeoAzur_2D_512_512	Uns.	complex	278784	2502724	Geo-Physics
GeoAzur_2D_1024_1024	Uns.	complex	1081600	9721924	Geo-Physics
GeoAzur_2D_2048_2048	Uns.	complex	4260096	38316100	Geo-Physics

Table 1: Set of test problems. N is the order of the matrix and NZ its number of nonzero elements. The matrices come from Tim Davis' collection (University of Florida), from the GridTLSE collection (University of Toulouse) and from geophysics applications [30, 34]. "Sym." is for symmetric matrices and "Uns." for unsymmetric matrices. For symmetric matrices, we work on the unsymmetric problem associated, although the value of NZ reported only represents the number of nonzeros in the lower triangle. The largest matrices, indicated by "(*)", will only be used on the largest machine, `dude`.

In our study, we rely on the two multi-core based computers below:

- **hidalgo**:
  - **Processor:** 2 × 4-Core Intel Xeon Processor E5520 2.27 GHz (Nehalem).
  - **Memory:** 16 GigaBytes.
  - **Compiler:** Intel compilers (icc and ifort) version 12.0.4 20110427.
  - **BLAS:** Intel(R) Math Kernel Library (`MKL`) version 10.3 update 4.
  - **Location:** ENSEEIHT-IRIT, Toulouse.
- **dude**:
  - **Processor:** 4 × 6-Core AMD Opteron Processor 8431 2.40 GHz (Istanbul).
  - **Memory:** 72 GigaBytes.
  - **Compiler:** Intel compilers (icc and ifort) version 12.0.4 20110427.

– **BLAS:** Intel(R) Math Kernel Library (`MKL`) version 10.3 update 4.

In one of the experiments, we also use a symmetric multiprocessor without NUMA effects from IDRIS[1]:

- **vargas**:
  - **Processor:** 32-Core IBM Power6 Processor 4.70 GHz.
  - **Memory:** 128 GigaBytes.
  - **Compiler:** xlf version 13.1 and xlc version 11.1.
  - **BLAS:** ESSL version 3.3.

3 Multithreaded node parallelism

In this section, we describe sources of multithreaded parallelism inside each computational task: multithreaded libraries and insertion of OpenMP directives. The combination of this type of shared-memory parallelism with distributed-memory parallelism is also discussed in Section 3.3.

3.1 Use of multithreaded libraries

The largest part of the multifrontal factorization time is spent in dense linear algebra kernels, namely, the `BLAS` library. Therefore, reducing the corresponding time is the first source of improvement. A straightforward way to do this consists in using existing optimized multithreaded `BLAS` libraries. This is completely transparent to the application in the sense that only the link phase is concerned, requiring no change to the algorithms, nor to the code.

In our case, the largest amount of computation is spent in the `GEMM` and `TRSM` calls from Algorithm 2. Because the update on F_{21} and F_{22} are done only after the fully-summed block is factorized (lines 12 and 13), the corresponding `TRSM` and `GEMM` operations operate on very large matrices, on which multithreaded `BLAS` libraries have freedom to organize the computations using an efficient parallelization.

Several optimized `BLAS` libraries exist, for example ATLAS [35], `OpenBLAS`, `MKL` (from Intel), `ACML` (from AMD), or `ESSL` (from IBM). As said in Section 2.2, we use `MKL`. One difficulty with Atlas is that the number of threads has to be defined at compile-time, and one difficulty we had with `OpenBLAS` (formerly `GotoBLAS`) was its interaction with OpenMP regions [11]. With the `MKL` version used, the `MKL_DYNAMIC` setting (similar to `OMP_DYNAMIC`) is activated by default, so that providing too many threads on small matrices does not result in speed-downs: extra threads are not used. This was not the case with some earlier versions of `MKL`, where it was necessary to manually set the number of threads to 1 (thanks to the OpenMP routine *omp_set_num_threads*) for fronts too small to benefit from threaded `BLAS`. Unless stated otherwise, the experiments reported are with `MKL_DYNAMIC` set to its default (true) value.

3.2 Directives-based loop parallelism

Loops can easily be parallelized in assembly and stack operations, as was initiated in [11]. Pivot search operations can also be multithreaded. The main difficulties encountered consisted in choosing, for each loop, the minimum granularity above which it was worth parallelizing it.

Concerning the assembly operations (Algorithm 1), the simplest way of parallelizing them consists in using a parallel `OpenMP` loop at line 6 of the algorithm. This way, all rows of a given

[1] Institut du développement et des ressources en informatique.

child are assembled in parallel. We observed experimentally that such a parallelization is only worth doing when the order of the contribution block to be assembled from the child is bigger than 300. Another approach to parallelize Algorithm 1, that would lead to a slightly larger granularity, would consist in splitting the rows of the parent node in a number of zones that would be assembled in parallel, where each zone sees all needed contribution rows assembled in it (from all children). Such an approach is used in the distributed version of our solver. In case of multithreading, it has not been experimented, although it might be interesting for wide trees in cases where the assembly costs may be large enough to compensate for the additional associated symbolic costs.

Stack operations (Algorithm 3), which are basically memory copy operations, can also be parallelized by using a parallel loop at line 2 of the algorithm. Again, a minimum granularity has to be ensured in order to avoid speed-downs on small fronts.

We use the default scheduling policy for `OpenMP`, which, in our environments, consists in using static chunks of maximum size. The larger the frontal matrices, the larger the gains obtained on assembly and stack operations.

3.3 Experiments on a multi-core architecture

In Table 2, we report the effects of using threaded `BLAS` and `OpenMP` directives on the factorization time on 8 cores of `hidalgo`; we also compare these results with an `MPI` parallelization using `MUMPS` 4.10.0, with different combinations of threads per process for a total of 8 cores. In case of multiple threads per `MPI` process, threaded `BLAS` and `OpenMP` directives are used within each `MPI` process, in such a way that the total number of threads is 8.

On the first set of matrices (3Dspectralwave, AUDI, sparsine, ultrasound80), the ratio of large fronts over small fronts in the associated elimination tree is high. Hence, the more threads per `MPI` process, the best the performance, because node parallelism and the underlying multithreaded `BLAS` routines can reach their full potential on many fronts. On the second set of matrices, the ratio of large fronts over small fronts is medium; the best computation times are generally reached when mixing tree parallelism at the `MPI` level with node parallelism at the `BLAS` level. On the third set of matrices, where the ratio of large fronts over small fronts is very small (most fronts are small), using only one core per `MPI` process is often the best solution: tree parallelism is critical whereas node parallelism does not bring any gain. This is because parallel `BLAS` is not efficient on small fronts where there is not enough work for all the threads. On the Geoazur series of matrices, we also observe that tree parallelism is more critical on 2D problems than on 3D problems: on 2D problems, the best results are obtained with more `MPI` processes (and less threads per `MPI` process).

We observe that `OpenMP` directives improve in general the amount of node parallelism (compare columns "Threaded `BLAS`" and "Threaded `BLAS` + `OpenMP` directives" in the "1 `MPI` × 8 threads" configuration), but that the gains are limited. With the increasing number of cores per machine, this approach is only scalable when most of the work is done in very large fronts (e.g., on very large 3D problems). Otherwise, tree parallelism is necessary. The fact that message-passing in our solver was primarily designed to tackle parallelism between computer nodes rather than inside multi-core processors, and the availability of high performance multithreaded `BLAS` libraries lead us to think that both node and tree parallelism should be exploited at the shared-memory level. The path we follow in the next section thus consists in introducing multithreaded tree parallelism.

Matrix	Sequential 1 MPI × 1 thread	threaded BLAS only 1 MPI × 8 threads	threaded BLAS + OpenMP directives 1 MPI × 8 threads	2 MPI × 4 threads	4 MPI × 2 threads	Pure MPI 8 MPI × 1 thread
3Dspectralwave	2061.95	372.83	**371.87**	392.98	387.57	N/A
AUDI	1270.20	251.14	**249.21**	250.87	300.43	315.85
sparsine	314.58	62.52	**61.87**	82.01	80.22	94.42
ultrasound80	441.84	89.05	**89.16**	95.67	124.07	124.10
dielFilterV3real	271.96	60.69	59.31	52.13	**47.85**	61.92
Haltere	691.72	121.29	120.81	**115.18**	140.34	145.55
ecl32	3.00	1.13	1.05	**0.93**	0.98	0.94
G3_circuit	16.99	8.84	8.73	6.24	4.21	**3.61**
QIMONDA07	25.78	27.42	28.49	18.21	9.63	**5.54**
GeoAzur_3D_32_32_32	75.74	16.09	**15.84**	16.28	18.68	19.62
GeoAzur_3D_48_48_48	410.78	73.90	72.96	**69.71**	95.02	106.86
GeoAzur_3D_64_64_64	1563.01	254.47	**254.38**	276.98	303.15	360.96
GeoAzur_2D_512_512	4.48	2.30	2.33	1.46	**1.40**	1.56
GeoAzur_2D_1024_1024	30.97	11.54	11.65	8.38	6.53	**6.21**
GeoAzur_2D_2048_2048	227.08	64.41	64.27	49.97	**43.33**	43.44

Table 2: Factorization times (seconds) on `hidalgo`, with different core-process configurations. Times are in seconds. N/A: the factorization ran out of memory. For each matrix, the best time obtained appears in bold.

4 Introduction of multithreaded tree parallelism

We now want to overcome the limitations of the previous approach, where we have observed limited gains from node parallelism on small frontal matrices. Even when there are large frontal matrices near the top of the tree, node parallelism may be insufficient in the bottom of the tree, where tree parallelism could be exploited instead. The objective of this section is thus to introduce tree parallelism at the threads level, allowing different frontal matrices to be treated by different threads. Many algorithms exist to exploit tree parallelism in sparse direct methods. Among them, the *proportional mapping* [32] and the *Geist-Ng algorithm* [17] have been widely used. They have for example inspired the mapping algorithms of `MUMPS` in distributed-memory environments. The Geist-Ng algorithm had been originally designed for shared-memory multiprocessor environments for parallel sparse Cholesky factorizations. Then, it has been adapted to distributed-memory environments and to other kinds of sparse factorizations. Its goal is to *"achieve load balancing and a high degree of concurrency among the processors while reducing the amount of processor-to-processor data communication"*[17]. For this, tree parallelism is exploited. Since the time of the development of this algorithm, multiprocessor architectures have evolved a lot, containing more and more cores, and new caches have appeared with increasing structure complexity. Therefore, we base our work on the Geist-Ng algorithm, trying to adapt it accordingly.

Sections 4.1 and 4.2 present two variants of the Geist-Ng algorithm, aiming at determining a layer in the tree under which only tree parallelism is used. This layer will be referred to as $\mathcal{L}_{th}$. In Section 4.3, we then explain how our existing multifrontal solver can be modified to take advantage of tree parallelism, without a deep redesign. We finally show in Section 4.4 some experimental results obtained with our variants of the Geist-Ng algorithm ($\mathcal{L}_{th}$ based algorithms) to exploit tree parallelism.

4.1 Balancing work among threads (AlgFlops algorithm)

The AlgFlops algorithm is essentially based on the Geist-Ng algorithm, whose main idea is *"Given an arbitrary tree and P processors, to find the smallest set of branches in the tree such that this set can be partitioned into exactly P subsets, all of which require approximately the same amount of work ..."*[17].

Geist and Ng proceed in two distinct steps: (a) find a layer separating the bottom from the top of the tree (which we call $\mathcal{L}_{th}$); and (b) factorizing the elimination tree. The goal of the layer is to identify a set of branches in the bottom of the tree that can be mapped onto the processors with an acceptable load balance. The remaining nodes, above $\mathcal{L}_{th}$, are then assigned to all the processors in a round-robin manner.

In order to find $\mathcal{L}_{th}$, the algorithm starts by defining it as the root of the tree. (In case of a forest, the initial layer contains the roots of all forest's trees.) As shown in Algorithm 4 and Figure 2, the iterative procedure replaces the largest subtrees by their children until finding a satisfactory layer. More precisely, the first phase consists in identifying the heaviest subtree and substituting its root with the roots of its children. Because of the dependencies between a parent node and its children, expressed by the elimination tree, the algorithm must respect the property that if a node is on $\mathcal{L}_{th}$, none of its ancestors can belong to this layer. The second phase consists in mapping the independent subtrees from the $\mathcal{L}_{th}$ layer on the processors. The third phase consists in checking whether the current layer respects a certain acceptance criterion, based on load balance. If this criterion is met, the algorithm then terminates. Algorithm 4 depends on the following points:

- **Subtree cost:** The cost of a subtree is here defined as the sum of the floating-point operations needed to work on the fronts that constitute it.
- **Subtree mapping:** The problem of mapping subtrees over processors is known as the multiprocessor scheduling problem. It is an NP-complete optimization problem which can be efficiently solved by the **LPT** (**L**ongest **P**rocessing **T**ime first) algorithm. The maximal runtime ratio between LPT and the optimal algorithm has been proved to be 4/3 - 1/(3p), where p is the number of processors [14].
- **Acceptance criterion:** The acceptance criterion is a user-defined tolerance corresponding to the minimal load balance of the processors under $\mathcal{L}_{th}$ (when the subtrees are mapped on the processors with LPT).

Algorithm 4 Geist-Ng analysis step: finding a satisfactory layer $\mathcal{L}_{th}$.

$\mathcal{L}_{th} \leftarrow$ roots of the elimination tree
repeat
  Find the node $\mathcal{N}$ in $\mathcal{L}_{th}$, whose subtree has the highest estimated cost {**Subtree cost**}
  $\mathcal{L}_{th} \leftarrow \mathcal{L}_{th} \cup \{children\ of\ \mathcal{N}\} \setminus \{\mathcal{N}\}$
  Map $\mathcal{L}_{th}$ subtrees onto the processors {**Subtree mapping**}
  Estimate load balance: $\frac{load(least\text{-}loaded\ processor)}{load(most\text{-}loaded\ processor)}$
until load balance > threshold {**Acceptance criterion**}

Concerning the numerical factorization, each thread picks the heaviest untreated subtree under $\mathcal{L}_{th}$ and factorizes it. If no more subtrees remain, the thread goes idle and waits for the others to finish. Then, whereas the Geist-Ng algorithm only used tree parallelism, our proposed AlgFlops algorithm uses tree parallelism under $\mathcal{L}_{th}$ but also node parallelism above it, for the

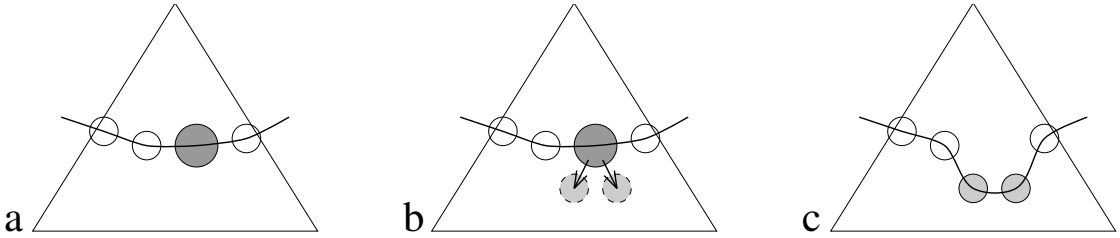


Figure 2: One step in the construction of the layer $\mathcal{L}_{th}$.

following reasons: (i) there are fewer nodes near the root of a tree, and more nodes near the leaves; (ii) fronts near the root often tend to be large, whereas fronts near the leaves tend to be small; (iii) this approach matches the pros and cons of each kind of parallelism observed in Section 3.3.

This approach still has a few limitations. First, the acceptance criterion threshold may need to be tuned manually, and may depend on the test problem and target computer. We observed that a threshold of 90% is an adequate default parameter for most problems, especially when reordering is performed with nested dissection-based techniques such as Metis [24] or Scotch [31]. However, with some matrices, it is not always possible to reach a too high threshold. In such cases, unless another arbitrary stopping criterion is used, the algorithm will show a poor performance as it may not stop until the $\mathcal{L}_{th}$ layer contains all the leaves of the elimination tree. Not only the determination of $\mathcal{L}_{th}$ will be costly but also the resulting $\mathcal{L}_{th}$ layer could be unadapted. Second, the 90% criterion is based on a flops metric for the subtrees and we often observe in practice a balance worse than 90% in terms of runtime under $\mathcal{L}_{th}$. This is due to the fact that the number of floating-point operations is not an accurate measure of the runtime on modern architectures with complex memory hierarchies: typically, the Gflops/s rate will be much bigger for large frontal matrices than for small ones. This limitation is amplified on unbalanced trees because the ratio of large vs. small frontal matrices may then be unbalanced over the subtrees. Third, and more fundamentally, a good load balance under $\mathcal{L}_{th}$ may not necessarily lead to an optimal total run time (sum of the run times under and above $\mathcal{L}_{th}$).

4.2 Minimizing the global time (AlgTime algorithm)

To overcome the aforementioned limitations, we propose a modification of the AlgFlops algorithm, which we refer to as AlgTime. This new version modifies the computation of $\mathcal{L}_{th}$, while the factorization step remains unchanged, with tree parallelism under$\mathcal{L}_{th}$ and node parallelism above $\mathcal{L}_{th}$.

One main characteristic of the AlgTime algorithm is that, instead of considering the number of floating-point operations, it focuses on the total runtime. Furthermore, the goal is not to achieve a good load balance but rather to minimize the total factorization time. It relies for that on a performance model of the mono-threaded (under $\mathcal{L}_{th}$) and multi-threaded (above $\mathcal{L}_{th}$) processing times, estimated on dense frontal matrices, as will be explained below (see Section 4.2.1). Thanks to this model, it becomes possible to get an estimate of the cost associated to a node both under and above $\mathcal{L}_{th}$, where mono-threaded and multi-threaded dense kernels are, respectively, applied.

The AlgTime algorithm (see Algorithm 5) computes layer $\mathcal{L}_{th}$ using the same main loop as in the Geist-Ng algorithm. However, at each step of the loop, it keeps track of the total factorization time induced by the current $\mathcal{L}_{th}$ layer as the sum of the estimated time that will

be spent under and above it. Hence, as long as the estimated total time decreases, we consider that the acceptance criterion has not been reached yet. Simulations showed, however, that this method can fail reaching the global minimum because the algorithm may be trapped in a local minimum.

In order to get out of local minima, the algorithm proceeds as follows. Once a (possibly local) minimum is found, the current $\mathcal{L}_{th}$ is temporarily saved as the optimal $\mathcal{L}_{th}$ found so far but the algorithm continues for a few extra iterations. The maximum number of additional iterations (called *extra_iterations*) is a user-defined parameter. The algorithm stops if no further decrease in time is observed within the authorized extra iterations. Otherwise, the algorithm continues after reseting the counter of additional iterations each time a layer better than all previous ones is reached. We observed that a value of 100 extra iterations is largely enough to reach the global minimum on all problems tested, without inducing any significant extra cost. By nature, this algorithm is meant to be robust against any shape of elimination tree, and is meant to be robust among $\mathcal{L}_{th}$-like algorithms. In particular, on unbalanced trees with large leaves, the algorithm is allowed to choose an $\mathcal{L}_{th}$ below some of the leaves (or below all leaves, in which case $\mathcal{L}_{th}$ is empty and the algorithm stops).

Algorithm 5 AlgTime algorithm.

$\mathcal{L}_{th} \leftarrow$ roots of the assembly tree
$\mathcal{L}_{th}\_best \leftarrow \mathcal{L}_{th}$
$best\_total\_time \leftarrow \infty$
$new\_total\_time \leftarrow \infty$
$cpt \leftarrow extra\_iterations$
repeat
  Find the node $\mathcal{N}$ in $\mathcal{L}_{th}$, whose subtree has the highest estimated serial time
  $\mathcal{L}_{th} \leftarrow \mathcal{L}_{th} \cup \{children\ of\ \mathcal{N}\} \setminus \{\mathcal{N}\}$
  Map $\mathcal{L}_{th}$ subtrees onto the processors
  Simulate $time\_under\_\mathcal{L}_{th}$
  $time\_above\_\mathcal{L}_{th} \leftarrow time\_above\_\mathcal{L}_{th} + cost(\mathcal{N}, nb_{threads})$
  $new\_total\_time \leftarrow time\_under\_\mathcal{L}_{th} + time\_above\_\mathcal{L}_{th}$
  if $new\_total\_time < best\_total\_time$ **then**
    $\mathcal{L}_{th}\_best \leftarrow \mathcal{L}_{th}$
    $best\_total\_time \leftarrow new\_total\_time$
    $cpt \leftarrow extra\_iterations$
  else
    $cpt \leftarrow cpt - 1$
  end if
until $cpt = 0$ or $\mathcal{L}_{th}$ is empty
$\mathcal{L}_{th} \leftarrow \mathcal{L}_{th}\_best$

4.2.1 Performance model

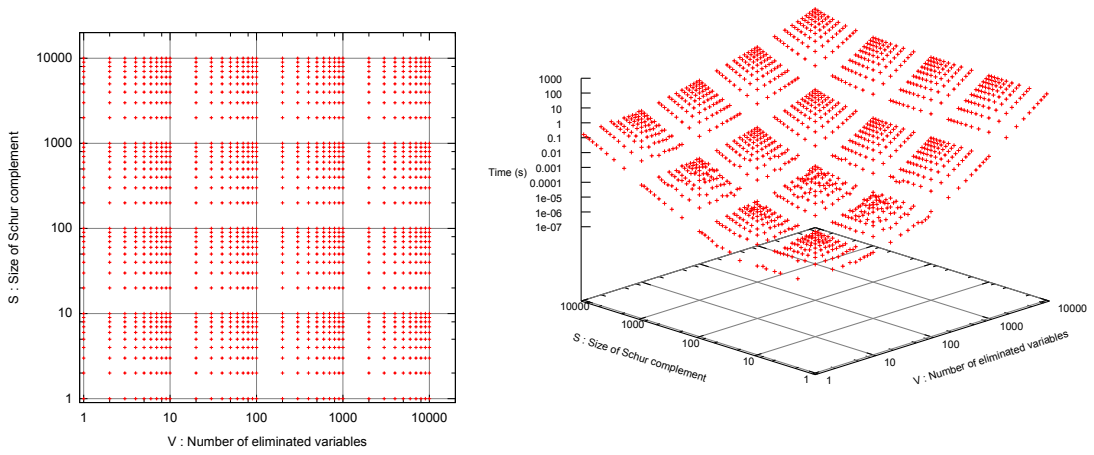
Let α be the GFlops/s rate of the dense factorization kernel described in Algorithm 2, which is responsible of the largest part of the execution time of our solver. α depends on the number v of eliminated variables and on the size s of the computed Schur complement. We may think of modeling the performance of dense factorization kernels by representing α under the form of a simple analytical formula parametrized experimentally. However, due to the great unpredictability of both hardware and software, it is difficult to find an accurate-enough formula. For this

reason, we have run a benchmarking campaign on dense factorization kernels on a large sample of well-chosen dense matrices for different numbers of cores. Then, using interpolation, we have obtained an empirical grid model of performance. Note that an approach based on performance models of `BLAS` routines has already been used in the context of the sparse solver `PaStiX` [20].

Given a two-dimensional field associated to v and s, we define a grid whose intersections represent the samples of the dense factorization kernel's performance benchmark. This grid must not be uniform. Indeed, α tends to vary greatly for small values of v and s, and tends to have a constant asymptotic behavior for large values. This is directly linked to the `BLAS` effects. Consequently, many samples must be chosen on the region with small v and s, whereas less and less samples are needed for large values of these variables. An exponential grid might be appropriate. However, not enough samples would be kept for large values of v and s. That is why we have adopted the following linear-exponential grid, melting linear samples on some regions, whose step grows exponentially between the regions:

$$\begin{cases} \text{v or s} \in [1, 10] & \text{step} = 1 \\ \text{v or s} \in [10, 100] & \text{step} = 10 \\ \text{v or s} \in [100, 1000] & \text{step} = 100 \\ \text{v or s} \in [1000, 10000] & \text{step} = 1000 \end{cases}$$

Figure 4.2.1 (a) shows this grid in log-scale and Figure 4.2.1 (b) shows the benchmark on one core on `hidalgo`. In order to give an idea of the performance of the dense factorization kernels



(a) Grid (b) Benchmark

Figure 3: Grid and benchmark on one core of `hidalgo`.

based on Algorithm 2, the GFlop/s rate of the partial factorization of a 4000×4000 matrix, with 1000 eliminated pivots, is $9.42 GFlops/s$ on one core, and is $56.00 GFlops/s$ on eight cores (a speed-up of 5.95). We note that working on the optimization of dense kernels is outside the scope of this paper.

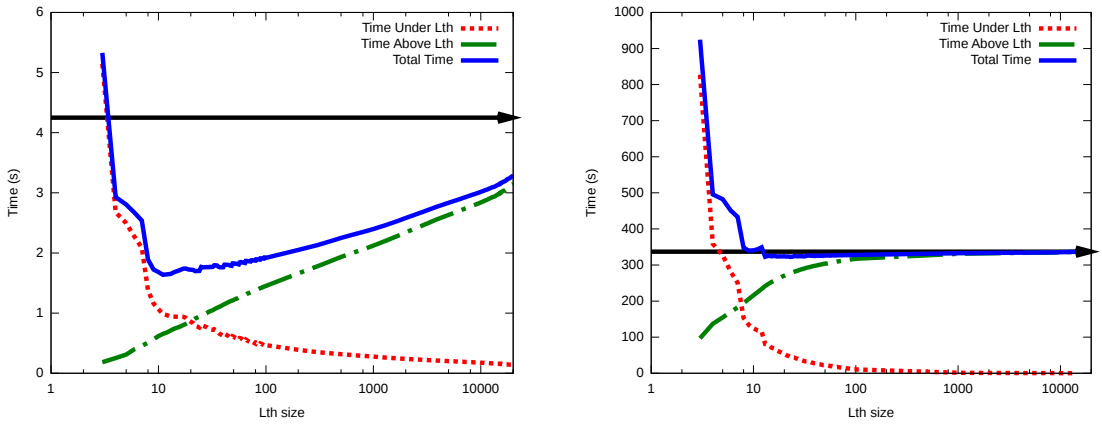
Once the benchmark is completed at each grid point, we can later estimate the performance

for any arbitrary desired point (v,s) using the following simple bilinear interpolation:

$$\begin{aligned}\alpha(v,s,NbCore) \approx \alpha(v_1,s_1,NbCore)\frac{(v_2-v_1)(s_2-s_1)}{(v_2-v)(s_2-s)}\\ +\alpha(v_2,s_1,NbCore)\frac{(v_2-v_1)(s_2-s_1)}{(v-v_1)(s_2-s)}\\ +\alpha(v_1,s_2,NbCore)\frac{(v_2-v_1)(s_2-s_1)}{(v_2-v)(s-s_1)}\\ +\alpha(v_2,s_2,NbCore)\frac{(v_2-v_1)(s_2-s_1)}{(v-v_1)(s-s_1)}.\end{aligned}$$

where (v_1,s_1) and (v_2,s_2) define the limits of the rectangle surrounding the desired value of (v,s). In cases where $(v,\ s)$ is outside the limits of the benchmark grid, the corresponding performance is chosen by default to be that of the limit of the grid.

4.2.2 Simulation



(a) 2D (11PTS_1000_1000) (b) 3D (11PTS_100_100_100)

Figure 4: 2D vs 3D: Simulated time as a function of the number of nodes in the $\mathcal{L}_{th}$ layer for two matrices of order 1 million for the AlgTime algorithm.

Before an actual implementation, we made some simulations in order to predict the effectiveness of our approach. The simulator was written in the *Python* programming language and relies on the performance model described above. Figure 4 shows results obtained for two different matrices generated from a finite-difference discretization. The 2D matrix uses a 9-point stencil on a square and the 3D matrix uses an 11-point stencil on a cube. We have chosen these two matrices because they represent typical cases of regular problems, with very different characteristics. The elimination tree related to the 2D matrix contains many small nodes at its bottom while nodes at the top are not very large comparatively. On the other hand, the elimination tree related to the 3D matrix contains nodes with a rapidly increasing size from bottom to top. Simulations consisted in estimating the time spent under and above $\mathcal{L}_{th}$, as well as the total factorization time, for all layers possibly reached by the algorithm (until the leaves).

The X-axis corresponds to the number of nodes contained in the different layers and the Y-axis to the estimated factorization time. The horizontal solid line represents the estimated

time that would be spent in using fine-grain node parallelism only (Section 3). The dotted (resp. solid-dotted) curve corresponds to the time spent under (resp. above) the $\mathcal{L}_{th}$ layers. As expected, the dotted curve decreases and the solid-dotted one increases with the numbers of nodes in the layers. The solid curve giving the total time (sum of the dotted and solid-dotted curves) seems to have a unique global minimum. We have run several simulations on several matrices and this behavior has been observed on all test cases.

The best $\mathcal{L}_{th}$ is obtained when the solid curve reaches its minimum. Hence, the difference between the horizontal line and the minimum of the solid curve represents the potential gain provided by the proposed AlgTime algorithm. This gain heavily depends on the kind of matrix, with large 3D problems such as the one from Figure 4 showing the smallest potential for $\mathcal{L}_{th}$-based algorithms exploiting tree parallelism: the smaller the fronts in the matrix, the better the gain we can expect from tree parallelism. This is the reason why there is a gap between the solid curve and the horizontal line at the right-most of the 2D problem in Figure 4, where $\mathcal{L}_{th}$ contains all leaves. This gap represents the gain of using tree parallelism on the leaves of the tree.

4.3 Implementation

Algorithm 6 Factorization phase using the $\mathcal{L}_{th}$-based algorithms.

1: **1. Process nodes under $\mathcal{L}_{th}$ (tree parallelism)**
2: **for all** subtrees $\mathcal{S}$, starting from the most costly ones, in parallel **do**
3: **for all** nodes $\mathcal{N} \in \mathcal{S}$, following a postorder **do**
4: Assemble, factorize and stack the frontal matrix of $\mathcal{N}$, using one thread (Algorithms 1, 2 and 3)
5: **end for**
6: **end for**
7: Wait for the other threads
8: Compute global information (reduction operations)
9: **2. Perform computations above $\mathcal{L}_{th}$ (node parallelism)**
10: **for all** nodes $\mathcal{N}$ above $\mathcal{L}_{th}$ **do**
11: Assemble, factorize and stack the frontal matrix of $\mathcal{N}$, using all threads (Algorithms 1, 2 and 3)
12: **end for**

The factorization algorithm (Algorithm 6) consists of two phases: first, $\mathcal{L}_{th}$ subtrees are processed using tree parallelism; then, the nodes above $\mathcal{L}_{th}$ are processed using node parallelism. At line 2 of the algorithm, each thread dynamically extracts the next most costly subtree. We have also implemented a static variant that follows the tentative mapping from the analysis phase. One important aspect of our approach is that we were able to directly call the computational kernels (assembly, factorization and stacking) and memory management routines of our existing solver. A possible risk is that, because the kernels use `OpenMP` themselves, calling them inside an `OpenMP` parallel region could generate many more threads than the number of cores available. This is not occurring when nested parallelism is disabled (this can be forced by using *omp_set_nested(.false.)*) and the existing kernels are executed sequentially within a nested region. Enabling `OMP_DYNAMIC` by a call *omp_set_dynamic (.true.)* also avoids creating too many threads inside a nested region. Finally, another possibility simply consists in explicitly setting the number of threads to one inside the loop on $\mathcal{L}_{th}$ subtrees.

We now discuss memory management. As said in Section 2.1, in our environment, one very large array is allocated once that will be used as workspace for all frontal matrices, factors and

stack of contribution blocks. Keeping a single work array for all threads (and for the top of the tree) is not a straightforward approach because the existing memory management algorithms do not easily generalize to multiple threads. First, the stack of contribution blocks is no more a stack when working with multiple threads. Second, the threads under $\mathcal{L}_{th}$ would require synchronizations for the reservation of their private fronts or contribution blocks in the work array; due to the very large number of fronts in the elimination tree, these synchronizations would be very costly. Third, smart memory management schemes including in-place assemblies and compression of factors have been developed in order to minimize the memory consumption, that would not generalize if threads work in parallel on the same work array. In order to avoid these difficulties, use the existing memory management routines without modification, and possibly be more cache-friendly, we have decided to create one private workspace for each thread, under $\mathcal{L}_{th}$, and still use the same shared workspace above $\mathcal{L}_{th}$ (although smaller than before since only the top of the tree is concerned). This approach raises a minor issue. Before factorizing a front, contribution blocks of its children must be assembled in it (Algorithm 1). This is not completely straightforward for the parents of the $\mathcal{L}_{th}$ fronts because different threads may handle different children of this parent front. We thus need to keep track of which thread handles which subtree under $\mathcal{L}_{th}$, so that one can locate the contribution blocks in the proper thread-private workspaces. This modification of the assembly algorithm is the only modification that had to be done to the existing kernels implementing Algorithms 1, 2 and 3. Remark that, when all contribution blocks in a local workspace have been consumed, it could be worth decreasing the size of the workspace so that only the memory pages containing the factors remain in memory. Depending on platforms, this can be done with the *realloc* routine.

Finally, local statistics are computed for each thread in private variables (number of operations, largest front size, etc.) and are reduced before switching from tree to node parallelism in the upper part of the tree, where they will also be updated by the main thread.

4.4 Experiments

We can see in Table 3 that $\mathcal{L}_{th}$-based algorithms improve the factorization time of all sparse matrices, in addition to the improvements previously discussed in Section 3. $\mathcal{L}_{th}$-based algorithms applied in a pure shared-memory environment also result in a better performance than when message-passing is used (see the results from Table 2).

We first observe that the gains of the proposed algorithms are very important on matrices whose elimination trees present the same characteristics as those of the 2D matrix presented above, namely: trees with many nodes at the bottom and few medium-sized nodes at the top. Such matrices arise, for example, from 2D finite-element and circuit-simulation problems. In those cases, the gain offered by $\mathcal{L}_{th}$-based algorithms seems independent from the size of the matrices. For the entire 2D GeoAzur set, the total factorization time has been divided by a factor of two. In the case of matrices whose elimination trees present the characteristics similar to those of the 3D case presented above, the proposed $\mathcal{L}_{th}$-based algorithms still manage to offer a gain. However, gains are generally much smaller than those observed in the 2D case. The reason for the difference of effectiveness between 3D-like and 2D-like problems is that, in the 3D case, most of the time is spent above $\mathcal{L}_{th}$ because the fronts in this region are very large, whereas in the 2D case, a significant proportion of the work is spent on small frontal matrices under $\mathcal{L}_{th}$. An extreme case is the QIMONDA07 matrix (from circuit simulation), where fronts are very small in all the regions of the elimination tree, with an overhead in multithreaded executions leading to speed-downs. Thus, AlgTime is extremely effective on such a matrix. On this matrix, the AlgFlops algorithm aiming at balancing the work under $\mathcal{L}_{th}$ was not able to find a good layer.

Matrix	Serial reference	Threaded BLAS + OpenMP	$\mathcal{L}_{th}$-based algorithms	
			AlgFlops	AlgTime
3Dspectralwave	2061.95	371.87	343.64	339.78
AUDI	1270.20	249.21	225.82	210.10
sparsine	314.58	61.87	59.46	57.91
ultrasound80	441.84	89.16	77.06	77.85
dielFilterV3real	271.96	59.31	46.10	44.52
Haltere	691.72	120.81	102.17	99.51
ecl32	3.00	1.05	3.07	0.72
G3_circuit	16.99	8.73	8.82	3.02
QIMONDA07	25.78	28.49	27.34	4.26
GeoAzur_3D_32_32_32	75.74	15.84	13.02	12.87
GeoAzur_3D_48_48_48	410.78	72.96	64.14	62.48
GeoAzur_3D_64_64_64	1563.01	254.38	228.69	228.12
GeoAzur_2D_512_512	4.48	2.33	0.88	0.84
GeoAzur_2D_1024_1024	30.97	11.65	5.37	5.02
GeoAzur_2D_2048_2048	227.08	64.27	35.47	34.56

Table 3: Experimental results with $\mathcal{L}_{th}$-based algorithms on `hidalgo` (8 cores).

As predicted by the simulations of Section 4.2.2, the loss of time due to the synchronization of the threads on $\mathcal{L}_{th}$ before starting the factorization above $\mathcal{L}_{th}$ is largely compensated by the gain of applying mono-threaded factorizations under $\mathcal{L}_{th}$. This is a key aspect. It shows that, in multi-core environments, making threads work on separate tasks is better than making them collaborate on the same tasks, even at the price of a strong synchronization. We will discuss a simple strategy to further reduce the overhead due to such a synchronization in Section 6.

In the case of homogeneous subtrees under $\mathcal{L}_{th}$, the difference in execution time between the AlgFlops and the AlgTime algorithms is small. Still, AlgTime is more efficient and the gap grows with the problem size. We now analyze the behaviour of the AlgFlops vs AlgTime algorithms in more details by making the following three observations.

- AlgTime induces a better load balance of the threads under $\mathcal{L}_{th}$ than does AlgFlops. For the AUDI matrix, the difference between the completion time of the first and last threads under $\mathcal{L}_{th}$, when using AlgFlops, is $90.22 - 72.50 = 17.72$ seconds, whereas the difference when using AlgTime is $94.77 - 82.06 = 12.71$ seconds. This difference is valuable since less time is wasted for a synchronization purpose.

- the $\mathcal{L}_{th}$ layer obtained with the AlgTime algorithm is higher in the elimination tree than that of the AlgFlops algorithm. For the AUDI matrix, the $\mathcal{L}_{th}$ of AlgFlops contains 24 subtrees, whereas that of AlgTime only contains 17 subtrees. This shows that AlgTime naturally detects that the time spent under $\mathcal{L}_{th}$ is more valuable than that spent above, as long as synchronization times remain reasonable.

- AlgFlops offers a gain in the majority of cases; but can yield catastrophic results, especially when elimination trees are unbalanced. One potential problem is that the threshold set in AlgFlops could not be reached, in which case, the algorithm will loop indefinitely until $\mathcal{L}_{th}$ reaches all the leaves of the tree. In order to avoid this behavior, one method is to limit the size of $\mathcal{L}_{th}$ artificially. The problem is that this new parameter is difficult to tune

up and that very unbalanced $\mathcal{L}_{th}$'s could lead to disastrous performance. The AlgTime algorithm is much more robust in such situations and brings important gains. For the ecl32 matrix, for example, the factorization time of AlgFlops is 3.07 seconds whereas that of the algorithm shown in Section 3 was 1.05 seconds. In comparison, AlgTime decreases the factorization time down to 0.72 seconds.

These results show that AlgTime is more efficient and more robust than AlgFlops, and brings significant gains.

5 Memory affinity issues on NUMA architectures

In multi-core environments, two main architectural trends have arisen: SMP and NUMA. SMP (**S**ymmetric **M**ulti**P**rocessor) architectures contain symmetric cores in the sense that the access to any piece of memory costs the same for any given core. On the other hand, NUMA (**N**on-**U**niform **M**emory **A**ccess) architectures contain cores whose memory access cost is different depending on the targeted piece of memory. Such architectures contain many sockets, each composed of a processor containing multiple cores, and local memory banks. Each core in a given socket accesses its local memory preferably, and accesses foreign memory banks with different (worse) costs, depending on the interconnection distance between the processors. Hence, SMP processors are fully connected whereas NUMA processors are partially connected and memory is organized hierarchically. Due to this characteristic, increasing the number of cores in an SMP fashion is of increasing difficulty. Hence, more and more modern architectures are NUMA due to its ease of scalability.

When comparing the estimated simulated time with the effective time spent under $\mathcal{L}_{th}$, it appeared that the estimation was optimistic. There are two possible reasons for that. First, the estimation only comprises the factorization time, whereas the effective time also includes assemblies and stack operations. However, the amount of time spent in assemblies is very small compared to the factorization time and is not enough to explain the discrepancy. Second, we have run the benchmarks on unloaded machines in order to obtain precise results. For instance, when we have run the mono-threaded benchmarks, only one CPU core was working, all the others being idle. However, in $\mathcal{L}_{th}$-based algorithms, all the cores are active under $\mathcal{L}_{th}$ at the same time. In such a case, resource sharing and racing could be the reason for the observed discrepancy under $\mathcal{L}_{th}$.

5.1 Performance of dense factorization kernels and NUMA effects

In order to understand the effects of SMP and NUMA architectures on sparse factorizations, we must first understand them on dense factorizations. We first study the effects of the load of the cores on the performance, and then discuss the effects of the memory affinity and allocation policies.

5.1.1 Machine load

The machine load is an effect we did not take into account until now. It depends on many variable parameters that are out of our control, making it difficult to model precisely. However, in the case of the multifrontal method, if we consider the stage when threads work under the $\mathcal{L}_{th}$ layer, we could consider that these threads are mainly in a factorization phase since, for each dense matrix, the time of factorization dominates that of assembly and stack operations.

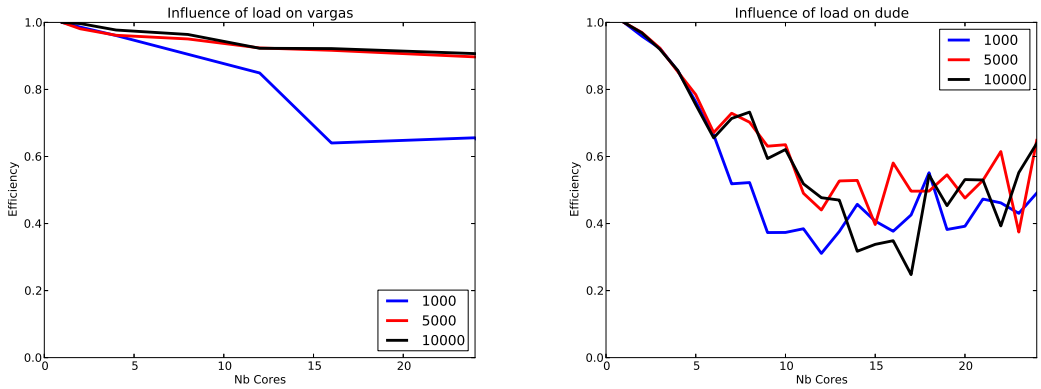
In order to study the effects of the machine load, we made an experiment on *concurrent mono-threaded* factorizations, in which different threads factorize different matrices with the same characteristics. Hence, we could vary the number and mapping of the threads as the size of the matrices for observing their effects on the machine load.

		Sequential	Multi-threaded		Concurrent mono-threaded	
Matrix size $(v+s)$	eliminated variables (v)	time (seconds)	time (seconds)	efficiency (%)	time (seconds)	efficiency (%)
100	100	2.41×10^{-4}	1.72×10^{-4}	17.5	2.51×10^{-4}	96.0
1000	100	2.00×10^{-2}	4.28×10^{-3}	58.4	2.33×10^{-2}	85.8
	1000	7.94×10^{-2}	1.81×10^{-2}	55.0	9.80×10^{-2}	81.0
10000	100	2.10×10^{0}	2.96×10^{-1}	88.6	2.26×10^{0}	92.7
	1000	1.88×10^{1}	2.69×10^{0}	87.3	2.03×10^{1}	92.4
	10000	7.63×10^{1}	1.14×10^{1}	83.9	8.50×10^{1}	89.8
15000	100	4.72×10^{0}	6.52×10^{-1}	90.6	5.12×10^{0}	92.3
	1000	4.35×10^{1}	6.04×10^{0}	90.0	4.72×10^{1}	92.1
	10000	2.37×10^{2}	3.41×10^{1}	87.1	2.61×10^{2}	91.2
	15000	2.57×10^{2}	3.70×10^{1}	87.0	3.43×10^{2}	**75.0**

Table 4: Factorization times on `hidalgo`, with varying matrix sizes and varying number of eliminated variables. 'Sequential' is the execution time for one task using one thread; 'Multi-threaded' represents the time spent in one task using eight cores, and 'Concurrent mono-threaded' represents the time spent in eight tasks using eight cores.

From the results exposed in Table 4, we can see that, on the `hidalgo` computer with 8 cores, concurrent mono-threaded factorizations are more efficient than multi-threaded ones. Moreover, concurrent mono-threaded factorizations are always efficient while muti-threaded ones need matrices of larger sizes in order to be efficient. We also note that the degradation of performance for both types of factorizations decreases as the number of eliminated variables increases (passing from partial to total factorizations: increasing v for a constant $v+s$). Surprisingly, we note that in the very special case of the total factorization of the 15000×15000 matrix, the performance of concurrent mono-threaded factorizations drops down. This contradicts the usual sense, and shows the limits of the concurrent mono-threaded approach. We can conclude by saying that the degradation in performance is between 5% and 20% in the majority of cases we could encounter under an $\mathcal{L}_{th}$ when applying an $\mathcal{L}_{th}$-based algorithm, but can reach up to 33% in very unusual cases.

From the results exposed in Figure 5, we can see how the architectures impact the performance. On `vargas`, which is a NUMA machine with such characteristics that it could be considered as an SMP machine, with nearly uniform memory access, the degradation of performance is progressive with the number of cores. Also, even if the degradation is important for small matrices (1000×1000), it becomes negligible for large ones (5000×5000 and 10000×10000). On `dude` though, we observe a very rapid degradation of performance independently of the size of the matrices. When choosing the cores on the same processor (6 first cores), the degradation is very smooth and predictable. However, once cores of different processors are used, the trend of the degradation becomes extremely chaotic.



(a) Influence of load on `vargas` (SMP)

(b) Influence of load on dude (NUMA)

Figure 5: Influence of machine load on a SMP machine (`vargas`, 32 cores) and on a NUMA machine (dude, 24 cores). The X-axis represents the number of cores involved in the 'concurrent mono-threaded' factorization, and the Y-axis represents the efficiency compared to an unloaded machine.

5.1.2 Memory locality and memory interleaving

The more a machine has NUMA effects, the more the cost of memory accesses to a given memory zone will be unbalanced among the processors. Thus, the way a dense matrix is mapped over the memory banks has important consequences on the performance on the factorization time. In Table 5, we extracted the main results of an experiment consisting in factorizing with Algorithm 2 a matrix of size 4000×4000 with $v = 1000$ variables to eliminate and a Schur complement of size $s = 3000$ with all possible combinations of cores used and memory allocation policies. From these

	Core ID	`membind 0`	`membind 1`	`localalloc` (OS default)	`interleave 0,1`
node 0	0	4.77	4.82	4.78	4.79
	1	4.74	4.78	4.73	4.75
	0...3	1.39	1.44	1.39	1.37
node 1	4	4.75	4.71	4.71	4.72
	4...7	1.44	1.39	1.39	1.37
node 0,1	all	1.10	1.11	1.09	**0.79**

Table 5: Effect of the `locallaloc` and `interleave` policies with different core configurations on `hidalgo`. The factorization times (seconds) are reported for a matrix of size 4000 with $v = 1000$ variables to eliminate. `membind` 0 (resp. 1) forces data allocation on the memory bank of node 0 (resp.1). `hidalgo` are in seconds on `hidalgo` of the factorization of a matrix of size 4000 and with $v = 1000$ eliminated variables with different core configurations.

results, we note that the `localalloc` policy (allocating memory on the same node as the core asking for it) is the best policy when dealing with serial factorizations. However, the `interleave` policy (interleaving pages on the memory banks in a round-robin manner) becomes the best one when dealing with multi-threaded factorizations, even when threads are mapped on cores of the same processor. However, this is partially due to the fact that the experiment was done on an

unloaded machine: taking advantage from unused resources from the idle neighbour processors is preferable to overwhelming local socket resources (memory banks, memory bus, caches, ...). When running the experiment on all cores, the `interleave` policy is by far the best one, and brings a huge gain over the local policies. Further experiments made with various matrix sizes and on the `dude` machine confirm this result.

Therefore, it seems that the best solution when working under $\mathcal{L}_{th}$ with concurrent mono-threaded factorizations is to allocate thread-private workspaces locally. On the other hand, when working above $\mathcal{L}_{th}$ with multi-threaded factorizations, it will be preferable to allocate the shared workspace using the interleave policy.

5.2 Application to sparse matrix factorizations

Controlling the memory policy can be done in several ways. A first, non intrusive one, consists in using the *numactl* utility. However, this utility sets the policy for all the allocations in the program. The second, more intrusive, approach consists in using the *libnuma* or *hwloc* [9] libraries. Inside the program, it is then possible to dynamically change the allocation policy. In order to apply the `interleave` policy only for the workspace shared by all processors above $\mathcal{L}_{th}$, while keeping the default `localalloc` policy for other allocations, in particular for the data local to each thread under $\mathcal{L}_{th}$, the second approach is necessary. However, applying the desired policy for each allocation is not enough to improve performance. A profiling of the memory mapping of the shared workspace's pages shows that the interleave policy is not applied at all. The reason is that when calling the *malloc* function to allocate a large array, only its first page is actually allocated, and the remaining pages are only allocated when an element of the array corresponding to that page is accessed for the first time. In order to effectively apply the `interleave` policy, one solution consists in accessing (writing one element of) each page of the array immediately after the allocation. Another portable solution that does not depend on external libraries is to make all threads access all pages that should be allocated on the same socket. Making this small modification dramatically improves the performance of the factorization, as will be discussed in Section 5.3.

Remark that the previous approach forces one to allocate all the pages of the shared workspace at once, while the physical memory also needs to hold the already allocated private workspaces. In the standard approach, memory pages related to the shared workspace are allocated only when needed. At the same time, we would like pages of private workspaces to be gradually recycled to pages of the shared workspace, allowing for a better usage of memory (see remark on *realloc* in Section 4.3). In general, the **first-touch** principle applies, which states that, with the `localalloc` memory allocation policy, a page is allocated on the same node as the thread that first touches it. While working on the parallelization of assembly operations (Algorithm 1), we observed that even without the *interleave* policy, it was possible to better share the memory pages between the threads by parallelizing the initialization of the frontal matrix to zero. While this had no effect on the time spent in assemblies, significant gains were observed regarding the performance of the factorization (Algorithm 2). Although those gains were not as large as the ones observed with the `interleave` policy (which are the ones we present in Section 5.3), this approach is an interesting alternative.

5.3 Performance analysis

5.3.1 Effects of the interleave policy

Table 6 shows the factorization time obtained on the `hidalgo` machine when using the `interleave` policy above $\mathcal{L}_{th}$ with the AlgTime algorithm. Column "Interleave/off" is identical to the last

column of Table 3. The gains are significant for all matrices except for the smallest problems and for the ones with too small frontal matrices, where interleaving does not help (as could be expected). In Table 7, we analyze further the impact of the `interleave` memory allocation policy

	AlgTime	
	Interleave	
Matrix	off	on
3Dspectralwave	339.78	295.84 (7.0)
AUDI	210.10	187.57 (6.8)
sparsine	57.91	48.60 (6.5)
ultrasound80	77.85	67.29 (6.6)
dielFilterV3real	44.52	40.88 (6.7)
Haltere	99.51	98.29 (7.0)
ecl32	0.72	0.70 (4.3)
G3_circuit	3.02	3.03 (5.6)

	AlgTime	
	Interleave	
Matrix	off	on
QIMONDA07	4.26	4.35 (5.9)
GeoAzur_3D_32_32_32	12.87	12.64 (6.0)
GeoAzur_3D_48_48_48	62.48	61.34 (6.7)
GeoAzur_3D_64_64_64	228.12	224.56 (7.0)
GeoAzur_2D_512_512	0.84	0.85 (5.3)
GeoAzur_2D_1024_1024	5.02	4.97 (6.2)
GeoAzur_2D_2048_2048	34.56	34.01 (6.7)

Table 6: Factorization times (seconds) without with the interleave policy on the factorization time with the AlgTime algorithm on `hidalgo`, 8 cores. The numbers in parenthesis correspond to the speed-ups with respect to sequential executions.

on the `dude` platform, which has more cores and shows more NUMA effects than `hidalgo`. The first columns correspond to runs in which only node parallelism is applied, whereas in the last columns, the AlgTime algorithm is applied. Parallel `BLAS` ("Threaded `BLAS`") in Algorithm 2 may be coupled with an `OpenMP` parallelization of Algorithms 1 and 3 (column "Threaded `BLAS` + `OpenMP`"). The first observation is that the addition of `OpenMP` directives on top of threaded `BLAS` brings significant gains compared to the use of threaded `BLAS` alone. We also observe in those cases that the `interleave` policy alone does not bring so much gain and does even bring losses in a few cases. Then, we used the AlgTime algorithm combined with the `interleave` policy (above $\mathcal{L}_{th}$). By comparing the timings in the last two columns, we then observe very impressive gains with the `interleave` policy.

These results can be explained by the fact that the `interleave` policy is harmful on small dense matrices, possibly because when CPU cores of different NUMA nodes collaborate, the cost of cache coherency will be much more important than the price of memory access. On the contrary, on medium-to-large dense matrices, the effects of the `interleave` policy are beneficial. The $\mathcal{L}_{th}$ layer separating the small fronts (bottom) from the large fronts (top) makes us benefit from interleaving without its negative effects. This is why the `interleave` policy alone does not bring much gain, whereas it brings huge gains when combined to the AlgTime algorithm.

Table 8 shows the time spent under and above $\mathcal{L}_{th}$, with and without interleaving, with and without $\mathcal{L}_{th}$ for the AUDI test case. In the first two columns corresponding to node parallelism, although the $\mathcal{L}_{th}$ layer is not used by the algorithm, we measure the times corresponding to the $\mathcal{L}_{th}$ layer defined by the AlgTime algorithm from the last two columns. We can see that $\mathcal{L}_{th}$-based algorithms improve the time spent under $\mathcal{L}_{th}$ (from 109.81 seconds to 36.02 seconds) thanks to tree parallelism; above, the time is identical since only node parallelism is used in both cases. When using the `interleave` policy, we can see that the time above $\mathcal{L}_{th}$ decreases a lot (from 121.95 seconds to 73.93 seconds) but that this is not the case for the time under $\mathcal{L}_{th}$. Moreover, we note that using the `interleave` policy without using $\mathcal{L}_{th}$-based algorithms is disastrous for the performance under the $\mathcal{L}_{th}$ layer (from 109.81 seconds to 151.54 seconds). This confirms that the `interleave` policy should not be used on small frontal matrices, especially in parallel.

On huge matrices, such as Serena (Table 7), the sole effect of the AlgTime algorithm is

Matrix	Serial reference	Node parallelism only: Threaded BLAS, Interleave off	Node parallelism only: Threaded BLAS, Interleave on	Node parallelism only: Threaded BLAS + OpenMP, Interleave off	Node parallelism only: Threaded BLAS + OpenMP, Interleave on	AlgTime algorithm, Interleave off	AlgTime algorithm, Interleave on
3Dspectralwave	2365.17	375.44	362.25	322.97	342.89	288.29	**174.74**
AUDI	1535.78	269.77	260.80	231.76	225.47	157.97	**110.04**
conv3D64	3001.43	518.51	563.09	497.52	496.87	438.98	**303.61**
Serena	7845.43	1147.59	1058.00	1081.42	1006.66	893.64	**530.63**
sparsine	365.43	64.90	67.37	57.07	58.63	60.16	**35.86**
ultrasound80	516.14	104.48	100.75	93.87	90.61	74.52	**44.89**
dielFilterV3real	324.50	81.07	80.75	68.42	69.91	36.17	**25.30**
Haltere	867.47	142.89	145.04	133.61	135.20	80.90	**56.54**
ecl32	3.91	2.10	2.12	1.74	1.70	1.10	**0.88**
G3_circuit	24.92	16.19	16.13	14.88	14.67	3.39	**2.81**
QIMONDA07	31.82	54.21	51.79	52.57	55.45	**4.23**	4.30
GeoAzur_3D_32_32_32	88.42	17.68	17.50	15.77	15.68	10.74	**8.54**
GeoAzur_3D_48_48_48	479.81	75.72	74.13	70.31	66.73	52.27	**37.45**
GeoAzur_3D_64_64_64	1774.57	240.40	239.86	221.73	225.45	195.69	**119.77**
GeoAzur_2D_512_512	5.28	18.42	18.18	21.24	21.12	1.91	**1.88**
GeoAzur_2D_1024_1024	39.91	86.66	102.17	97.2	152.81	**15.65**	19.38
GeoAzur_2D_2048_2048	309.64	98.81	158.22	96.26	436.65	**44.70**	55.28

Table 7: Factorization times in seconds and effects of the `interleave` memory allocation policy with node parallelism and with AlgTime on `dude`.

not so large (1081.42 seconds down to 893.64 seconds) but the effect of memory interleaving without $\mathcal{L}_{th}$ is even smaller (1081.42 seconds down to 1006.66 seconds). Again, the combined use of AlgTime and memory interleaving brings a huge gain: 1081.42 seconds down to 530.63 seconds (increasing the speed-up from 7.3 to 14.7 on 24 cores). Hence, on very large matrices, the main benefit of AlgTime is to make the interleaving become very efficient by applying it only above $\mathcal{L}_{th}$. This also shows that in our implementation, it was critical to separate the work arrays for local threads under $\mathcal{L}_{th}$ and for the more global approach in the upper part of the tree, in order to be able to apply different memory policies under and above $\mathcal{L}_{th}$ (`localalloc` and `interleave`, respectively).

	Node parallelism only (Threaded BLAS + OpenMP directives)		$\mathcal{L}_{th}$-based algorithm (AlgTime)	
Time	without interleaving	with interleaving	without interleaving	with interleaving
Under $\mathcal{L}_{th}$	109.81	151.54	36.02	36.11
Above $\mathcal{L}_{th}$	121.95	73.93	121.95	73.93
Total	231.76	225.47	157.97	110.04

Table 8: Interactions of $\mathcal{L}_{th}$ and memory interleaving on `dude`, for matrix AUDI. Times are in seconds.

5.3.2 Effects of modifying the performance models

As said before, the performance model is slightly optimistic under $\mathcal{L}_{th}$ because load, and to a minor extent assemblies and stack operations, were not taken into account in the dense benchmarks we rely on. On the other hand, the performance model is pessimistic above $\mathcal{L}_{th}$ because it does not use the `interleave` policy. Table 9 shows the effects of modifying the performance models by taking into account the load under $\mathcal{L}_{th}$ and an "interleaved" benchmark above $\mathcal{L}_{th}$. The results are again for the AUDI matrix on the `dude` machine, for which the penalty ratio for the load was set experimentally to 1.4.

Variant	Time under $\mathcal{L}_{th}$		Time above $\mathcal{L}_{th}$		Total Time
	Predicted	Observed	Predicted	Observed	Observed
no load + normal benchmark	28.08	36.93	137.56	73.37	110.30
load + interleaved benchmark	26.28	25.76	94.39	82.68	108.44

Table 9: Results of taking into account load under $\mathcal{L}_{th}$ and interleaved benchmark above on dude on the AUDI matrix. AlgTime is used with `localalloc` and `interleave` policies under and above $\mathcal{L}_{th}$, respectively.

We observe that the predicted times under and above $\mathcal{L}_{th}$ are more accurate when taking into account the machine load and the interleaved benchmark. For the AUDI matrix, the prediction error under $\mathcal{L}_{th}$ is of 24% when ignoring load and of 4.6% when taking it into account. Similarly, the prediction error above $\mathcal{L}_{th}$ is of 47% with normal benchmark and of 13% with the interleaved benchmark. It may look surprising that the interleaved benchmark still leads to pessimistic estimates. This may be due to the sequence of parallel assembly, factorization and stack operations in `MUMPS` that keep the caches and TLB's in a better state than in the benchmarking code.

Furthermore, the improved accuracy of predictions allows in turn the AlgTime algorithm to make smarter choices of the $\mathcal{L}_{th}$'s. This then yields better total factorization times. For the AUDI matrix, $\mathcal{L}_{th}$ contains 42 subtrees when taking into account load and interleaved benchmarks, and only 29 subtrees when ignoring them. This means that the $\mathcal{L}_{th}$ layer has been lowered down the tree, which is expected since we added a penalty on the computations under the $\mathcal{L}_{th}$ and improved that above. Consequently, less time is spent under $\mathcal{L}_{th}$ (25.76 seconds vs. 36.93 seconds) and more time is spent above (82.68 second vs. 73.37 second). Finally, the total factorization time is decreased from 110.30 second to 108.44 seconds.

Further improvements to the performance model would be hard to achieve because of the difficulty of modeling further cache effects, assemblies and stack operations. Also, the performance on a matrix with given characteristics may vary depending on the machine state and numerical considerations such as pivoting. In the next section, we present another approach, more dynamic, where we show that it is possible to be less dependent on the accuracy of the benchmark and the precise choice of the $\mathcal{L}_{th}$ layer.

6 Recycling idle cores

When using an $\mathcal{L}_{th}$-based algorithm, many cores become idle when they finish their share of work under $\mathcal{L}_{th}$. Consequently, one must pay the price of the corresponding synchronization, where the $\mathcal{L}_{th}$ layer represents a barrier. The greater the unbalance under $\mathcal{L}_{th}$, the greater the synchronization cost. Moreover, we only used `BLAS` with either one core or with all cores, although `BLAS` performance is not ideal on all cores and an arbitrary number of cores could be

used. The efficiency of sequential `BLAS` is high, but decreases progressively with the number of cores used. For example, on `dude`, the efficiency of Algorithm 2 is of about 76% on 6 cores but goes down to 51% on 24 cores. Consequently, $\mathcal{L}_{th}$-based algorithms make good use of sequential `BLAS` but not of parallel `BLAS`. This suggests that applying a smoother transition between the bottom and the top of the tree could make better use of `BLAS` while exploiting idle cores under $\mathcal{L}_{th}$.

6.1 Core idea and algorithm

In order to reduce the number of idle cores, we can think of two possibilities: either making them start the work above $\mathcal{L}_{th}$; or making them help the others under $\mathcal{L}_{th}$.

The former solution is attractive but presents complications. First, because most memory is consumed on the large nodes above $\mathcal{L}_{th}$, the order of traversal of the elimination tree above $\mathcal{L}_{th}$ is usually not arbitrary and is constrained to be a given postorder [28]. This would imply constraints on the order of the nodes under $\mathcal{L}_{th}$: for instance, one must guarantee that the roots of $\mathcal{L}_{th}$ whose parent is the first node to be processed above $\mathcal{L}_{th}$, have been processed early enough. This could lead to a loss of time, which may not be compensated by the gains of an idle core recycling strategy. Second, once a `BLAS` call on a large front is initiated in a node above $\mathcal{L}_{th}$ (typically, at lines 12 or 13 of Algorithm 2), the number of threads in that call can no longer be modified. Since nodes above $\mathcal{L}_{th}$ are larger than those under $\mathcal{L}_{th}$, it is likely that many threads working under $\mathcal{L}_{th}$ would complete their work before those working above. In that case, even if new idle cores show up, they may not be used to help the thread(s) that started working on a node above $\mathcal{L}_{th}$. Third, if we introduce tree parallelism above $\mathcal{L}_{th}$ in order to remedy the previous problems, a whole new memory management strategy would have to be designed in order to be able to make many threads work in a single shared workspace. Such a deep modification is besides the objectives of our study, in which we aim at adapting a distributed-memory code to make use of multi-core without a deep re-design.

The latter solution, consisting in re-using idle cores to help active threads under $\mathcal{L}_{th}$, is simpler and more natural. The main question is: how to dispatch idle cores over active threads that are still working under $\mathcal{L}_{th}$? When using node parallelism, the least the cores, the more efficient they will be. Therefore, in order to achieve the fairest possible core dispatching, we use a strategy consisting in assigning repeatedly each new idle core to the thread (or thread team) that has the minimum number of cores (or threads) at its disposal.

Algorithm 7 (Mapping of idle cores over threads) gives a possible mapping strategy for idle cores on threads over time. It is executed in a critical section, exactly once by each thread that has finished its share of the work under $\mathcal{L}_{th}$ and discovers that the pool of unprocessed subtrees is empty. From time to time, when a thread starts the factorization of a new panel in a front, or between two `BLAS` calls, it applies Algorithm 7 (Detection of available cores), to check whether the number of cores at its disposal has changed. This is done by comparing the entry in array *nb_cores* in the index corresponding to *my_thread_id* with is current number of cores, stored in the private variable *my_nb_cores*. If this is the case, it then updates its variable *my_nb_cores* and updates its number of cores with a call to *omp_set_num_threads*. In our implementation, we do not use a mutex (or lock) because we assume atomic (exclusive) unitary read/writes of the small integer (aligned 32-bit word) entries in the array *nb_cores*. If this was not the case, one could replace the critical section from the idle core mapping by a mutex, and use that mutex in the algorithm detecting available cores. The mutex should be inside the *if*-block at line 2 in order to limit its cost: it is only used in case *nb_cores*(*my_thread_id*) is being, or has been, modified.

The implementation of Algorithm 7 is straightforward. However, dynamically changing the number of resources inside a parallel region was not, mainly because of the interactions between

Algorithm 7 $\mathcal{L}_{th}$ with idle core recycling.

Initialization:
shared:
    $nb\_cores \leftarrow (1, \ldots, 1)$ {Number of cores of each thread}
private:
    $my\_thread\_id \leftarrow$ Id of the current thread
    $nb\_threads \leftarrow$ Total number of threads
    $my\_nb\_cores \leftarrow 1$ {Number of cores currently used by the current thread}

Mapping of idle cores over threads:
while $nb\_cores(my\_thread\_id) > 0$ **do**
  {Find thread with least number of cores}
  $id\_thread\_to\_help \leftarrow 0$
  for $i = 1$ **to** $nb\_threads$ **do**
    if $i \neq my\_thread\_id$ **and** $nb\_cores(i) \neq 0$ **and** ($id\_thread\_to\_help = 0$ **or** $nb\_cores(i) < nb\_cores(id\_thread\_to\_help)$) **then**
      {thread i is not me and has not finished yet and (is the first we encountered or has less cores than the current thread we wish to help)}
      $id\_thread\_to\_help \leftarrow i$
    end if
  end for
  if $id\_thread\_to\_help \neq 0$ **then**
    {Notify thread $id\_thread\_to\_help$ that it has more cores}
    $nb\_cores(id\_thread\_to\_help) \leftarrow nb\_cores(id\_thread\_to\_help) + 1$
    $nb\_cores(my\_thread\_id) \leftarrow nb\_cores(my\_thread\_id) - 1$
  else
    Break
  end if
end while

Detection of available cores:
if $my\_nb\_cores < nb\_cores(my\_thread\_id)$ **then**
  $my\_nb\_cores = nb\_cores(my\_thread\_id)$
  $omp\_set\_num\_threads(my\_nb\_cores)$
  {Change the number of cores to be used by current thread (my_thread_id)}
end if

`OpenMP` and `BLAS` libraries. In Algorithm 7, the *omp_set_num_threads* function sets the number of cores for future `OpenMP` parallel regions of the calling threads. Nested parallelism should be enabled, and this can be done thanks to the *omp_set_nested* function. Also, the *omp_set_dynamic* function must be used in order to disable automatic dynamic control of the number of threads. When enabled, this dynamic control typically forbids *omp_set_num_threads* to increase the number of threads within a nested region up to a number larger than the original number of threads available for the inner parallel regions. Even after using these functions, we still had problems within the `MKL BLAS` library, which still automatically limited the effective number of cores used. After setting the `MKL` dynamic behaviour to false by calling *mkl_set_dynamic* with argument *.false.*, we finally observed that `BLAS` kernels took advantage of the extra cores provided. This shows that the portability of such approaches regarding current technologies is still an issue.

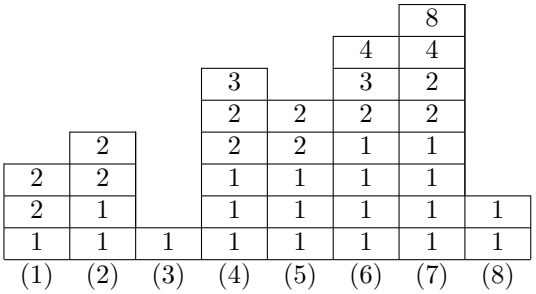


Figure 6: Distribution of idle cores over time in the $\mathcal{L}_{th}$ idle cores recycling algorithm on `hidalgo` (8 cores) on matrix AUDI. Thread ids are at the bottom inside parenthesis. Initially (bottom), all threads have one core available. When core 3 becomes idle, the task on thread 1 starts using 2 cores instead of 1. At the end (top), all cores are assigned to the task initially processed by thread 7.

Figure 6 illustrates the execution of the $\mathcal{L}_{th}$ idle core recycling algorithm for the factorization of the AUDI matrix on `hidalgo`. Each row represents the number of cores at the disposal of each thread (column). When a thread finishes, it sets its number of cores to 0. Each time a thread ends up its work under $\mathcal{L}_{th}$, it updates the array *nb_core* (see Algorithm 7).

Notice that our approach has the same purpose as a work-stealing mechanism. However, the view is opposite to the one of work-stealing mechanisms: instead of stealing work from other threads' ready tasks pools, an idle thread offers itself to help active threads (on already started tasks).

6.2 Analysis and discussion

When comparing the time spent under $\mathcal{L}_{th}$ by each thread, we observed that several threads finished significantly earlier when using Algorithm 7. However, we also observed two minor problems.

- First, many threads never used the additional cores put at their disposal. In several cases, the time spent in the root of a subtree under $\mathcal{L}_{th}$ is between one third and one half of the time spent in the whole subtree. Thus, when the first thread finishes its subtrees, it often happens that others already started the computation of the root of their final subtree, or the last large `BLAS` call at the end of the factorization (line 13 of Algorithm 2). Therefore, they did not realize that new cores are actually available. In less extreme cases, some threads did notice, at first, the availability of new cores, but failed to notice it afterwards.

As we can see in Figure 6, the number of cores available for the last working thread (in our example the seventh) converges progressively to the total number of cores of the machine. However, when this last thread started its last `BLAS` calls at the root of its last subtree, only two cores were available; yet, more cores became available after that (4 then 8). We could observe that, on well-balanced $\mathcal{L}_{th}$'s, this phenomenon occurs frequently, whereas on badly-balanced $\mathcal{L}_{th}$'s, the algorithm is more efficient since more threads are able to notice the availability of new cores.

- Second, it happens that many threads take advantage of available idle cores and finish sooner than expected, but that the time spent under $\mathcal{L}_{th}$ remains unchanged. This could be explained by the fact that not enough cores were given to the thread with the largest amount of remaining work (like a critical path of execution), on which depends the time under $\mathcal{L}_{th}$.

Many variants of the idle core recycling algorithm could be applied, particularly on the mapping of idle cores on active threads. One variant could be to postpone the attribution of idle cores from the moment a thread completes its work to the moment a thread looks for idle cores to help it, thus helping with the first problem. Another approach would consist in giving idle cores in priority to the thread on the critical path, with an estimation of the remaining work of each thread, thus solving the second problem. However, the approach discussed in the next subsection (`BLAS` split) will be enough to solve both problems.

6.3 Early idle core detection and NUMA environments

As explained before, one difficulty to detect idle cores comes from the large time spent in the last `TRSM` and `GEMM BLAS` calls from Algorithm 2. To test for the availability of idle cores more often, we decided to split the `BLAS` calls into pieces: we first update a block of rows of L, then update the corresponding block of rows of the Schur complement, then work on the next block of L, followed by the next block of the Schur complement, etc. Because we need large blocks in the `BLAS` calls, especially when we want to exploit multithreaded `BLAS`, we only split the `BLAS` calls in two or three, only on the roots of $\mathcal{L}_{th}$ subtrees. This decision is strengthened by the fact that the situation where idle cores can help active threads often happens only when all these threads work on the roots of $\mathcal{L}_{th}$ subtrees.

Another improvement concerns NUMA architectures, where care must be taken when mapping idle cores over active threads. For this, we still map idle cores on threads with a minimum number of cores at their disposal; however, in case of equality, preference is given to the threads mapped on the same processor (or NUMA node). Remark that this requires knowledge of the thread-to-socket mapping; this was done by initially binding threads to processors. For example, on `hidalgo`, we map threads 1 to 4 on the first processor and threads 5 to 8 on the second processor.

Table 6 shows comparative times spent under $\mathcal{L}_{th}$ by threads for the AUDI matrix on `hidalgo`. Splitting the last `BLAS` calls makes threads detect the availability of idle cores much more often, and decreases the time spent under $\mathcal{L}_{th}$. This also implies that the problem of identifying the critical path of execution under $\mathcal{L}_{th}$ is no longer an issue, since the thread on this critical path will finally take advantage of a large number of cores. We can also observe that the NUMA-friendly approach brings further improvement. This gain is not very large; however, the `hidalgo` computer being composed of two quad-core Nehalem processors, it experiences limited NUMA effects. On `dude` (which has more NUMA effects), we observed that with 24 cores, the $\mathcal{L}_{th}$ layer resulting from the AlgTime algorithm was always sufficiently well balanced so that

idle core recycling had no gain to bring. We discuss in Section 6.4 how a modification of the $\mathcal{L}_{th}$ layer can impact performance.

Thread IDs	No core recycling	Core recycling		
		original	Split `BLAS`	Split `BLAS` + NUMA-friendly
1	82.73	82.33	82.20	82.99
2	84.35	83.72	83.86	83.94
3	91.08	87.17	87.56	88.78
4	91.45	90.35	88.73	89.28
5	91.53	90.62	89.32	90.27
6	93.23	92.99	90.83	91.23
7	94.06	93.53	91.78	91.63
8	95.59	94.77	92.23	92.03

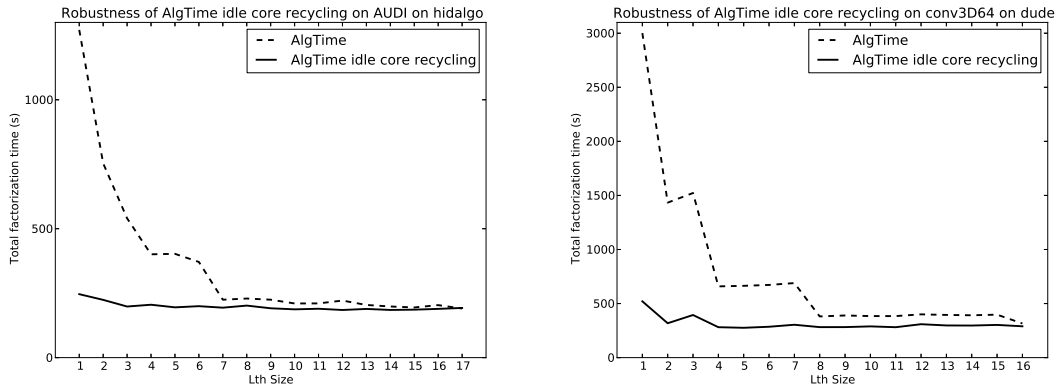
Table 10: Time spent under $\mathcal{L}_{th}$ by each thread on the AUDI matrix on `hidalgo` (8 cores) without the idle core recycling algorithm, with it, and with the split `BLAS` extension. In the last three columns, when a thread finishes its last subtree, the corresponding core is reassigned to other threads.

6.4 Sensitivity to $\mathcal{L}_{th}$ height

One consequence of re-using idle cores in Algorithm 7 is that the performance of the factorization is less sensitive to the height of $\mathcal{L}_{th}$. The algorithm may even work better when choosing an $\mathcal{L}_{th}$ higher than that found by AlgFlops or AlgTime: this way, the nodes in the tree handled sequentially with a normal $\mathcal{L}_{th}$ will still be handled sequentially, but a part of the nodes that were originally treated using all cores will now be treated with less cores, leading to higher efficiency. Doing so moves Algorithm 7 from the initial aim of making a good load balancing under $\mathcal{L}_{th}$ to the additional goal of melting tree parallelism with node parallelism. For example, if on a system with p cores, we define an $\mathcal{L}_{th}$ layer with exactly p subtrees, each core will work on one subtree, exploiting tree parallelism up to a certain level, and node parallelism will start smoothly by reassigning cores of finished subtrees to the other subtrees.

Figure 7 shows the total factorization time for AUDI on `hidalgo` and for conv3D64 on `dude`, for $\mathcal{L}_{th}$ layers of different sizes, with and without the use of Algorithm 7. Using Algorithm 7, we can see that a higher than normal $\mathcal{L}_{th}$ layer does not bring gain on AUDI, but at least provides a stable performance: Algorithm 7 acts as a safeguard of the unbalance of $\mathcal{L}_{th}$'s.

On the conv3D64 problem with more cores, however, one can see some loss with certain sizes of $\mathcal{L}_{th}$'s but gains with others. For example, with an $\mathcal{L}_{th}$ of size 16, the total computation time is 303.61 seconds whereas, with an $\mathcal{L}_{th}$ of size 5, the total computation time drops down to 275.87 seconds. Even though this gain of time is valuable, raising the $\mathcal{L}_{th}$ is different from what we have studied so far: such an $\mathcal{L}_{th}$ contains fewer subtrees (5 subtrees) than there are cores on the machine (24 cores), which means that most nodes under $\mathcal{L}_{th}$ have been treated with 5 cores each. In such a situation, whole subtrees, and more particularly small nodes in the bottom of these subtrees, are computed with more than one core, which is not ideal for efficiency. This means that the gains are obtained higher in the tree: the side effect of raising the $\mathcal{L}_{th}$ layer is that tree parallelism is used higher in the tree, whereas less node parallelism is used, with a smoother transition between tree and node parallelism. Since the scalability of the dense factorization kernels we experimented with is not ideal on 24 cores, on this matrix, this can be more efficient.



(a) AUDI matrix on `hidalgo` (b) CONV3D64 matrix on `dude`

Figure 7: Robustness of the idle core recycling algorithm when $\mathcal{L}_{th}$ is raised. Remark that the leftmost parts of the dotted and plain curves ($\mathcal{L}_{th}$ composed of one node) correspond to the serial time and to the time with node parallelism only, respectively.

7 Conclusion

In this paper, we have presented a methodology to adapt an existing, fully-featured, distributed-memory code to shared-memory architectures. We studied the gains of using efficient multithreaded libraries (in our case, optimized `BLAS`) in combination to message-passing, then introduced multithreading in our main computation kernels thanks to `OpenMP`. We then exploited higher-level parallelism, taking advantage of the characteristics of the task graph arising from the problems. Because the task graph is a tree, serial kernels are first used to process independent subtrees in parallel, and multithreaded kernels are then applied to process the nodes from the top of the tree. We proposed an efficient algorithm based on a performance model of individual tasks to determine when to switch from tree parallelism to node parallelism. Because this switch implies a synchronization, we showed how it is possible to reduce the associated cost by dynamically re-assigning idle CPU cores to active tasks. We note that this last approach depends on the interaction between the underlying compilers and `BLAS` libraries and requires a careful configuration. We also considered NUMA environments, showed how memory allocation policies affect performance, and how memory affinity can be efficiently exploited in practice.

All along this study, we relied as much as possible on third-party optimized multithreaded libraries (in our case, `BLAS`). The proposed approaches, although some of them are very general, were illustrated with experiments done in the context of an existing solver, `MUMPS`. Large performance gains were obtained, while reusing the existing computational kernels and memory management algorithms, and keeping the existing numerical functionalities.

This study is complete and will impact future versions of the `MUMPS` solver; yet it opens doors to new perspectives. As shown in Section 6, using a smooth transition between tree to node parallelism can bring valuable gains when increasing the number of cores; in order to go further in that direction, one would need to design new scheduling and memory management algorithms. The specific dense kernels currently used in our software were not initially designed for multithreaded environments and could thus be improved. Although this is out of the scope of this study, and although our methodology is independent from their absolute performance, optimized kernels would result in better overall speed-ups on large numbers of cores, would

postpone the need to redesign the scheduling and memory management schemes, while limiting extra-memory usage when too much tree parallelism is used. Finally, although we only focused on computations with a single `MPI` process, we now have an hybrid distributed-memory/shared-memory code available that can take advantage of modern clusters of multi-core nodes. A preliminary experiment shows that, on 8 nodes with 8 cores each of the *bonobo* machine from the *plafrim* platform at Inria-Bordeaux, the factorization time of a GeoAzur_3D matrix of size 96×96×96 using 64 `MPI` processes (one core per `MPI` process) takes **657 seconds** and falls down to **297 seconds** on 8 `MPI` processes with 8 cores per `MPI` when using ideas described in this paper. There is still significant room for improvements, particularly in the dense kernels that use both `OpenMP` threads and `MPI` processes near the root. The optimization of such an hybrid `MPI-OpenMP` approach will be the object of future work.

Acknowledgements

We are grateful to LIP, ENSEEIHT-IRIT and Inria for the access to their multicore machines. We thank Patrick Amestoy, Alfredo Buttari and Abdou Guermouche for valuable discussions and remarks on this work.

References

[1] The OpenMP application program interface. http://www.openmp.org.

[2] P. R. Amestoy. *Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment*. PhD thesis, Institut National Polytechnique de Toulouse, 1991. Available as CERFACS report TH/PA/91/2.

[3] 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.

[4] 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.

[5] 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.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, Feb. 2011.

[7] H. Avron, G. Shklarski, and S. Toledo. Parallel unsymmetric-pattern multifrontal sparse LU with column preordering. *ACM Transactions on Mathematical Software*, 34(2):8:1–8:31, 2008.

[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*, 2011.

[9] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In M. Danelutto, J. Bourgeois, and T. Gross, editors, *PDP*, pages 180–186. IEEE Computer Society, 2010.

[10] A. Buttari. Fine granularity QR factorization for multicore based systems. Technical Report RT-APO-11-6, ENSEEIHT, Toulouse, France, 2011.

[11] I. Chowdhury and J.-Y. L'Excellent. Some experiments and issues to exploit multicore parallelism in a distributed-memory parallel sparse direct solver. Research Report RR-4711, INRIA and LIP-ENS Lyon, Oct. 2010.

[12] T. A. Davis. Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse qr factorization. *ACM Trans. Math. Softw.*, 38(1):8:1–8:22, Dec. 2011.

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

[14] M. Drozdowski. Scheduling multiprocessor next term tasks: An overview. *European Journal of Operational Research*, 94(2):215–230, Oct. 1996.

[15] 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.

[16] M. Faverge. *Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-coeurs.* PhD thesis, LaBRI, Université Bordeaux I, Talence, Talence, France, Dec. 2009.

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

[18] A. Gupta. A shared- and distributed-memory parallel general sparse direct solver. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):263–277, 2007.

[19] P. Hénon, P. Ramet, and J. Roman. PaStiX: A parallel sparse direct solver based on a static scheduling for mixed 1D/2D block distributions. In *Proceedings of Irregular'2000, Cancun, Mexique*, number 1800 in Lecture Notes in Computer Science, pages 519–525. Springer Verlag, May 2000.

[20] P. Hénon, P. Ramet, and J. Roman. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing*, 28(2):301–321, Jan. 2002.

[21] J. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse Cholesky factorization using DAGs. Technical Report RAL-TR-2009-027, Rutherford Appleton Laboratory, 2009.

[22] J. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse cholesky factorization using dags. *SIAM Journal on Scientific Computing*, 6(32):3627–3649, 2010.

[23] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal sparse Cholesky. *Future Generation Computer Systems*, 20(3):425–440, 2004.

[24] G. Karypis and V. Kumar. MeTiS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0.* University of Minnesota, Sept. 1998.

[25] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[26] X. S. Li. Evaluation of SuperLU on multicore architectures. *Journal of Physics: Conference Series*, 125:012079, 2008.

[27] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2), 2003.

[28] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12:127–148, 1986.

[29] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and Practice. *SIAM Review*, 34:82–109, 1992.

[30] S. Operto, J. Virieux, P. R. Amestoy, J.-Y. L'Excellent, L. Giraud, and H. Ben Hadj Ali. 3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study. *Geophysics*, 72(5):195–211, 2007.

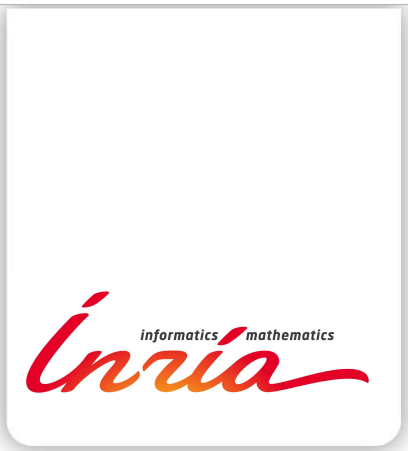
[31] F. Pellegrini. SCOTCH and LIBSCOTCH 5.0 User's guide. Technical Report, LaBRI, Université Bordeaux I, 2007.

[32] A. Pothen and C. Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 14(5):1253–1257, 1993.

[33] O. Schenk and K. Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical Report CS-2004-004, CS Department, University of Basel, August 2004.

[34] F. Sourbier, S. Operto, J. Virieux, P. R. Amestoy, and J.-Y. L'Excellent. FWT2D: a massively parallel program for frequency-domain full-waveform tomography of wide-aperture seismic data – part 2: numerical examples and scalability analysis. *Computer and Geosciences*, 35(3):496–514, 2009.

[35] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399