Sparse direct solvers with accelerators over DAG runtimes

Xavier Lacoste, Pierre Ramet, Mathieu Faverge, Yamazaki Ichitaro, Jack Dongarra

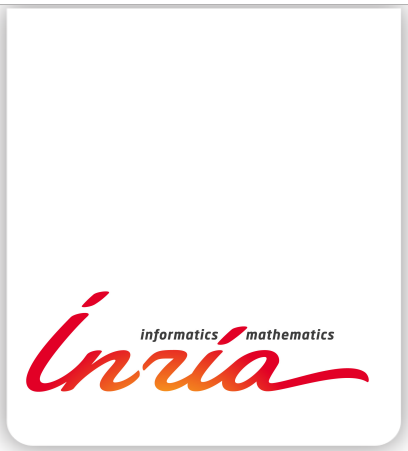
Sparse direct solvers with accelerators over DAG runtimes

Xavier Lacoste, Pierre Ramet
INRIA
University of Bordeaux
Bordeaux, France
Email : {xavier.lacoste,pierre.ramet}@inria.fr

Mathieu Faverge, Ichitaro Yamazaki, Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
Knoxville, Tennessee, USA
Email : {mfaverge,iyamazak,dongarra}@eecs.utk.edu

RESEARCH REPORT
N° 7972
27 April 2012
Project-Team Bacchus

ISSN 0249-6399 ISRN INRIA/RR--7972--FR+ENG



Sparse direct solvers with accelerators over DAG runtimes

Xavier Lacoste, Pierre Ramet
INRIA
University of Bordeaux
Bordeaux, France
Email : {xavier.lacoste,pierre.ramet}@inria.fr

Mathieu Faverge, Ichitaro Yamazaki, Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
Knoxville, Tennessee, USA
Email : {mfaverge,iyamazak,dongarra}@eecs.utk.edu

Équipe-Projet Bacchus

Rapport de recherche n° 7972 — 27 April 2012 — 11 pages

Résumé : Les architectures de calcul intègrent de plus en plus de coeurs de calcul partageant une même mémoire nécessairement hiérarchique. Les algorithmes, en particulier ceux relatifs à l'algèbre linéaire, nécessitent d'être adaptés à ces nouvelles architectures pour être efficaces. PASTIX [*] est un solveur direct parallèle pour matrices creuses qui intègre un ordonnanceur dynamique pour des architectures hiérarchiques de grande taille. Dans ce papier, nous étudions la possibilité de remplacer cette stratégie interne d'ordonnancement par deux supports d'exécution génériques : DAGUE [†] et STARPU [‡]. Ces supports d'exécution offrent la possibilité de dérouler le graphe de tâches de la factorisation numérique sur des noeuds de calcul disposant d'accélérateurs. Comme pour de précédents travaux réalisés pour les noyaux denses de l'algèbre linéaire, nous présentons nos noyaux, pour des structures creuses, issus de la librairie MAGMA et des algorithmes DAG utilisés par ces deux supports d'exécution. Nous présentons une étude comparative des performances de notre solveur supernodal avec ces trois ordonnanceurs sur des architectures multicoeurs et multigpus, et en particulier les gains obtenus sur les accélérateurs avec le support d'exécution STARPU. Ces résultats montrent qu'une approche basée sur un DAG offre une interface de programmation uniforme pour réaliser du calcul haute performance sur des problèmes irréguliers comme ceux de l'algèbre linéaire creuse.

Mots-clés : Algèbre linéaire creuse, Calcul haute performance DAG, GPU

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

Sparse direct solvers with accelerators over DAG runtimes

Abstract: The current trend in the high performance computing shows a dramatic increase in the number of cores on the shared memory compute nodes. Algorithms, especially those related to linear algebra, need to be adapted to these new computer architectures in order to be efficient. PASTIX [4] is a sparse parallel direct solver, that incorporates a dynamic scheduler for strongly hierarchical modern architectures. In this paper, we study the replacement of this internal highly integrated scheduling strategy by two generic runtime frameworks : DAGUE [5] and STARPU [6]. Those runtimes will give the opportunity to execute the factorization tasks graph on emerging computers equipped with accelerators. As for previous work done in dense linear algebra, we present the kernels used for GPU computations inspired by the MAGMA library and the DAG algorithm used with those two runtimes. A comparative study of the performances of the supernodal solver with the three different schedulers is performed on manycore architectures and the improvements obtained with accelerators are presented with the STARPU runtime. These results demonstrate that these DAG runtimes provide uniform programming interfaces to obtain high performance on different architectures on irregular problems as sparse direct factorizations.

Key-words: Sparse linear algebra, High performance computing, DAG, GPU

I. Introduction

Solving a large sparse general or symmetric positive definite linear system of equations, $Ax = b$, is a crucial and time-consuming part in many scientific and engineering applications. Due to their robustness, direct solvers are often used in industrial codes, despite their large memory consumption. In addition, the factorization used in the recent direct solvers can take advantage of the superscalar capabilities of the processors using blockwise algorithms and BLAS primitives. Consequently, many parallel techniques for sparse matrix factorization have been studied and implemented. For a complete survey on direct methods, we refer to [1]–[3]. The goal of this paper is to design algorithms that can plainly take advantage of the vast computing power found in modern heterogeneous computer architectures. In our current work, we focused on matrices with symmetric sparsity patterns (the sparsity patterns of $A + A^T$ is used for unsymmetric cases), and focused on the factorization with static pivoting. In this context, the block structure of the factors and the numerical operations are known in advance, and consequently, we can use a static (i.e. before the actual numerical factorization) algorithm for scheduling the communication and the computational tasks.

In previous works [4], [5], we have proposed a static mapping and scheduling algorithm based on a combination of 1D and 2D block distributions for sparse supernodal factorization with static pivoting. This algorithm achieves very good performance by taking into account the communication and computation requirements of each factorization as well as the communication and computation capabilities of the parallel architecture. In addition, we have developed a strategy to control the memory overhead and to reduce the communication volume needed for the message buffering. In PaStiX, this buffering corresponds to the local aggregation approach in which all local contributions for the same non-local block are summed into a temporary block buffer before being sent. The new strategy improves the mechanism of this local aggregation that often lead to great reduction in the memory consumption, especially for 3D problems.

Emerging supercomputers consist of many microprocessors, each of which may have many computational cores. Hence, these emerging architectures exhibit strongly hierarchical topologies, both in terms of memories and processor interconnect. Achieving good performance requires a mapping of the algorithms on the computational resources, and scheduling algorithms specifically designed for NUMA architectures.

In PaStiX the internal data structures of the solver, as well as the communication patterns, have been modified for the dynamic scheduling on these architectures [6]. A dynamic scheduler based on a work-stealing algorithm has been also developed to fill in communication idle times while preserving a good locality for data mapping [7]. Furthermore, we have integrated the dynamic adaptation of the computational task grain to efficiently use multi-core architectures and shared memory. Experiments on several numerical test cases have been performed to prove the efficiency of these approach on different architectures.

Modern GPUs can substantially outperform high-end multicore CPUs both in terms of data processing rate and memory bandwidth. In the past, porting general purpose codes onto GPUs required a considerable programming effort, mostly due to the lack of tools and interfaces. However, API for GPUs such as CUDA or OpenCL have been rapidly evolving in the last few years, bringing accelerator programming into the mainstream. Hence, GPUs are becoming a more and more attractive alternative to traditional CPUs, in particular for the more interesting ratio of cost-per-flop and watts-per-flop. However, efficient GPU programming remains a laborious challenge. In this paper, in order to exploit the computing power of GPUs, we have integrated a runtime system into the sparse direct solver. We divided the algorithm into computational tasks, and use a Directed Acyclic Graph (DAG) to represent the dependencies between these tasks. This DAG representation of an algorithm enables a clear separation between the flow of data between tasks and the data distribution. Then, a runtime system is used to schedule these tasks while respecting the dependencies. Since the runtime system offers a uniform programming interface for a specific subset of hardware or low-level software entities, applications can use these uniform programming interfaces in a portable manner. Furthermore, the runtime system can optimize application requests by dynamically mapping the tasks onto resources as efficiently as possible. The work presented in this paper is part of the MORSE project [1] to design dense and sparse linear algebra methods that achieve the fastest possible time to an accurate solution on large-scale heterogeneous multicore systems with GPU accelerators.

Concerning accelerator-based platforms for sparse direct solvers, a lot of attention has been recently paid to design new algorithms that can exploit the huge potential of GPUs. For a multifrontal sparse direct solver, some preliminary works have been proposed in the community [8], [9], resulting in single-GPU implementations based on off-loading parts of the computations to the GPU. The main idea is to treat some parts of the task dependency graph entirely on the GPU. Therefore, the main originality of these efforts was in the methods and algorithms used to decide whether or not a task can be processed on a GPU. In most cases this was achieved through a threshold based criterion on the size of the computational tasks. From the software point of view, most of these studies have only produced software prototypes and few sparse direct solvers exploiting GPUs have been made available to the users, most of them being developed by private software companies such as MatrixPro [2], Acceleware [3] and BCSLib-GPU [4]. As far as we know, there are no publications nor reports where the algorithmic choices are depicted. A recent progress towards a multifrontal sparse QR factorization on GPU have been presented in [10].

1. http ://icl.eecs.utk.edu/morse
2. http://www.matrixprosoftware.com/
3. http://www.acceleware.com/matrix-solvers
4. http://www.boeing.com/phantom/bcslib/

The rest of the paper is organized as follows : In the next section, we present the libraries and software used for our current study. There, both DAGUE and STARPU runtimes are introduced, and we summarize the main features of the PASTIX solver. Then, in Section III, we describe the algorithms and main ideas that have been implemented to optimize the task scheduling and the granularity of the sparse factorization. In Section IV, we present experiments we performed with our sparse direct solver to validate our approach on industrial challenging matrices. Finally, in Section V, we conclude with some prospects of the current work.

II. RUNTIME AND SOLVER

DAGUE [11] is a distributed runtime system designed to achieve extreme computational scalability by exploiting an algebraic representation of Direct Acyclic Graphs that efficiently captures the totality of tasks involved in a computation and the flow of data between them. Its primary goal is to maximize parallelism while automatically orchestrating task execution so as to minimize both communication and load imbalance. Unlike other available DAG-based runtimes, the concise symbolic representation of the algorithm that DAGUE uses minimizes the memory required to express the map of tasks ; at the same time, it provides extreme flexibility during the scheduling process. This algebraic representation allows the DAG to be traversed at very high speed, while tracking any flow of data from task to task. By combining this underlying mechanism with an understanding of the specific hardware capabilities of the target architecture, DAGUE is able to schedule tasks in ways that creatively adapt alternative work-stealing strategies to the unique features of the system. These capabilities enable DAGUE to optimize data movement between available computational resources, including both different nodes of the full system and different accelerators on the same node.

The DAGUE runtime aimed first at providing a scheduler for large distributed system of multicore nodes and is able to handle heterogeneous architectures to relocate the most compute intensive kernel on GPUs. A full description of DAGUE, and the implementation of classical linear algebra factorizations in this environment, can be found in [11], [12].

STARPU [13] is a software tool aiming to allow programmers to exploit the computing power of the available CPUs and GPUs, while relieving them from the need to specially adapt their programs to the target machine and processing units. The STARPU run-time supports a *task-based programming model*. Applications submit computational tasks, with CPU and/or GPU implementations, and StarPU schedules these tasks and associated data transfers on available CPUs and GPUs. The data that a task manipulates is automatically transferred among accelerators and the main memory, so that programmers are freed from the scheduling issues and technical details associated with these transfers. STARPU takes particular care of scheduling tasks efficiently, using well-known algorithms from the literature. In addition, it allows scheduling experts, such as compiler or computational library developers, to implement custom scheduling policies in a portable fashion.

The main differences between DAGUE and STARPU are the tasks submission process, the centralized scheduling and the data movement strategy. DAGUE uses its own parametrized language to describe the DAG in comparison to the simple sequential submission loops used by STARPU. Therefore STARPU relies on a centralized strategy which analyzes the dependencies between tasks and schedules these tasks on the available resources, while each computational unit of DAGUE immediately release the dependencies of the completed task solely using the local knowledge of the DAG. At last, STARPU scheduling strategy exploits cost models of the computation and data movements to schedule a task to the right resource (CPU or GPU) in order to minimize overall execution time. However it has no data movement policy on shared memory systems resulting in lower efficiencies when no GPUs are used compared to the data-reuse heuristic of DAGUE.

Hence, the research around STARPU has focused mainly on the case of an heterogeneous multicore node enhanced with multiple GPUs, while research around DAGUE has more focused on scalability issues on a large number of homogeneous nodes.

PASTIX is a scientific library that provides a high performance parallel direct solver for very large sparse linear systems. Numerical algorithms are implemented in single or double precision (real or complex) using LL^T or LDL^T factorizations for symmetric matrices, and LU factorizations with static pivoting for non symmetric matrices having symmetric patterns. Theres is a version of PASTIX for multicore node architectures, which uses a hybrid MPI-thread programming to fully exploit the advantage of shared memory and to reduce the memory overhead. Direct methods are numerically robust methods, but very large three dimensional problems may require a large amount of memory, even with any memory optimization. For this type of problems, PASTIX provides an adaptive blockwise incomplete factorization that is much more accurate (and numerically more robust) than the scalar incomplete factorizations which are commonly used as preconditioner for iterative solvers. Such incomplete factorization can take advantage of the latest breakthroughs in sparse blocked direct methods, and particularly should be very competitive in CPU time (effective power used from processors and good scalability), while avoiding the memory limitation encountered by direct methods.

III. ALGORITHMS

In this paper, we study and design algorithms and parallel programming models for implementing sparse supernodal direct methods on an emerging computers equipped with GPU accelerators. Our ultimate goal is to release a manycores (CPU+GPU) version of the PASTIX solver [5]. In this paper, we consider the Cholesky, LDL^T, and LU factorization algorithms which are already present in PASTIX.

The task dependencies of numerical factorization can be represented by a tree whose nodes represent computational

tasks and whose edges represent transfer of data between the tasks. The distributed memory version of PaStiX uses a right-looking formulation which, having computed the factorization of a column-block corresponding to a node of the tree, immediately sends the data to update the column-blocks corresponding to its ancestors in the tree. In the actual implementation, we locally aggregate contributions to the same block before sending the contributions. This can significantly reduce the number of messages, and is known to limit the memory overhead induced by the direct methods.

Either a static or dynamic scheduling of block computations can be used, independently from these different approaches described above. For homogeneous parallel architectures, it is useful to have an efficient static scheduling scheme. For the PaStiX solver, we have recently developed a dynamic scheduling scheme, specifically designed for modern supercomputers, that have many microprocessors, each of which consists of one or many computational cores, and induces a strong hierarchical topology. To address the needs of dynamic scheduling, the data structures of the solver, as well as the patterns of communication, have been also modified [6]. Thanks to these efforts on the multicore implementation [14], we have a platform ready to deal with bigger problem sizes on today's supercomputers.

The main contributions of our current paper can be subdivided into three subtasks described hereafter :

– **Kernels for sparse factorization**. Our first step of developing a sparse direct solver for clusters of multi-GPU nodes is to develop *supernodal computational kernels* designed specifically for this purpose. In particular, one of the most important kernels is the matrix/matrix-product involved in the updates of the trailing matrix. In the case of supernodes, data structures can be compacted in order to improve the efficiency of the computations, but we still need to improve the matrix-add operations that correspond with each block of the sparse updates. This kernel must also prefetch the data to minimize the data transfers on the GPU devices.

– **Scheduling of computations and data transfers**. The main aim of this paper is to evaluate the potential of replacing the current scheduler of PaStiX with two generic runtime frameworks DAGuE and StarPU for executing the task graph corresponding to a sparse factorization.

  In the current scheduling scheme of PaStiX, a task corresponds to a supernode (1D block distribution), see Fig. 1. To improve the efficiency of the sparse factorization on a multicore implementation, we introduced a way of controlling the granularity of the BLAS operations (referred to as an ESP option for Enhanced Sparse Parallelism). This functionality dynamically splits a single task of computing the contribution to the trailing submatrix, using the current panel into subtasks, so that the critical path of the algorithm can be reduced.

  In this paper, for both DAGuE and StarPU scheduling, one computational task corresponds to the computation of the contributions to a remaining column-block using a single block in a panel. Hence, the number of tasks to schedule is equal to the number of blocks in the factored matrix.

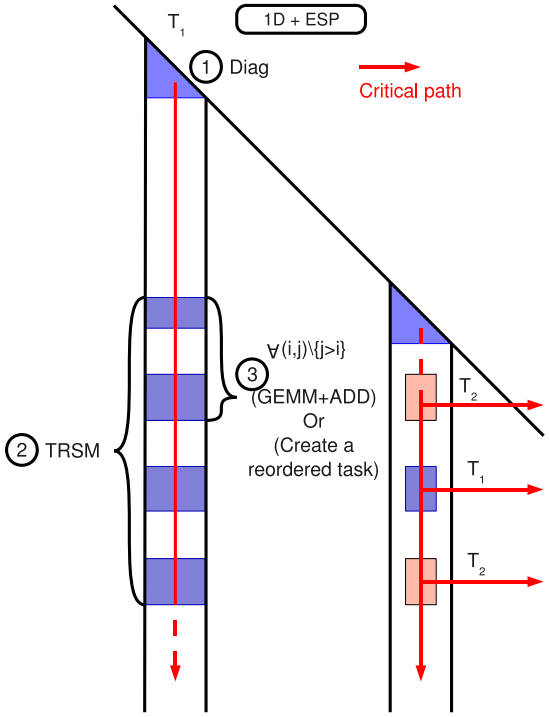
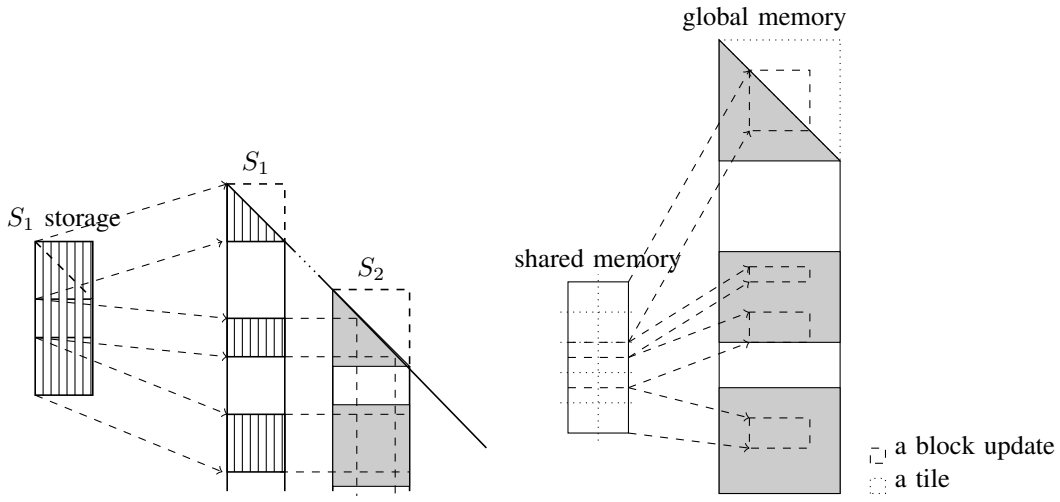


FIGURE 1: Task graph for supernodal factorization

– **Sparse algorithms**. In order to control the granularity of the computational tasks, the automatic criteria used to set adaptive block sizes has to be extended to heterogeneous architectures. Some references for benchmarking dense linear algebra kernels are described in [15] and show that efficiency could be obtained on GPU devices only on relatively large blocks – that can be found on top of the elimination tree. Similarly, the amalgamation algorithm [16], reused from the implementation of an incomplete factorization, is a crucial step to obtaining larger supernodes and efficiency on GPU devices. The default parameter for amalgamation has been slightly increased ; we allow up to 12% more fill-in to build larger blocks while keeping a good a high level of parallelism.

A. *Sparse* GEMM *on* GPU

Fig. 2a illustrates the data structure of PaStiX storing the sparse matrix. The nonzero entries are grouped into supernodes which are composed of several dense blocks. Each column block is stored as a dense block - represented by S_1 *storage* in the figure. When we update the column block S_2 facing the first extradiagonal block of S_1, all blocks of S_1 are included in the blocks of S_2. Thus, in PaStiX, we first factorize the diagonal block of S_1, and we then update the off-diagonal blocks of S_1. Next, for each extradiagonal block B_i of S_1, we compute the contribution $B_k B_i^T$ to the (k, i)-th block for



(a) Sparse structure of the matrix (b) Shared memory to global memory

FIGURE 2: Data management

$k > i$, and store the contribution in a temporary buffer. Finally, we update S_i for each extradiagonal block B_i. On a GPU, the computation is exactly the same except that our buffer is in the shared memory of the GPU.

To have an efficient usage of the GPU during sparse factorization, we wrote a new GPU kernel that takes the sparsity of our matrix into account. Our kernel is based on the framework proposed in [17], which also allows us to use an auto-tuning script to select near-optimal block sizes for the kernel. Specifically, this kernel computes $C = \alpha AB + \beta C$, where C is divided into a 2D grid of tiles (see Fig. 2b), and each thread block computes a tile of the matrix-multiplication, AB, which is stored in the shared GPU memory. Each thread is in charge of computing several entries of this tile. Each of these entries is computed one by one, and to compute each entry, the corresponding row of A and the corresponding column of B are loaded into the thread registers. Once the tile is computed, it is added into C. In the supernodal solver PASTIX, C may span several discontinuous supernodal blocks in a column-block (see Fig. 2b). Hence, during the summation of αAB into βC, we use an offset that specifies the row of C, to which the computed values of αAB should be accumulated into. For this, we added two pairs of integer arrays as input arguments to the kernel, which store the first and last row indexes of each blocks of A and C.

B. DAG *scheduling*

In this section, we describe the different approaches used to describe the DAGs in DAGUE and STARPU. For both codes, we define two kinds of tasks in our LL^T factorization :

- *panel(s)* : factorization of the diagonal block of the supernode s (POTRF) and update all the extradiagonal blocks of this supernode (TRSM) ;
- *gemm(b,s)* : perform the GEMM product of all the blocks beneath B_b in its supernode by B_b^T and subtract the result from the supernode s which has its diagonal block facing B_b.

For the LU factorization, we are using the same tasks with addition of the update of U in both kernels. The $panel$ task does one extra solve on the upper part, and the $gemm$ task does two matrix-matrix multiplications : one for the lower part and one for the upper part. Finally, the LDL^t factorization follows the same scheme as Cholesky with different kernels to integrate the storage and the computation of the diagonal. The three factorizations have then the same DAG representation with different kernels, LU having more dataflow to move the upper part.

1) JDF representation of a DAG: In DAGUE, the data distribution and dependencies are specified using the Job Data Flow (JDF) format. Fig. 3 shows our JDF representation of the sparse Cholesky factorization using the tasks `panel` and `gemm` described previously. The second one is based only on the block id parameter and computes internally the supernode (`fcblk`) in which to apply the update. On Line 2 of `panel(j)`, `cblknbr` is the number of block columns in the Cholesky factor. Once the j-th panel is factorized, the trailing submatrix can be updated using the j-th panel. This data dependency of the submatrix update on the panel factorization is specified on Line 6, where `firstblock` is the block index of the j-th diagonal block, and `lastblock` is the block index of the last block in the j-th block column. The output dependency on Line 7 indicates that the j-th panel is written to memory at the completion of the panel factorization. The input dependency of the j-th panel factorization is specified on Lines 4 and 5, where `leaf` is *true* if the j-th panel is a leaf in the elimination-tree, and `lastbrow` is the index of the last block updating the j-th panel. Hence, if the j-th panel is a leaf, the panel is read from memory. Otherwise, the panel is passed in as the output of the last update on the panel.

Similarly, `gemm(k)` updates the `fcblk`-th block column

```
1. panel(j) [high_priority = on]
2. j = 0 .. cblknbr-1
3. ... set up parameters for the j-th task ...
4.  :A(j)
5. RW A ← ( leaf ) ? A(j) : C gemm( lastbrow )
6.          → A gemm(firstblock+1..lastblock)
7.          → A(j)
```

(a) Panel factorization

```
1. gemm(k)
2. k = 0 .. blocknbr-1
3. ... set up parameters for the k-th task ...
4.  :A(fcblk)
5. READ A ← diag ? A(fcblk) : A panel(cblk)
6. RW C ← first ? A(fcblk) : C gemm( prev )
7.          → diag ? A(fcblk)
8.          → (( !diag) && (next == 0)) ? A panel( fcblk )
9.          → (( !diag) && (next != 0)) ? C gemm( next )
```

(b) Trailing submatrix update

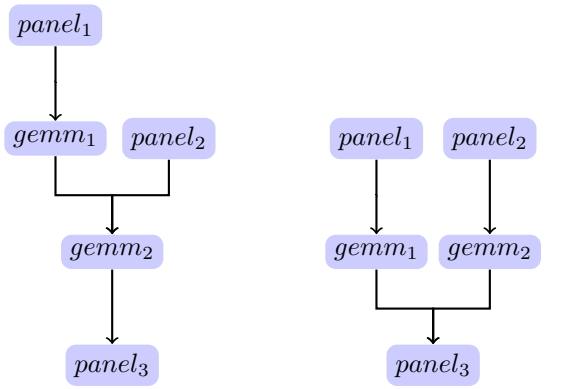
FIGURE 3: JDF representation of Cholesky.

using the k-th block, where `fcblk` is the index of the block row that the k-th block belongs to, and `blocknbr` on Line 2 is the number of blocks in the Cholesky factor. The input dependencies of `gemm` are specified on Lines 4 through 6, where the `cblk`-th panel `A` is being used to update the `fcblk`-th column `C`. Specifically, on these lines, `diag` is *true* if the k-th block is a diagonal block, and it is *false* otherwise, and `prev` is *false* if the k-th block is the first block in the `fcblk`-th block row, and it is the index of the block in the block row just before the k-th block otherwise. Hence, the `prev`-th block updated the `fcblk`-th column just before the k-th block does. Hence, the data dependency of `gemm(k)` is resolved once the `cblk`-th panel is factorized, and the `fcblk`-th column is updated using the `prev`-th block. Notice that the diagonal blocks are not used to update the trailing submatrix, but it is included in the code to have a continuous space of execution for the task required by DAGUE. Finally, Lines 7 through 9 specify the output dependencies of `gemm(k)`, where `next` is *false* if the k-th block is a diagonal block, and it is the index of the next block after the k-th block in the `fcblk`-th row otherwise. Hence, the completion of `gemm(k)` resolves the data dependency of the `fcblk`-th panel factorization if this is the last update on the panel, or it resolves the dependency of updating the `fcblk`-th block column using the `next`-th block otherwise.

2) STARPU *tasks submission:* STARPU builds its DAG following the tasks ordering provided by the user and by using data dependencies. The following pseudocode shows the STARPU tasks submission loop for the LL^T decomposition.

```
for all Supernode S_1 do
    submit_panel(S_1) {update of the panel}
    for all extra diagonal block b_i of S_1 do
        S_2 ← supernode_in_front_of(B_i)
        submit_gemm(S_1, S_2) {sparse GEMM B_{k,k≥i} × B_i^T substracted from S_2}
    end for
end for
```

By default, STARPU generates the tasks graph following sequential consistency (Fig. 4a). Since the order in which the facing supernode receives contribution is not relevant in our factorization, we disabled STARPU sequential consistency (Fig. 4b). Then, we notify the scheduler that a *panel* task



(a) with sequential consistency (b) without sequential consistency

FIGURE 4: Task graph

depends on all the GEMM updates on the supernode and that this task triggers the GEMM on the trailling supernodes.

We are not using a reduction on the target panel, because for each update only a subarea is impacted. Moreover, a reduction requires a copy of the destination supernode on each computing device, that could represent a lot of memory overhead. This allows only an out of order execution of the updates while guarantying the mutual exclusion of the computations. DAGUE in its actual release doesn't allow those reduction operations and keep the order defined by the dependencies. As the order of the reduction is imposed by the data dependencies, the potential concurrency is reduced but the accuracy of the numerical results is maintained between successive runs.

C. Sparse mapping

Our mapping algorithm is based on a static scheduling based on a performance model. Thus, the partitioning and mapping step generates a fully ordered schedule that can be used in the parallel factorization. This schedule aims at statically regulating all of the issues that are classically managed at runtime. To make our scheme very reliable, we estimate the workload and message passing latency by using a BLAS and communication network time model, which is automatically calibrated on the target architecture.

Unlike usual algorithms, our partitioning and distribution strategy is divided into two distinct phases. The partitioning

algorithm is based on a recursive top-down strategy over the block elimination tree provided by block symbolic factorization. Pothen and Sun presented such a strategy in [18]. The partition phase splits column-blocks associated with large supernodes, and then for each column-block, it builds a set of candidate threads for its mapping. Once the partitioning step is over, the task graph is built. In this graph, each task is associated with the set of candidate threads for its column-block. The mapping and scheduling phase then try to optimally map each task onto one of these sets of threads. An important constraint is that once a task has been mapped to a thread, then all the data accessed by this thread are also mapped on the process associated with the thread. This means that an unmapped task that accesses a block that has already been mapped should be mapped on the same thread to preserve the data locality.

IV. Results

To compare the factorization times of PaStiX using generic runtime systems DAGuE and StarPU, with that using the internal scheduler of PaStiX, we present experimental results on challenging matrices from industrial applications. Some properties of the three different test matrices used for our scaling studies are shown in Table I. For all three approaches, the same partitioning and mapping were used, while the fully ordered schedule was used only for experiments using the internal scheduler of PaStiX.

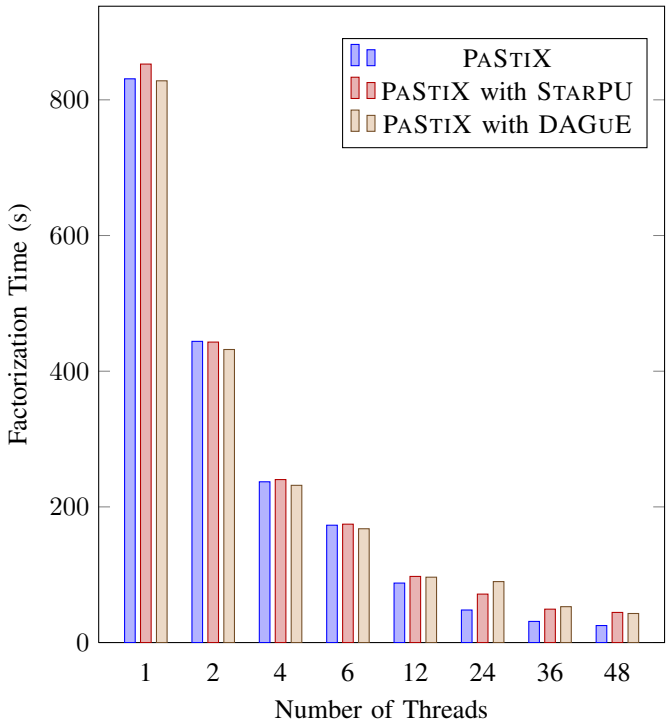


Figure 5: LL^T decomposition on Audi (double precision)

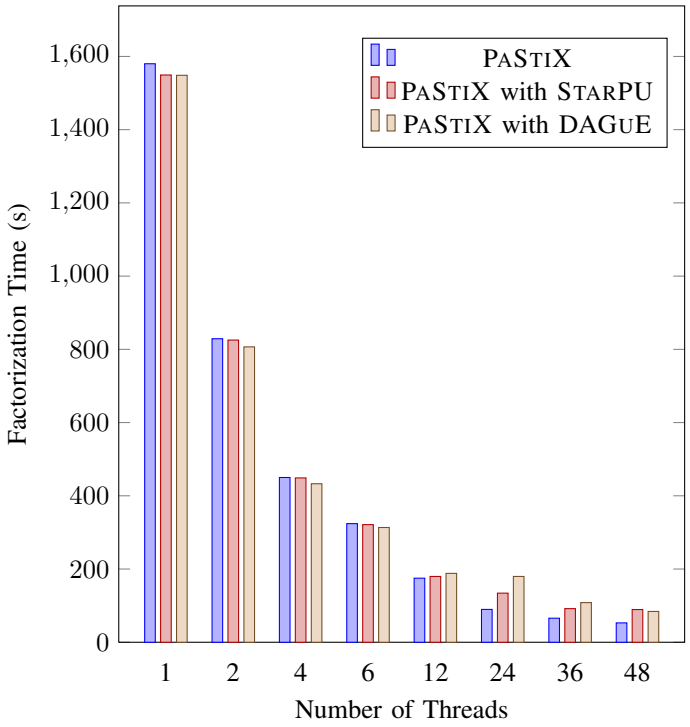


Figure 6: LU decomposition on MHD (double precision)

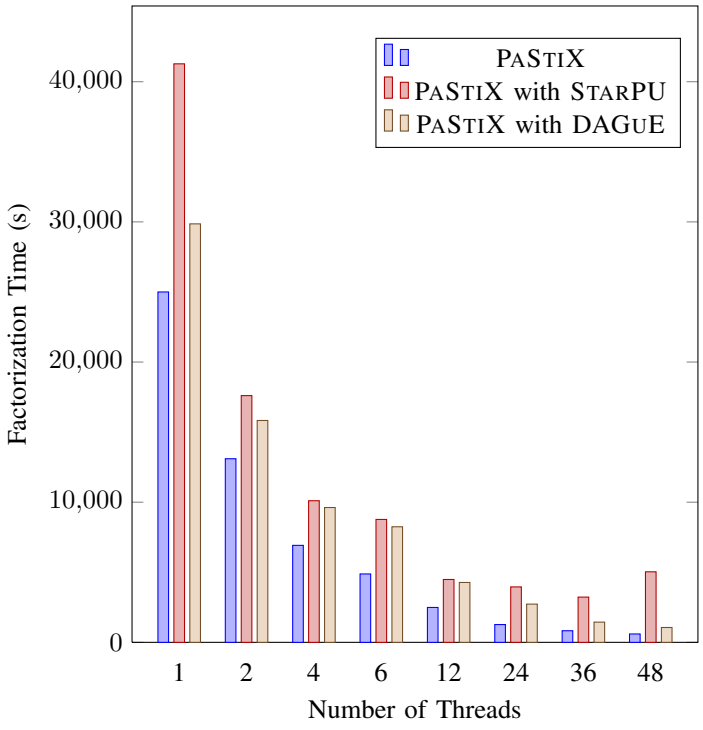


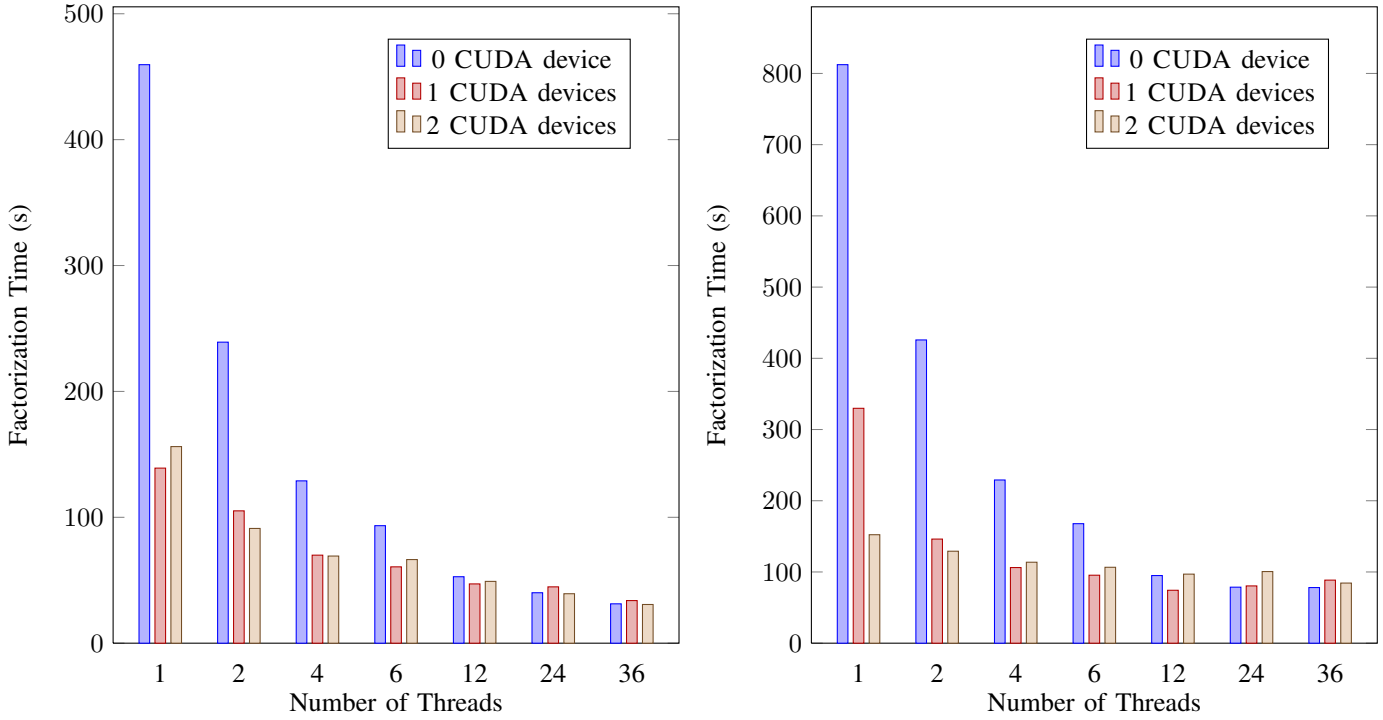
Figure 7: LDL^T decomposition on 10M (double complex)

A. Experiments on multicores

The first set of experiments were conducted on the Romulus machine at the university of Tennessee. Romulus has four twelve-core AMD Opteron 6180 SE CPUs (2.5 GHz) with

Name	N	NNZ_A	Fill ratio	OPC	Type	Factorization	Source
MHD	485,597	12,359,369	61.20	9.84e+12	Real	LU	University of Minnesota
Audi	943,695	39,297,771	31.28	5.23e+12	Real	LL^T	PARASOL Collection
10M	10,423737	89,072,871	75.66	1.72e+14	Complex	LDL^T	French CEA-Cesta

TABLE I: Matrices description



(a) Audi LL^T decomposition

(b) MHD LU decomposition

FIGURE 8: GPU results on Romulus (single precision)

256GB of total RAM, and is equipped with two Tesla T20 GPUs.

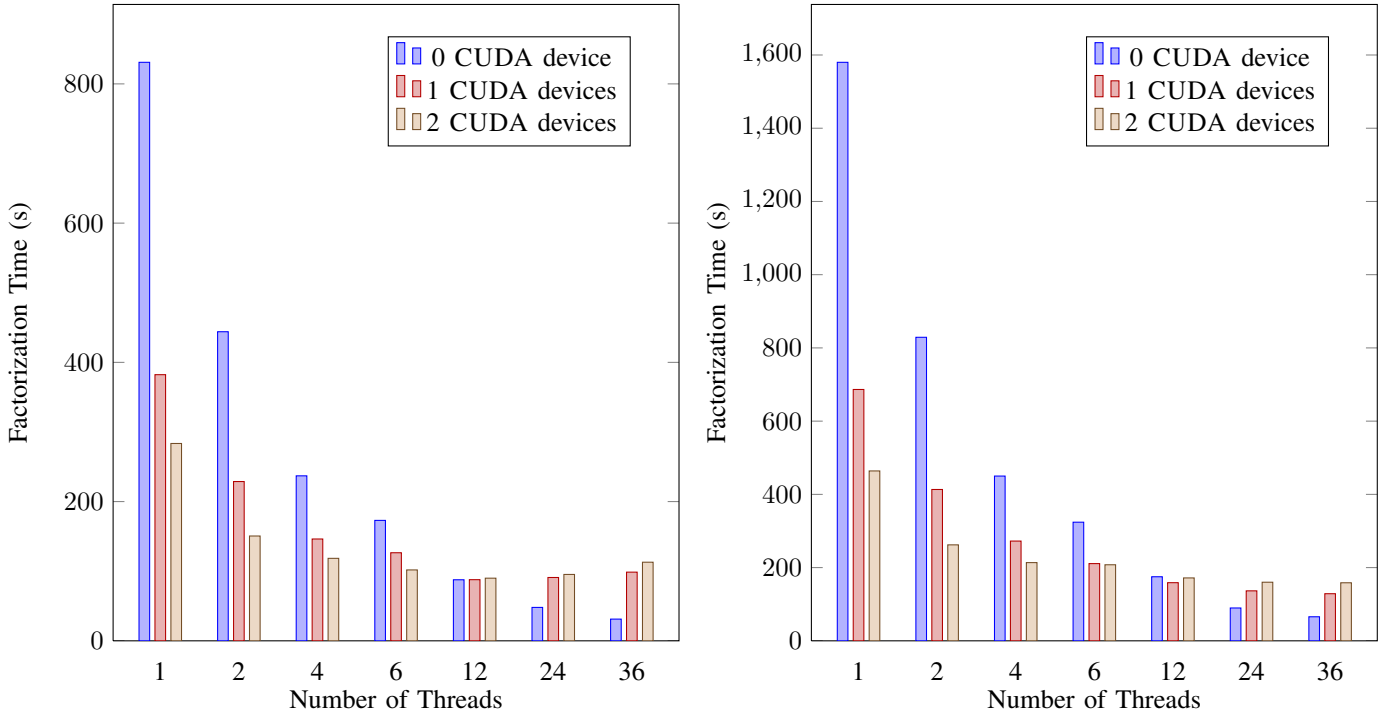
Fig. 5 and 6 compare the performance of PASTIX using DAGUE and STARPU runtimes, with PASTIX using its original scheduler on Audi and MHD test matrices. A good scalability was obtained using all three approaches. On these two test cases, generic schedulers behaved quite well, and obtained the performances similar to the fine-tuned PASTIX scheduler. On a small number of processors, we could even obtain better results using the generic schedulers. DAGUE lost some performance when more than a socket (12 cores) was used, but recovered as soon as the computation spawned over more than 2 sockets. The DAGUE scheduler seemed to not be able to extract any performance gain between 12 and 24 cores.

With the *10 Millions* test matrix (Fig. 7), the finely-tuned scheduler of PASTIX outperformed the generic runtimes. Specifically, STARPU showed its limitation as the number of threads increased, while DAGUE maintained a good scalability. This can be explained by the fact that STARPU does not take data locality into account.

B. Experiments with GPUs

Fig. 8 and 9 show the results of using one and two GPUs. For these experiments, we did not use all 48 available cores for computation as one core was dedicated to each GPU. The CUDA kernel gave good acceleration on both single and double precision. The factorization time was reduced significantly using the first GPU when the number of cores was small. The speedups of up to 5 was obtained using one GPU. However, the second GPU was relevant only with a small number of cores (less than 4 threads in single precision (Fig. 8), and less than 12 threads in double precision (Fig. 9)). With one GPU, once the cores on a socket (12 core on Romulus, and 6 on Mirage) were fully utilized, the GPU had no effect on the factorization time.

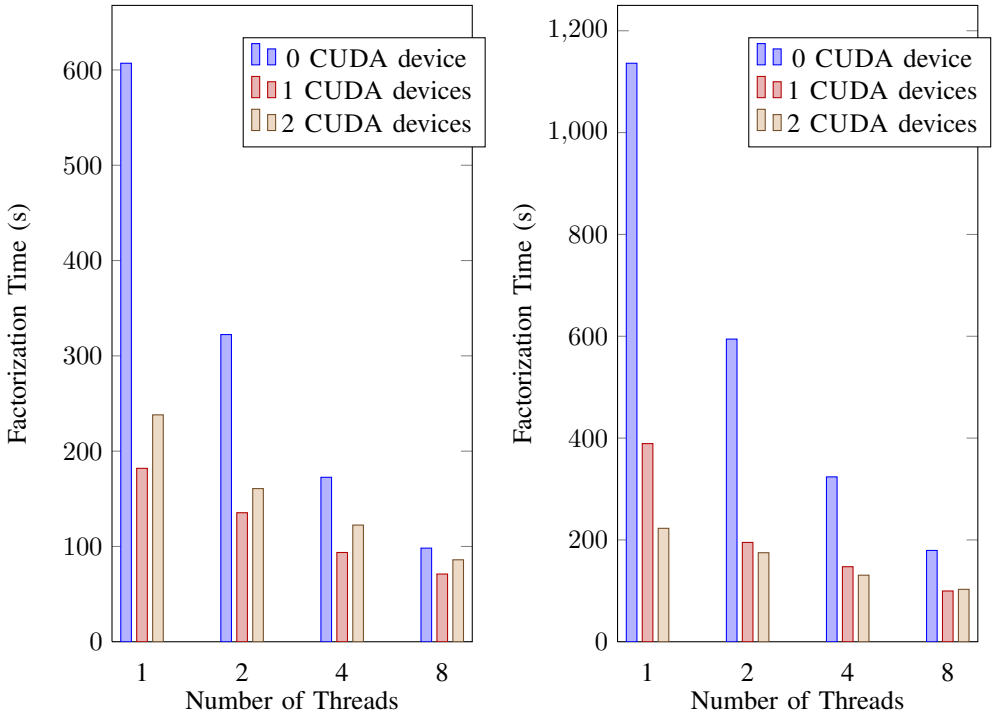
We also conducted additional GPU experiments using a compute node of the Mirage machine from INRIA - Bordeaux. Mirage nodes are composed of 2 Hexa-core (Westmere Intel® Xeon® X5650), with 36GB of RAM. The results on Mirage (Fig. 10) were similar to the ones on Romulus.



(a) Audi LL^T decomposition

(b) MHD LU decomposition

FIGURE 9: GPU results on Romulus (double precision)



(a) Audi LL^T decomposition

(b) MHD LU decomposition

FIGURE 10: GPU results on Mirage (double precision)

V. Conclusion

In this paper, we examined the potential benefits of using generic runtime systems, DAGuE and StarPU, in a parallel sparse direct solver PaStiX. The experimental results using up to 48 cores and two NVIDIA Tesla GPUs demonstrated the potential of this approach to design a sparse direct solver on heterogeneous manycore architectures with accelerators using a uniform interface.

Through the study presented in this paper, we have identified three potential research path, which we plan to continue to investigate in the future.

First, in order to minimize the overhead induced by the scheduler, we need to increase the granularity of the tasks at the bottom of the elimination tree. These leaves or subtrees may be merged into bigger tasks to achieve this goal.

Second, we would like to pursue a similar experiment in distributed heterogeneous environments, composed by many-cores nodes with multiGPUs. On such an platform, when a supernode update another non-local supernode, the update blocks are stored in a local extra-memory space (this is called "fan-in" approach). In order to reduce communication time overhead, we delay sending these updates until the last updates to the supernodes are accumulated, trading latency for bandwidth. We will study potential approaches to implement such an challenging optimization using generic runtime systems, StarPU and DAGuE.

More specifically, in the context of StarPU, for successive solution steps, the performance models used by our direct solver for the static mapping and scheduling step, could benefit from an online model refined during the execution.

Finally, the availability of extra computational resources, highlight the potential to dynamically build or rebuild the supernodal structures according to the load on the cores and the GPUs. A first approach will be to adjust the task granularity at runtime. Simultaneously, we will work on the distributed version of our solver, and on the challenging problem of defining an initial mapping of the data compatible with heterogeneous capabilities of the distributed memory architectures.

Acknowledgment

The authors would like to thank the DAGuE and StarPU teams for their support and assistance with this project. Special thanks also go to Sam Crawford for the valuable comments on our submitted manuscript and to Abdou Guermouche for his advice.

Références

[1] I. S. Duff, A. M. Erisman, and J. K. Reid, "Direct methods for sparse matrices," *Oxford University Press*, London 1986.

[2] A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng, "Sparse Cholesky factorization on a local memory multiprocessor," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, pp. 327–340, 1988.

[3] A. George and J. W.-H. Liu, *Computer solution of large sparse positive definite systems*. Prentice Hall, 1981.

[4] 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 *Irregular'2000*, ser. LNCS, vol. 1800, Cancun, Mexique, May 2000, pp. 519–525.

[5] ——, "PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems," *Parallel Computing*, vol. 28, no. 2, pp. 301–321, Jan. 2002.

[6] M. Faverge and P. Ramet, "Dynamic scheduling for sparse direct solver on NUMA architectures," in *Proceedings of PARA'2008*, Trondheim, Norway, May 2008.

[7] ——, "Fine grain scheduling for sparse solver on manycore architectures," in *15th SIAM Conference on Parallel Processing for Scientific Computing*, Savannah, USA, Feb. 2012.

[8] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury, "Multifrontal Factorization of Sparse SPD Matrices on GPUs," *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 372–383, May 2011.

[9] C. D. Yu, W. Wang, and D. Pierce, "A CPU-GPU Hybrid Approach for the Unsymmetric Multifrontal Method," *Parallel Computing*, vol. 37, no. 12, pp. 759–770, Oct. 2011.

[10] T. Davis, "Multifrontal sparse qr factorization : Multicore, and gpu work in progress," in *15th SIAM Conference on Parallel Processing for Scientific Computing*, Savannah, USA, Feb. 2012.

[11] G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra, "DAGuE : A generic distributed DAG engine for High Performance Computing," *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[12] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'11)*, 2011.

[13] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU : A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency Computat. Pract. Exper.*, 2010, (to appear).

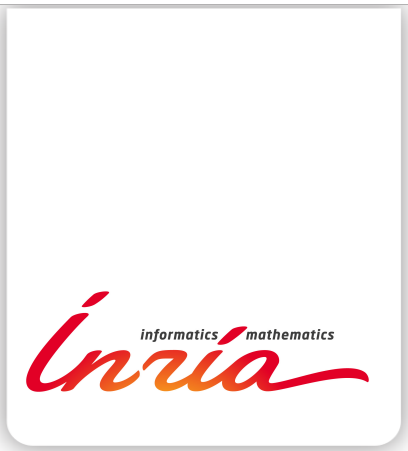
[14] P. Hénon, P. Ramet, and J. Roman, "On using an hybrid MPI-Thread programming for the implementation of a parallel sparse direct solver on a network of SMP nodes," in *PPAM'05*, ser. LNCS, vol. 3911, Poznan, Pologne, Sep. 2005, pp. 1050–1057.

[15] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *Supercomputing'08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, no. November, 2008.

[16] P. Hénon, P. Ramet, and J. Roman, "On finding approximate supernodes for an efficient ILU(k) factorization," *Parallel Computing*, vol. 34, pp. 345–362, 2008.

[17] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning gemm kernels for the fermi gpu," *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[18] A. Pothen and C. Sun, "A mapping algorithm for parallel sparse Cholesky factorization," *SIAM J. Sci. Comput.*, vol. 14(5), pp. 1253–1257, Sep. 1993.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

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

ISSN 0249-6399