Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers

Luka Stanisic, Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Arnaud Legrand, Florent Lopez, Brice Videau

Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers

Luka Stanisic*, Emmanuel Agullo†, Alfredo Buttari‡, Abdou Guermouche†,
Arnaud Legrand*, Florent Lopez‡, Brice Videau*

*CNRS/Inria/University of Grenoble Alpes, France; firstname.lastname@imag.fr
†Inria/University of Bordeaux, France; firstname.lastname@labri.fr
‡CNRS/University Paul Sabatier, Toulouse, France; firstname.lastname@irit.fr

***Abstract*—The ever growing complexity and scale of parallel architectures imposes to rewrite classical monolithic HPC scientific applications and libraries as their portability and performance optimization only comes at a prohibitive cost. There is thus a recent and general trend in using instead a modular approach where numerical algorithms are written at a high level independently of the hardware architecture as Directed Acyclic Graphs (DAG) of tasks. A task-based runtime system then dynamically schedules the resulting DAG on the different computing resources, automatically taking care of data movement and taking into account the possible speed heterogeneity and variability. Evaluating the performance of such complex and dynamic systems is extremely challenging especially for irregular codes. In this article, we explain how we crafted a faithful simulation, both in terms of performance and memory usage, of the behavior of `qr_mumps`, a fully-featured sparse linear algebra library, on multi-core architectures. In our approach, the target high-end machines are calibrated only once to derive sound performance models. These models can then be used at will to quickly predict and study in a reproducible way the performance of such irregular and resource-demanding applications using solely a commodity laptop.**

I. Introduction

To answer the ever increasing degree of parallelism required to process extremely large academic and industrial problems, the High Performance Computing (HPC) community is facing a threefold challenge. First, at large scale, **algorithms** [1], [2], [3] that used to be studied mostly from a mathematical point of view or for specific problems are becoming more and more popular and practical for applications in many scientific fields. The common property of these algorithms is their **very irregular pattern**, making their design a strong challenge for the HPC community. Second, the **diversity of hardware architectures** and their respective complexity make it another challenge to design codes whose performance is portable across architectures. Third, **evaluating their performance** subsequently also becomes highly challenging.

Until a few years ago, the dominant trend for designing HPC libraries mainly consisted of designing scientific software as a single whole that aims to cope with both the algorithmic and architectural needs. This approach may indeed lead to extremely high performance because the developer has the opportunity to optimize all parts of the code, from the high level design of the algorithm down to low level technical details such as data movements. But such a development often requires a tremendous effort, and is very difficult to maintain. Achieving portable and scalable performance has become extremely challenging, especially for irregular codes.

To address such challenge, a modular approach can be employed. First, numerical algorithms are written at high level, independently of the hardware architecture, as a Directed Acyclic Graph (DAG) of tasks (or kernels) where each vertex represents a computation task and each edge represents a dependency between tasks, possibly involving data transfers. A second layer is in charge of scheduling the DAG, i.e., of deciding when and where to execute each task. In the third layer, a runtime engine takes care of implementing the scheduling decisions, of retrieving the data necessary for the execution of a task (taking care of ensuring data coherency), of triggering its execution and of updating the state of the DAG upon task completion. The fourth layer consists of the tasks code optimized for the target architecture. In most cases, the bottom three layers need not to be written by the application developer since most runtime systems embed off-the-shelf generic scheduling algorithms that efficiently exploit the target architecture.

In this article, we address the third aforementioned challenge related to the performance evaluation of complex algorithms implemented over dynamic task-based runtime systems. A previous study [4] has shown that the performance of regular algorithms (such as those occuring in dense linear algebra) on hybrid multi-core machines could be accurately predicted through simulation. We show that it is also possible to conduct faithful simulation of the behavior of an *irregular* fully-featured library both in terms of *performance* and *memory* on multi-core architectures. To this end, we consider the `qr_mumps` [5] (see Section III-A for more details) sparse direct solver which implements a multifrontal QR factorization (a highly irregular algorithm) on top of the StarPU [6] runtime system and simulate part of the execution with the SimGrid [7] simulation engine (see Section III-B).

The rest of the paper is organized as follows. Section II provides some related work on the use of task-based runtime systems in particular in the context of dense and sparse linear algebra solvers. Very few propose a fully integrated simulation mode allowing to predict performance. Then we present in Section III and IV the internals of `qr_mumps` and how it has been modified to run on top of SimGrid. In Section V, we

explain how we carefully modeled the different components of `qr_mumps`. Finally we present in Section VI and VII numerous and detailed experimental results that assess how simulation predictions faithfully match with real experiments. We conclude this article in Section VIII with a presentation of current limitations of our work and an opening on envisioned future work.

II. Related Work

A. Task-based runtime systems and the Sequential Task Flow model

Whereas task-based runtime systems were mainly research tools in the past years, their recent progress makes them now solid candidates for designing advanced scientific software as they provide programming paradigms that allow the programmer to express concurrency in a simple yet effective way and relieve him from the burden of dealing with low-level architectural details.

The *Sequential Task Flow* (STF) model simply consists of submitting a sequence of tasks through a non blocking function call that delegates the execution of the task to the runtime system. Upon submission, the runtime system adds the task to the current DAG along with its dependencies which are automatically computed through data dependency analysis [8]. The actual execution of the task is then postponed to the moment when its dependencies are satisfied. This paradigm is also sometimes referred to as *superscalar* since it mimics the functioning of superscalar processors where instructions are issued sequentially from a single stream but can actually be executed in a different order and, possibly, in parallel depending on their mutual dependencies.

Recently, the runtime system community has delivered more and more reliable and effective task-based runtime systems [6], [9], [10], [11], [12] (a complete review of them is out of the scope of this paper) supporting this paradigm up to the point that the OpenMP board has included similar features in the latest OpenMP standard 4.0 (e.g., the `task` construct with the recently introduced `depend` clause). Although this OpenMP feature is already available on some compilers (`gcc` and `gfortran`, for instance) we chose to rely on StarPU as it provides a very wide set of features that allows for a better control of the task scheduling policy and because it supports accelerators and distributed memory parallelism which will be addressed in future work.

B. Task-based linear algebra solvers

The dense linear algebra community has strongly adopted such a modular approach over the past few years [9], [13], [14], [15] and delivered production-quality software relying on it. The quality improvement of runtime systems makes it possible to design more irregular algorithms such as sparse direct methods with this approach.

The PaSTIX solver, has been extended [16] to rely on either the StarPU or PaRSEC runtime systems for performing supernodal Cholesky and LU factorizations on multi-core architectures possibly equipped with GPUs. Kim *et al.* [17] presented a DAG-based approach for sparse LDL^T factorizations where OpenMP tasks are used and dependencies between nodes of the elimination tree are implicitly handled through a recursive submission of tasks, whereas intra-node dependencies are essentially handled manually. The baseline code used in this article is a QR multifrontal solver running on top the StarPU runtime system [5].

Interestingly, to better extract potential parallelism, these solvers (e.g., PaSTIX [16] and `qr_mumps` [18]) were already often designed with, to some extent, the concept of *task* before having in mind the goal of being executed on top of a runtime system. It is also the case of the SuperLU [19] supernodal solver. This design offered the opportunity to produce an advanced mechanism for simulating its performance [20]. The present work however differs from [20] mostly by the fact that the execution of `qr_mumps` is dynamic and deeply non-deterministic, which greatly complicates its performance evaluation.

III. Presentation of SimGrid/StarPU/`qr_mumps`

A. `qr_mumps`

The multifrontal method, introduced by Duff and Reid [21] as a method for the factorization of sparse, symmetric linear systems, can be adapted to the QR factorization of a sparse matrix thanks to the fact that the R factor of a matrix A and the Cholesky factor of the normal equation matrix A^TA share the same structure under the hypothesis that the matrix A is *Strong Hall*. As in the Cholesky case, the multifrontal QR factorization is based on the concept of *elimination tree* introduced by Schreiber [22] expressing the dependencies between elimination of unknowns. Each vertex f of the tree is associated with k_f unknowns of A. The coefficients of the corresponding k_f columns and all the other coefficients affected by their elimination are assembled together into a relatively small dense matrix, called *frontal matrix* or, simply, *front*, associated with the tree node (see Figure 2). An edge of the tree represents a dependency between such fronts. The elimination tree is thus a topological order for the elimination of the unknowns; a front can only be eliminated after its children. We refer to [18], [23], [24] for further details on high performance implementation of multifrontal QR methods.

The multifrontal QR factorization then consists in a tree traversal following a **topological order** (see line 1 in Figure 1 (left)) for eliminating the fronts. First, the **activation** (line 3) allocates and initializes the front data structure. The front can then be **assembled** (lines 5-12) by stacking the matrix rows associated with the k_f unknowns with uneliminated rows resulting from the processing of child nodes. Once assembled, the k_f unknowns are eliminated through a **complete** QR **factorization** of the front (lines 14-21); for these last two operations we assume that a 1D, block-column, partitioning is applied to the fronts. This produces k_f rows of the global R factor, a number of Householder reflectors that implicitly represent the global Q factor and a *contribution block* formed by the remaining rows. These rows will be assembled into the

Sequential version

```
forall fronts f in topological order
   ! allocate and initialize front
   call activate(f)

   forall children c of f
      forall blockcolumns j=1...n in c
         ! assemble column j of c into f
         call assemble(c(j), f)
      end do
      ! Deactivate child
      call deactivate(c)
   end do

   forall panels p=1...n in f
      ! panel reduction of column p
      call panel(f(p))
      forall blockcolumns u=p+1...n in f
         ! update of column u with panel p
         call update(f(p), f(u))
      end do
   end do
end do
```

STF version

```
forall fronts f in topological order
   ! allocate and initialize front
   call submit(activate, f:RW, children(f):R)

   forall children c of f
      forall blockcolumns j=1...n in c
         ! assemble column j of c into f
         call submit(assemble, c(j):R, f:RW)
      end do
      ! Deactivate child
      call submit(deactivate, c:RW)
   end do

   forall panels p=1...n in f
      ! panel reduction of column p
      call submit(panel, f(p):RW)
      forall blockcolumns u=p+1...n in f
         ! update of column u with panel p
         call submit(update, f(p):R, f(u):RW)
      end do
   end do
end do
call wait_tasks_completion()
```

Figure 1. Sequential version (*left*) and corresponding STF version from [5] (*right*) of the multifrontal QR factorization with 1D partitioning of frontal matrices.

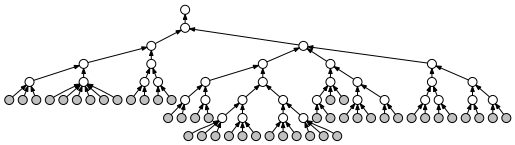

Figure 2. Typical elimination tree: each node corresponds to a front and the resulting tree is traversed from the bottom to the top. To reduce the overhead incurred by managing a large number of fronts, subtrees are pruned and aggregated into optimized sequential tasks (`Do_subtree`) depicted in gray.

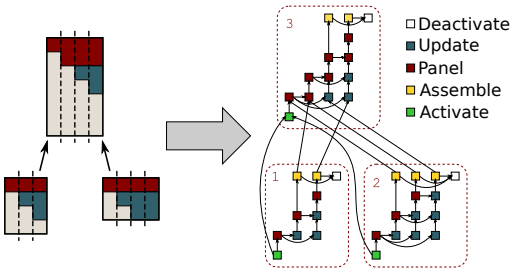

Figure 3. The STF code allows to express the complex dependencies induced by the staircase. The resulting DAG is dynamically scheduled by the runtime system.

parent front together with the contribution blocks from all the sibling fronts.

The multifrontal method provides two distinct sources of concurrency: tree and node parallelism. The first one stems from the fact that fronts in separate branches are independent and can thus be processed concurrently; the second one from the fact that, if a front is large enough, multiple threads can be used to assemble and factorize it. Modern implementations exploit both sources of concurrency which makes scheduling difficult to predict, especially when relying on dynamic scheduling which is necessary to fully exploit the parallelism delivered by such an irregular application.

One distinctive feature of the multifrontal QR factorization is that frontal matrices are not entirely full but, prior to their factorization, can be permuted into a staircase structure that allows for moving many zero coefficients in the bottom-left corner of the front and for ignoring them in the subsequent computation. Although this allows for a considerable saving in the number of operations, it makes the workload extremely irregular and the cost of kernels extremely hard to predict even in the case where a regular partitioning is applied to fronts, which makes it challenging to model.

Figure 1 (right) shows the STF version from [5] (used in the present study) of the multifrontal QR factorization described above. Instead of making direct function calls (`activate`, `assemble`, `deactivate`, `panel`, `update`), the equivalent STF code submits the corresponding tasks (see Figure 3). Since the data onto which these functions operate as well as their access mode (Read, Write or Read/Write) are also specified, the runtime system can perform the superscalar analysis while the submission of task is progressing. For instance, because an `assemble` task accesses a block-column `f(i)` before a `panel` task accesses the same block-column in Write mode, a dependency between those two tasks is inferred. Figure 3 shows (on the right side) the resulting DAG associated with a small, artificial, elimination tree (on the left side). It must be noted that in real life problems the elimination tree commonly contains thousands of nodes and the associated DAG hundreds of thousands of tasks.

Because the number of fronts in the elimination tree is commonly much larger than the number of resources, the partitioning is not applied to all of them. A *logical tree pruning* technique [18] is used similar to what proposed by Geist and Ng [25]. Through this technique, a layer in the elimination tree is identified such that each subtree rooted at this layer is treated in a single task with a purely sequential code. This new type of tasks, which are named `do_subtree`, is represented in Figure 2 as the gray layer.

B. StarPU/SimGrid

In [4], we explained how we crafted a coarse-grain hybrid simulation/emulation of StarPU [6], a dynamic runtime system for heterogeneous multi-core architectures, on top of SimGrid [7], a simulation toolkit initially designed for distributed system simulation. In our approach, among the four software layers mentioned in Section~ III-B (high-level DAG description, scheduling, runtime and kernels), the first three ones are emulated in the SimGrid framework. Their actual code is indeed executed except that a few primitives (such a thread synchronizations) are handled by SimGrid. In the fourth layer, the task execution is simulated. An initial calibration of the target machine is run once to derive performance profiles and models (interconnect topology, data transfers, computation kernels) that are given as an input to SimGrid. Subsequent simulations can then be performed in a reproducible way on personal commodity laptops without requiring any further access to the target machines. We showed in [4] how this approach enables to obtain accurate performance prediction of

two dense linear algebra applications on a wide variety of hybrid (comprising several CPUs and GPUs) architectures. Our extensive study demonstrated that non-trivial scheduling and load balancing properties can be faithfully studied with such approach. It has also been used to study subtle performance issues that would be quite difficult and costly to evaluate using solely real experiments [26].

IV. Challenges for porting qr_mumps on top of SimGrid

The extension of this methodology to an highly irregular algorithm such as the multifrontal QR method is not immediate because the fourth layer (kernel execution) is much harder to simulate. When working with dense matrices, it is common to use a global fixed block size and a given kernel type (e.g., dgemm) is therefore always called with the same parameters throughout the execution, which makes its duration on the same processing unit very stable and easy to model. On the contrary, in the qr_mumps factorization, the amount of work that has to be done by a given kernel greatly depends on its input parameters. Furthermore, these parameters may or may not be explicitly given to the StarPU runtime in the original qr_mumps code.

The challenges for designing a reliable simulation of qr_mumps thus consists in identifying which parameters significantly impact the task durations and propagating these information to SimGrid. Some parts of the StarPU code responsible for interacting with SimGrid were also modified to detect specific parametric kernels and predict durations based on the captured parameter values. We have also extended the StarPU tracing mechanism so that such kernel parameters are traced as well, which is indispensable to obtain traces that can be both analyzed and compared between real and simulated executions.

V. Modeling qr_mumps Kernels

We conducted an in-depth analysis of the temporal behavior of the different kernels invoked in qr_mumps. We discovered and assessed that Panel and Update have a similar structure and that the only parameters that influence their duration can be obtained from the corresponding task parameters and therefore determined by inspecting the matrix structure, without even doing the factorization for real. This eases the modeling of such kernels since such information can be used to define the region in which such parameters evolve and to design an informed experiment design to characterize the performance of the machine.

The other three kernels (Do_subtree, Activate and Assemble) show different behavior and the explaining parameters were more difficult to identify. In the following we describe our modeling choices for each kernel, detailing how and why particular parameters proved to be crucial.

A. Simple Negligible Kernels

As illustrated by Figure 8 (top-right histogram), the Deactivate kernel has a very simple duration distribution. We remind that this kernel is responsible solely for deallocating the memory at the end of the matrix block factorization. It is thus not surprising that these tasks are very short (a few milliseconds) and are negligible compared to the other kernels. Furthermore, there are only a few instances of such tasks even for large matrices and the cumulative duration of the Deactivate kernels is thus generally less than 1% of the overall application duration (makespan). Therefore, we decided to simply ignore this kernel in the simulation, injecting zero delay whenever it occurs. So far, this simplification has not endangered the accuracy of our simulation tool.

B. Parameter Dependent Kernels

Certain qr_mumps kernels (Panel and Update) are mostly wrappers of LAPACK/BLAS routines in which the vast majority of the total execution time is spent. Their duration depends essentially on their input arguments, which define the geometry of the submatrix on which the routines work. Although these routines execute very different kind of operations, they can be modeled in a similar way.

1) Panel: The duration of Panel kernel mostly depends on the geometry of the data block it operates upon, i.e., on MB (height of the block), NB (width of the block) and BK (number of rows that should be skipped). This kernel simply encapsulates the standard dgeqrt LAPACK subroutine that performs the QR factorization of a dense matrix of size $m \times n$ with $m = MB - (BK - 1) \times NB$ and $n = NB$. Therefore, its *a priori* complexity is:

$$T_{\texttt{Panel}} = a + 2b(NB^2 \times MB) - 2c(NB^3 \times BK) + \frac{4d}{3} NB^3,$$

where a, b, c and d are machine and memory hierarchy dependent constant coefficients.

Table I
Linear Regression of Panel

	Panel Duration	
NB^3	1.50×10^{-5} (1.30×10^{-5}, 1.70×10^{-5})	***
$NB^2 * MB$	5.49×10^{-7} (5.46×10^{-7}, 5.51×10^{-7})	***
$NB^3 * BK$	-5.52×10^{-7} (-5.57×10^{-7}, -5.48×10^{-7})	***
Constant	-2.49×10^{1} (-2.83×10^{1}, -2.14×10^{1})	***
Observations	493	
R^2	0.999	
Note:	*p<0.1; **p<0.05; ***p<0.01	

Such linear combination of parameter products fits the linear modeling framework and the summary of the corresponding linear regression is given in Table I. For each parameter combination in the first column (NB^3, $NB^2 \times MB$, and $NB^3 \times BK$), we provide an estimation of the corresponding coefficient along with the 95% confidence interval. These values correspond to the a, $2b$, $2c$ and $\frac{4d}{3}$ from the previous formula. The standard errors are always at least one order of magnitude lower than the corresponding estimated values, which means that the coefficient estimates is accurate. Furthermore, the three stars for each parameter in the last column indicate that

the estimates of the coefficients are all significantly different from 0, which means that these parameters are significant. This hints that the model is minimal and that we can not simplify it further by removing parameters without damaging its precision. Finally, the most important indicator is the adjusted R^2 value[1], which in our case is almost 1, which indicates that this model has a very good predictive power.

We also checked the linear model hypothesis (linearity, homoscedasticity, normality) by analyzing the corresponding standard plots provided by the statistical language R. Only the normality assumption did not hold perfectly in our case, which is known to not harm the quality of the regression.

2) `Update`: The duration of the `Update` kernel also depends on the geometry of the data it operates upon, defined by the same MB, NB, and BK parameters. This kernels simply wraps the LAPACK `dgemqrt` routine which applies k Householder reflections of size $m, m-1, ..., m-k+1$ on a matrix of size $m \times n$ where m, n and k are equal to $MB-(BK-1) \times NB$, NB and NB, respectively. Therefore, its *a priori* complexity is defined as:

$$T_{\texttt{Update}} = a' + 4b'(NB^2 \times MB) - 4c'(NB^3 \times BK) + 3d'NB^3$$

Table II
LINEAR REGRESSION OF `Update`

	`Update` Duration	
NB^3	1.59×10^{-9} (-6.93×10^{-8}, 7.25×10^{-8})	
$NB^2 * MB$	4.37×10^{-7} (4.36×10^{-7}, 4.37×10^{-7})	***
$NB^3 * BK$	-4.37×10^{-7} (-4.38×10^{-7}, -4.36×10^{-7})	***
Constant	8.33×10^{-1} (7.12×10^{-1}, 9.54×10^{-1})	***
Observations	20,893	
R^2	0.998	
Note:	*p<0.1; **p<0.05; ***p<0.01	

The same approach can thus be used and the R regression summary is provided in Table II. The coefficient estimates are obviously different from the ones of the `Panel` kernel since the nature of the two kernels is different. The most notable difference is certainly the NB^3 coefficient whose influence cannot be accurately estimated and may thus appear as insignificant. However, this can be explained by the fact that for this particular matrix, the parameter range of BK is limited, which leads to a confounding of the effects of NB^3 with the ones of $NB^3 \times BK$.

It is also interesting to note that this model is based on a much larger set of observations, which is expected since one `Panel` is followed by many `Updates`. Finally, the R^2 value reported in Table II is again very close to 1, which shows the excellent predictive power of this simple model.

3) Simulation: From the previous R linear regressions (tables I and II), we automatically generate C code for the simulation of these kernels (see Figure 4) and link it to SimGrid. When simulating `qr_mumps`, whenever `Panel` or `Update` is called, their parameters are given as an input to these functions. The calling thread is then blocked during the corresponding duration estimation, thereby increasing the simulation time without actually executing the tasks.

[1]The coefficient of determination, denoted R^2, indicates how well data fit a statistical model and ranges from 0 to 1. An R^2 of 0 indicates that the model explains none of the variability of the response data around its mean while an R^2 of 1 indicates that the regression line perfectly fits the data.

```
/* Injecting panel time */
static inline double xbt_panel_time(double MB,
                                    double NB, double BK)
{
  // Computed from matrix: e18.mtx
  // Adjusted R-squared: 0.999
  return  -24.89 + (NB*NB*NB)*(1.50e-05) +
    (NB*NB)*MB*(5.49e-07) + (NB*NB*NB)*BK*(-5.52e-07);
}
/* Injecting update time */
static inline double xbt_update_time(double MB,
                                    double NB, double BK)
{
  // Computed from matrix:  e18.mtx
  // Adjusted R-squared:  0.999
  return  0.83 + (NB*NB*NB)*(1.59e-09) +
    (NB*NB)*MB*(4.37e-07) + (NB*NB*NB)*BK*(-4.37e-07);
}
```

Figure 4. Automatically generated code for computing the duration of `Panel` and `Update` kernels.

It is important to understand that these two kernels are the most critical ones regarding overall simulation accuracy and that the precision of their estimation greatly influences both the makespan and the dynamic scheduling.

C. Matrix Dependent Kernels

Modeling kernels based on their signature is quite natural but it is unfortunately not applicable to all kernels. Some of them are more than simple calls to regular LAPACK/BLAS subroutines. These tasks execute a sequence of operations that depends on the matrix structure as well as on the organization of the previously executed tasks. Therefore, such kernels cannot be modeled simply from their input parameters. The actual amount of work performed by the task can however be estimated by taking into account the size of the submatrix and its internal structure. Such complexity is required as large parts of the matrix blocks are filled with zeros. These regions are thus skipped by kernel operations. Three kernels therefore require a specific expertise on the multifrontal QR method and the `qr_mumps` code to provide such workload estimates.

- **`Do_subtree`**: Both an estimation of the number of floating point operations it needs to perform and the number of nodes it has to manage are required to model the duration of this kernel.
- **`Activate`**: The duration of this kernel is mainly governed by two factors: the number of coefficients that have to be set to zero and the number of non-zero coefficients it has to assemble from children nodes.
- **`Assemble`**: The complexity of this kernel is directly linked to the total number of non-zero coefficients that needs to be copied to the parent node. However such memory

intensive operation is more subject to variability than the other computing kernels.

Analyzing the execution of these kernels and constructing models can then be performed as in Subsection V-B using the R language and simple linear regressions. Table III presents a summary of the prediction quality for each kernel as well as the minimal number of parameters that have to be taken into account. For all of them, the adjusted R^2 is close to 1, which indicates an excellent predictive power.

Table III
SUMMARY OF THE MODELING OF EACH KERNEL BASED ON THE E18 MATRIX ON FOURMI (SEE SECTION VI FOR MORE DETAILS).

	Panel	Update	Do_subtree	Activate	Assemble
1.	*NB*	*NB*	#Flops	#Zeros	#Coeff
2.	*MB*	*MB*	#Nodes	#Assemble	/
3.	*BK*	*BK*	/	/	/
R^2	0.99	0.99	0.99	0.99	0.86

VI. EXPERIMENTAL METHODOLOGY

To evaluate the quality of our approach, we used two different kind of nodes from the Plafrim[2] platform. The *fourmi* nodes comprise 2 Quad-core Nehalem Intel Xeon X5550 with a frequency of 2,66 GHz and 24 GB of RAM memory and the *riri* nodes comprise 4 Deca-core Intel Xeon E7-4870 with a frequency of 2,40 GHz and 1 TB of RAM memory. The *fourmi* nodes proved to be easier to model as their CPU architecture is well balanced with 4 cores sharing L3 cache on each of the 2 NUMA nodes. Such configuration leads to little cache contention. However, the RAM of these nodes is limited and thereby limits the matrices that can be factorized to a certain size. Although the huge memory of the *riri* machine puts almost no restriction on the matrix choice, its memory hierarchy with 10 cores sharing the same L3 cache lead to cache contention that can be tricky to model.

The matrices we used for evaluating our approach are presented in the Table IV and come from the UF Sparse Matrix Collection[3] plus one from the HIRLAM[4] research program.

Table IV
MATRICES USED FOR THE EXPERIMENTS

Matrix	m	n	nz	GFlops
tp-6	143 000	1 010 000	11 500 000	277.7
karted	46 500	133 000	1 770 000	279.9
EternityII_E	11 100	262 000	1 570 000	566.7
degme	186 000	659 000	8 130 000	629.0
hirlam	1 390 000	452 000	2 710 000	2401.3
TF16	15 400	19 300	216 000	2656.0
e18	24 600	38 600	156 000	3399.1
Rucci1	1 980 000	110 000	7 790 000	12768.1
sls	1 750 000	62 700	6 800 000	22716.6
TF17	38 100	48 600	586 000	38209.3

The execution time of a single kernel on a certain machine greatly depends on the machine characteristics (namely CPU frequency, memory hierarchy, compiler optimization, etc.). Obtaining accurate timing is thus a critical step of the modeling. To predict the performance of the factorization of a set of matrices on a given experimental platform, we first benchmark the kernels identified in the previous section.

For the kernels that have clear dependency on the matrix geometry (`Panel` and `Update`), we wrote simple sequential benchmarking scripts, that pseudo-randomly choose different parameter values, allocate the corresponding matrix and finally run the kernel, capturing its execution time. However, for kernels (`Do_subtree`, `Activate` and `Assemble`) whose code is much more complex and depends on many factors, including even dependencies on previously executed tasks, creating a simplistic artificial program that would mimic such a sophisticated code is very difficult. Since each sparse matrix has a unique structure, the corresponding DAG is very different and the kernel parameters (such as height and width) greatly vary from one matrix factorization to another. For example, the `qr_mumps` factorization of cat_ears_4_4 executes a very large number of "small" `Do_subtree` kernels (each with a small amount of work), while some others matrices, such as e18, have much fewer instances of this kernel, but with a bigger workload. Consequently, it is very hard to construct a single linear model that is appropriate for both use cases. The inaccuracies caused by such model imperfection can produce either underestimation or overestimation of the kernel duration and thus of the whole application makespan as well. Therefore, to benchmark such kernels we rely on traces generated by a real `qr_mumps` execution (possibly on different matrices than the ones that need to be studied) instead of a careful experiment design.

The result of this benchmark is analyzed with R to obtain linear models that are then provided to the simulation. At each step of the regression, we control that the models are adequate through a careful inspection of the regression summaries and of the residual plots. These models are then linked with the simulator and the experimental platform is then of no longer use as `qr_mumps` can then be run in simulation mode on a commodity laptop. Using a recent and more powerful machine only improves the simulation speed and possibly allows for running several simulations in parallel.

To evaluate the validity of our approach, we need to compare real execution outcomes (denoted by Native in the sequel) with simulations outcomes (denoted by SimGrid in the sequel). Therefore, we execute `qr_mumps` for the different matrices and collect not only the execution time but also an execution trace with information for each kernel as well as when memory is allocated and deallocated. When running on simulation, due to the structure of our integration, we can collect a trace of the same nature and thus compare the real execution to the simulation in details.

To summarize, our experimentation workflow can be described in the following four steps:

1) Run once a designed calibration campaign on the target machine that spans the desired parameter space corresponding to the different matrices.

[2]https://plafrim.bordeaux.inria.fr/
[3]http://www.cise.ufl.edu/research/sparse/matrices/
[4]http://hirlam.org/

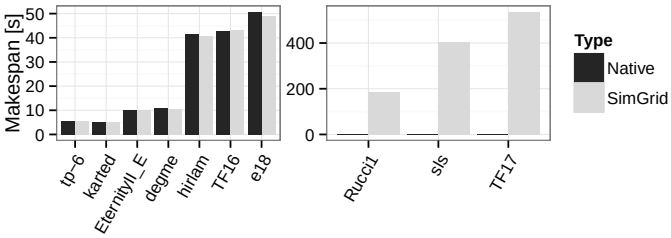

Figure 5. Makespans on the 8 CPU cores fourmi machine.

2) Analyze benchmarking outputs and execution traces, fitting the observations into linear models for each kernel. Add such models to SimGrid.
3) Run simulations on a commodity machine.
4) Validate simulation accuracy by comparing makespans, traces and memory consumption with the Native execution.

Following the principles and the Git/Org-mode workflow presented in [27], all the results that are given in this document are also available online[5] for further inspection. This repository additionally contains source code of `qr_mumps`, StarPU and SimGrid along with all the scripts for running the experiments, the calibrations and conducting the analysis, making our work as reproducible as possible. Supplementary data (e.g., produced by "unsuccessful" experiments and that can be very informative to the reader) can also be found at the same location.

VII. Simulation Quality Evaluation

A. Makespan and Execution Evaluation

1) Evaluation on the fourmi machine: Figure 5 depicts the overall execution time of `qr_mumps` when factorizing the different matrices of Table IV. The SimGrid predictions are very accurate, as they are never larger than 3%. It should be noted that our predictions are systematically slight underestimations of the actual execution time as our coarse grain approach ignores the runtime overhead and a few cache effects.

Still, focusing solely on a single number at the end of the execution hides all the details about the operations performed during the execution. Therefore, we also investigated the whole scheduling in details, comparing Native to SimGrid execution traces. An example of such investigation for the e18 matrix (it is the one exhibiting the largest difference between Native and SimGrid makespans) is shown in Figure 6. To make the Gantt charts as readable as possible, we retain only the modeled kernels and idle state, filtering overlapping states related to the runtime control.

`qr_mumps` starts by executing many `Do_subtree` kernels and executes all the remaining ones soon after. Most of the time is spent running `Update` while the `Panel` is executed regularly. Towards the end, there are fewer and fewer tasks with more and more dependencies between them, and many cores have thus to remain idle. The Native and SimGrid traces are extremely close. A noticeable difference can be seen at the very beginning as in simulation all workers start exactly at time 0, they all pick `Do_subtree` tasks that are the leaves of the DAG and thus the first ready tasks. Although the real execution tends to run in the same manner, it is not so strict and several workers pick `Activate` tasks available right after the first `Do_subtree` terminates. Another small discrepancy between the two traces can be noticed at the end of the execution where the idle time is distributed in a quite different way. This is however related to the difference of scheduling decisions taken by the runtime. Indeed, we remind that the StarPU runtime schedules tasks dynamically and thus even two consecutive Native executions can lead to quite different execution structures even if their total duration is generally similar. Idle time distributions similar to the one of the SimGrid trace can also be observed in real executions.

However, gantt charts of such densely packed traces of dynamic schedules can sometimes be misleading. The first issue comes from the fact that there is a huge number of very small states that have to be aggregated during the graphical representation which is limited by the screen or printer resolution. Many valuable information can be lost in such process and the result may be biased [28]. Second, it is very hard to quantify the resemblance of the two traces that are dynamically scheduled, as even when the task graph is fixed, the tasks will naturally have very different starting times from an execution to another. Therefore, we compared different and more controlled aggregates.

For example, Figure 8 compares the distributions of the duration of each kernel for a real execution and a simulation (we use the same two traces of the e18 matrix that were used earlier). The upper row presents kernel distributions from the Native trace, while in the bottom row are the ones predicted by the SimGrid simulation. The modeling technique described in Section V proves very satisfactory, since distributions match quite accurately. The only kernel for which we can observe a slightly larger discrepancy is `Assemble`, which was indeed very hard to model and had the worst R^2 value. However, in practice this kernel is rarely on the critical path of the `qr_mumps` execution and is often covered by other kernels. Additionally, the overall duration of all `Assemble` tasks is relatively small compared to others and thus such inaccuracies of the model do not greatly affect the final simulation prediction.

Studying distribution applies a temporal aggregation and discards any notion of time and when a specific task was executed. This can hide interesting facts about certain events or a particular group of tasks that occurred in a distinct period of time during the whole application run. Figure 7 tracks the execution of the `Panel` kernel and indicates the duration of the tasks at each time. To ease the correspondence between the real execution and the simulation, the color of each point is related to the job id of the task. Both colors and the pattern of the points suggest that the traces match quite well. Even though the scheduling is not exactly the same, it is still very close. Similar analysis have been performed for all the other

[5] http://starpu-simgrid.gforge.inria.fr/. We also provide on github an example of an easy to access pretty-printed report on a trace as well as information on how the trace was captured.

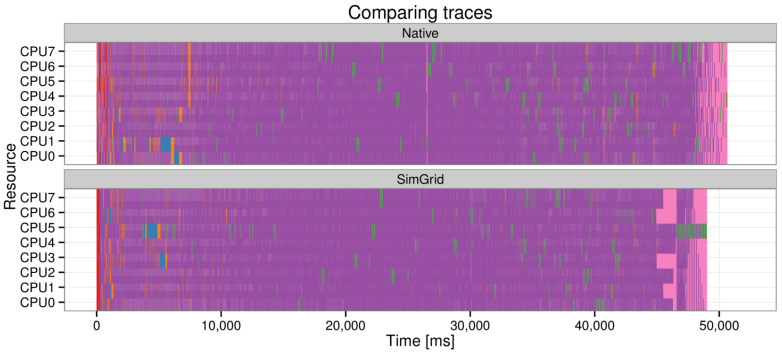

Figure 6. Gantt chart comparison on the 8 CPU cores fourmi machine.

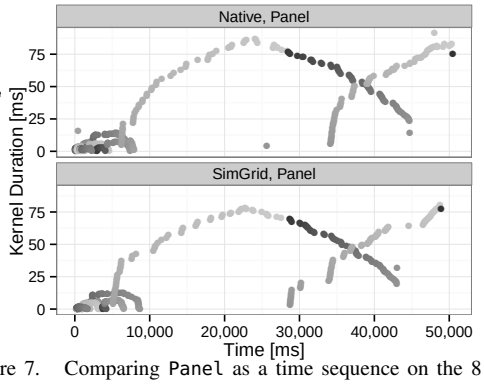

Figure 7. Comparing `Panel` as a time sequence on the 8 CPU cores fourmi machine. Color is related to the task id.

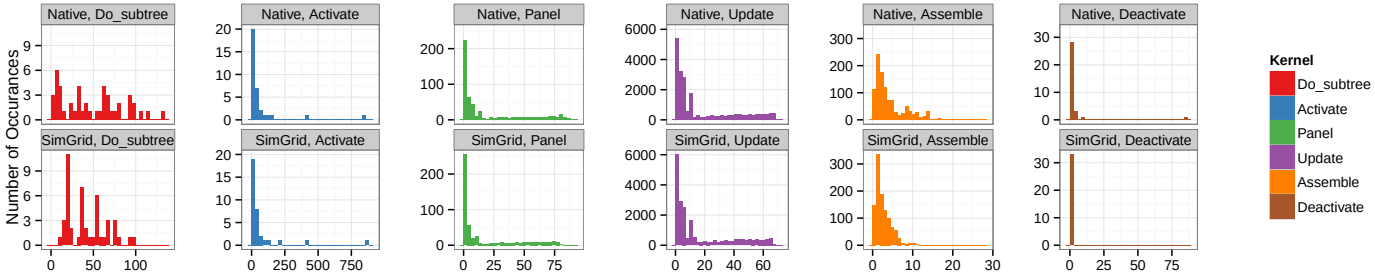

Figure 8. Comparing kernel distribution duration on 8 CPU cores fourmi machine.

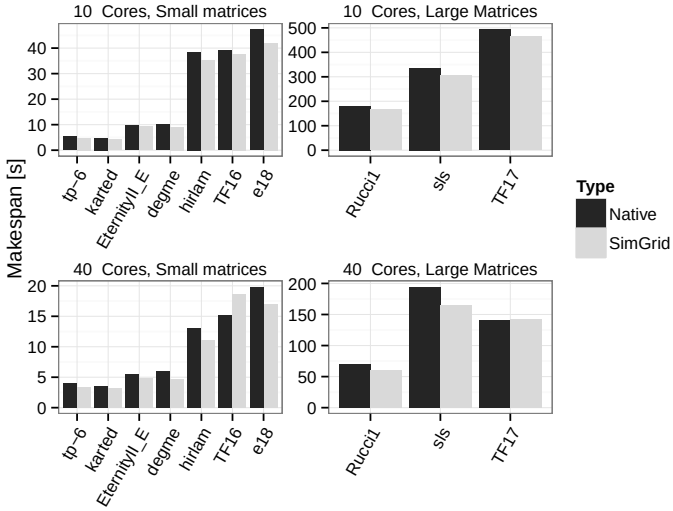

Figure 9. Results on the riri machine using 10 or 40 CPU cores. When using a single node (10 cores), the results match relatively well although not as well as for the fourmi machine due to a more complex and packed processor architecture. When using 4 nodes (40 cores), the results are still within a reasonnable bound despite the NUMA effects.

kernels as well and the results were very much alike.

2) Evaluation on the riri machine: To validate our approach, we have experimented on a different architecture. The results presented in Figure 9 show a more important error of the SimGrid predictions, that is now averaging 8.5%. Such inaccuracy mostly comes from the fact that *riri* has a specific architecture, where the 10 cores used for the experiments all share the same L3 cache. The pressure on the cache produced by all the workers executing kernels in parallel decreases the overall performance, which is not correctly captured by our models. Still, the SimGrid predictions stay reasonably close to the Native ones and thus very useful to users and developers.

One step further in our experimental campaign was to compare executions on the full machine, using all 40 cores. As expected, the largest prediction error doubled (Figure 9), but the results can still be considered as good since all the tendencies are well captured. In particular, non trivial results can be obtained such as the fact that the TF17 matrix benefits much more from using several nodes than the sls matrix.

B. Memory Consumption

Working on several parts of the elimination tree in parallel provides more scheduling opportunities, which improves processor occupancy. But it also increases memory requirements, which can have a very negative influence on performance. The most critical criteria regarding memory is the peak memory consumption of the application. A single wrong scheduling decision can dramatically increase it and potentially result in memory swapping between the RAM and the disk.

Since the amount of work that can be executed in parallel is generally limited and very matrix dependent, finding the right trade-off between memory consumption and the efficient use of the whole set of available cores is crucial for obtaining the best performance [29]. To evaluate a new factorization algorithm or a different scheduling strategy, one thus has to perform a large number of costly experiments on various matrices. Using simulation can greatly reduce the cost of such study as it does not require the access to the actual experimental machines, often shared between many users. It

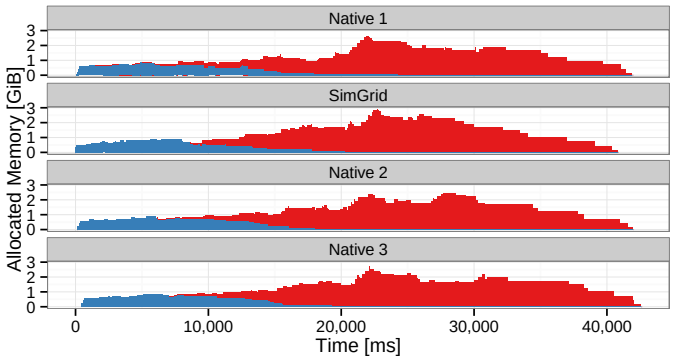

Figure 10. Memory consumption evolution. The blue and red parts correspond to the `Do_subtree` and `Activate` contribution.

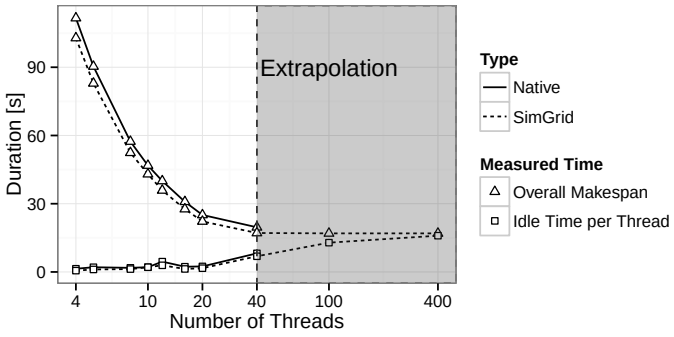

Figure 11. Extrapolating results for e18 matrix on 100 and 400 CPU cores.

can be performed much faster and several simulations with different parameters can even be run in parallel. In our solution no actual memory is neither allocated nor deallocated for the data, as the corresponding *malloc* calls are only simulated by SimGrid. However, the size for the required array allocation and deallocation is still traced. This allows for reconstructing memory usage, providing the memory peak prediction of the simulation. Since SimGrid faithfully represents the runtime execution (as it was presented in the previous subsections), its execution will go through the DAG in a very similar way to the Native run. Therefore, the memory peak predicted by SimGrid will be very close to the one observed in Native experiments.

Beyond the memory peak, it is also interesting to study the evolution of memory consumption throughout the execution. Figure 10 shows the evolution of the total amount of memory allocated by `qr_mumps` when factorizing the hirlam matrix for three Native executions and for a simulation. The three Native executions correspond to three consecutive runs performed with exactly the same source code and environment. Such analysis allows for identifying where the scheduling was not optimal and what parts of the application should be improved to increase the overall performance. Although the memory consumption evolution is very similar between different experiments it is far from being identical since runtime scheduling decisions are made dynamically. The evolution predicted by SimGrid is remarkably difficult to distinguish from the three other ones, which shows that our approach allows for faithfully predicting memory consumption.

C. Extrapolation

When sufficient care is taken on benchmarking and kernel profiling, we believe that our approach allows for performing faithful performance prediction of the execution on large computer platforms. Based on models obtained from a single CPU calibration, we can extrapolate the simulation on a larger numbers of cores. These simulation results are certainly not as accurate as the ones presented in Section VII, but can still show general trends. New phenomena that haven't been observed yet may occur at large scale but the simulation somehow allows for obtaining an "optimistic" performance prediction that will be achieved "if nothing goes wrong". Researchers can thus observe how their matrix factorization would perform in an ideal context. Since certain parts tend to be underestimated (e.g., contention) in the simulations, SimGrid results provide theoretical performance bounds, above which application could not pass on the target machine.

Figure 11 shows the performance obtained when factorizing the e18 matrix with different numbers of cores. Together with the overall makespan, we indicate how much time in average each core spends idle. With a small number of workers, each thread has enough work as `qr_mumps` is well parallelized. However, the execution is limited by the critical path in the DAG of tasks and thus above a given platform size, most of the cores remain idle, waiting for task dependencies to be satisfied. Increasing the number of threads beyond this point does not improve the overall performance, but only decreases the efficiency of the workers. After comparing our simulation results to the Native execution for up to 40 cores on the riri machine, we decided to use the same kernel models and investigate what performance could be expected from a larger machine comprising the same kind of nodes. The simulation results actually predict that the makespan will not improve any further and that most cores will be idle. Investigating more in detail the trace simulated for 100 cores would allow to know whether the critical path is hit or if further improvements can still be expected with a better scheduling.

VIII. Conclusion

This article extends our previous work [4] that addressed the simulation of dynamic dense linear algebra solvers on hybrid machines, by considering a sparse multifrontal linear algebra solver. Modeling the irregular internals of such application is much more challenging and required a careful study. We show through extensive experimental results that we manage to accurately predict both the performance and the memory usage of such applications. Our proposal also allows for quickly simulating such dynamic applications using only commodity hardware instead of expensive high-end machines. As an illustration, factorizing the TF17 matrix on a 40 core machine (see Section VI) requires 157s and 58GiB of RAM while simulating on a laptop its execution only takes 57s and 1.5GiB of RAM. Being able to quickly obtain performance and details of memory consumption of such applications on a

commodity laptop is a very useful feature to the `qr_mumps` users and developers, as they can easily test the influence of various scheduling, parameters or even code modifications. The SimGrid simulation mode is thus integrated in the latest versions of StarPU and of `qr_mumps`.

The current main limitation of our work concern the faithful modeling of very large NUMA machines were data transfers are implicit and where performance degradation of computation kernels due to cache sharing can be significant. We think such issues can be addressed through a careful model calibration taking into account potential interferences and through the use of explicit data transfers. This could be done through the use of StarPU-MPI [30] that was designed exploit large distributed machines. The simulation of StarPU-MPI based applications with SimGrid is underway as well as the exploitation of hybrid machines (comprising GPUs) in `qr_mumps`. These developments will allow us to investigate performance evaluation of truly complex, irregular and dynamic applications at very large scale.

ACKNOWLEDGMENTS

This work is partially supported by the SONGS (11-ANR-INFRA-13) and SOLHAR (ANR-13-MONU-0007) ANR project. Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS.

REFERENCES

[1] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, no. 2, pp. 325–348, Dec. 1987.

[2] W. Hackbusch, "A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$–matrices," *Computing*, vol. 62, no. 2, pp. 89–108, 1999.

[3] A. Björck, *Numerical methods for least squares problems*. Siam, 1996.

[4] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut, "Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures," *Concurrency and Computation: Practice and Experience*, May 2015.

[5] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems," IRIT, Tech. Rep. IRI/RT–2014-03–FR, 2014, submitted to ACM TOMS.

[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*, vol. 23, pp. 187–198, Feb. 2011.

[7] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014.

[8] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[9] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Parallelizing dense and banded linear algebra libraries using SMPSs," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 18, pp. 2438–2456, 2009.

[10] J. Kurzak and J. Dongarra, "Fully dynamic scheduler for numerical computing on multicore processors," *LAPACK working note*, vol. lawn220, 2009.

[11] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and multi-CPU parallelization for interactive physics simulations," in *Euro-Par*, 2010, pp. 235–246.

[12] 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.

[13] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. Van De Geijn, F. G. Van Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Trans. Math. Softw.*, vol. 36, no. 3, 2009.

[14] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012037, 2009.

[15] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. Dongarra, "Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach," *Scalable Computing and Communications: Theory and Practice*, 2013.

[16] X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca, "Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes," in *23rd International Heterogeneity in Computing Workshop, IPDPS 2014*, IEEE. IEEE, 2014.

[17] K. Kim and V. Eijkhout, "A parallel sparse direct solver via hierarchical DAG scheduling," *ACM Trans. Math. Softw.*, vol. 41, no. 1, pp. 3:1–3:27, Oct. 2014.

[18] A. Buttari, "Fine-grained multithreading for the multifrontal QR factorization of sparse matrices," *SIAM Journal on Scientific Computing*, vol. 35, no. 4, pp. C323–C345, 2013.

[19] X. S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 302–325, 2005.

[20] P. Cicotti, X. S. Li, and S. B. Baden, "Performance modeling tools for parallel sparse linear algebra computations," in *Parallel Computing: From Multicores and GPU's to Petascale*, 2009, pp. 83–90.

[21] I. S. Duff and J. K. Reid, "The multifrontal solution of indefinite sparse symmetric linear systems," *ACM Transactions On Mathematical Software*, vol. 9, pp. 302–325, 1983.

[22] R. Schreiber, "A new implementation of sparse Gaussian elimination," *ACM Transactions On Mathematical Software*, vol. 8, pp. 256–276, 1982.

[23] P. R. Amestoy, I. S. Duff, and C. Puglisi, "Multifrontal QR factorization in a multiprocessor environment," *Int. Journal of Num. Linear Alg. and Appl.*, vol. 3(4), pp. 275–300, 1996.

[24] T. A. Davis, "Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 8:1–8:22, Dec. 2011.

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

[26] E. Agullo, O. Beaumont, L. Eyraud-Dubois, J. Herrmann, S. Kumar, L. Marchal, and S. Thibault, "Bridging the gap between performance and bounds of cholesky factorization on heterogeneous platforms," in *Proceedings of the 24th International Heterogeneity in Computing Workshop*, 2015.

[27] L. Stanisic, A. Legrand, and V. Danjean, "An effective git and org-mode based workflow for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 61–70, Jan. 2015.

[28] L. Mello Schnorr and A. Legrand, "Visualizing More Performance Data Than What Fits on Your Screen," in *Tools for High Performance Computing 2012*, A. Cheptsov, S. Brinkmann, J. Gracia, M. M. Resch, and W. E. Nagel, Eds. Springer Berlin Heidelberg, 2013, pp. 149–162.

[29] L. Marchal, O. Sinnen, and F. Vivien, "Scheduling tree-shaped task graphs to minimize memory and makespan," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS. IEEE Computer Society, 2013, pp. 839–850.

[30] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, "StarPU-MPI: Task programming over clusters of machines enhanced with accelerators," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, J. Träff, S. Benkner, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2012, vol. 7490, pp. 298–299.