Performance Analysis of Irregular Task-Based Applications on Hybrid Platforms: Structure Matters

Marcelo Cogo Miletto, Lucas Leandro Nesi, Lucas Mello Schnorr, Arnaud Legrand

Performance Analysis of Task-Based Multi-Frontal Sparse Linear Solvers: Structure Matters

Marcelo Cogo Miletto[a], Lucas Leandro Nesi[a], Lucas Mello Schnorr[a,*], Arnaud Legrand[b]

[a] *Institute of Informatics, Federal University of Rio Grande do Sul – UFRGS, 91501-970, Porto Alegre, RS, Brazil*
[b] *Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, Grenoble, F-38000, Isère, France*

Abstract

Efficiently exploiting computational resources in heterogeneous platforms is a real challenge which has motivated the adoption of the task-based programming paradigm where resource usage is dynamic and adaptive. Unfortunately, classical performance visualization techniques used in routine performance analysis often fail to provide any insight in this new context, especially when the application structure is irregular. In this paper, we propose several performance visualization techniques and modeling strategies motivated by the analysis of task-based multifrontal sparse linear solvers whose structure is particularly complex. We show that by building on both a performance model of irregular tasks and on structure of the application (in particular the elimination tree), we can detect and highlight anomalies and understand resource utilization from the application point-of-view in a very insightful way. We validate these novel performance analysis techniques with the `QR_mumps` sparse parallel solver by describing a series of case studies where we identify and address non trivial performance issues thanks to our visualization methodology.

Keywords:
Performance Analysis, Task-based Scheduling, Trace Visualization, Performance Modeling, Irregular Tasks, Elimination Tree

1. Introduction

High-Performance Computing (HPC) applications rely on hardware parallelism to accelerate computations. The construction of efficient parallel programs remains challenging as HPC embraces hybrid architectures comprising multi-core CPUs, GPUs, and TPUs. The task-based programming paradigm has emerged as the easiest and most efficient way for programmers to develop applications with portable performance over such systems. Nowadays, it is supported by several libraries such as OpenMP [47], StarPU [33], OmpSs [34], Kaapi [41], and OpenCL [39]. This paradigm allows describing the program as a set of high-level computational tasks handled by a runtime system that builds on decades of research in scheduling theory.

Yet, due to the complexity of hybrid platforms and parallel applications, the efficiency of task-based applications and schedulers remain susceptible to many performance degradation factors. Numerous studies show that different runtimes achieve significantly different performance for the same application in the same environment [16, 12, 7]. This variation can be due to various factors such as the different overhead costs for creating and submitting tasks to the runtime system, bad decisions made by the scheduler, poorly implemented applications or computation kernels, or inadequate application parameters. Identifying and optimizing these problems is laborious since they can occur at many levels. Furthermore, this has become an increasingly complex effort as applications have to handle multicore processors with non-uniform memory hierarchies, together with GPUs, and network communication [25].

Performance visualization tools commonly aid analysts and developers throughout the performance analysis process by providing a Gantt chart depicting application states through time (along the X-axis) using the hierarchy of computational resources (Y-axis). Although this type of generic visualization can pinpoint many performance issues, it misses essential application-specific aspects. Likewise, most approaches expect the task cost to be homogeneous, which is well suited for regular and well-behaved applications like dense linear algebra. Unfortunately, many real scenarios are not so well behaved.

For example, sparse matrix factorization algorithms, present in many computer applications [15], are way more complicated than their dense counterparts. The input problems have to go through a symbolic analysis phase, before the numerical factorization, to extract the parallelism. A classical approach to exploit this parallelism is the Multifrontal method [52, 45] that breaks the whole factorization problem into a heterogeneous collection of smaller and denser subproblems, called frontal matrices. A structure called *Elimination Tree*, which lies at the heart of the multifrontal method, connects these subproblems through dependencies. It is crucial to consider these specialized structures when analyzing application performance since they shape and define the application execution behavior.

In this article we propose several new performance visualization techniques to exploit the structure of task-based sparse ma-

*Corresponding author
Email addresses: `marcelo.miletto@inf.ufrgs.br` (Marcelo Cogo Miletto), `lucas.nesi@inf.ufrgs.br` (Lucas Leandro Nesi), `schnorr@inf.ufrgs.br` (Lucas Mello Schnorr), `arnaud.legrand@imag.fr` (Arnaud Legrand)

trix solvers. These techniques are validated with an extensive analysis of the multifrontal task-based parallel sparse solver `QR_mumps` [17]. Our main contributions are as follows.

(1) We propose a statistical model of the irregular tasks of `QR_mumps` to detect tasks with anomalous duration given their expected floating-point operation count and the computational resource type (Section 4). We illustrate through four case studies how this mechanism allows us to uncover and fix simple to non-trivial issues that would easily go unnoticed. **(2)** We propose a visualization of the temporal traversal of the elimination tree structure, including derived information such as memory usage and the number of active tree nodes, along with the computational effort in terms of tree nodes and depth (Section 5). We illustrate through three case studies how this visualization enables understanding on how the factorization unfolds, to identify and address non-trivial scheduling problems that would be hard to understand solely with generic Gantt chart.

Our contributions benefit both runtime and application developers by visually highlighting anomalous tasks, malfunctioning application structures (elimination tree), and inefficient scheduling. We show that these techniques allow the identification and the correction of possible performance problems at various levels, from the tree partitioning and matrix reordering to runtime scheduler decisions and parameters.

Section 2 presents the fundamental concepts behind parallel multifrontal sparse matrix factorization and task-based implementations. This section also covers some related work on performance modeling and performance analysis of task-based applications. Section 3 describes the computational environment, workload characterization, and the design of our experiments. Section 4 is devoted to our contribution on the exploitation of the task structure while Section 5 is devoted to the exploitation of the application structure. Section 6 concludes the paper and presents future directions for this work.

2. Background and Related Work

In this section, we discuss the recent implementations of high-performance sparse direct solvers (Section 2.1), and the process of collecting application information, and how to use this data for in-depth performance analysis (Section 2.2).

2.1. Parallel Sparse Matrix Factorization

In this work, we study a particular set of task-based applications that arises in many research problems: solving large and sparse systems of equations. This kind of problem is a source of extremely irregular workloads, presenting tasks of different types, computational weights, and variable memory consumption [27]. We focus on solvers that incorporate parallel versions of the multifrontal method for obtaining the direct solution of sparse equation systems [18]. Implementing these solvers requires us to carefully consider many aspects that may harm application performance, from the sparse matrix data structure representation to the efficient scheduling and implementation of the computational kernels in a parallel environment. Furthermore, the implementation is responsible for balancing the load over a complex architecture comprising heterogeneous compute resources while considering communication costs and memory management. All those aspects are vital to building efficient software, which should be portable and scalable regardless of the diversity of current architectures.

Since dense linear algebra kernels are the core of these solvers, they systematically rely on the standard set of basic routines from BLAS [50], their hand-tuned implementations like OpenBLAS [30], and MKL [26] for CPU, and CUBLAS [40] and MAGMA BLAS [35] for GPUs. This variety of libraries allows to adapt to the variety of computational resources and select the implementations which are the best suited to the platform at hand.

To address the load balancing issue, a traditional approach among solvers consists in relying on an in-house scheduler specifically designed for the sparse factorization context[10]. This approach enables the developers to describe the application as a Directed Acyclic Graph (DAG), where the DAG nodes represent the computational tasks and the edges, the data dependencies among them. This approach simplifies programming since it delegates the control flow management and load balancing to the scheduler. Task-based programming has been used in this context since the pre-multicore era in MPI-based implementations such as MUMPS [46] and is still used nowadays. These in-house schedulers are lightweight because they often rely on a specific algorithm's knowledge. However, such in-house and application-dependent schedulers commonly have limited features and fail to scale well on heterogeneous systems. Therefore, there is a move toward the use of general-purpose runtime systems such as StarPU, which has proven to be a well-suited option for the parallelization of sparse factorization methods [27]. In the next sections, we give more details on the stages and structure of the multifrontal method and give practical details of the `QR_mumps` implementation on top of StarPU but we believe our proposal could be equally applied to any other sparse solver and runtime.

The Multifrontal Method

The multifrontal method [52] is an extension of the frontal method [53], focusing on the factorization of symmetric matrices using Cholesky. However, the method provides a structure that can be adapted to factorize sparse unsymmetric matrices using LU or QR factorization as well. This method breaks the whole matrix factorization problem into smaller and denser subproblems, as partial factorization steps. These subproblems are known as being the frontal matrices or just fronts.

In classical approaches, each frontal matrix represents one elimination step related to a column j. Now, because of the matrix sparsity, some elimination steps operate in disjoint subsets of matrix coefficients. Then, multiple fronts can be factorized in parallel, which gives the name to the method. However, if the elimination of one column changes the coefficients used in another step, there is a dependency between these eliminations. These dependencies are captured by a structure that is the heart of the multifrontal method: the *elimination tree*. This tree structure holds the fronts in its nodes and expresses the dependencies

between them as a parent and child relation in the tree. A parent node can only be factorized after all its child nodes were already factorized, and its contribution blocks were assembled into the parent node. The whole matrix factorization is done by traversing the tree in the topological order, from bottom to top.

Considering a QR Householder factorization, Figure 1 presents an example of a sparse matrix structure and its elimination tree with some detail in its the frontal matrices. In the figure, we can observe the inherent parallelism that arises from this structure regarding eliminating columns that reside in different branches of the tree simultaneously (e.g., 1, 2, and 5). This source of parallelism is commonly referred to as the *tree parallelism*. We can also notice how the fronts form dense submatrices of the problem by looking at the detailed fronts 1, 2, and {3,4}. At each column elimination step, one row of the final factor R is produced, along with a set of coefficients that form the contribution block that goes in the parent node matrix (the blue and red dots in the figure). Lastly, the Q factor is implicitly represented by the Householder reflector vectors computed in each frontal matrix. Another thing that can be observed in the frontal matrix that represents the elimination of columns 3 and 4 is the so-called *staircase structure* which appears in bigger fronts, where we have many zero elements in the bottom left of the matrix. Note that we have reordered the front rows to observe this structure better.

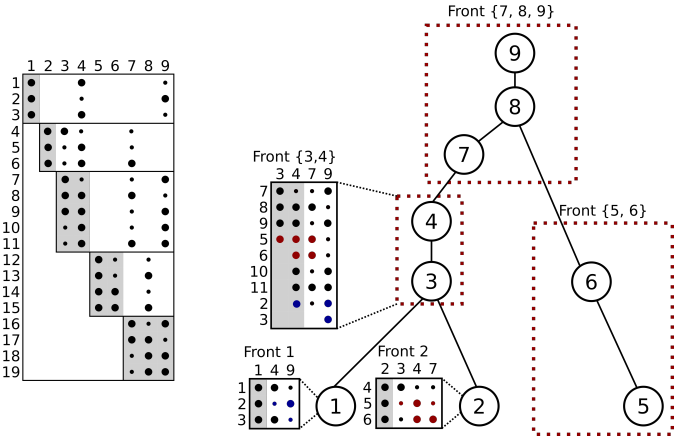

Figure 1: Example of a sparse matrix (left) and its elimination tree (right). Original matrix coefficients are marked as black dots, while fill-in coefficients are small dots. The gray-shaded area represents the columns that are being eliminated in that elimination step. Blue and red coefficients represent the contribution blocks in the detailed fronts in the elimination tree. The red dashed areas mean that those tree nodes were amalgamated to form a supernode.

We can also observe that each column elimination has a dependency on another, representing the above-mentioned classical approach. However, this classical strategy has the drawback of generating small fronts, limiting the efficiency that could be achieved by Level-3 BLAS operations. Other strategies consider the amalgamation of nodes with a similar structure for the R factor, at the cost of generating some additional fill-in. Figure 1 represents this strategy through the amalgamation of the nodes {3,4}, {5,6}, and {7,8,9}, forming what is commonly called supernodes. This amalgamation of nodes transforms the elimination tree by creating bigger frontal matrices where the high efficiency of BLAS-3 routines can be better explored at the cost of some additional fill-in. However, the efficiency of BLAS-3 routines pays off this additional cost and improve overall performance. Further improvements to this method consist of exploring an intra-node parallel front factorization technique such as multithreaded BLAS or tiled factorization algorithms, enabling even more concurrent work through *node parallelism*.

The complete matrix factorization and solution is thus organized in three different phases in the multifrontal method. The **analysis phase** handles a major concern in sparse matrix factorization: reducing or keeping the generation of new nonzero coefficients (fill-in effect) under control. In this phase, software libraries like COLAMD [43], Scotch [48], and Metis [1] use matrix reordering algorithms such as Approximate Minimum Degree, Nested Dissection, and Cuthill-McKee [29] to provide a matrix permutation that reduces the fill-in during the factorization. Applications also perform a symbolic factorization step that enables them to preallocate the necessary memory for the final structure by calculating beforehand the final structure of the matrix after the factorization. At the end of this phase, we have the elimination tree structure ready to be computed by the next phase. In sequence, the **factorization phase** is responsible for traversing the elimination tree from the leaves to the root, computing the partial factorization in each front, and combining the child node contribution blocks to the parent frontal matrix. These front factorizations can be done in parallel. For example, the method can process all the leaves of the tree at the same time. There is a restriction in starting the parent node factorization because all its child nodes need to be already computed to assemble their contribution blocks. Then, as the computations move towards the tree root, the *tree parallelism* becomes more scarce, and fronts get bigger, and this is why we should use some other techniques like tiled factorization to explore the *node parallelism*. Finally, in the **solve phase** has the last tree traversal to apply forward and backward substitutions, and a triangular solve operation for each front to group their results.

QR_mumps: A Fine-Grained Task-Based Multifrontal Method

The `QR_mumps` application [17] is an example of a multifrontal sparse direct solver that uses the elimination tree structure to partition and parallelize the problem. Using a fine-grained task-based approach relying on the StarPU runtime system, it partitions the frontal matrices in tiles on which tasks will work. This approach allows exploring a new level of parallelism called interlevel parallelism [17], which refers to the limitation of starting a parent node only once all children contribution blocks have been assembled into it. With this finer partitioning, as soon as a part of the parent node is assembled, the factorization in that region can start, allowing computation overlap between children and parent. The resulting performance gains comes at the cost of making harder the understanding of the application execution.

The application relies on four LAPACK kernels for the factorization: `geqrt`, `gemqrt`, `tpqrt`, and `tpmqrt`. The partitioning of the frontal matrices in `QR_mumps` follows a 2D block partitioning, breaking a single front into many smaller blocks where these tasks operate. The block and task dimensions are

controlled by the user-defined parameters mb, nb, and an internal blocking size ib. The first two controls the number of rows and columns of the block, and the latter is a parameter used in LAPACK routines to decrease the number of extra flops needed because of the 2D partitioning, as explained by [36]. Despite this beneficial effect from the LAPACK ib parameter, the routines were modified to control fill-in level inside the blocks where there are zeroes in the bottom left part of the block, forming the staircase structure. The effect of these parameters in the routines regarding the fill-in and task irregularity is presented by Figure 2. The figure represents a frontal matrix (dense) with its rows sorted by the leftmost nonzero to clearly observe the staircase structure, leading to many zero elements in the bottom left of the matrix. The dashed lines represent the matrix partitioning following the mb and nb parameters, for which there is no restriction for its values so that we can have rectangular blocks. The only restriction is that the value of nb must be a multiple of ib, which controls the internal blocking effect, illustrated in the left part of Figure 2, where the dark gray squares represent the fill-in coefficients. This part of the figure shows the effect of two different values for ib: nb/6, and nb/3. Note how the fill-in is reduced with smaller ib values, but this comes at the cost of lower efficiency in the BLAS-3 operations.

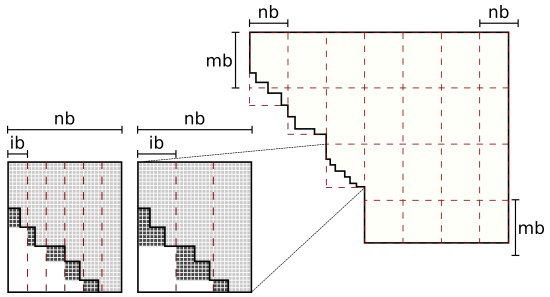

Figure 2: Example of a front partitioning using mb and nb, sources of task irregularity, and the ib size effect in fill-in. Matrix coefficients are represented as light gray squares and fill-in coefficients as dark gray.

Figure 2 also demonstrates the irregularity sources in the tasks, as the blocks with the staircase structure will have fewer coefficients to compute. Furthermore, the frontal matrix partitioning may not result in an exact number of blocks given its numbers of rows and columns. This way, the blocks residing in the staircase structure and blocks in the bottom or right borders of the matrix may have a smaller number of rows and columns, as can be observed by the mb and nb sizes at the right of the figure. This irregularity leads to tasks with many different sizes and computational weights. Moreover, tasks of the same size can have different computational weights like the ones that reside in the staircase structure.

The frontal matrices are very small at the bottom of the tree, and the number of tree nodes at this level is commonly much higher than the number of processing units. To avoid creating too many tasks and limit the overhead of the runtime system, the QR_mumps application uses a logical pruning technique described by [28] to compute entire subtrees within a unique sequential task: the do_subtree tasks. This optimization also makes the operations in those regions of the tree more efficient. Assuming that there is enough tree parallelism, the pruning algorithm determines a layer in the tree structure such that all subtrees rooted at this layer have a computational weight smaller than a given threshold (e.g., 1% of the total factorization cost).

Another optimization brought by this fine-grained approach is memory consumption control. The QR_mumps application allows the definition of a memory usage upper bound based on how much memory the application would use in a sequential traversal of the elimination tree. Previous experiments demonstrate that the application can keep performance while saving memory usage [21].

All these optimizations specified at the application level in QR_mumps are architecture-independent thanks to flexibility offered by the task-based paradigm which leverages multiple kernel implementations and handles all the above-mentioned strategies as a DAG scheduling problem. The DAG is entirely handled by the StarPU runtime system to ensure data coherence and dynamic load balancing using one of its many scheduling policies that take into account task priorities and the compute and communication costs among the computational resources.

2.2. Task-Based Performance Analysis: Tools and Techniques

Performance analysis is an essential step to understand and improve any application. Traditional tools like ViTE [31] or Paraver [49], for instance, depict the behavior of application traces through Gantt charts. Unfortunately, since the execution of task-based programs is stochastic, it turns out they are a much more challenging scenario to analyze than traditional parallel applications that have well-identified regular computation and communication phases. Not only is a Gantt chart very difficult to read in this context but generally these tools lack important features for task-based applications, like task dependencies and critical path analysis. We revisit performance analysis and visualization techniques for task-based applications.

There are a few task-oriented performance analysis tools, such as DAGViz [20] and Temanejo [32]. Even though they display the application DAG, they either focus on DAG task debugging or on a timeline with workers and available parallelism. The later idea can be useful to visually represent the tree and node parallelism in the multifrontal method but this representation does not handle well heterogeneous resources. More recently, StarVZ [11, 6] was developed to build task-based performance visualization for the StarPU library. This tool provides a multi-level and complete view of the application, runtime aspects, and DAG analysis.

Other performance analysis techniques focus on the modeling of the behavior of individual tasks. From a more low-level perspective, focusing on individual tasks, TaskInsight [14] implements a technique that allows evaluating the scheduler decisions in terms of data reuse by using task trace information from hardware counters. Another common approach consists in developing analytical models of frequently used computation kernels like the BLAS based ones and many studies detail how to model the task cost mathematically in the context of sparse matrix operations [51, 22, 42, 37]. Their most frequent use is weight partitioning, scheduling hints, and performance prediction of whole execution application either through a regression

model [8] or through a simulator such as SimGrid [24]. However, most of the time, these models remain absent from the visual performance analysis.

Therefore, traditional trace visualization tools lack both DAG-related and task-related features and rarely exploit application-specific characteristics (e.g., the tree structure), which makes them completely unfit to understand how an application made of heterogeneous tasks unfolds on a heterogeneous set of resources.

3. Experimental Design

Hardware and Software Configuration. Table 1 lists the computational platforms used in our experiments. They provide contrasting configurations in terms of computational resources count, implying different elimination trees due to pruning, and diverse CPU and GPU computing capabilities. All machines run Debian 10, kernel version 4.19.0-8-amd64 in a controlled environment with exclusive access during experiments. The StarPU version used in the experiments comes from the development branch [2] linked against CUDA 10.2.89. We have used Scotch 6.0.8 [9] and Metis 5.1.0 [1] for matrix reordering. The `QR_mumps` code was compiled using GCC 8.4.0 [3] and linked against OpenBlas 0.3.9 [5]. We collect enriched information using application-injected data registered by StarPU in application traces. We convert the binary data towards the visualization using StarVZ [11, 4].

Table 1: Hardware specification of the three platforms.

Machine	CPU Cores	GPU Cores
`Tupi`	E52620v4, 1×8	2× GTX 1080Ti, 3584
`Hype`	E52650v3, 2×10	2× Tesla K80, 2496
`Draco`	E52640, 2×8	2× Tesla K20m, 2496

Application Configuration and Workload. The `QR_mumps` application is exceptionally configurable. Globally, we used two different versions of `QR_mumps` depending on whether we wanted to use GPUs or not. We use a fixed block size (nb=320) and internal block size (ib=32) for executions using only CPU and larger sizes (nb=600, ib=60) when using GPUs. While these values provide a good task granularity for both CPU and GPU setups of our experimental platform, they have no influence in the elimination tree structure.

We also have investigated the impact of the **memory constraint** parameter on performance. This constraint (limited or unlimited) regulates the total amount of memory that the application can use during the factorization. When limited, the application respects a memory usage constraint defined by its computed sequential peak. The memory limitation directly impacts the total amount of parallelism available and changes the elimination tree traversal. Another important parameter for our experiments is the StarPU's **scheduler** algorithm. Among the many possibilities proposed by [33], we have considered the `lws` and `prio` for CPU executions, and `heteroprio`, `dmda`, and `dmdasd` schedulers for cases including GPUs (see [33] for further details about these schedulers) . Finally, we also employ two **ordering algorithms** in the application, one based on the Scotch and another on the Metis library. Both handle matrix reordering but with different strategies. As consequence, their elimination tree and floating-point operation cost for the matrix factorization are different.

Table 2 lists those sparse matrices from real problems (Matrix Market and SuiteSparse Matrix Collection repositories) that we use in this work. We have many possible combinations for each workload and application configuration (memory constraint, scheduler, and ordering). We avoid exploring all combinations since some schedulers only make sense when we have GPUs, like `heteroprio`, `dmda`, and `dmdasd`.

Table 2: Matrices used as workload for `QR_mumps`.

Name	Rows	Cols	NNZ
`ch8-8-b3`	117.600	18.816	470.400
`flower_8_4`	55.081	125.361	375.266
`e18`	24.617	38.602	156.466
`degme`	185.501	659.415	8.127.528
`karted`	46.502	133.115	1.770.349
`Rucci1`	1.977.885	109.900	7.791.168
`TF17`	38.132	48.630	586.218

Table 3 lists the specific configuration (Scheduler, Ordering, Machine, Input and other) for every case and figure that we detail in Sections 4 and 5, for a total of seven case-studies. All runs use 320 as block size and 32 as internal block size, except for the case described in Section 5.4, which uses 900/90.

4. Performance Modeling and Abnormality detection of Irregular QR Sparse Tasks

Performance models of compute kernels can be used to improve scheduling and load balancing, predict performance, and post-mortem performance analysis. For example, StarPU uses such models to guide task scheduling policies at runtime. Models can also reveal many possible performance problems and hint the analyst toward a more in-depth analysis of specific application regions where those anomalies occurred. Enriching space-time visualizations with such information has been successfully applied in the context of dense linear algebra [11]. However, the irregularity of the sparse factorization tasks discussed in the previous section calls for more elaborate techniques. We first detail how three regression strategies can model the irregular tasks of `QR_mumps` and how they enrich the Gantt-chart. Then we present four scenarios that showcase how identified anomalies allowed us to discover and address simple to non trivial performance issues.

4.1. Regression Models for the Irregular Tasks of `QR_mumps`

The duration of a task obviously depends on the kernel type (`geqrt`, `gemqrt`, `tpqrt`, and `tpmqrt`) and whether it is executed on a CPU or a GPU. A good estimation of the amount of Flops incurred by these tasks in the dense case can easily be obtained using the `nb`, `mb`, `ib` geometry and granularity parameters [36] and then used as a surrogate for the task duration. For example, the `gemqrt` kernel which multiplies an arbitrary real

Table 3: Specific configurations used for all runs used in the showcase of Sections 4 and 5.

Section	Figure	Place	Scheduler	Ordering	Machine	Input	Memory
4.2	5	–	lws	Scotch	Hype	flower-8-4	Limited
4.3	6	(A).Left	lws	Metis	Hype	flower-8-4	Limited
		(A).Right	prio	Metis	Hype	flower-8-4	Limited
		(B)	prio	Metis	Hype	Rucci1	Limited
		(C).Left	dmdasd	Metis	Draco	Rucci1	Unlimited
		(C).Right	prio	Metis	Draco	Rucci1	Unlimited
4.4	7	Left	lws	Scotch	Draco	ch8-8-b3	Unlimited
		Center	lws	Scotch	Draco	ch8-8-b3	Unlimited
		Right	lws	Scotch	Hype	ch8-8-b3	Limited
4.5	8	–	lws	Scotch	Draco	flower-8-4	–
	9	Top, Up	lws	Metis	Draco	flower-8-4	Unlimited
		Top, Down	lws	Metis	Draco	flower-8-4	Unlimited
		Bottom, Up	lws	Scotch	Hype	karted	Unlimited
		Bottom, Down	lws	Scotch	Hype	karted	Unlimited
5.2	11	–	lws	Metis	Hype	e18	Unlimited
	12	–	lws	Metis	Hype	e18	Limited
5.3	13	Top	prio	Metis	Tupi	degme	Unlimited
		Bottom	lws	Metis	Tupi	degme	Unlimited
5.4	14	Top	heteroprio	Scotch	Hype	TF17	Limited
		Bottom	dmda	Scotch	Hype	TF17	Limited

$m \times n$ matrix C (using block size b) by the real orthogonal matrix Q obtained from a QR factorization and represented by the product of k elementary reflectors incurs:

$$\texttt{gemqrt}(m,n,k,b) = \sum_{j=0}^{k/b-1} \left[4(m-jb)nb - b^2 n\right] \approx 2nkb(2m - \frac{k}{b}) - bnk$$

In such case, it is natural to assume that the duration of the task will be proportional to this amount of Flops. However, we deal with a sparse structure and each front may have a stair structure (as depicted in Figure 2). `QR_mumps` wisely considers that and invoke a dense BLAS call for each stair step. As a consequence, the real amount of Flops does not depend solely on the geometry and granularity parameters but also on the stair structure. Figure 3 presents the actual duration of the `gemqrt` task versus the prediction obtained using a simple linear model based on the above Flop estimate (Left) and the prediction using a full polynomial model with the parameters m, n, k, b (Right), that generally leads to better estimates as it is more flexible and may account for memory access and for subtle cost differences in arithmetic operations. Unfortunately, in both cases, the dense estimation is generally very pessimistic and predicts duration much higher than the actual duration. Fortunately, it is not too difficult in `QR_mumps` to combine the above dense model with the staircase structure and ignore the left-lower zero part of the block to better estimate the actual computational load (in GFlops) incurred by each task.

StarPU propagates this theoretical GFlops estimation for each task to the trace allowing us to relate against the task duration (in milliseconds), as shown in Figure 4. Unlike what could be observed in when comparing Figure 3 with the upper left facet of Figure 4, there is no more gross underestimation. Except for a few tasks, the predicted duration (in green) closely matches the actual duration. As one would expect, the behavior

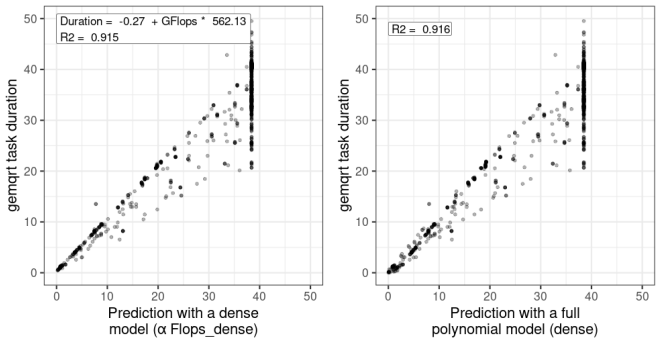

Figure 3: Task duration in milliseconds against the prediction obtained with a dense model for `gemqrt` (BLAS 3.8.0). Point transparency indicates measurement intensity.

is broadly linear and the efficiency (the slope of the regression) is very different from a compute kernel to another and from a BLAS implementation to another. Yet, the classical linear regression assumptions do not hold. In particular the variability is neither normal not constant (duration for larger theoretical flop counts are more variable and generally positively skewed) and this assumption is more or less strongly violated depending on kernels and BLAS versions. Since the duration is always positive and the variability appears to grow linearly with the flop count, we handle this heteroscedasticity using a simple log-log transformation before the linear regression but it remains limited and is generally insufficient to efficiently detect outliers. Consequently, following an exploratory data analysis approach, we leave the analyst the choice of the modeling strategy for each compute kernel depending on what appears the more suited. In our investigation, we propose three: classical linear regression (for simple well-behaved cases), robust linear

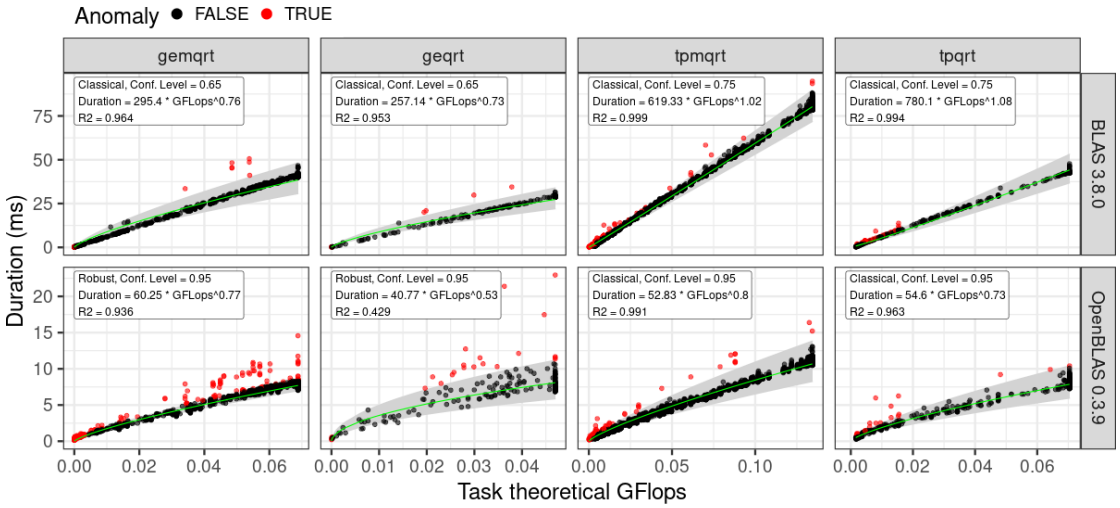

Figure 4: Task duration in milliseconds as a function of the theoretical GFlop count. The green line and the gray ribbon represent the prediction and prediction interval derived from the model formula $\log(Duration) \approx \log(GFlops)$ fit over the data with different strategies (Classical, Robust) and confidence levels (from 0.65 to 0.95). Red tasks are the ones classified as potential anomalies by the model (those outside the ribbon). Point transparency indicates measurement intensity.

regression (when the noise structure follows a skewed heavy-tailed distributions) [38], and mixture linear regression (when the behavior is multi-modal) [44]. Figure 4 depicts the usage of the classical and robust linear regression, while Section 4.5 presents the usage of the mixture linear regression. Although even the simple linear models allow for an excellent modeling of standard BLAS implementations like NetLib BLAS (upper row of Figure 4), it is interesting to note that optimized BLAS implementations like OpenBLAS (lower row of Figure 4) appear to have a much larger variance than the former ones and may seem more unstable or difficult to model (visually there are more outliers and more variance), especially for the `gemqrt` and `geqrt` tasks. This is why it is preferable to use the robust linear regression for the `gemqrt` and `geqrt` tasks, and a classical linear regression for the other tasks. Despite this variability, such libraries have much better performance than the BLAS implementation, and are thus heavily used in practice.

By analyzing the model prediction intervals, we can check the model adequacy in fitting the data and detect outliers within task and resource types using the prediction upper limit given a confidence level. By checking whether the observations lie above this confidence line and, the tasks are classified as a potential anomaly and represented in red in Figure 4. The anomalous task classification can then be used to enrich the space-time Gantt chart by giving anomalous tasks more intense colors than those whose duration is near what is expected. As we will show in the next subsections, unless they appear as being completely random, the spatial and temporal location of these tasks is generally a sign of performance issues.

4.2. *Case 1: Influence of the Submission Thread*

During our investigations it was common for outlier tasks to appear on a single core and to be rather grouped in time, as if there was short temporal perturbations. Figure 5 contains a representative example of such configuration, combining the task submission panel, with the number of submitted tasks along time (top), with the space-time view, enriched with anomaly detection by our performance model (bottom). We can see that task anomalies (B.1, B.2, and B.3) coincide with a steep increase of task submissions (A.1, A.2, and A.3). The reason is that StarPU has a main thread responsible for handling task submission, which can occur at any moment of the application execution. In most cases, the thread unrolls the graph of tasks at the beginning of the execution. Nevertheless, for scenarios with memory limitations, as the one investigated here, task submission can be postponed according to memory availability. In Figure 5, the submission thread, pinned to CPU9, causes computational tasks to have a slightly longer duration because it competes for the same resources. When binding the submitter thread to a dedicated core, the anomalous tasks disappear.

Although this scenario clarifies the cause of task anomalies,

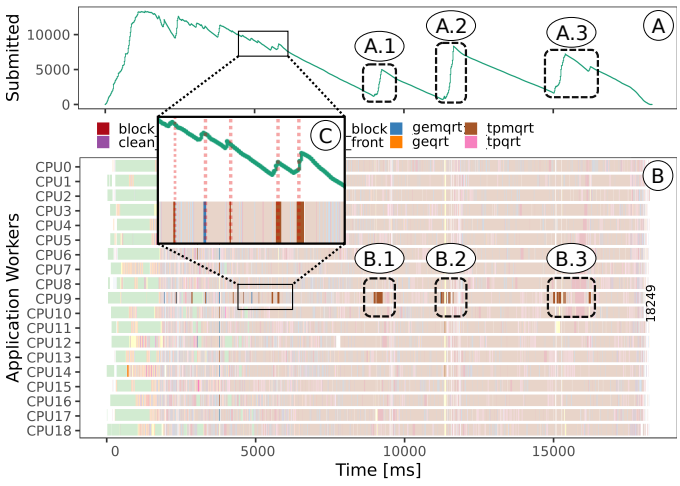

Figure 5: Panel A shows the number of tasks submitted over time, while panel B presents the application workers and the tasks they executed. Anomalies are associated with the task submissions (A-B 1, 2, and 3 pairs). The submission thread is fixed in CPU9, where these anomalies occur. A zoom (Panel C) shows that even small numbers of task submissions can also cause anomalies.

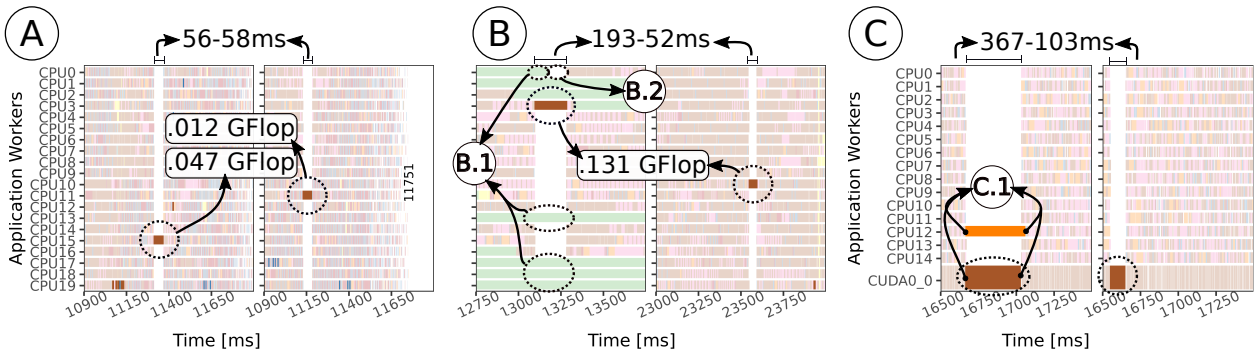

Figure 6: Six examples demonstrating the presence of task anomalies in the time span of ≈1 second. Case (A) has two different executions for the `Hype` machine (with the `lws` scheduler on the left and with the `prio` scheduler on the right). Case (B) has two time slices of the same execution in the Hype machine, and case (C) has two different executions for the Draco machine with GPU (with the `dmdasd` scheduler on the left and `prio` scheduler on the right).

it does not characterize a runtime-level performance problem. Indeed, such overhead is expected albeit rarely visible and our outlier detection mechanism gracefully revealed it. Task submission is unavoidable but as our experiments indicate, the additional cost of these submissions is negligible and it is generally better not to dedicate a core for submission. We can thus simply treat this core separately, having its outliers aligned with task submissions disregarded, focusing on other potentially anomalous tasks as we see next.

4.3. Case 2: Tracing-related Perturbations

Another common behavior we noticed during our experiments appears as a global idle time spanning all workers. Such absence of tasks may have several explanations: a natural lack of parallelism, bad scheduling decisions, data transfers limitation, and so on. Commonly, it remains complex to identify the reason for each noticeable behavior. Here, we describe another interesting, albeit more trivial, cause that appeared in many different combinations of workload, machine, and computational resources and which is illustrated through six instances on Figure 6 (the six facets). It appears to be a time-dependent phenomenon for some runs, but it looks fairly random for others. These cases share a common characteristic that, when idling, workers are in a so-called "overhead" runtime state (among other states such as scheduling, fetch, sleeping). Sometimes, only one task continues its execution, while other tasks remain dormant, as shown by the two (A) graphic cases where outlier tasks have very different observed GFlops but with similar duration. In other scenarios, other non-anomalous tasks that have already started are capable of continuing their execution, as shown by the left-case of the (B) graphic where outlier tasks have the same observed GFlops but very different duration. Sometimes, as in the left execution of (C), this idle period was responsible for up to 14% of the total worker idleness. Interestingly, our performance model always identifies an anomalous task which coincides perfectly with the idle period. There is no reason why a task shortage or a bad scheduling decision would suddenly cause a task slowdown, which allows to rule out many possible explanations. Furthermore, as shown in (A) and (C) the problem is almost reproducible: although the outlier tasks are different among executions, the perturbation generally occurs roughly at the same time regardless of the scheduler.

We found out that this anomalous event is related to the trace dump during the application execution. This occurs when the trace buffer, limited by the `STARPU_TRACE_BUFFER_SIZE` variable, gets full. Increasing the buffer size to a huge value actually removed this phenomenon and allowed us to concentrate on more intricate performance problems.

4.4. Case 3: Uncovering Numerical Stability Issues

We have also identified some consistent workload-dependent anomalies that did not correlate with particular moments nor with particular cores. Figure 7 illustrates an example when factorizing the `ch8-8-b3` matrix in the `Hype` and `Draco` machines, using the `lws` scheduler. We have confirmed that all tasks tagged as anomalies by our model were not caused by tracing or submission perturbations as in our previous analysis. Furthermore, the task's duration difference magnitude was enormous, from ≈9ms up to 150ms for `gemqrt`, and from 7ms to 65ms for `geqrt`. The increased cache misses for some tasks identified as anomalies do often explain a longer duration but neither cache misses nor other hardware counters like the total floating-point operations changed for those tasks, yet their duration always stands out compared to the other tasks.

Still on Figure 7, we noticed that the `gemqrt` anomalies were coming from all the same 20 task identifiers, preceded by an equally anomalous `geqrt` task, even when we executed with different schedulers and in different machines. A sequential execution pointed out to the same 21 anomalous tasks. The first two images from left to right are executions on the Draco machine with the `lws` scheduler. The first does not use GPU, and the anomalies are spread along with the execution, while the second figure, with one GPU, depicts the same tasks executed very close to each other. The last figure also presents the same `geqrt` and `gemqrt` tasks classified as anomalies but for the `Hype` machine execution. We have also noticed that many of the `tpmqrt` anomalies are part of the same elimination tree node. However, their variability is much smaller than the other anomalies. Furthermore, the amount of `tpmqrt` anomalous tasks is different from an execution to an other unlike what happens for the presented `geqrt` and `gemqrt` tasks.

As they consistently occur over the same tasks that work in the same matrix block, we investigate whether this difference

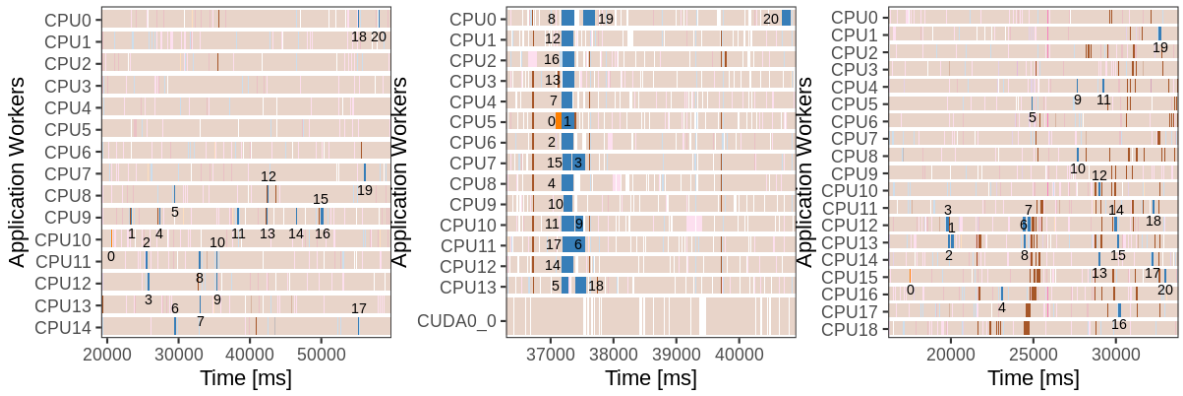

Figure 7: The same 20 blue `GEMQRT` tasks classified as anomalies are dependent from the same orange `GEQRT` task, identified with 0 in the plots.

comes from the block's spatial position and its numerical content, guiding or preventing some architecture-specific optimizations. Thus, we dumped the binary content of the blocks that those tasks use to investigate their content. We found that some of those blocks contain many denormal numbers, which is any nonzero number smaller than the smaller number in the IEEE standard for floating-point arithmetic. When present, the Intel processor executes multiply, divides, and square root operations with longer latency. If the application does not need denormal precision, we can improve application performance by enabling specific control flags. For example, the SSE/AVX floating-point units of the x86_64 processors architecture have the control flags flush-to-zero (FTZ) and denormal-as-zero (DNZ) to define the operations' behavior when encountering a denormal number [23]. This way, we recompiled the application using these flags, and the anomalies disappear. Although such deactivation removes the anomalies, we recommend the adoption of the Metis ordering in this case to guarantee numerical stability.

4.5. Case 4: Identifying Locality Efficiency Issues

Finally, during our investigation, we stumbled upon situations where the usage of our model with classical or robust linear regression (see Section 4.1) did not fit data as appropriately as it should. Such a case study is depicted in Figure 8 where variability is much more important and where a more complex model is needed to describe the data more correctly. This lack of adequacy of a simple model is easily checked by inspecting the residuals and is notably interesting as it is generally the sign of a more profound problem in the execution. In such cases, we resorted to finite mixture models technique to classify the data in different clusters (Figure 8) where `geqrt` and `tpqrt` have two regression lines instead of one. We can then investigate where those clusters are located in Gantt chart's space and time, checking if the clusterized tasks are time or space-related, giving us insights about this unexpected behavior.

The top row of Figure 9 depicts the corresponding Gantt chart for the `flower_8_4` matrices where many tasks (without transparency) are considered as anomalies by our model (see, in particular, the "slow" tasks). They appear throughout the execution at moments different from those indicating new task submission or those aligned with overhead states. A careful analysis of these anomalous tasks against our model indicates that the theoretical GFlops provided by the application is no longer capable to explain the task duration.

We hypothesize that they suffer from an increase in the number of cache misses, affecting our model's prediction capability. To check if the number of cache misses could explain the increased duration, we linked StarPU with the PAPI library to capture the total number of L1, L2, and L3 cache misses for each application task. We first verified if there was some correlation between the total miss number for caches L1, L2, and L3 and task duration for other experiments. In general, GFlops, L1, and L2 misses have a strong positive correlation with task duration. The L3 cache misses explain less of the variability for well-behaved tasks. However, in the cases where a mixture model seems more appropriate, it is the opposite: GFlop, L1, and L2 misses fail to be a good explanatory variable while the L3 total cache misses is a better explanatory variable.

Table 4 presents evidence of the enormous difference between the sequential and the parallel total time per task for the `flower_8_4` input matrix. In the sequential version, tasks have no interference from other concurrent tasks, allowing us to capture their expected behavior. We observe that the parallel `geqrt` tasks is 3.37× slower than the sequential case. The worst-case is the `do_subtree` tasks, 12.8× slower. The slow `do_subtree`

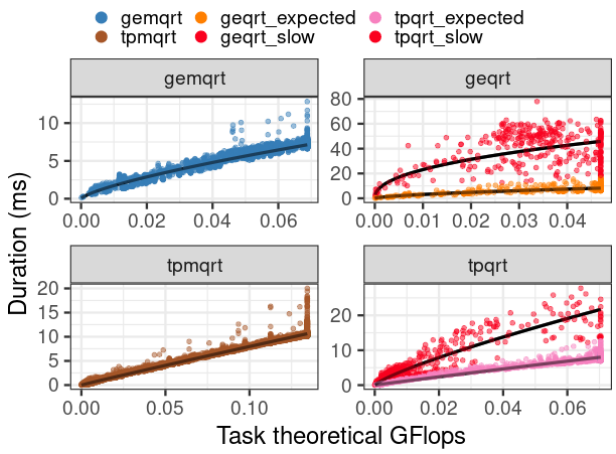

Figure 8: Using finite mixture models to fit multiple models over the data.

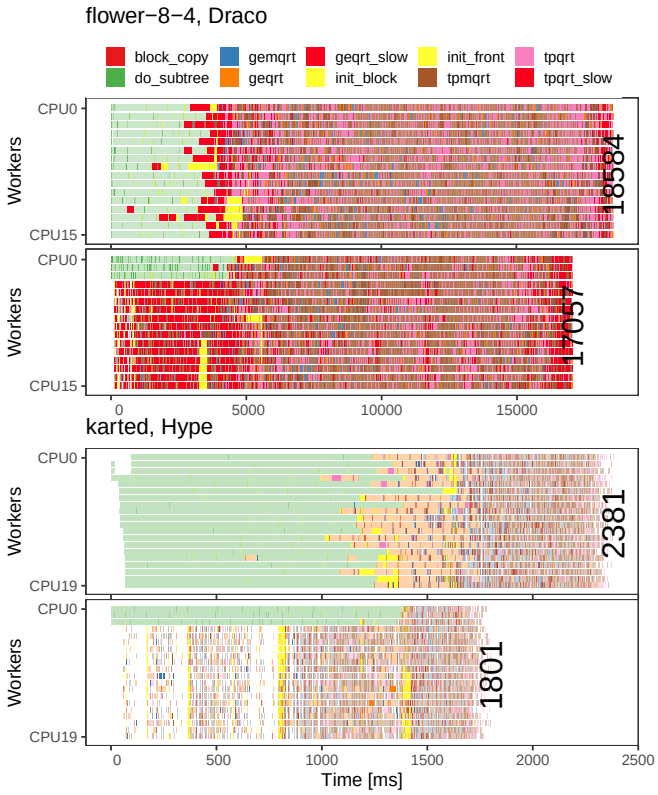

Figure 9: Anomalous `do_subtree`, `geqrt` and `tpqrt` tasks (marked as slow) when carrying the factorization of the `flower_8_4` (top) and `karted` (bottom) matrices using all available workers (top row, parallel); Factorization of the `flower_8_4` and `karted` matrices when restricting the `do_subtree` tasks to run only in three CPU resources (bottom row, throttling). Despite the presence of remaining anomalous tasks when throttling, this execution is faster than that of all resources available for all tasks, as shown in the top row.

tasks share the same explanation as for the `geqrt` and `tpqrt` tasks since the `do_subtrees` have the same kernels. These results confirm the locality efficiency problem [17].

Table 4: Task time of `flower_8_4` factorization in the `Draco` Machine.

Task Type	**Total time** sequential	**Total time** parallel	**Total time** throttling
`do_subtree`	3.89s	49.98s (12.8×)	13.09s (3.36×)
`geqrt`	4.41s	14.87s (3.37×)	13.86s (3.14×)
`tpqrt`	11.39s	15.53s (1.36×)	16.81s (1.48×)
`gemqrt`	26.88s	32.48s (1.21×)	33.02s (1.23×)
`tpmqrt`	152.21s	171.88s (1.13×)	173.09s (1.14×)
`block_copy`	1.40s	1.85s (1.32×)	1.83s (1.3×)

A careful observation of the Figure 9 (top row) indicates that slow tasks concentrate in regions where there are many `do_subtree` and other simultaneous `geqrt` tasks, which suggests they affect cache reuse in two ways: (1) `do_subtree` use a significant amount of memory without much reuse as the other 2D tasks do, and (2), `geqrt` tasks are executed spatially far from each other. They are either the starting factorization task of a tree node or its trailing submatrix which does not not share matrix blocks with other `geqrt` tasks. Such characteristic can be the case for the `tpqrt` tasks too, which traverses the matrix row by row.

To alleviate this locality efficiency problem, we execute the application by limiting the execution of `do_subtree` tasks to only three CPU cores. The bottom row of Figure 9 depicts the resulting behavior once again for the `flower-8-4` matrix. To illustrate how this strategy works for other inputs, we show in the bottom of Figure 9 the same situation for the `karted` input matrix. The first three CPUs run all `do_subtree` tasks while all cores are responsible to execute remaining task types. By restricting the execution of these memory-bound tasks, we limit the available parallelism, which creates idle time in the beginning but we also reduce the makespan from 18.5s to 17s (≈8% reduction) for `flower-8-4` and from 2.3s to 1.8s (≈22%) for `karted`. Specifically for the `flower-8-4` input, Table 4 provides, in the throttling column, the total time to compute all tasks of a given type. In this scenario, throttling improves the efficiency of `do_subtree` tasks making it closer to the sequential total time (without any interference), which improves performance overall.

5. Visualizing how the Multifrontal Factorization Unfolds

As explained in Section 2.1, sparse solvers based on the multifrontal method rely on an elimination tree structure. In this section, we first provide a set of visualization panels related to this multifrontal structure to depict application behavior along time. They include panels that show the tree structure enriched with application computation, the aggregated resource utilization by tree node and tree depth, and memory utilization along time. The combination of these panels allows to perceive correlations since they are temporally synchronized.

We start by explaining these different panels in details in Section 5.1 and then we present three scenarios that showcase how these panels can be used in practice for the performance analysis of `QR_mumps`. These panels enable us to carry out interesting comparisons that are frequently questioned during the evaluation of a sparse task-based solver, such as the effect of memory limitation, and the adoption of different runtime schedulers.

5.1. Visualization Panels inspired by the Elimination Tree

Main Panel: Elimination Tree Visualization

The Elimination Tree panel provides us a macroscopic view of the execution guided by the tree structure, as defined by the experiment's ordering algorithm and the fronts' submission order. Figure 10 details the data transformations to obtain the final visual design. We start with the graph of the elimination tree as given by `QR_mumps` application (A). We then position the elimination tree nodes in the vertical axis according to their submission order, and prepare the positioning of all events (e.g., memory allocation/deallocation) of each node in the time axis (horizontally) (B), and finally enrich such a view with several elements from the traces such as main events and resource usage (C) and which we detail hereafter.

Figure 11.(A) depicts a representative example of the elimination tree panel, showing for how long the elimination tree nodes (listed in the Y-axis according to their submission order)

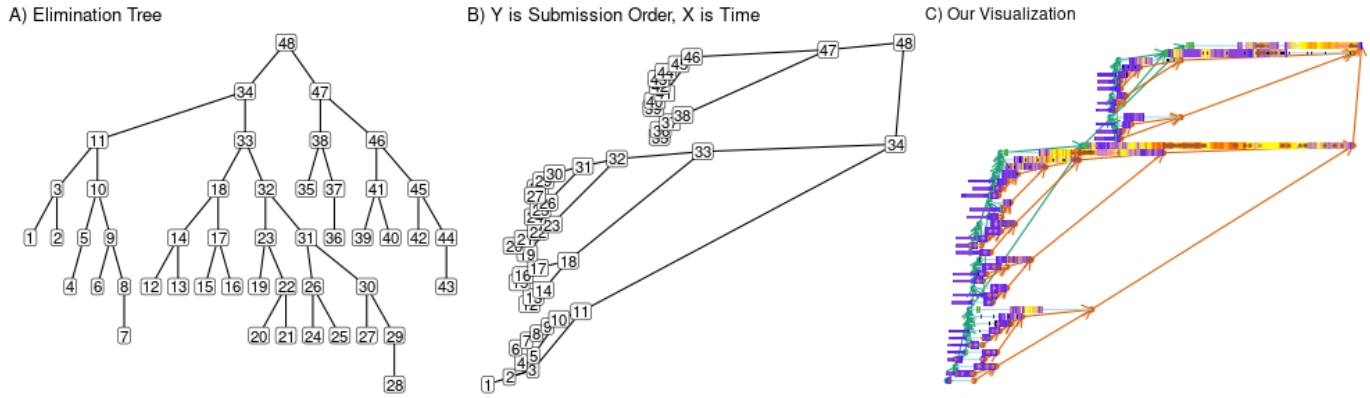

Figure 10: From the Elimination Tree graph generated by `QR_mumps` to our elimination tree visualization: an intermediary step rotates the graph and lists nodes by their submission order (in the vertical axis) and time (horizontal) prior to the enrichment in our view.

exist along time (the X-axis). This representation has two main elements: the nodes of the tree and their parent-child connection. Each node of the tree occupies a horizontal coordinate in the panel. A line for each node starts with the first memory allocation task (represented by a green rectangle) and ends with the task that releases the node memory. The lifetime purely indicates memory footprint, not meaning that there are computations over this node this whole time. Green and orange arrows indicate the parent-child connection. The green arrows depart from the beginning of a node's line and points to its parent. The same happens for orange arrows but considering the end of a node's line. To represent computations over the structure, and considering that many resources compute tasks of a given elimination tree node, we employ the color gradient to represent the computational load intensity (based only on factorization tasks) as a percentage of the total number of computing resources. For example, nodes 34 and 35 share resources between 2s and 2.8s (yellow color), but then, the scheduler drives resources to compute tasks of node 35, as we can see by the red color from 2.8s until 3.2s. This color gradient representation suffers temporal aggregation in user-configurable time intervals (100ms in this case) because there may be too many events per node. We can alternatively represent other performance metrics, like the instantaneous GFlops throughput. For simplification, we group the sequential nodes pruned by the application by their common parent, reusing its Y position, and spatially aggregating their computations because they generally are numerous. Pruned nodes also share the same color gradient that represents compute intensity used for intermediary tree nodes. Thus, to differentiate them, we use half of the height of the non-pruned nodes for their representation.

Furthermore, we can represent other aspects of the application in this tree plot, like the communication tasks between parent and child nodes and the anomalous tasks' location. The black rectangles (inset within each tree node line) in Figure 11.(A) represent the tree's communication, which comes from the assembly of the contribution block of a child in its parent front (such situations are not very frequent in this example but can be more clearly seen in Figure 13 for example). We use the raw communication tasks duration to represent them with transparency to know when we have a higher concentration of these tasks, which also has half of the height of its node representation to avoid a complete overlap in the plot. Lastly, the elimination tree plot uses dots inside the node's computation marker to depict the anomalous task space/time location in the tree, with the same color used in the Gantt to identify their type.

By evidencing where are the computations, initializations, communications, and anomalies, our elimination tree panel shows precisely how the scheduler traverses the tree, including the prioritized or postponed paths or nodes. This panel also depicts the three levels of parallelism: the tree parallelism when representing concurrent nodes in different Y positions, the node parallelism with the color gradient, and the interlevel parallelism by depicting overlapping computations between parent and child nodes. We can use this complete elimination tree view, allied to all the other plots, to identify specific application moments and have a clear view of its behavior. Also, one can promptly see if the tree is tall, short, thin, or wide. The tree structure impacts the number of available tasks, memory consumption, and how the scheduler traverses the tree to provide enough parallelism, serving as a general signature of how scheduling decisions evolve during the execution.

Auxiliary Panels: Node and Depth Resource Usage

The elimination tree panel is handy to indicate details of how the multifrontal method evolves, but it lacks an aggregated view of computational power. In this sense, two additional panels provide a more succinct view of how much computing power is dedicated to processing each elimination tree node. The panel of Figure 11.(B) indicates the tree node parallelism, that is, the cumulative resource utilization of the computational tasks associated with each elimination tree node (colors) as a function of time. Alternatively, the panel of Figure 11.(C) provides an option to fill colors of the same plot using the tree node depth (the tree distance from the root node), illustrating how the scheduler traversed the tree concerning this depth property. The node panel colors purely differentiate one tree node from another, not identifying each node individually. So the node panel reuses colors to represent nodes without overlapping task executions. However, for the depth panel, the color scale has a meaning

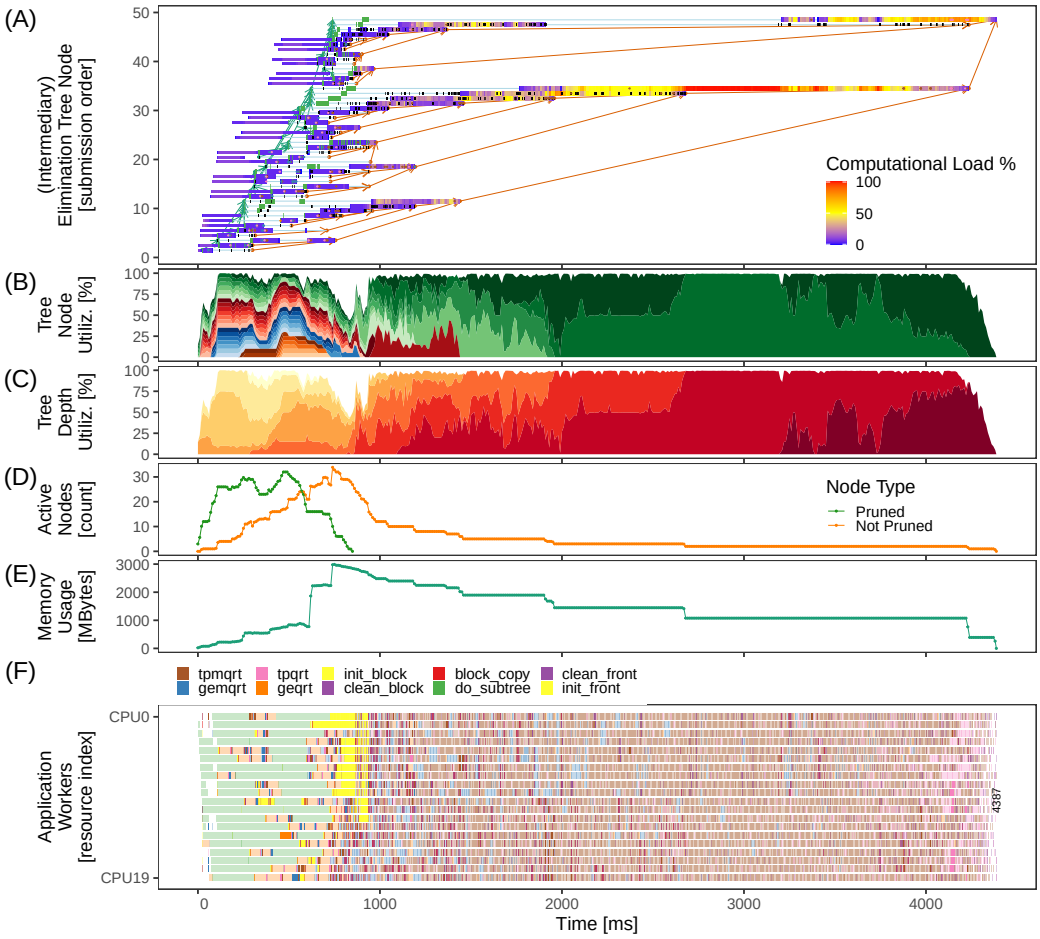

Figure 11: Overview of visualization panels tailored for the elimination tree, using the `e18` matrix on `Hype` as an example. The elimination tree panel (A) shows the tree structure and the computational signature along time. Panels (B) and (C) depict the resource utilization by the computational tasks highlighting the usage per tree node and tree depth. For every time interval, the colors indicate an elimination tree node (B) and the depth of the tree (C). Panels (D) and (E) show the number of parallel and sequential active nodes in memory, and the memory used by these nodes. For reference, we also depict Panel (F), a classical Gantt chart.

that represents the distance from the root, represented with a darker color. For both panels, we temporally aggregate the time spent in computational tasks in user-configurable intervals to define the resource usage of a given elimination tree node or depth (100ms in this case). When combined with elimination tree visualization (see Figure 11.(A) for an example), this plot's specific shape can be considered a signature of the scheduler regarding other application properties like task priorities, memory availability. We can see, for instance, how computing power tackles the parallelism of the tree structure.

Auxiliary Panels: Active Nodes and Memory Usage

Tree traversal may significantly impact memory consumption, and reversely, any memory restriction can significantly impact tree traversal. Therefore, the visualization of the available parallelism and tree traversal is relevant to the analysis.

In the multifrontal method, the child nodes need to merge their contribution blocks to their parent node. This dependency implies that all nodes involved in this operation must be present on memory at that moment. Because memory is a finite resource, such applications can easily consume a large portion of the available memory. Panels (D) and (E) of Figure 11 provide a summary to understand better how the number of in-memory active nodes and current memory usage evolves through execution time. Since `QR_mumps` can work with memory usage constraints to keep memory usage under control, these panels help understand how the application handles memory-related issues in scenarios with a memory constraint. The two lines in the in-memory active nodes panel indicate whether nodes are sequential (pruned by the application) or parallel tree nodes. Depending on the tree traversal, it is normal that sequential nodes only exist at the beginning of the execution because they are the leaves of the tree. That is the case shown in Figure 11.(D).

5.2. *Case 1: The Influence of the Memory Limitation*

For all experiments so far, we have noticed that, except in cases where the memory peak threshold is particularly limiting, the `QR_mumps` application can keep the performance very similar to cases without memory usage restrictions and even provide better results sometimes. This effect has been discussed by [21], but now, we can observe the interplay between the memory limitations and the elimination tree exploration by the scheduler. Figures 11 and 12 show two different executions with the factorization of the `e18` matrix using the Metis reordering in the `Hype`

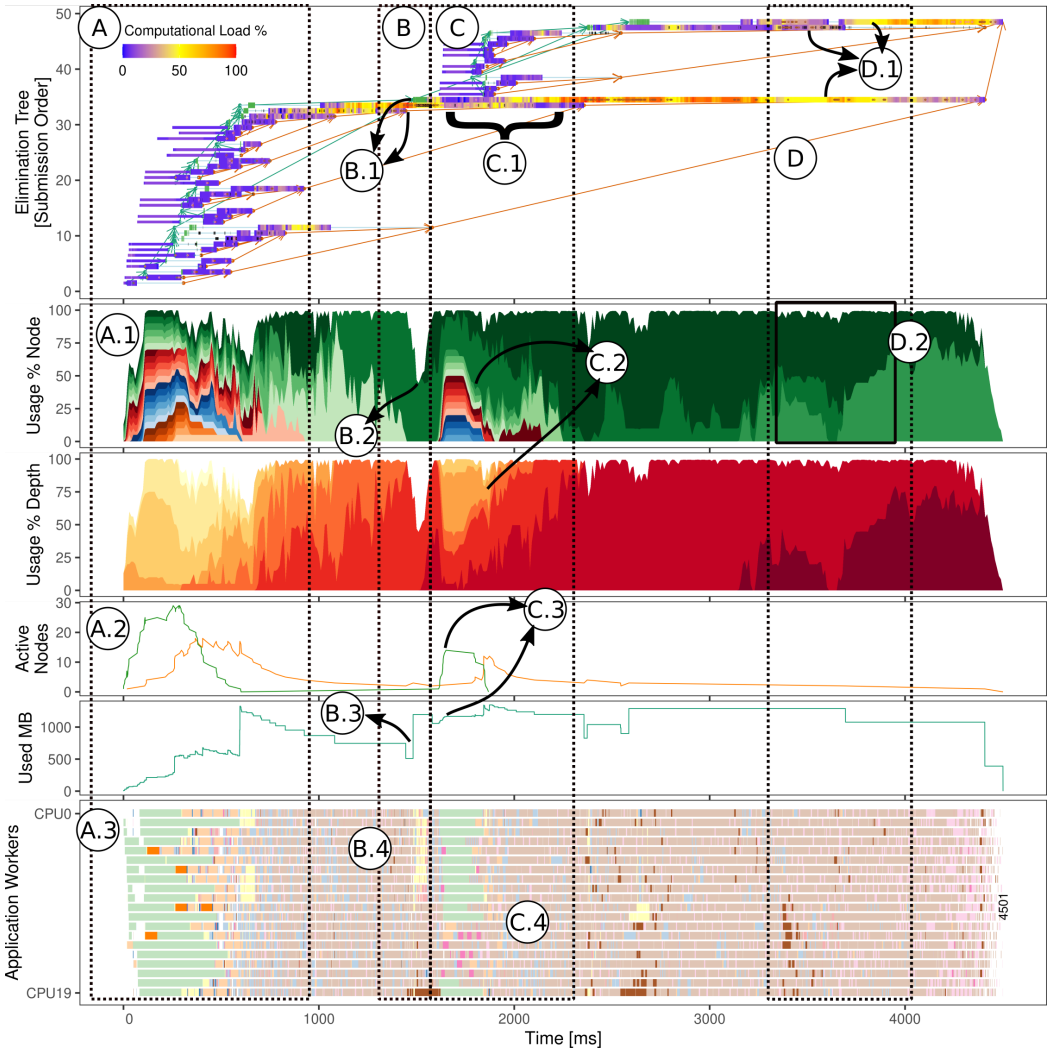

Figure 12: Tree-related plots and the Gantt chart for an execution of matrix `e18` in `Hype` machine.

Machine. The only difference between them is that the former depicts a case without memory limitation (the peak is 3GBytes), while the latter depicts the execution when limiting the memory to the sequential peak (≈1.4GBytes). The makespan comparison between the unlimited (Figure 11) and limited (Figure 12) executions indicates that the latter is insignificantly degraded but their internal structures are very different. Without limit, we observe in Figure 11 that the entire tree is allocated early in the execution. When limiting memory usage, memory allocations are delayed until the last moments of the factorization (see Figure 12). This exploration impacts the available tree-parallelism. In Figure 12.(B), we observe that around 1.5 secs of execution, half of the tree intermediary nodes were not touched yet by the execution with memory constraint, while the unlimited case has started and even finished computing many other nodes.

We can see in Figure 12 (A.1) that in the beginning of the execution, tree-level parallelism is available, as shown by the many colors that appear in the resource usage per node plot. The bottom part of the tree is composed of many small nodes, indicating a tree that is sufficiently wide to provide enough parallel work to application workers. Furthermore, in (A.2), we observe that the vast majority of the nodes active in memory are pruned nodes, which is also presented by the number of `do_subtree` tasks in (A.3). The (B) and (C) areas point out to multiple delays in node allocation imposed by the memory constraint parameter. For example, we can see how the freeing of the node pointed in (B.1) allows the allocation of many other nodes. The amount of memory initialization tasks (`init_front` and `init_block`) reduces the compute resource utilization as shown in (B.2). The memory usage plot in (B.3) depicts exactly when the memory becomes available. Because the newly allocated node would use much memory, it had to wait for some other node to free the needed memory. All these memory initialization tasks can also be seen in the Gantt chart in (B.4, yellow color) but would be hard to interpret without the tree view. A similar case appear in the (C) area, where the set of nodes (C.1) get delayed because the execution has already reached the memory limit (C.3). We confirm that most of the nodes are the leaves of the tree (C.2), as also shown by the `do_subtree` tasks in green (C.4). In the last scenario (D), in (D.1) and (D.2), we can observe how the application can overlap computations between the last three dependent nodes of the

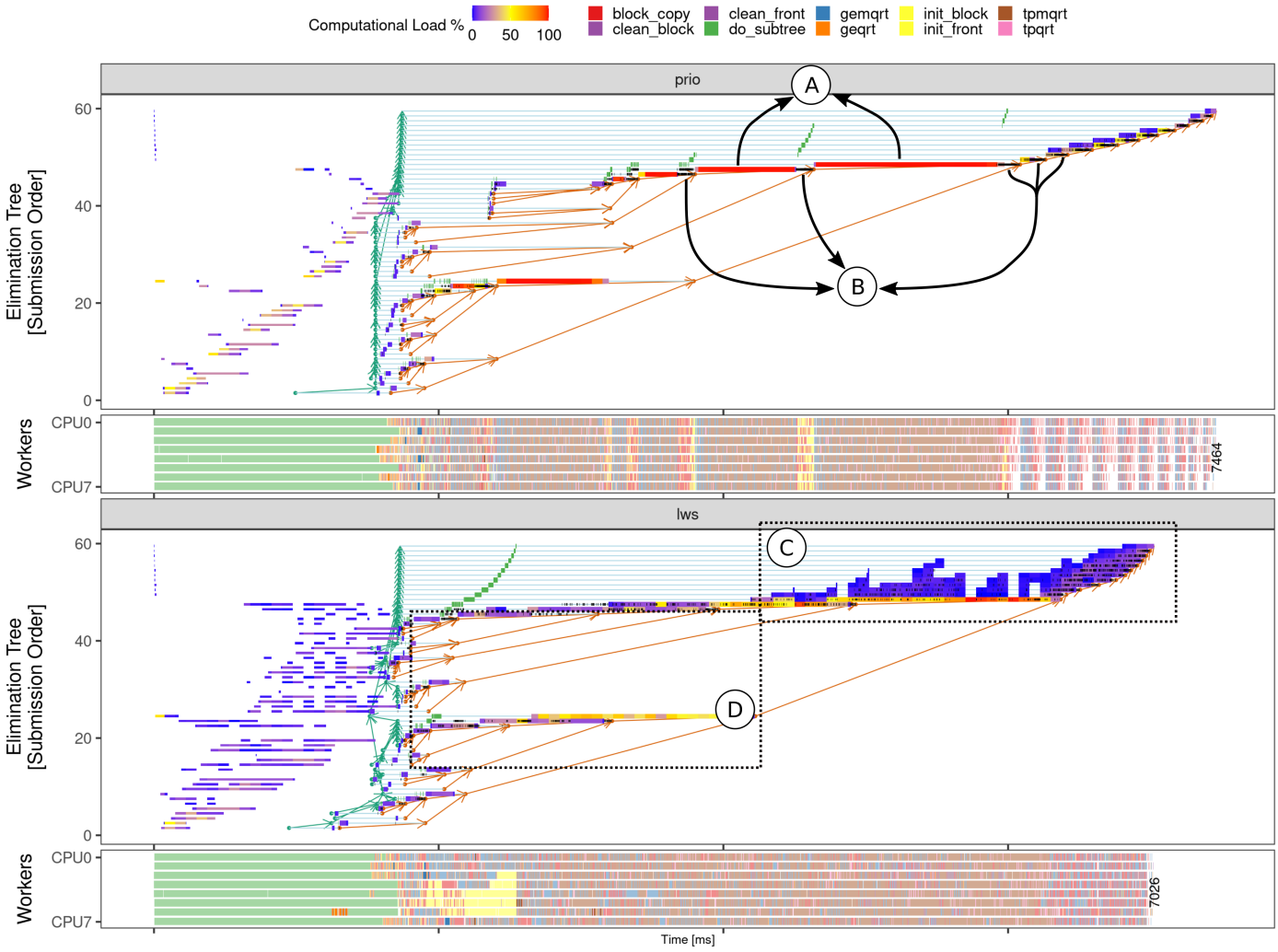

Figure 13: Comparing `prio` (top, with the elimination tree and gantt-chart) and `lws` (bottom) schedulers for the `degme` matrix in the Tupi Machine.

tree. This is possible thanks to the finer-grained tree partitioning using a DAG structure as already discussed in Section 2.1.

Interestingly, the memory restriction distributes the execution of memory-bound operations (such as `do_subtree` tasks) throughout the execution, which generally improves the locality efficiency and could have provided a similar effect as when using a small number of cores dedicated to memory-bound tasks, as previously discussed in Section 4.5. Unfortunately, delaying subtree initialization (the yellow areas) seems to significantly impact `tpmqrt` tasks (as indicated by the many outliers simultaneous). Relaxing a bit the memory constraint could allow to spread a bit more these initialization tasks throughout the execution and to limit their influence on the other tasks.

5.3. Case 2: Identifying Task Priority Issues

The elimination tree panel can be used to compare different schedulers' behavior while traversing the tree for diverse hardware and software configurations. In Figure 13, we compare the performance of two StarPU schedulers: the locality-aware work stealing (`lws`) against the heteroprio (`prio`). Other configurations remains fixed: the same `degme` matrix as input using the Metis ordering without memory limitation in the Tupi Machine using only CPUs. We can observe that the makespan is lower for the `lws` scheduler and the bad scheduling decisions of `prio` are clear from the Gantt-charts although the reason why such decisions are taken is not really clear. As shown in (A), it seems that `prio` focuses computations on one node at a time because `QR_mumps` assigns increasing priorities according to the tree node submission order and that the `prio` scheduler has a single central ready task queue which sorts all tasks according to the basic tree node priority. Another `prio` behavior appears in (B), where there seems to be a clear communication pattern. Indeed the `block_copy` and `init_*` tasks are consistently scheduled after the final tasks of a tree node as they have a lower priority, delaying the child and parent communications.

While restricting computations to one or a few tree nodes at a time may improve spatial data locality, the restriction in communication reduces the availability of the tree and inter-level parallelism compared to what is achieved by `lws` which efficiently exploits the interlevel parallelism computation of the last elimination tree nodes in the area highlighted by (C), and the tree parallelism in (D). The `lws` scheduler explores more

tree parallelism and interlevel parallelism, making node computations last longer, which in this specific case is beneficial because the lack of tree parallelism at the end of the application is compensated by the interlevel parallelism and gives `lws` a clear advantage over the `prio` scheduler. This case provides an excellent example of how the exploitation of the application data structures with the performance visualization shows clearly what is happening and can help developers devise smarter scheduling strategies.

5.4. Case 3: Resource Usage of two Runtime Schedulers

We compare the `dmda` and `heteroprio` schedulers using as input the `TF17` matrix reordered with Scotch, in a scenario with memory limitation in the `Hype` Machine. Figure 14 depicts such a comparison between `heteroprio` (top panels) and `dmda` (bottom) with the elimination tree panel, workers, and ready tasks panels. The Gantt charts allow to readily observe that the makespan of the `dmda` is ≈8.5% smaller than that of `heteroprio` and suggest that the main flaws of `heteroprio` happened in (B.2) and (C.2) where many resources are idling. Yet, the reason behind this behavior remains unclear and one may wonder whether `dmda` can be improved further or not.

The panel on the bottom of the Figure depict the GFlops throughput difference between the two schedulers, with the red (resp. blue) color highlighting when `heteroprio` (resp. `dmda`) scheduler has completed more flops. Although `heteroprio` has a slightly better start than `dmda`, as shown with the red line in the left part of the rectangle (A), the advantage quickly changes in favor of `dmda`, slowly and constantly increasing the difference over time until the moment where `heteroprio` leaves many idle resources, which leads to an even steeper increase in the difference for `dmda`. Since in the first 13 seconds both schedulers have many available tasks and seem to explore the tree roughly in the same way, the only reason why `dmda` would go faster than `heteroprio` is that it makes a better use of available resources. If we first focus on the [2s-5s] time interval where both schedulers seem to keep all resources very busy, it turns out that `heteroprio` leaves the CPU idle 2.57% of the time (compared to 0.3% for `dmda`) and the GPU idle 9% of the time (compared to 1.2% for `dmda`). This is likely to be explained by the fact that `heteroprio` does not account well for transfer times between the CPUs and GPU. Given the speed difference between the GPU and the CPUs, this resource usage difference explains most of the throughput difference between `heteroprio` and `dmda` but there is also a more subtle reason.

Although `heteroprio` tries to put the tasks which are the better accelerated on the GPUs, it seems to have difficulties handling the variety of small and large tasks. Similarly to what was proposed by [11] in the dense case and using our performance model for all kernel types, we can write linear program to compute the optimal allocation of tasks to CPUs and GPUs when ignoring all dependencies. This absolute lower bound is depicted with a vertical line Area Bound Estimation (ABE) in the Gantt chart and allows to see that `dmda` is quite close to the optimal and is mostly limited by the lack of parallelism at the very end of the execution. More interestingly, since the linear program provides an estimation of the amount of tasks of each type that should be run on each resource, it can be compared with the actual allocation for both schedulers (rightmost panel) for the main task types: `gemqrt` (in blue in the Gantt chart) and `tpmqrt` (in brown in the Gantt chart), distinguishing between small and large tasks. In the ideal allocation, only the costlier `tmpqrt` tasks (with ≈3GFlops) should be allocated on GPU resources. The small ones, which are not as well accelerated should rather be executed on the CPUs. Overall, the `heteroprio` scheduler does not take into account data locality, uses a very naive cost model and disregards the theoretical GFlops cost of every task, thereby using GPUs even for tasks with a very low GFlop count for which CPUs are more suited. The `dmda` scheduler, on the other hand, is equipped with our performance model and better differentiates the situations where CPU contribution is interesting. This is all the more important for large tasks `tpmqrt` tasks which represent the majority of the work. A similar interpretation works for the `gemqrt` tasks as well: when dependencies are ignored, no tasks of this type should ideally be run on the GPUs. However, this deviation is harmless for `dmda` as `gemqrt` (in blue) run on the GPU only in the very beginning of the execution, when it helps releasing tasks. Overall, this finer analysis also explains why `dmda` manages to move faster along the tree than `heteroprio` although it is not directly visible from the Gantt charts.

We can also observe very visible differences at the end, when both runs are working on the last elimination tree nodes (74 and 75). As shown by the C.1 and C.2 annotations, the slope of the idle time in the end is much steeper for `dmda`, indicating again that it seems to make a better usage of resources than `heteroprio`. The Figure also highlights a segment of the critical path for the last `tpmqrt` task in both cases (The tasks with a red border in the end). The critical path in `heteroprio` occurs mainly on the CPU, where it schedules almost exclusively `tpqrt` tasks (in pink). Conversely, `dmda` manages to accelerate its critical path by placing more such tasks on the GPU.

Coming back to the more obvious idling situation of `heteroprio` in B.2, one can notice that it corresponds to a situation where there are relatively few ready tasks as only one node of the elimination tree (74) is active. Yet, a closer inspection to the Ready Tasks panel reveals that there are way enough tasks to keep all CPUs busy in the beginning and that task shortage only occurs in the end of B.2 (red rectangles indicate whenever there are fewer tasks than the total number of workers). It means that `heteroprio` deliberately leaves CPU idle, which happens again because it uses a naive cost model and globally considers that CPU are 25× slower than GPUs.

It is interesting to note, by looking for the corresponding part of the tree, that a similar situation occurs in B.1 for `dmda` although it is way less visible. This lack of ready tasks perfectly correlates with the beginning of a new elimination tree node (75) and to the end of two children nodes (33 and 54). Even though three nodes are active at the same time, a close inspection of the GFlops Throughput for these nodes allows to notice that the throughput is low (blue) for the children nodes while the throughput is moderate (yellow) for the father. It is very likely that the two subtrees have a sequence of tasks in the end on which most tasks of the father node depend, hence a poor

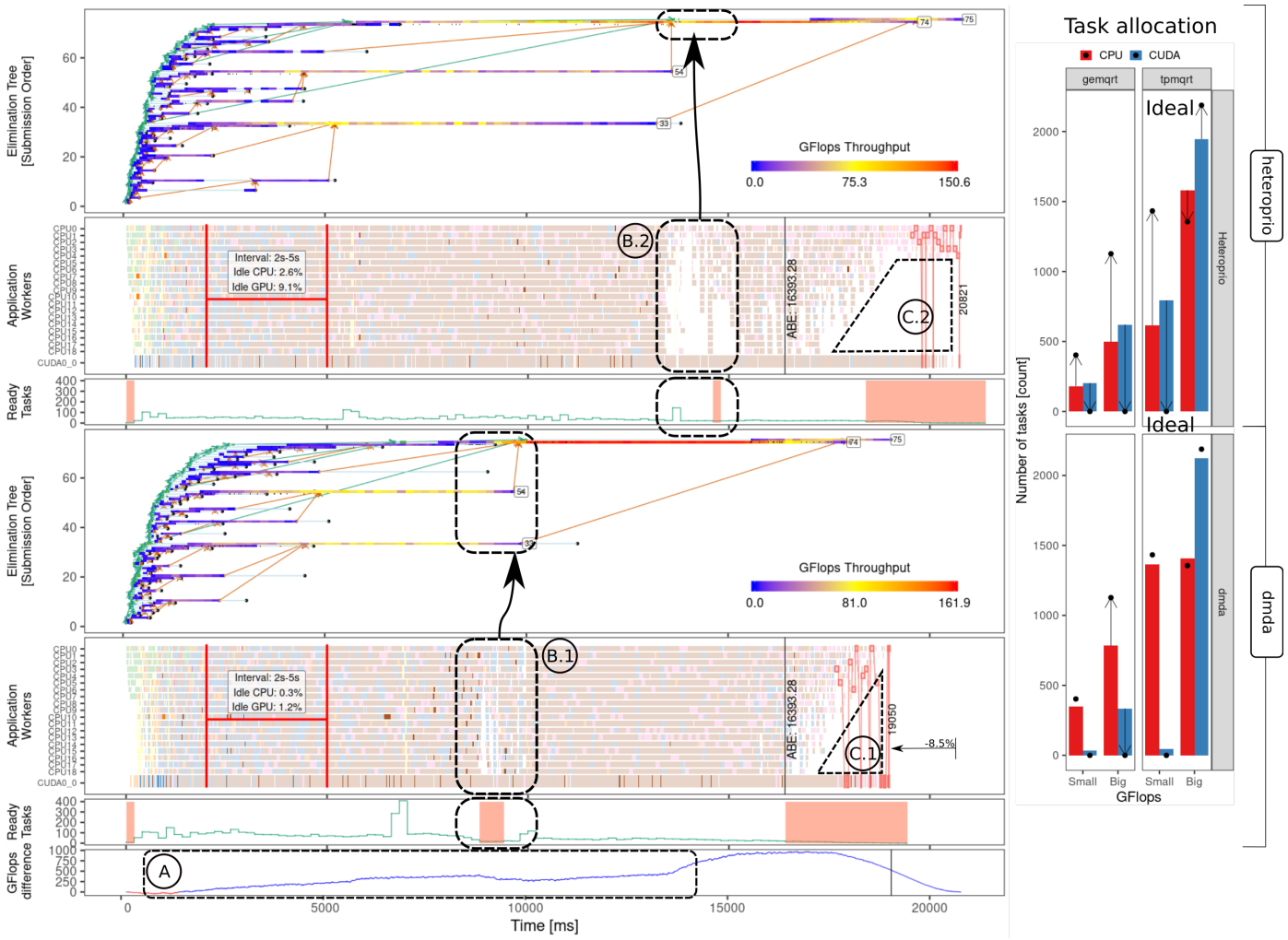

Figure 14: Left: a comparison of the application behavior when using the `heteroprio` (top, with the elimination tree, Gantt chart and ready tasks) and `dmda` (middle) schedulers, with the GFlops difference between them (bottom); Right: GFlops histograms per resource type (CPU and CUDA) for two task types (`gemqrt` and `tpmqrt`) for the `heteroprio` (top) and `dmda` (bottom) schedulers, including the ABE target as points and arrow pointing to them (to indicate departure from such an ideal case).

pipelining along the tree at this very particular moment. There is a priori not much the scheduler can do to alleviate this lack of parallelism in the DAG which acts as a loose synchronization point and would be particularly harmful if we had more or faster computing resources. The fine-grain dependencies between nodes originate from the inner row/column ordering of the fronts, which is computed independently for each node so as to minimize the amount of flops they will incur (Figure 2). By imposing an ordering for the children closer to the one of the father nodes, a better pipeline of nodes would be possible although it would come at the cost of more work.

The comparison illustrated in Figure 14 demonstrates the usefulness of the elimination tree panel to understand the unfolding of such complex applications and potential performance issues. Although it does not play much role in this particular scenario, it is for example visible that we are dealing with memory limitation scenarios. Indeed, in both cases, there is a green arrow standing out from the others and pointing from the beginning of node 33 to the beginning of node 75 and the memory allocation of node 75 is delayed after the completion of node 54.

It would be tempting to believe that the completion of nodes 33 and 54 explain most of the performance (e.g., the sooner the completion of nodes 33 and 54, the sooner the completion of nodes 74 and 75, hence the smaller the makespan). Yet, when comparing many executions (both for `dmda` and `heteroprio`), although the makespan is very stable (less than 1% of variability) and the Gantt charts all appear to be very similar, the completion of nodes 33 and 54 is very variable from an execution to an other: sometimes node 33 ends way earlier than node 54, sometimes it is the other way around, and sometimes, they both complete quite late (at around 15 seconds). In some executions the scheduler focuses mostly on one node before working on the other, and sometimes, it alternates between both. This variability in the tree traversal comes from the fact that although `QR_mumps` assigns increasing priorities to tasks according to the tree node submission order, these priorities are ignored by `dmda` and `heteroprio`: the scheduling is mostly driven by the order in which tasks become ready.

6. Conclusion

This work presents two techniques to better analyze the performance of an irregular application by exploiting its structure. Our intent is to propose a method to identify problems arising in complex scenarios. Our contributions include (1) the modeling of irregular computation kernels to identify tasks with abnormal duration, and (2) to go beyond the Gantt chart by leveraging the global structure of the application (in the `QR_mumps` case, driven by the elimination tree). The anomalous task detection mechanism for irregular tasks proves its usefulness to identify performance issues in many levels of the runtime system while the representation of the elimination tree from the multifrontal method provides a deep understanding of how the application unfolds. We identify perturbations whose origin is as diverse as the task submission thread (Section 4.2), tracing overhead (Section 4.3), numerical instability (Section 4.4) and cache contention (Section 4.5). For all cases, we provide a fix or an alternate solution. We also demonstrate that our elimination tree view is paramount to better understand the very sophisticated interplay between the application structure and the runtime scheduler. We illustrate its effectiveness in three case studies where we compare two executions and explain why they lead to similar or different performance. In particular, we investigate the influence of memory limitation (Section 5.2), of task priorities (Section 5.3), and of task repartition between the CPUs and the GPU (Section 5.4). With an analysis solely based on a traditional Gantt chart, none of these situations would be intelligible. The new resulting performance visualization panels and techniques have been added to the StarVZ tool.

Although we illustrate our proposal with `QR_mumps`, the information we rely on (flop estimation of dense kernels and the elimination tree) can easily be obtained for any other implementation of the multifrontal methods such as the Cholesky and LU factorization of PaStiX. Other methods like Fast Multipole Method (FMM) [19, 13] are also composed by irregular tasks whose execution is organized around a tree structure and are thus likely to exhibit similar issues as the ones we found with `QR_mumps` and benefit from the techniques presented herein.

As future work, we expect to investigate our method in multinode plataforms. Also, there are some opportunities to develop panels to relate tree computation to NUMA nodes and PAPI hardware counters. Finally, since scheduling decisions are naturally stochastic, a more challenging path is how the internal structure of the application (in the `QR_mumps` and for multifrontal methods in general, the elimination tree) could be leveraged to compare not just two traces but two sets of executions.

Acknowledgements

This research was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, the National Council for Scientific and Technological Development (CNPq), and the projects: FAPERGS (Data Science – 19/711-6, MultiGPU 16/354-8, and GreenCloud – 16/488-9), the CNPq project 447311/2014-0, the CNPq scholarships 131347/2019-5 and 141971/2020-7, the CAPES/Brafitec EcoSud 182/15, and the CAPES/Cofecub 899/18. We finally thank Alfredo Buttari, Emmanuel Agullo, and Abdou Guermouche for the fruitful discussions we had along these years in the context of the `QR_mumps` application.

References

[1] [SW Rel.] G. Karypis, V. Kumar, and A. Gladky, *METIS* Feb. 2021. SWHID: ⟨`swh:1:rev:ce7fd39dec097f7fb5fa661260ee13b4808873e7`⟩.

[2] [SW Rel.] C. Augonnet et al., *StarPU* Oct. 2020. SWHID: ⟨`swh:1:rev:cc8d6e7fea87e825b90631bd9a164082cbb1ae5b;origin=https://gitlab.inria.fr/starpu/starpu.git;visit=swh:1:snp:7a6c8616d125a4aa5d5b639cce8623e7c6b7e8ba`⟩.

[3] [SW Rel.], *gcc* version 8.4.0, Mar. 2020.

[4] [SW Rel.] L. M. Schnorr, V. Garcia Pinto, M. C. Miletto, and L. Leandro Nesi, *StarVZ* version 0.4.0, 2020. SWHID: ⟨`swh:1:dir:10c1ac0d164a5771132468ac2a4aa4eca1169467`⟩.

[5] [SW Rel.], *OpenBLAS* version 0.3.9, Mar. 2020.

[6] L. Leandro Nesi, S. Thibault, L. Stanisic, and L. Mello Schnorr. "Visual Performance Analysis of Memory Behavior in a Task-Based Runtime on Hybrid Platforms". In: *IEEE/ACM CCGRID*. 2019.

[7] M. C. Miletto and L. Schnorr. "OpenMP and StarPU Abreast: the Impact of Runtime in Task-Based Block QR Factorization Performance". In: *XX Simpósio em Sist. Comp. de Alto Desempenho*. SBC. 2019.

[8] I. Oz, M. K. Bhatti, K. Popov, and M. Brorsson. "Regression-Based Prediction for Task-Based Program Performance". In: *Circ., Syst. and Comp.* 28.04 (2019).

[9] [SW Rel.] F. Pellegrini, C. Chevalier, S. Fourestier, J.-H. Her, C. Lachat, A. Jacques, and L. Scarano, *SCOTCH* version 6.0.8, Aug. 2019. SWHID: ⟨`swh:1:dir:d86bd752ba36f07295c468c869469c4304853120;origin=https://gitlab.inria.fr/scotch/scotch;visit=swh:1:snp:01909a42259b0ea9b619ce4abab3d2642773ff79;anchor=swh:1:rev:6ac391912e21820e71748153d0f4e5107e628352`⟩.

[10] I. Duff, J. Hogg, and F. Lopez. "Experiments with sparse Cholesky using a sequential task-flow implementation". In: *Numer. Alg., Contr. & Opt.* 8.2 (2018).

[11] V. Garcia Pinto, L. Mello Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean. "A visual performance analysis framework for task-based parallel applic. running on hybrid clusters". In: *CCPE* 30.18 (2018).

[12] E. Agullo, O. Aumage, B. Bramas, O. Coulaud, and S. Pitoiset. "Bridging the gap between OpenMP and task-based runtime systems for the fast multipole method". In: *IEEE Trans. on Par. and Distr. Syst.* 28.10 (2017).

[13] E. Agullo, B. Bramas, O. Coulaud, L. Stanisic, and S. Thibault. *Modeling Irregular Kernels of Task-based codes: Illustration with the Fast Multipole Method*. Research Report RR-9036. INRIA Bordeaux, Feb. 2017.

[14] G. Ceballos, T. Grass, A. Hugo, and D. Black-Schaffer. "Task-insight: Understanding task schedules effects on memory and performance". In: *The 8th Intl. Workshop on Prog. Models and Applications for Multicores and Manycores*. ACM. 2017.

[15] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Oxford University Press, 2017.

[16] J. V. Lima, I. Raïs, L. Lefèvre, and T. Gauthier. "Performance and energy analysis of openmp runtime systems with dense linear algebra algorithms". In: *Intl. Symp. on Comp. Arch. and High Perf. Comp. Workshops (SBAC-PADW)*. IEEE. 2017.

[17] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. "Implementing multifrontal sparse solvers for multicore arch. with sequential task flow runtime systems". In: *TOMS* 43.2 (2016).

[18] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. "A survey of direct methods for sparse linear systems". In: *Acta Numerica* 25 (2016).

[19] P. Blanchard, B. Bramas, O. Coulaud, E. Darve, L. Dupuy, A. Etcheverry, and G. Sylvand. "ScalFMM: A Generic Parallel Fast Multipole Library". In: *SIAM Conf. on Comp. Sci. and Eng.* Salt Lake City, United States, Mar. 2015.

[20] A. Huynh, D. Thain, M. Pericàs, and K. Taura. "DAGViz: a DAG visualization tool for analyzing task-parallel program traces". In: *The 2nd Workshop on Visual Perf. Analysis*. 2015.

[21] F. Lopez. "Task-based multifrontal QR solver for heterogeneous architectures". PhD thesis. Université de Toulouse, Université Toulouse III-Paul Sabatier, 2015.

[22] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau. "Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers". In: *Intl. Conf. on Par. and Distr. Syst.* IEEE, 2015, pp. 481–490.

[23] M. Wittmann, T. Zeiser, G. Hager, and G. Wellein. "Short note on costs of floating point operations on current x86-64 architectures: Denormals, overflow, underflow, and division by zero". In: *arXiv preprint arXiv:1506.03997* (2015).

[24] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. "Versatile, scalable, and accurate simulation of distributed applications and platforms". In: *Journal of Parallel and Distributed Computing* 74.10 (2014).

[25] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. "State of the Art of Performance Visualization". In: *EuroVis - STARs*. Ed. by R. Borgo et al. The Eurographics Association, 2014.

[26] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. "Intel math kernel library". In: *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014.

[27] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. "Multifrontal QR factorization for multicore architectures over runtime systems". In: *European Conf. on Par. Proc.* 2013.

[28] A. Buttari. "Fine-grained multithreading for the multifrontal QR factorization of sparse matrices". In: *SIAM Journal on Scientific Computing* 35.4 (2013).

[29] C. F. Van Loan and G. H. Golub. *Matrix computations 4th ed.* Johns Hopkins Univ. Press Baltimore, 2013.

[30] Z. Xianyi. *OpenBLAS: An optimized BLAS library*. 2013. URL: `http://www.openblas.net/`.

[31] K. Coulomb, A. Degomme, M. Faverge, and F. Trahay. "An open-source tool-chain for performance analysis". In: *Tools for High Perf. Comp. 2011*. Springer, 2012.

[32] R. Keller, S. Brinkmann, J. Gracia, and C. Niethammer. "Temanejo: Debugging of thread-based task-parallel programs in starss". In: *Tools for High Perf. Comp.* Springer, 2012.

[33] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures". In: *Concurrency and Computation: Practice and Experience* 23.2 (2011).

[34] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. "Ompss: a proposal for programming heterogeneous multi-core architectures". In: *Parallel processing letters* 21.02 (2011).

[35] R. Nath, S. Tomov, and J. Dongarra. "Accelerating GPU Kernels for Dense Linear Algebra". In: *The 2009 Intl. Meeting on High Performance Computing for Computational Science, VECPAR'10*. Berkeley, CA: Springer, June 2010.

[36] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. "A class of parallel tiled linear algebra algorithms for multicore architectures". In: *Parallel Computing* 35.1 (2009).

[37] 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*. IOS Press, 2009.

[38] V. Todorov and P. Filzmoser. "An Object-Oriented Framework for Robust Multivariate Analysis". In: *Journal of Statistical Software* 32.3 (2009), pp. 1–47.

[39] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*. Dec. 2008.

[40] C. Nvidia. "Cublas library". In: *NVIDIA Corporation, Santa Clara, California* 15.27 (2008).

[41] T. Gautier, X. Besseron, and L. Pigeon. "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors". In: *Intl. Workshop on Par. Symb. Comp.* ACM. 2007.

[42] L. Grigori and X. S. Li. "Towards an accurate performance modeling of parallel sparse factorization". In: *Applicable Algebra in Eng., Comm. and Comp.* 18.3 (2007).

[43] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. "Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm". In: *ACM TOMS* 30.3 (2004).

[44] F. Leisch. "FlexMix: A General Framework for Finite Mixture Models and Latent Class Regression in R". In: *Journal of Statistical Software, Articles* 11.8 (2004).

[45] P. Hénon, P. Ramet, and J. Roman. "PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems". In: *Par. Comp.* 28.2 (2002).

[46] P. R. Amestoy, I. S. Duff, and J.-Y. L'excellent. "Multifrontal parallel distributed symmetric and unsymmetric solvers". In: *Comp. methods in applied mech. and eng.* 184.2-4 (2000).

[47] L. Dagum and R. Menon. "OpenMP: An industry-standard API for shared-memory programming". In: *Computing in Science & Engineering* 1 (1998).

[48] F. Pellegrini and J. Roman. "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs". In: *Intl. Conf. on High-Perf. Comp. and Networking*. Springer. 1996.

[49] V. Pillet, J. Labarta, T. Cortes, and S. Girona. "Paraver: A tool to visualize and analyze parallel code". In: *Proceedings of WoTUG-18: transputer and occam developments*. Vol. 44. 1. Citeseer. 1995.

[50] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. "An extended set of FORTRAN basic linear algebra subprograms". In: *ACM TOMS* 14.1 (1988).

[51] J. W. Liu. "Computational models and task scheduling for parallel sparse Cholesky factorization". In: *Par. Comp.* 3.4 (1986).

[52] I. S. Duff and J. K. Reid. "The multifrontal solution of indefinite sparse symmetric linear". In: *ACM Transactions on Mathematical Software (TOMS)* 9.3 (1983).

[53] B. M. Irons. "A frontal solution program for finite element analysis". In: *International Journal for Numerical Methods in Engineering* 2.1 (1970).