



HAL
open science

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

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

► **To cite this version:**

Marcelo Cogo Miletto, Lucas Leandro Nesi, Lucas Mello Schnorr, Arnaud Legrand. Performance Analysis of Irregular Task-Based Applications on Hybrid Platforms: Structure Matters. 2021. hal-03298021v1

HAL Id: hal-03298021

<https://inria.hal.science/hal-03298021v1>

Preprint submitted on 23 Jul 2021 (v1), last revised 9 Jan 2023 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Performance Analysis of Irregular Task-Based Applications on Hybrid Platforms: 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 tailored for 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

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 [43], StarPU [31], OmpSs [32], Kaapi [38], and OpenCL [36]. 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 [15, 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 [23].

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). Figure 1 presents this kind of classical visualization using ViTE [29] (top) and StarVZ [11] (bottom), using the same trace. 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 [14], 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 [49, 41] 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.

*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)

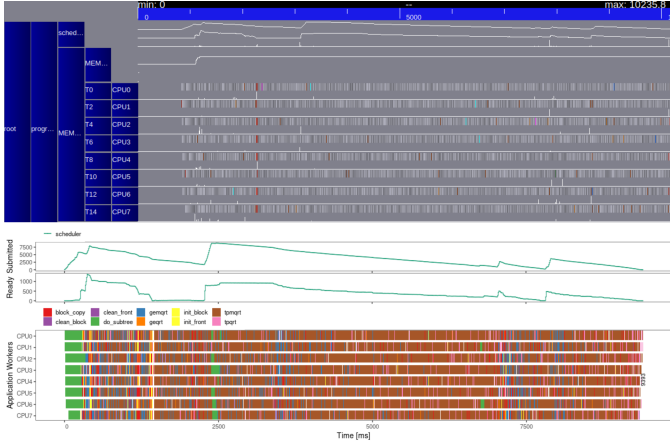


Figure 1: Task submission, ready tasks, and the Gantt chart for the same dataset using ViTE (top) and StarVZ (bottom).

In this article we propose several new performance visualization techniques to exploit the structure of task-based sparse matrix solvers. These techniques are implemented in the StarVZ tool [4] and applied to the multifrontal task-based parallel sparse solver QR_mumps [16]. Figure 2 illustrates the resulting visualization when using the same traces as for Figure 1. Our main contributions are as follows. (1) We propose a statistical model of the irregular tasks of QR_mumps to automatically detect tasks with anomalous duration given their expected floating-point operation count and the computational resource type (Section 4). We illustrate through four different 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 progression of the elimination tree structure along time, including derived information such as the memory consumption with the number of active tree nodes and the computational effort in terms of tree nodes and depth (Section 5). We illustrate through four different case studies how this visualization enables detailed understanding on how the factorization unfolds, to identify and address non-trivial scheduling problems that would be impossible to understand with a 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.

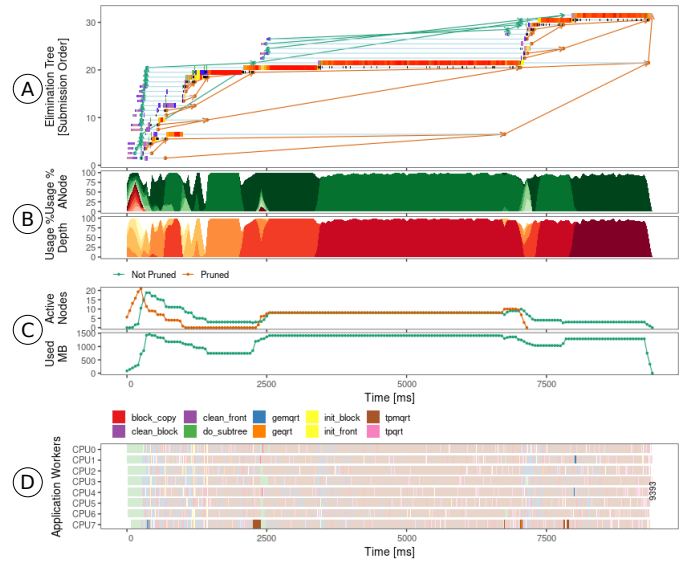


Figure 2: The proposed application-specific panels and improvements. (A) the elimination tree tasks location along time, (B) the resource utilization by computational tasks by tree node and depth, (C) memory and structure related panels, and (D) anomalous tasks highlighting considering their irregularity.

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 [25]. We focus on solvers that incorporate parallel versions of the multifrontal method for obtaining the direct solution of sparse equation systems [17]. Implementing these solvers requires 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 [47], their hand-tuned implementations like OpenBLAS [28], and MKL [24] for CPU, and CUBLAS [37] and MAGMA BLAS [33] 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.

125 To address the load balancing issue, a traditional approach¹⁸¹
 126 among solvers consists in relying on an in-house scheduler¹⁸²
 127 specifically designed for the sparse factorization context¹⁸³.
 128 This approach enables the developers to describe the applica¹⁸⁴
 129 tion as a Directed Acyclic Graph (DAG), where the DAG nodes¹⁸⁵
 130 represent the computational tasks and the edges, the data de¹⁸⁶
 131 pendencies among them. This approach simplifies program¹⁸⁷
 132 ming since it delegates the control flow management and load¹⁸⁸
 133 balancing to the scheduler. Task-based programming has been¹⁸⁹
 134 used in this context since the pre-multicore era in MPI-based¹⁹⁰
 135 implementations such as MUMPS [42] and is still used nowa¹⁹¹
 136 days. These in-house schedulers are lightweight because they¹⁹²
 137 often rely on a specific algorithm’s knowledge. However, such
 138 in-house and application-dependent schedulers commonly have
 139 limited features and fail to scale well on heterogeneous systems.
 140 Therefore, there is a move toward the use of general-
 141 purpose runtime systems such as StarPU, which has proven to
 142 be a well-suited option for the parallelization of sparse factorization
 143 methods by [25]. In the next sections, we give more
 144 details on the stages and structure of the multifrontal method
 145 and give practical details of the QR_mumps implementation on
 146 top of StarPU but we believe our proposal could be equally applied
 147 to any other sparse solver and runtime.

148 The Multifrontal Method

149 The multifrontal method was developed by [49] as an ex-
 150 tension of frontal method proposed by [50], focusing on the
 151 factorization of symmetric matrices using Cholesky. However,
 152 the method provides a structure that can be adapted to factorize
 153 sparse unsymmetric matrices using LU or QR factorization as
 154 well. This method breaks the whole matrix factorization prob-
 155 lem into smaller and denser subproblems, as partial factoriza-
 156 tion steps. These subproblems are known as being the frontal
 157 matrices or just fronts. 193

158 In classical approaches, each frontal matrix represents one¹⁹⁴
 159 elimination step related to a column j . Now, because of the ma¹⁹⁵
 160 trix sparsity, some elimination steps operate in disjoint subsets¹⁹⁶
 161 of matrix coefficients. Then, multiple fronts can be factorized¹⁹⁷
 162 in parallel, which gives the name to the method. However, if¹⁹⁸
 163 the elimination of one column changes the coefficients used in¹⁹⁹
 164 another step, there is a dependency between these eliminations.²⁰⁰
 165 These dependencies are captured by a structure that is the heart²⁰¹
 166 of the multifrontal method: the *elimination tree*. This tree struc²⁰²
 167 ture holds the fronts in its nodes and expresses the dependencies²⁰³
 168 between them as a parent and child relation in the tree. A par²⁰⁴
 169 ent node can only be factorized after all its child nodes were²⁰⁵
 170 already factorized, and its contribution blocks were assembled²⁰⁶
 171 into the parent node. The whole matrix factorization is done by²⁰⁷
 172 traversing the tree in the topological order, from bottom to top. ²⁰⁸

173 Considering a QR Householder factorization, Figure 3²⁰⁹
 174 presents an example of a sparse matrix structure and its elimi²¹⁰
 175 nation tree with some detail in its the frontal matrices. In the²¹¹
 176 figure, we can observe the inherent parallelism that arises from²¹²
 177 this structure regarding eliminating columns that reside in dif²¹³
 178 ferent branches of the tree simultaneously (e.g., 1, 2, and 5).²¹⁴
 179 This source of parallelism is commonly referred to as the *tree*²¹⁵
 180 *parallelism*. We can also notice how the fronts form dense sub²¹⁶

matrices 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.

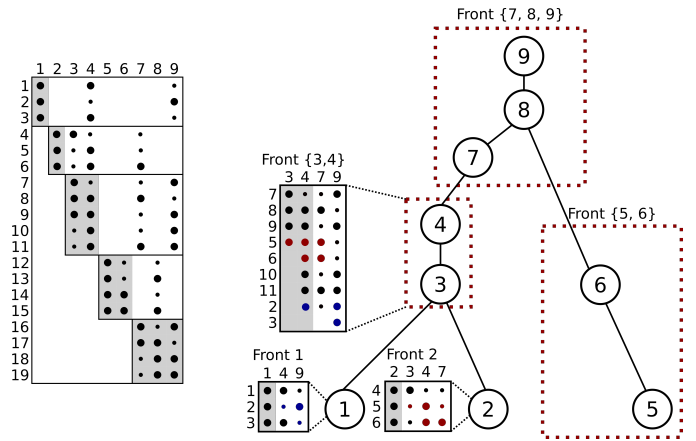


Figure 3: Example of a sparse matrix (left) and its elimination tree (right). Original matrix coefficients are marked as black dots, while fill-in coefficients are small dots. The gray-shaded area represents the columns that are being eliminated in that elimination step. Blue and red coefficients represent 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 3 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 [40], Scotch [45], and Metis [44] use

217 matrix reordering algorithms such as Approximate Minimum
 218 Degree, Nested Dissection, and Cuthill-McKee [27] to provide
 219 a matrix permutation that reduces the fill-in during the factor-
 220 ization. Applications also perform a symbolic factorization step
 221 that enables them to preallocate the necessary memory for the
 222 final structure by calculating beforehand the final structure of
 223 the matrix after the factorization. At the end of this phase, we
 224 have the elimination tree structure ready to be computed by the
 225 next phase. In sequence, the **factorization phase** is responsible
 226 for traversing the elimination tree from the leaves to the root,
 227 computing the partial factorization in each front, and combining
 228 the child node contribution blocks to the parent frontal matrix.
 229 These front factorizations can be done in parallel. For example,
 230 the method can process all the leaves of the tree at the same
 231 time. There is a restriction in starting the parent node factor-
 232 ization because all its child nodes need to be already computed
 233 to assemble their contribution blocks. Then, as the computa-
 234 tions move towards the tree root, the *tree parallelism* becomes
 235 more scarce, and fronts get bigger, and this is why we should
 236 use some other techniques like tiled factorization to explore the
 237 *node parallelism*. Finally, in the **solve phase** has the last tree
 238 traversal to apply forward and backward substitutions, and a
 239 triangular solve operation for each front to group their results.

240 *QR_mumps: A Fine-Grained Task-Based Multifrontal Method*

241 The `QR_mumps` application [16] is an example of a multi-
 242 frontal sparse direct solver that uses the elimination tree struc-
 243 ture to partition and parallelize the problem. Using a fine-
 244 grained task-based approach relying on the StarPU runtime sys-
 245 tem, it partitions the frontal matrices in tiles on which the com-
 246 putational tasks will work. This approach allows exploring a
 247 new level of parallelism referred to as *interlevel parallelism*
 248 by [16]. The interlevel parallelism refers to the limitation of
 249 starting a parent node only once all its child nodes contribu-
 250 tion blocks have been assembled into it. With this finer-grained
 251 partitioning, as soon as a part of the parent node is completely
 252 assembled, the factorization in that region can start, allowing to
 253 overlap computations between a child and a parent node. Al-
 254 though this optimization brings significant performance gain, it
 255 also makes the execution of the whole application even more
 256 challenging to understand.

257 The application relies on four LAPACK kernels for the fac-
 258 torization: `geqrt`, `gemqrt`, `tpqrt`, and `tpmqrt`. The parti-
 259 tioning of the frontal matrices in `QR_mumps` follows a 2D block
 260 partitioning, breaking a single front into many smaller blocks
 261 where these tasks operate. The block and task dimensions are
 262 controlled by the user-defined parameters `mb`, `nb`, and an in-
 263 ternal blocking size `ib`. The first two controls the number of
 264 rows and columns of the block, and the latter is a parame-
 265 ter used in LAPACK routines to decrease the number of ex-
 266 tra flops needed because of the 2D partitioning, as explained
 267 by [34]. Despite this beneficial effect from the LAPACK `ib`
 268 parameter, the routines were modified to control fill-in level in-
 269 side the blocks where there are zeroes in the bottom left part
 270 of the block, forming what is referred to as the staircase struc-
 271 ture. The effect of these parameters in the routines regarding the
 272 fill-in and task irregularity is presented by Figure 4. The figure

represents a frontal matrix (dense) with its rows sorted by the
 leftmost nonzero to clearly observe the staircase structure, lead-
 ing 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 val-
 ues so that we can have rectangular blocks. The only restriction
 is that the value of `nb` must be a multiple of `ib`, which con-
 trols the internal blocking effect, illustrated in the left part of
 Figure 4, where the dark gray squares represent the fill-in coef-
 ficients. 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 effi-
 ciency in the BLAS-3 operations.

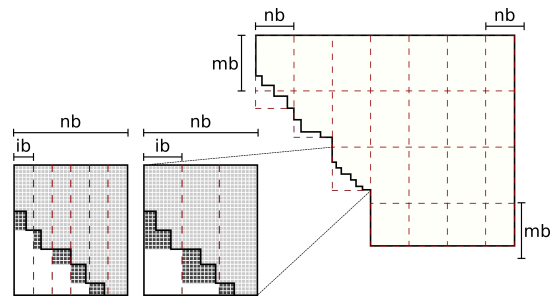


Figure 4: Example of a front partitioning using `mb` and `nb`, sources of task irregularity, and the `ib` size effect in fill-in. Matrix coefficients are represented as light gray squares and fill-in coefficients as dark gray.

Figure 4 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 [26] 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 tasks `init/clean_front` and `init/clean_block` allow to allocate and deallocate the frontal matrix structures and their blocks. In addition, the `block_copy` task allows assembling specific contribution blocks in a node parent, even before the

complete factorization, which enables to free the assembled front block from memory earlier. The QR_mumps application has a parameter that allows the user to define a memory usage upper bound based on how much memory the application would use in a sequential traversal of the elimination tree. Previous experiments proved that the application could maintain performance while saving memory usage [19].

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 [29] or Paraver [46], 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 [18] and Temanejo [30]. 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 [13] 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 [48, 20, 39, 35]. 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 [22]. 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) whenever 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 [31], we have considered the `lws` and `prio` for CPU executions, and `heteroprio`, `dmda`, and `dmdasd` schedulers for cases including GPUs. 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 conse-

417 quence, their elimination tree and floating-point operation cost
 418 for the matrix factorization are different.

419 Table 2 lists those sparse matrices from real problems (Ma-
 420 trix Market and SuiteSparse Matrix Collection repositories) that
 421 we use in this work. We have many possible combinations
 422 for each workload and application configuration (memory con-
 423 straint, scheduler, and ordering). We avoid exploring all combi-
 424 nations since some schedulers only make sense when we have
 425 GPUs, like heteroprrio, 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

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

428 Performance models of compute kernels can be used to im-
 429 prove scheduling and load balancing, predict performance, and
 430 post-mortem performance analysis. For example, StarPU uses
 431 such models to guide task scheduling policies at runtime. Mod-
 432 els can reveal many possible performance problems and hint the
 433 analyst toward conducting a more in-depth analysis of specific
 434 application regions where those anomalies occurred. Enriching
 435 space-time visualizations with such information has been suc-
 436 cessfully applied in the context of dense linear algebra by [11].
 437 However, the irregularity of the sparse factorization tasks dis-
 438 cussed in the previous section calls for more elaborate tech-
 439 niques. We first present a regression model of the irregular tasks
 440 of QR_mumps and how it can be used to enrich the Gantt-chart.
 441 Then we present four scenarios that showcase how this informa-
 442 tion about anomalies allowed us to discover and address simple
 443 to non trivial performance issues.

4.1. A Regression Model of the Irregular Tasks of QR_mumps

445 The duration of a task depends obviously on the kernel type
 446 (geqrt, gemqrt, tpqrt, and tpmqrt) and whether it is exe-
 447 cuted on a CPU or on a GPU. But in the sparse case it also
 448 depends on the geometry of the matrices, on the nb, mb, ib
 449 granularity parameters, and on the staircase structure. Based
 450 on all this information, it is possible to estimate the theoretic-
 451 al computational load (in GFlops) incurred by each task. This
 452 theoretical cost is estimated by the QR_mumps application dur-
 453 ing the task submissions to the StarPU runtime system and is
 454 propagated in the trace, which allows us to relate task duration
 455 (in milliseconds) to the number of theoretical Gigaflops per-
 456 formed by a task, as shown in Figure 5. As one would expect,
 457 the behavior is broadly linear and the efficiency (the slope of
 458 the regression) is very different from a compute kernel to an-
 459 other and from a BLAS implementation to another. Yet, all the

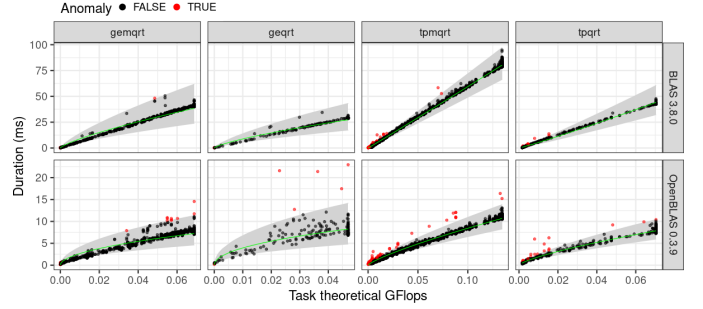


Figure 5: Task duration in milliseconds against the task theoretical GFlop count. The ribbon represents the model formula $\log(\text{Duration}) \approx \log(\text{GFlops})$ fit over the data. Red tasks are the ones classified as anomalies by the model.

460 classical linear regression assumptions do not hold, in particu-
 461 lar the variability is not constant (durations for larger theoretical
 462 flop counts are more variable). Consequently, we need to con-
 463 sider the residuals heteroscedasticity, especially when using an
 464 optimized version of BLAS, as shown in the bottom row of Fig-
 465 ure 5. Since the duration is always positive and the variability
 466 appears to grow linearly with the flop count, we handle this het-
 467 eroscedasticity using a simple log-log transformation before the
 468 linear regression but other techniques could be used as well. Al-
 469 though our model allows modeling both standard BLAS imple-
 470 mentations like NetLib BLAS and optimized BLAS implemen-
 471 tations like OpenBlas, the latter appear to have a much larger
 472 variance than the former ones and may seem more unstable or
 473 difficult to model. Nevertheless, such optimized libraries have
 474 much better performance 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. The goal is to check whether the observations lie above this confidence line and in such a case, the task is classified as an anomaly and represented in red in Figure 5. Note that since this test is applied to all the tasks, the confidence level is adapted with a Bonferoni correction. 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 depicted by Figure 2.(D) (see CPU7 for instance). 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 the sign of deeper performance issues.

4.2. Case 1: Influence of the Submission Thread

During our investigations it was common for outlier tasks to appear on a given core and to be rather grouped in time, as if there was short temporal perturbations. Figure 6 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 for this behavior is that StarPU has a main thread responsible

501 for handling task submission, which can occur at any moments⁵³⁴
 502 of the application execution. In most cases, the thread unrolls⁵³⁵
 503 the graph of tasks at the beginning of the execution. Never⁵³⁶
 504 theless, for scenarios with memory limitations, as the one in⁵³⁷
 505 vestigated here, task submission can be postponed according to⁵³⁸
 506 memory availability. In Figure 6, the submission thread, pinned⁵³⁹
 507 to CPU9, causes computational tasks to have a slightly longer⁵⁴⁰
 508 duration because it competes for the same resources. When⁵⁴¹
 509 binding the submitter thread to a dedicated core, the anomalous⁵⁴²
 510 tasks disappear.

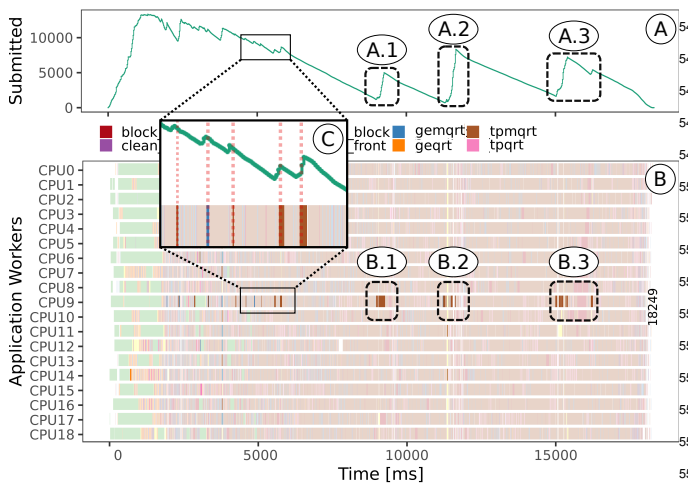


Figure 6: Panel A shows the number of tasks submitted over time, while panel B presents the application workers and the tasks they executed. We can see in the A-B 1, 2, and 3 pairs that the anomalies are associated with the task⁵⁶¹ submissions. In this figure, the submission thread is fixed in CPU9, where⁵⁶² these anomalies occur. In the C inset plot, we have a zoom that shows that even⁵⁶³ small numbers of task submissions can also cause anomalies.

511 Although this scenario clarifies the cause of task anomalies,
 512 it does not characterize a runtime-level performance problem.
 513 Indeed, such overhead is expected albeit rarely visible and our
 514 outlier detection mechanism gracefully revealed it. Task sub-
 515 mission is unavoidable but as our experiments indicate, the ad-
 516 ditional cost of these submissions is negligible and it is gen-
 517 erally better not to dedicate a core for submission. We can
 518 thus simply treat this core separately, having its outliers aligned
 519 with task submissions disregarded, focusing on other poten-
 520 tially anomalous tasks as we see next.

521 4.3. Case 2: Tracing-related Perturbations

522 Another common behavior we noticed during our experi-⁵⁷⁸
 523 ments appears as a global idle time spanning all workers. Such⁵⁷⁹
 524 absence of tasks may have several explanations: natural lack⁵⁸⁰
 525 of parallelism (few ready tasks as can be seen, for example,⁵⁸¹
 526 on Figure 6 at B.2/A.2 or Figure 13 at the end of B.4), bad⁵⁸²
 527 scheduling decisions (on Figure 15 at B.2), data transfers limi-⁵⁸³
 528 tation, and so on. Commonly, it remains complex to identify the⁵⁸⁴
 529 reason for each noticeable behavior. Here, we describe an other⁵⁸⁵
 530 interesting, albeit more trivial, cause that appeared in many⁵⁸⁶
 531 different combinations of workload, machine, and computa-⁵⁸⁷
 532 tional resources and which is illustrated through six instances⁵⁸⁸
 533 on Figure 7. It appears to be a time-dependent phenomenon⁵⁸⁹

for some runs, but it looks fairly random for others. These⁵⁴³
 case share a common characteristic that, when idling, workers⁵⁴⁴
 are in a so-called “overhead” runtime state (among other states⁵⁴⁵
 such as scheduling, fetch, sleeping). Sometimes, only one task⁵⁴⁶
 continues its execution, while other tasks remain dormant, as⁵⁴⁷
 shown by the two (A) graphic cases. In other scenarios, other⁵⁴⁸
 non-anomalous tasks that have already started are capable of⁵⁴⁹
 continuing their execution, as shown by the left-case of the⁵⁵⁰
 (B) graphic. Sometimes, for example in the left execution of⁵⁵¹
 (C), this idle period was responsible for up to 14% of the total⁵⁵²
 worker idleness. Interestingly, our performance model systemat-⁵⁵³
 ically identifies an anomalous task which coincides perfectly⁵⁵⁴
 with this idle period. There is no reason why a task shortage or⁵⁵⁵
 a bad scheduling decision would suddenly cause a task slow-⁵⁵⁶
 down, which allows to rule out many possible explanations.
 Furthermore, as illustrated in (A) and (C) the problem is al-
 most reproducible: although the outlier tasks are different from
 a run to an other, the perturbation generally occurs roughly at
 the same time regardless of the scheduler.

We found out that this anomalous event is related to the
 FxT trace dump during the application execution. This occurs
 when the trace buffer, which has a limited size controlled by
 the STARPU_TRACE_BUFFER_SIZE variable, gets full. Increas-
 ing the buffer size to a huge value actually removed this phe-
 nomenon and allowed us to concentrate on more intricate per-
 formance problems.

521 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 8 illustrates an example when fac-
 torizing 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. Further-
 more, the task’s duration difference magnitude was enormous,
 from ≈ 9 ms up to 150ms for gemqrt, and from 7ms to 65ms for
 geqrt. 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 8, 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 ex-
 ecuted 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 anoma-
 lous tasks is different among many executions as what happens
 for the presented geqrt and gemqrt tasks.

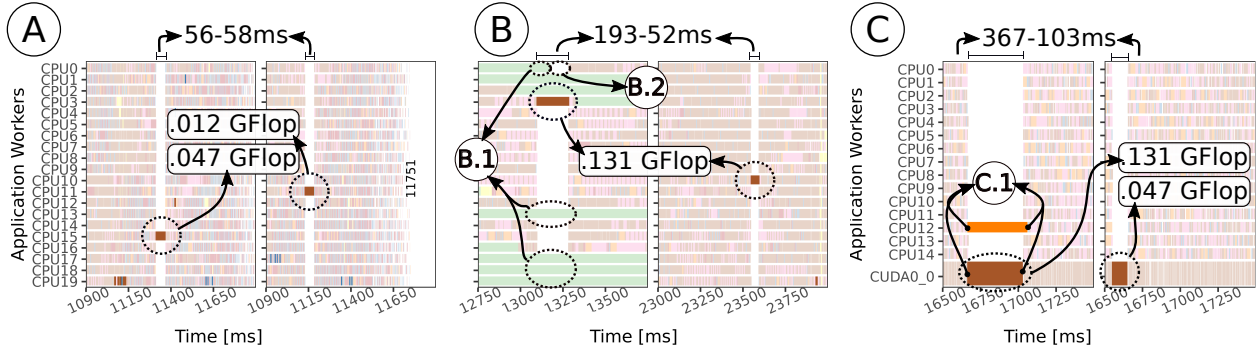


Figure 7: Six examples demonstrating the presence of task anomalies in the time span of ≈ 1 second. Case (A) has two different executions for the Hype machine (with the `1ws` scheduler on the left and with the `prio` scheduler on the right). Case (B) has only one execution in the Hype machine, and case (C) has two different executions for the Draco machine with GPU (with the `dmdasd` scheduler on the left and `prio` scheduler on the right).

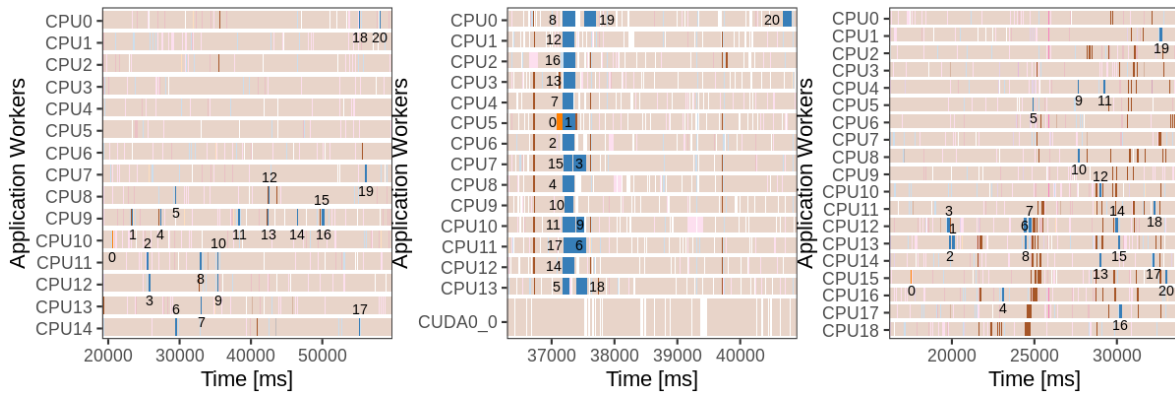


Figure 8: The same 20 blue GEMQRT tasks classified as anomalies are dependent from the same orange GEQRT task, identified with 0 in the plots.

590 The increased cache misses for some tasks identified as anomalies do explain a longer duration. Despite these signs that the cache misses may explain the variability in task duration, we hypothesize that another factor is causing this anomaly. As they consistently occur over the same tasks that work in the same matrix block, we investigate whether this difference 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.

600 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 [21]. 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 our model did not fit data as appropriately as for Figure 5 and where none of the previous problems could explain outlier tasks. Such a case study is depicted in Figure 9 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 9) 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 left of the top row of Figure 10 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 num-

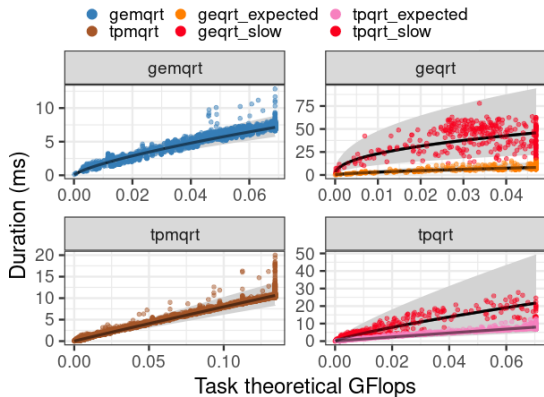


Figure 9: Using finite mixture models to fit multiple models over the data.

641 ber of cache misses, affecting our model’s prediction capabil-685
 642 ity. To check if the number of cache misses could explain the 686
 643 increased duration, we linked StarPU with the PAPI library to 687
 644 capture the total number of L1, L2, and L3 cache misses for 688
 645 each application task. We first verified if there was some cor-689
 646 relation between the total miss number for caches L1, L2, and 690
 647 L3 and task duration for other experiments. In general, GFlops, 691
 648 L1, and L2 misses have a strong positive correlation with task 692
 649 duration. The L3 cache misses explain less of the variability 693
 650 for well-behaved tasks. However, in the cases where a mixture 694
 651 model seems more appropriate, it is the opposite: GFlop, L1, 695
 652 and L2 misses fail to be a good explanatory variable while the
 653 L3 total cache misses is a better explanatory variable.

654 Table 3 presents evidence of the enormous difference be-
 655 tween the sequential and the parallel total time per task for the 696
 656 `flower_8_4` input matrix. In the sequential version, tasks do 697
 657 not suffer interference from other concurrent tasks, allowing us 698
 658 to capture their expected behavior. We observe that the parallel 699
 659 `geqrt` tasks take $3.37\times$ more time than in the sequential case. 700
 660 The worst-case occurs with the `do_subtree` tasks, present- 701
 661 ing an increase of $12.8\times$. The reason for slower `do_subtree` 702
 662 tasks is the same as for the `geqrt` and `tpqrt` tasks since the 703
 663 `do_subtrees` are composed of these very same kernels. These 704
 664 results confirm the memory contention problem, also known as 705
 665 locality efficiency [16].

Table 3: Task time of `flower_8_4` factorization in the Draco Machine.

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

666 A careful observation of the Figure 10 (left of top row) in- 718
 667 dicates that slow tasks concentrate in regions where there are 719
 668 many `do_subtree` and other simultaneous `geqrt` tasks, which 720

669 suggests they affect cache reuse in two ways: (1) `do_subtree`
 670 use a significant amount of memory without much reuse as the
 671 other 2D tasks do, and (2), `geqrt` tasks are executed spatially
 672 far from each other. They are either the starting factorization
 673 task of a tree node or its trailing submatrix which does not
 674 share matrix blocks with other `geqrt` tasks. Such characteris-
 675 tic can be the case for the `tpqrt` tasks too, which traverses the
 676 matrix row by row.

677 To alleviate this locality efficiency problem, we execute the
 678 application by limiting the execution of `do_subtree` tasks to
 679 only three CPU cores. The left of the bottom row of Figure 10
 680 depicts the resulting behavior once again for the `flower_8_4`
 681 matrix. To illustrate how this strategy works for other inputs,
 682 we show in the right of Figure 10 the same situation for the
 683 `karted` input matrix. The first three CPUs run all `do_subtree`
 684 tasks while all cores are responsible to execute remaining task
 685 types. By restricting the execution of these memory-bound
 686 tasks, we limit the available parallelism, which creates idle time
 687 in the beginning but we also reduce the makespan from 18.5s
 688 to 17s ($\approx 8\%$ reduction) for `flower_8_4` and from 2.3s to 1.8s
 689 ($\approx 22\%$) for `karted`. Specifically for the `flower_8_4` input,
 690 Table 3 provides, in the throttling column, the total time to
 691 compute all tasks of a given type. In this scenario, throttling
 692 improves the efficiency of `do_subtree` tasks making it closer
 693 to the sequential total time (without any interference), which
 694 improves performance overall.

5. Visualizing how the Multifrontal Factorization Unfolds

As explained in Section 2.1, sparse solvers based on the mul-
 tifrontal method rely on an elimination tree structure. In this
 section, we first provide a set of visualization panels related
 to this multifrontal structure, as illustrated in Figure 11, to de-
 pict 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 Sec-
 tion 5.1 and then we present four scenarios that showcase how
 these panels can be used in practice for the performance analy-
 sis of `QR_mumps`. These panels enable us to carry out interesting
 comparisons that are frequently questioned during the evalua-
 tion of a sparse task-based solvers, such as the effect of memory
 limitation, the adoption of different runtime schedulers and fill-
 reducing ordering operations.

5.1. Visualization Panels inspired by the Elimination Tree

Main Panel: Elimination Tree Visualization

The Elimination Tree panel provides us the perception of the
 tree structure, as defined by the experiment’s ordering algo-
 rithm and the fronts’ submission order. Figure 11.(A) depicts a
 representative example of this panel, showing for how long the
 elimination tree nodes (listed in the Y-axis according to their

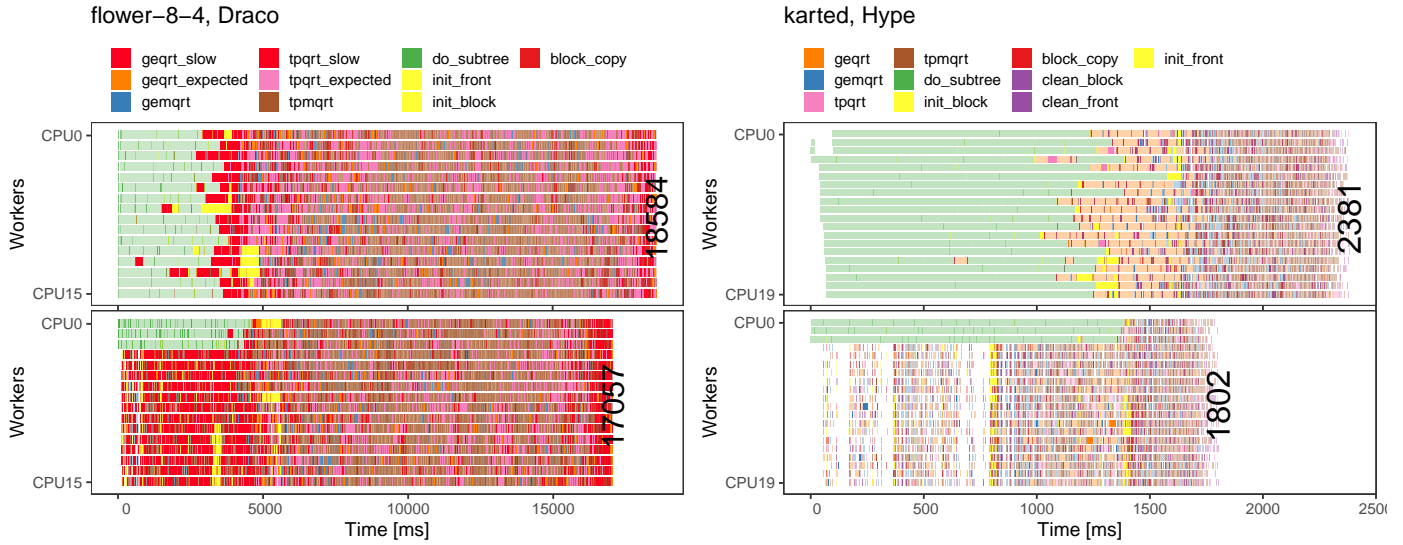


Figure 10: Anomalous `do_subtree`, `geqrt` and `tpqrt` tasks (marked as slow) when carrying the factorization of the `flower_8_4` (left) and `karterd` (right) matrices using all available workers (top row, parallel); Factorization of the `flower_8_4` and `karterd` matrices when restricting the `do_subtree` tasks to run only in three CPU resources (bottom, 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.

721 submission order) exist along time (the X-axis). This represen-754
722 tation has two main elements: the nodes of the tree and their-755
723 parent-child connection. Each node of the tree occupies a hor-756
724 izontal line in the panel. The line starts with the first memory-757
725 allocation task (represented by a green rectangle) and ends with-758
726 the task that releases the node memory. The lifetime purely-759
727 indicates memory footprint, not meaning that there were com-760
728 putations over this node this whole time. To represent compu-761
729 tations over the structure, and considering that many resources-762
730 compute tasks of a given node of the elimination tree, we em-763
731 ploy the color gradient to represent the computational load in-764
732 tensity (based only on factorization tasks) as a percentage of the-765
733 total number of computing resources. We can alternatively rep-766
734 resent other performance metrics, like the GFlops throughput.-767
735 Green and orange arrows indicate the parent-child connection.-768
736 The green arrows depart from the beginning of a node's line-769
737 and points to its parent. The same happens for orange arrows
738 but considering the end of a node's line. For simplification, we
739 group the sequential nodes pruned by the application by their-770
740 common parent, reusing the same Y position, and spatially ag-771
741 gregating their computations because they generally are numer-772
742 ous. Besides the Y position, pruned nodes share the same gra-773
743 dient color legend used in the other tree nodes. Thus, to differ-774
744 entiate them, we use half of the height of the non pruned nodes-775
745 for their representation. 776
777

746 Besides the spatial aggregation of computational load per-778
747 node of the elimination tree, we also aggregate the behavior-779
748 within a node in user-configurable time intervals (represented-780
749 by the color gradient). These aggregations help to handle cases-781
750 where tasks are too numerous, and application makespan is very-782
751 long. In Figure 11.(A), the time interval for the aggregation is-783
752 100ms. Combined, both spatial and temporal aggregations pro-784
753 vide a clearer behavior view maintaining temporal details. 785

Furthermore, we can represent other aspects of the applica-
tion in this tree plot, like the communication tasks between par-
ent and child nodes and the anomalous tasks' location. The
black rectangles (inset within each tree node line) in Fig-
ure 11.(A) represent the tree's communication, which comes
from the assembly of the contribution block of a child in its par-
ent front (such situations are not very frequent in this example
but can be more clearly seen in Figure 14 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 rep-
resentation to avoid a complete overlap in the plot. Lastly, the
elimination tree plot can also depict the anomalous task location
in space and time related to the tree as dots inside the node's
computation marker, following the Gantt chart tasks' colors 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, includ-
ing 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 paral-
lelism 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 applica-
tion 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 pro-
vide enough parallelism, serving as a general signature of how
scheduling decisions evolve during the execution.

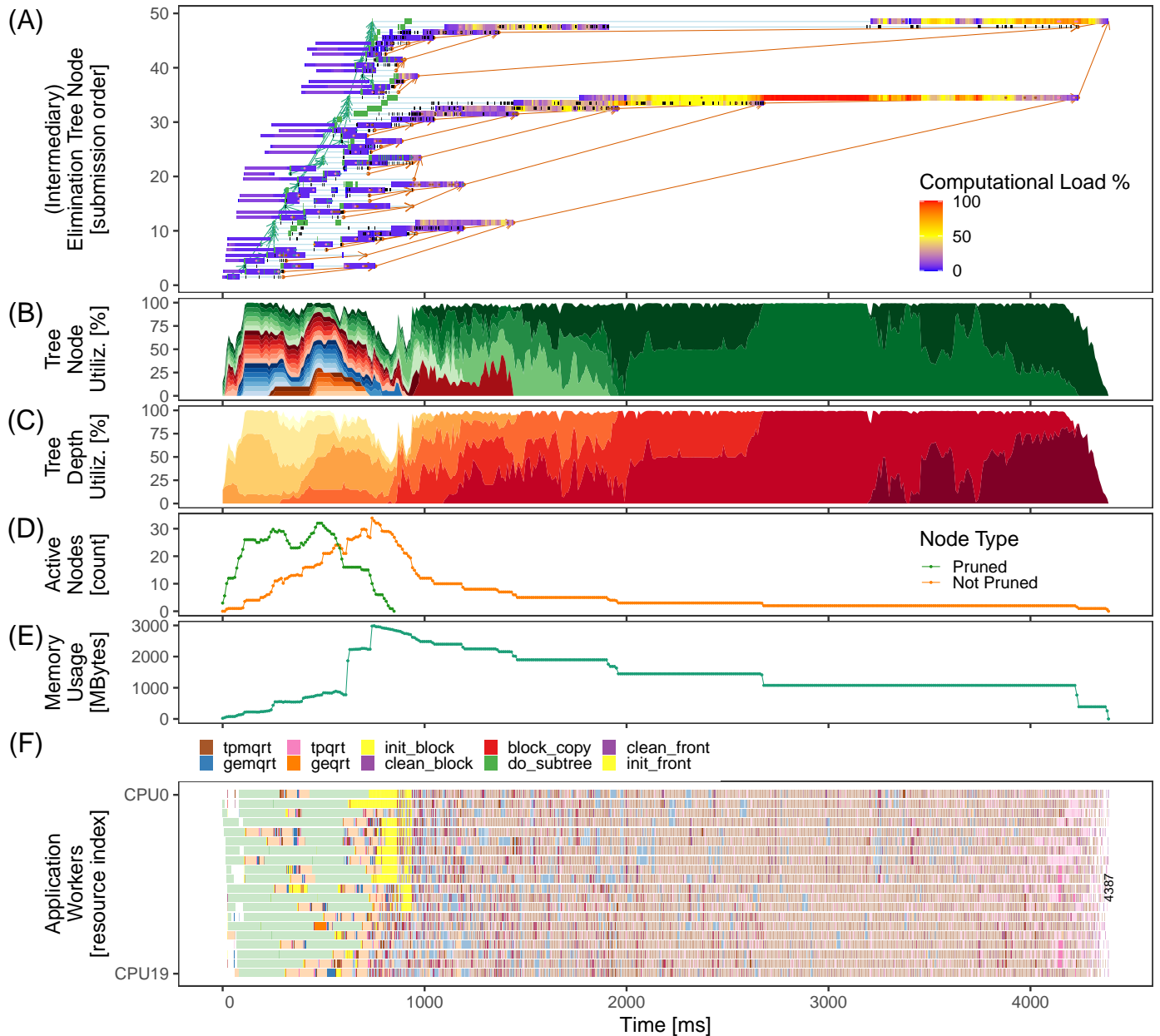


Figure 11: Overview of visualization panels tailored for the elimination tree, using the e18 matrix on hype as an example. The elimination tree panel (A) shows the tree structure and the computational signature along time. Panels (B) and (C) depict the resource utilization by the computational tasks highlighting the usage per tree node and tree depth. 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.

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 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

810 regarding other application properties like task priorities, mem-845
 811 ory availability. We can see, for instance, how computing power846
 812 tackles the parallelism of the tree structure. 847

813 *Auxiliary Panels: Active Nodes and Memory Usage* 849

814 Tree traversal may significantly impact memory consump-850
 815 tion, and memory consumption restrictions can have a signifi-851
 816 cant impact on tree traversal. The available parallelism and tree852
 817 traversal are very dependent on these memory-related aspects,853
 818 making the visualization of this information relevant to appli-854
 819 cation analysis. 855

820 In the multifrontal method, the child nodes need to merge856
 821 their contribution blocks to their parent node. This dependency857
 822 implies that all nodes involved in this operation must be present858
 823 on memory at that moment. Because memory is a finite resour-859
 824 ce, such applications can easily consume a large portion of860
 825 the available memory. Panels (D) and (E) of Figure 11 provide861
 826 a summary to understand better how the number of in-memory862
 827 active nodes and current memory usage evolves through execu-863
 828 tion time. Since QR_mumps can work with memory usage864
 829 constraints to keep memory usage under control, these panels865
 830 help understand how the application handles memory-related866
 831 issues in scenarios with a memory constraint. The two lines in867
 832 the in-memory active nodes panel indicate whether these tree868
 833 nodes are sequential (pruned by the application) or parallel tree869
 834 nodes. Depending on the traversal of the tree, it is normal that870
 835 sequential nodes only exist at the beginning of the execution871
 836 because they are the leaves of the tree. That is the case shown872
 837 in Figure 11.(D).

838 *5.2. Case 1: Evaluating fill-reducing Ordering Operations* 875

839 The most obvious use for our tree visualization is the analys-876
 840 is of the fill-reducing ordering operations performed in the877
 841 matrices in the analysis phase. In this work, we have used the878
 842 Metis and Scotch fill-reducing ordering packages. Figure 12879
 843 shows the tree structure plot for two executions with the same880
 844 parameter configurations, except for the ordering. 881

The factorization time for these two executions is different845
 because the total number of floating-point operations to perform846
 the factorization is different. Beyond this difference, we can see847
 that their structure is quite different too. If we observe the par-848
 titioning, we instantly notice that the Metis ordering produced849
 a poorly balanced tree, with few small nodes and a huge node850
 that dominates the application execution pointed by (A). On the851
 other hand, Scotch produced a more balanced tree with more852
 nodes, dividing the computational load better, as highlighted by853
 the comparison in (B), where the dashed lines cut both elimina-854
 tion trees at level 2. They have in common that they produced855
 very few pruned nodes for the do_subtree tasks and a small856
 node at the root. Such visualization might help application de-857
 velopers and analysts relate a specific ordering algorithm and858
 its generated tree structure to application performance. 859

860 *5.3. Case 2: The Influence of the Memory Limitation* 875

861 For all experiments so far, we have noticed that, except in862
 cases where the memory peak threshold is particularly limiting,863
 the QR_mumps application can keep the performance very simi-864
 lar to cases without memory usage restrictions and even provide865
 better results sometimes. This effect has been discussed by [19],866
 but now, we can observe the interplay between the memory limi-867
 tations and the elimination tree exploration by the scheduler.868
 Figures 11 and 13 show two different executions with the factor-869
 ization of the e18 matrix using the Metis reordering in the Hype870
 Machine. The only difference between them is that the former871
 depicts a case without memory limitation (the peak is 3GBytes),872
 while the latter depicts the execution when limiting the memory873
 to the sequential peak (≈ 1.4 GBytes). The makespan compari-874
 son between the unlimited (Figure 11) and limited (Figure 13)875
 executions indicates that the latter is insignificantly degraded876
 but their internal structures are very different. Without limit,877
 we observe in Figure 11 that the entire tree is allocated early878
 in the execution. When limiting memory usage, memory allo-879
 cations are delayed until the last moments of the factorization,880
 as illustrated in Figure 13. This exploration impacts the avail-881
 able tree-parallelism. In Figure 13.(B), we observe that around882
 1.5 seconds of execution, half of the tree intermediary nodes883
 were not touched yet by the execution with memory constraint,884
 while the unlimited case has started and even finished comput-885
 ing many other nodes. 886

887 We can see in Figure 13 (A.1) that in the beginning of the888
 execution, tree-level parallelism is available, as shown by the889
 many colors that appear in the resource usage per node plot.890
 The bottom part of the tree is composed of many small nodes,891
 indicating a tree that is sufficiently wide to provide enough par-892
 allel work to application workers. Furthermore, in (A.2), we893
 observe that the vast majority of the nodes active in memory894
 are pruned nodes, which is also presented by the number of895
 do_subtree tasks in (A.3). The (B) and (C) areas point out896
 to multiple delays in node allocation imposed by the mem-897
 ory constraint parameter. For example, we can see how the898
 freeing of the node pointed in (B.1) allows the allocation of899
 many other nodes. The amount of memory initialization tasks900
 (init_front and init_block) reduces the compute resource901
 utilization as shown in (B.2). The memory usage plot in (B.3)

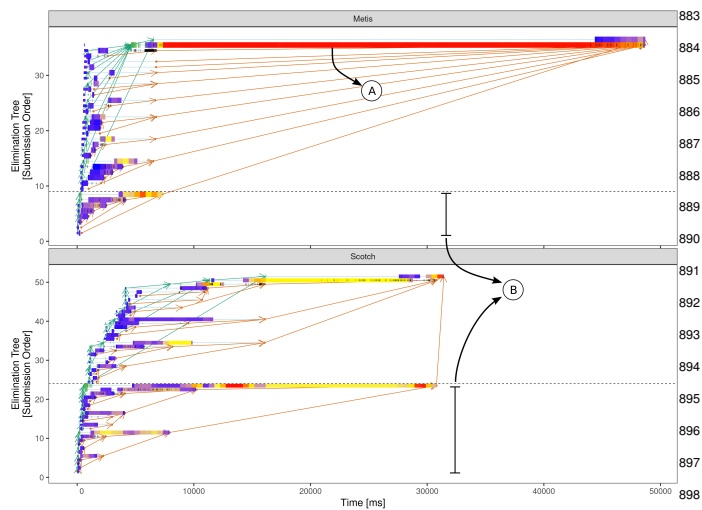


Figure 12: Metis against Scotch ordering for ch8-8-b3 matrix in Hype Machine.900

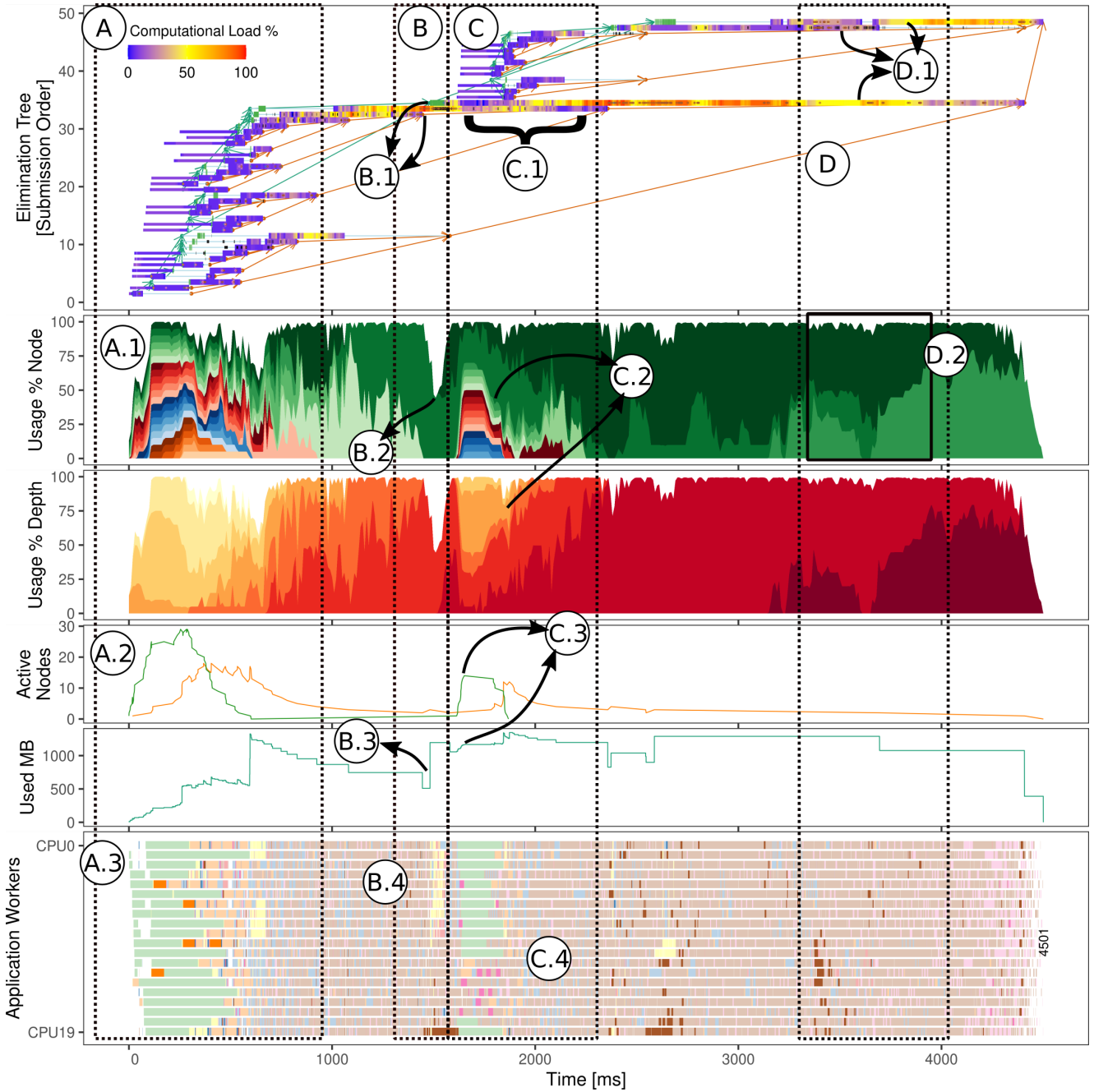


Figure 13: Tree-related plots and the Gantt chart for an execution of matrix e18 in Hype machine.

901 depicts exactly when the memory becomes available. Because
 902 the newly allocated node would use much memory, it had to
 903 wait for some other node to free the needed memory. All these
 904 memory initialization tasks can also be seen in the Gantt chart
 905 in (B.4, yellow color) but would be hard to interpret without
 906 the tree view. A similar case appear in the (C) area, where
 907 the set of nodes (C.1) get delayed because the execution has al-
 908 ready reached the memory limit (C.3). We confirm that most of
 909 the nodes are the leaves of the tree (C.2), as also shown by the
 910 do_subtree tasks in green (C.4). In the last scenario (D), in
 911 (D.1) and (D.2), we can observe how the application can over-
 912 lap computations between the last three dependent nodes of the

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 ex-

cution and to limit their influence on the other tasks.

5.4. Case 3: Identifying Task Priority Issues

The elimination tree panel, as described in Figure 11 (A), can be used to compare different schedulers' behavior while traversing the tree for diverse hardware and software configurations. In Figure 14, we compare the performance of the `lws` against the `prio` scheduler for the `degme` matrix 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 pointed in (A), it seems that the `prio` scheduler focuses computations on one node at a time. This is due to the fact that `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 noticeable behavior of the `prio` scheduler is pointed in (B), where there seems to be a clear communication pattern. Indeed the `block_copy` and `init_*` tasks are systematically scheduled after the final computations of a tree node as they have a lower priority, which regularly delays 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.

5.5. Case 4: Resource Usage of two Runtime Schedulers

In this section, we compare the `dmda` and `heteroprio` schedulers using as input the TF17 matrix reordered with Scotch, in a scenario with memory limitation in the Hypo Machine. Figure 15 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 $\approx 8.5\%$ 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 a closer inspection to all the other panels will allow to reveal that this idling is part of the reason only.

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. Similarly to what was proposed by [11] in the dense case, we can compute the optimal allocation of tasks to CPUs and GPUs when ignoring all dependencies using our performance model for all kernel types. 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. We can also see from the CPU/GPU division provided on the right for the main task types (`gemqrt` and `tpmqrt`) that `dmda` makes a better usage of resources in general and ultimately makes decisions that are closer to that of the ABE. In the ideal allocation, only the costlier `tpmqrt` tasks (with ≈ 3 GFlops) should be allocated on GPU resources. The `dmda` scheduler only takes myopic decisions but comes closer to this decision than `heteroprio`. A similar interpretation works for the `gemqrt` tasks as well, except that no tasks of this type should ideally be run on the GPUs. Overall, the `heteroprio` scheduler 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 explanation is the reason why the `dmda` scheduler manages to move faster along the tree than `heteroprio` although it is not directly visible from the Gantt charts.

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 (although there are way enough tasks to keep all CPUs busy in the beginning as can be seen from the Ready Tasks panel). 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 mainly comes from the memory limitation imposed to this execution and perfectly correlates with the beginning of a new elimination tree node as shown by the arrows towards the elimination tree panel. The `heteroprio` scheduler manages very poorly the lack of ready tasks when deciding to not use the CPUs because it globally considers that CPU are 25 times slower than GPUs. That is not the case for the `dmda` schedulers that manages to exploit CPUs in this difficult situation.

Finally, we can also observe differences at the end, when both runs are working in the last elimination tree node. 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 that once again it makes a

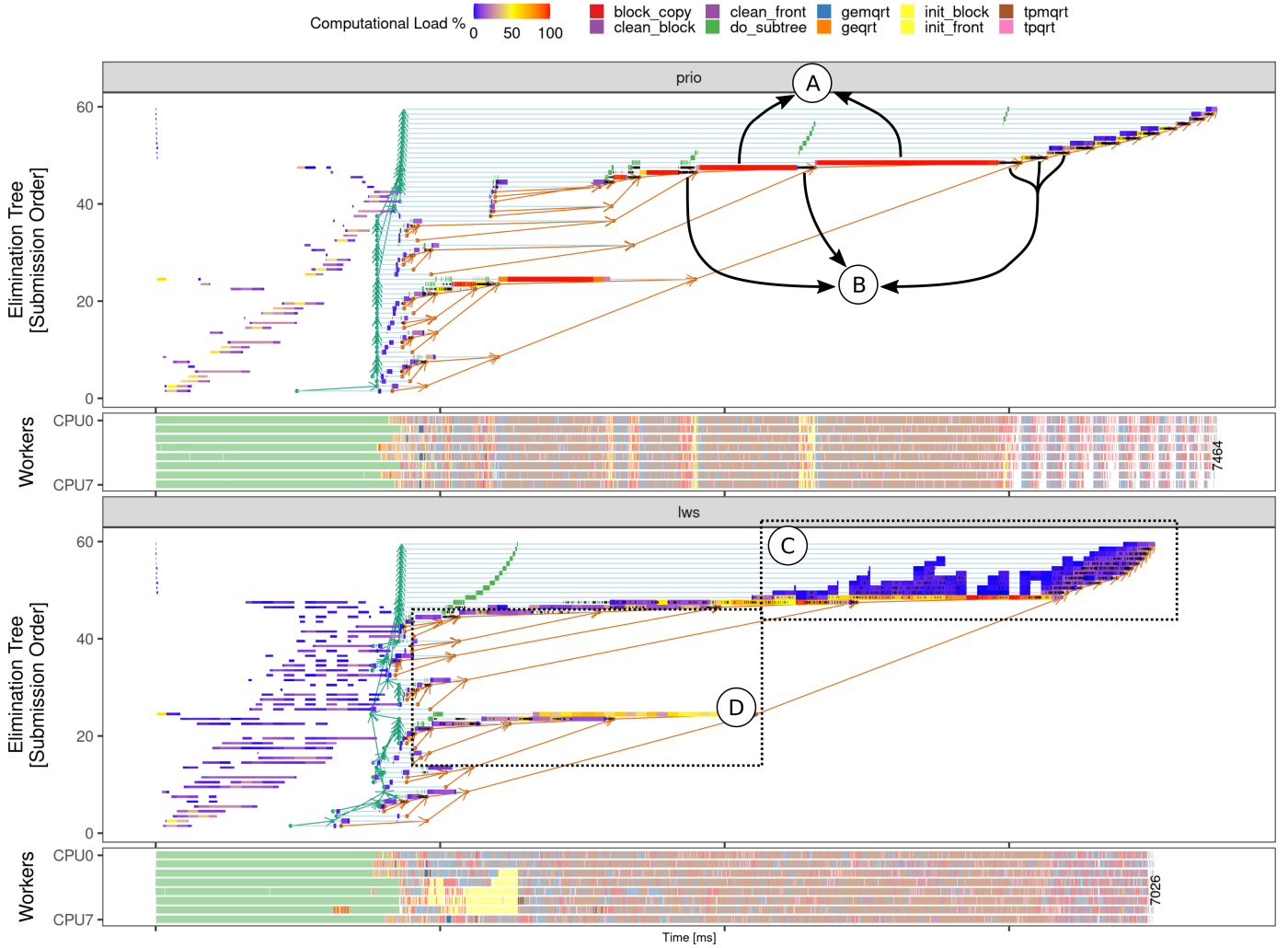


Figure 14: Comparing prio (top, with the elimination tree and gantt-chart) and lws (bottom) schedulers for the degme matrix in the Tupi Machine.

better usage of resources than heteroprio.

The comparison illustrated in Figure 15 demonstrates the usefulness of the elimination tree panel to understand the performance of such complex applications. We can indeed correlate the elimination tree events and the workers resource usage. For instance, the idle time shown in the B markers correlates with the elimination tree nodes. Because of the memory limitation, the application is incapable to release more tasks to be executed. This behavior clearly suggests to release the memory limitation. When doing so the idle time indeed disappears but the memory-intensive task interference illustrated in Section 4.5 unfortunately comes back. This interplay suggests the need for a finer control of the memory limitation, active in the beginning but relaxed toward the end.

6. Conclusion

This work presented new performance visualization panels and techniques that were added to the StarVZ tool. The task regression modeling for performance analysis and these new panels are related to the irregularity of sparse factorization applica-

tions and the elimination tree structure from the multifrontal method. These panels give the specialists an application-specific performance point of view. At the same time, the anomalous task detection mechanism for irregular tasks proved its usefulness to help identify performance issues in many levels of an application.

We have seen that combining our proposed techniques can help us identify problems on different levels. At the runtime level, for example, we have the task submission interference. The overhead states associated with the anomalous task revealed the tracing flush problem and the block_copy task priorities related to the different runtime schedulers, which led to unwanted behavior in the application regarding the DAG/tree traversal. At the application level, we can have a set of parameters that degrade application performance, like memory consumption control, which leads to other side effects such as reducing the task's locality efficiency, increasing cache misses and task duration. The later effect is captured by fitting multiple models in the data, which help us automatically detecting those regions and also can help to provide better simulations. Furthermore, we have seen that some anomalies may come from

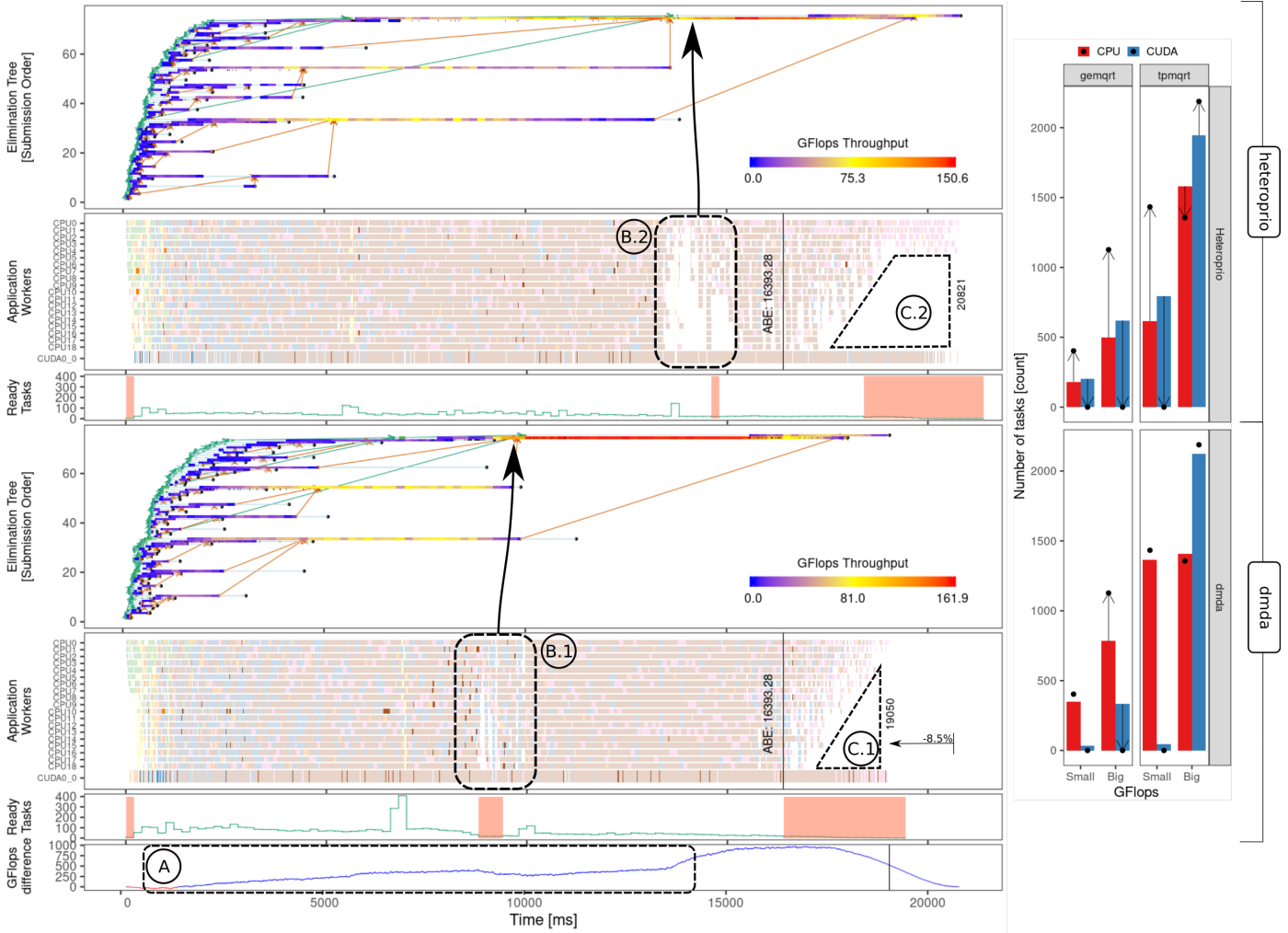


Figure 15: Left: a comparison of the application behavior when using the heteroprio (top, with the elimination tree, gantt-chart and ready tasks) and dmda (middle) schedulers, with the GFlops difference between them (bottom); Right: GFlops histograms per resource type (CPU and CUDA) for two task types (gemqrt and tpmqrt) for the heteroprio (top) and dmda (bottom) schedulers, including the ABE target as points and arrow pointing to them (to depict distance from such an ideal case).

1076 the workload given to the application, changing the task data¹⁰⁹⁵
 1077 blocks content, and the way the underlying architecture handles
 1078 denormal numbers impacts performance. ¹⁰⁹⁶

1079 The information used to create these multifrontal method-¹⁰⁹⁷
 1080 related visualizations can be commonly found in implementa-¹⁰⁹⁸
 1081 tions of the method, which helps to make the workflow generic¹⁰⁹⁹
 1082 enough to work with other multifrontal-based applications than¹¹⁰⁰
 1083 QR_mumps, or even other sparse factorization methods for the¹¹⁰¹
 1084 anomalies detection. ¹¹⁰²

1085 Using the developed analysis and visualization techniques in¹¹⁰⁴
 1086 other scenarios like with multi computational nodes or other ap-¹¹⁰⁵
 1087 plications may reveal other interesting contributions, helping to¹¹⁰⁶
 1088 understand application performance in even more challenging¹¹⁰⁷
 1089 scenarios. Also, there are some opportunities to develop panels¹¹⁰⁸
 1090 to relate tree computation to NUMA nodes and PAPI hardware¹¹⁰⁹
 1091 counters. In addition, we consider exploring these performance
 1092 visualization techniques in other task-based sparse solvers that
 1093 implement the multifrontal method, such as the Cholesky and
 1094 LU factorization of PaStiX.

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. The companion material of this paper, including experimental details, source code, and code snippets to regenerate figures, is hosted by ZENODO, for which we are also grateful. 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.

1110 **References**

- 1111 [1] [Software Release] G. Karypis, V. Kumar, and A. Gladky, *METIS* version 5.1.0, Feb. 2021. SWHID: `<swh:1:rev:ce7fd39dec097f7fb5fa661260ee13b4808873e7>`. 1163-1168
- 1112 [2] [Software Release] C. Augonnet et al., *StarPU* version devel, Oct. 2020. SWHID: `<swh:1:rev:cc8d6e7fea87e825b90631bd9a164082cbb1ae5b;origin=https://gitlab.inria.fr/starpu/starpu.git;visit=swh:1:snp:7a6c8616d125a4aa5d5b639cce8623e7c6b7e8ba>`. 1169-1174
- 1113 [3] [Software Release], *gcc* version 8.4.0, Mar. 2020. 1175
- 1114 [4] [Software Release] L. Mello Schnorr, V. Garcia Pinto, M. C. Miletto, and L. Leandro Nesi, *StarVZ* version 0.4.0, 2020. SWHID: `<swh:1:dir:10c1ac0d164a5771132468ac2a4aa4eca1169467>`. 1176-1179
- 1115 [5] [Software Release], *OpenBLAS* version 0.3.9, Mar. 2020. 1180
- 1116 [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: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2019. 1181-1185
- 1117 [7] M. C. Miletto and L. Schnorr. “OpenMP and StarPU Abreast: the Impact of Runtime in Task-Based Block QR Factorization Performance”. In: *Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*. SBC, 2019. 1186-1190
- 1118 [8] I. Oz, M. K. Bhatti, K. Popov, and M. Brorsson. “Regression-Based Prediction for Task-Based Program Performance”. In: *Journal of Circuits, Systems and Computers* 28.04 (2019). 1191-1194
- 1119 [9] [Software Release] 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>`. 1195-1202
- 1120 [10] I. Duff, J. Hogg, and F. Lopez. “Experiments with sparse Cholesky using a sequential task-flow implementation”. In: *Numerical Algebra, Control & Optimization* 8.2 (2018). 1203-1206
- 1121 [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 applications running on hybrid clusters”. In: *CCPE* 30.18 (2018). 1207-1211
- 1122 [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 Transactions on Parallel and Distributed Systems* 28.10 (2017). 1212-1216
- 1123 [13] G. Ceballos, T. Grass, A. Hugo, and D. Black-Schaffer. “Taskinsight: Understanding task schedules effects on memory and performance”. In: *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2017. 1163-1168
- 1124 [14] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Oxford University Press, 2017. 1169-1174
- 1125 [15] J. V. Lima, I. Raïs, L. Lefèvre, and T. Gauthier. “Performance and energy analysis of openmp runtime systems with dense linear algebra algorithms”. In: *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE, 2017. 1175-1179
- 1126 [16] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. “Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems”. In: *Acm transactions on mathematical software (toms)* 43.2 (2016). 1180-1185
- 1127 [17] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. “A survey of direct methods for sparse linear systems”. In: *Acta Numerica* 25 (2016). 1186-1190
- 1128 [18] A. Huynh, D. Thain, M. Pericàs, and K. Taura. “DAGViz: a DAG visualization tool for analyzing task-parallel program traces”. In: *Proceedings of the 2nd Workshop on Visual Performance Analysis*. 2015. 1191-1194
- 1129 [19] F. Lopez. “Task-based multifrontal QR solver for heterogeneous architectures”. PhD thesis. Université de Toulouse, Université Toulouse III-Paul Sabatier, 2015. 1195-1202
- 1130 [20] 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: *21st IEEE International Conference on Parallel and Distributed Systems, ICPADS 2015, Melbourne, Australia, December 14-17, 2015*. IEEE Computer Society, 2015, pages 481–490. 1203-1206
- 1131 [21] 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). 1207-1211
- 1132 [22] 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). 1212-1216
- 1133 [23] 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*. Edited by R. Borgo et al. The Eurographics Association, 2014. ISBN: 978-3-03868-028-4. 1217-1221
- 1134 [24] 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. 1222-1226

- 1217 [25] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez1269
1218 “Multifrontal QR factorization for multicore architec1270
1219 tures over runtime systems”. In: *European Conference*1271
1220 *on Parallel Processing*. Springer. 2013. 1272
- 1221 [26] A. Buttari. “Fine-grained multithreading for the multi1273
1222 frontal QR factorization of sparse matrices”. In: *SIAM*1274
1223 *Journal on Scientific Computing* 35.4 (2013). 1275
- 1224 [27] C. F. Van Loan and G. H. Golub. *Matrix computations*1276
1225 *fourth edition*. Johns Hopkins University Press Balti1277
1226 more, 2013. 1278
- 1227 [28] Z. Xianyi. *OpenBLAS: An optimized BLAS library*. 20131279
1228 URL: <http://www.openblas.net/> (visited on1280
1229 05/15/2020). 1281
- 1230 [29] K. Coulomb, A. Degomme, M. Faverge, and F. Tra1282
1231 hay. “An open-source tool-chain for performance anal1283
1232 ysis”. In: *Tools for High Performance Computing 2011*1284
1233 Springer, 2012. 1285
- 1234 [30] R. Keller, S. Brinkmann, J. Gracia, and C. Niethammer1286
1235 “Temanejo: Debugging of thread-based task-parallel1287
1236 programs in starss”. In: *Tools for High Performance*1288
1237 *Computing 2011*. Springer, 2012. 1289
- 1238 [31] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacre1290
1239 nier. In: *Concurrency and Computation: Practice and*1291
1240 *Experience* 23.2 (2011). 1292
- 1241 [32] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Mar1293
1242 tinell, X. Martorell, and J. Planas. “Ompss: a proposal for1294
1243 programming heterogeneous multi-core architectures”.1295
1244 In: *Parallel processing letters* 21.02 (2011). 1296
- 1245 [33] R. Nath, S. Tomov, and J. Dongarra. “Accelerating1297
1246 GPU Kernels for Dense Linear Algebra”. In: *Proceed1298
1247 ings of the 2009 International Meeting on High Per1299
1248 formance Computing for Computational Science, VEC1300
1249 PAR’10*. Berkeley, CA: Springer, June 2010. 1301
- 1250 [34] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. “A1302
1251 class of parallel tiled linear algebra algorithms for multi1303
1252 core architectures”. In: *Parallel Computing* 35.1 (2009)1304
1253 1305
- 1254 [35] P. Cicotti, X. S. Li, and S. B. Baden. “Performance mod1305
1254 eling tools for parallel sparse linear algebra computa1306
1255 tions”. In: *Parallel Computing*. Citeseer. 2009. 1307
- 1256 [36] Khronos OpenCL Working Group. *The OpenCL Specifi1308
1257 cation, version 1.0.29*. Dec. 2008. 1309
- 1258 [37] C. Nvidia. “Cublas library”. In: *NVIDIA Corporation,*
1259 *Santa Clara, California* 15.27 (2008).
- 1260 [38] T. Gautier, X. Besseron, and L. Pigeon. “Kaapi: A thread
1261 scheduling runtime system for data flow computations
1262 on cluster of multi-processors”. In: *Proceedings of the*
1263 *2007 international workshop on Parallel symbolic com-*
1264 *putation*. ACM. 2007.
- 1265 [39] L. Grigori and X. S. Li. “Towards an accurate perfor-
1266 mance modeling of parallel sparse factorization”. In: *Ap-*
1267 *licable Algebra in Engineering, Communication and*
1268 *Computing* 18.3 (2007).
- [40] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng.
“Algorithm 836: COLAMD, a column approximate min-
imum degree ordering algorithm”. In: *ACM Transactions*
on Mathematical Software (TOMS) 30.3 (2004).
- [41] P. Hénon, P. Ramet, and J. Roman. “PASTIX: a high-
performance parallel direct solver for sparse symmetric
positive definite systems”. In: *Parallel Computing* 28.2
(2002).
- [42] P. R. Amestoy, I. S. Duff, and J.-Y. L’excellent. “Mul-
tifrontal parallel distributed symmetric and unsymmet-
ric solvers”. In: *Computer methods in applied mechanics*
and engineering 184.2-4 (2000).
- [43] L. Dagum and R. Menon. “OpenMP: An industry-
standard API for shared-memory programming”. In:
Computing in Science & Engineering 1 (1998).
- [44] G. Karypis and V. Kumar. “METIS: A software pack-
age for partitioning unstructured graphs, partitioning
meshes, and computing fill-reducing orderings of sparse
matrices”. In: (1997).
- [45] F. Pellegrini and J. Roman. “Scotch: A software pack-
age for static mapping by dual recursive bipartitioning of
process and architecture graphs”. In: *International Con-*
ference on High-Performance Computing and Network-
ing. Springer. 1996.
- [46] V. Pillet, J. Labarta, T. Cortes, and S. Girona. “Paraver: A
tool to visualize and analyze parallel code”. In: *Proced-*
ings of WoTUG-18: transputer and occam developments.
Volume 44. 1. Citeseer. 1995.
- [47] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J.
Hanson. “An extended set of FORTRAN basic linear al-
gebra subprograms”. In: *ACM Transactions on Mathe-*
matical Software (TOMS) 14.1 (1988).
- [48] J. W. Liu. “Computational models and task scheduling
for parallel sparse Cholesky factorization”. In: *Parallel*
computing 3.4 (1986).
- [49] I. S. Duff and J. K. Reid. “The multifrontal solution of in-
definite sparse symmetric linear”. In: *ACM Transactions*
on Mathematical Software (TOMS) 9.3 (1983).
- [50] B. M. Irons. “A frontal solution program for finite ele-
ment analysis”. In: *International Journal for Numerical*
Methods in Engineering 2.1 (1970).