UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCELO COGO MILETTO

Combining Prediction Models and Visualization Techniques for Enhanced Performance Analysis of Irregular Task-based Applications

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Lucas Mello Schnorr
Coadvisor: Prof. Dr. Claudio Schepke

Porto Alegre
February 2021

CIP — CATALOGING-IN-PUBLICATION

Cogo Miletto, Marcelo

Combining Prediction Models and Visualization Techniques for Enhanced Performance Analysis of Irregular Task-based Applications / Marcelo Cogo Miletto. – Porto Alegre: PPGC da UFRGS, 2021.

140 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Lucas Mello Schnorr; Coadvisor: Claudio Schepke.

1. HPC, Performance Visualization, Performance Model, Multifrontal Method, Task-based Applications. I. Mello Schnorr, Lucas. II. Schepke, Claudio. III. Título.

Combining Prediction Models and Visualization Techniques for Enhanced Performance Analysis of Irregular Task-based Applications

ABSTRACT

Parallel application performance analysis is an essential and a continuous step towards understanding and optimizing any high-performance program. Nowadays, ubiquitous and complex heterogeneous architectures turn this job even more burdensome. While paradigms like task-based ease programming through its abstractions and its runtime system, the analysis of such applications demand attention because of its specific view of the applications. Likewise, the analysis of irregular applications built upon specific data structures need to consider its abstractions and behavior to improve and facilitate an analyst's work. Thus, the current work proposes strategies to enhance the performance analysis of irregular task-based applications and propose application-centric visualization panels to represent performance according to the elimination tree structure, the foundation of many direct sparse factorization methods. The strategies rely on tracing information for collecting task performance data. Since task-based applications can create many tasks and huge trace files, the proposed automatic mechanism for anomalous task classification based on regression models allows highlighting specific groups of problematic tasks and guiding the analysis process. The visualization techniques represent the tree structure and describe application-specific concepts like tree and node parallelism, child and parent dependencies, and communications. Those strategies are applied to the `qr_mumps` sparse task-based solver in an extensive set of experiments. The anomalous detection mechanism exposed four different task anomaly sources, guiding a solution that improved performance by up to 24% by reducing task interference. The elimination tree visualization panels allowed detailed comparisons between different application and runtime configurations, revealing other sources of inefficiency. The experiments also involved testing the `qr_mumps` application in a real computational simulation application, where it presented better performance than other parallel solvers. The results demonstrate the usefulness of the proposed strategies to guide the performance analysis of irregular task-based applications and enhance the performance representation of elimination-tree based applications.

Keywords: HPC, Performance Visualization, Performance Model, Multifrontal Method, Task-based Applications.

Combinando Modelos de Predição e Técnicas de Visualização para Melhorar a Análise de Desempenho de Aplicações Baseadas em Tarefas Irregulares

RESUMO

A análise de desempenho de aplicações paralelas trata-se de uma etapa essencial e contínua para entender e otimizar aplicações de alto desempenho. Arquiteturas heterogêneas hoje estão onipresentes e tornam esse trabalho ainda mais oneroso. Enquanto paradigmas como a programação baseada em tarefas facilitam o desenvolvimento por meio de abstrações e o sistema de *runtime*, sua análise exige mais atenção devido a sua visão específica da aplicação. Da mesma forma, análises de aplicações irregulares e construídas sobre estruturas de dados específicas precisam considerar tais características para facilitar o trabalho de analistas. Assim, este trabalho propõe estratégias para aprimorar a análise de desempenho de aplicações baseadas em tarefas irregulares usando painéis de visualização específicos, representando o desempenho de acordo com a estrutura da árvore de eliminação, alicerce de muitos métodos de fatoração esparsa direta. As estratégias utilizam informações de rastreamento para coletar dados de desempenho de tarefas. Como aplicações baseadas em tarefas podem gerar grandes arquivos de rastreamento, é proposto um mecanismo para classificação de tarefas anômalas com base em modelos de regressão que permite destacar tarefas problemáticas automaticamente, direcionando a análise. As técnicas de visualização representam a estrutura da árvore e comportamentos específicos da aplicação, como o paralelismo da árvore e dos nós, dependências entre nós filhos e pais, e comunicações. Essas estratégias são aplicadas ao *solver* esparso baseado em tarefas `qr_mumps` em um conjunto de experimentos. Os modelos de regressão expuseram quatro fontes de anomalias, guiando uma solução que melhorou o desempenho em até 24% ao reduzir a interferência entre tarefas. Os painéis de visualização da árvore de eliminação permitiram comparações detalhadas entre diferentes configurações da aplicação e *runtime*, revelando outras fontes de ineficiência. Também usamos o `qr_mumps` em uma aplicação de simulação computacional, onde ele apresentou melhor desempenho do que outros *solvers* paralelos. O estudo demonstrou a utilidade das técnicas propostas para guiar a análise de desempenho de aplicações baseadas em tarefas irregulares e melhorar a representação do desempenho de aplicações construídas sobre árvores de eliminação.

Palavras-chave: HPC, Método Multifrontal, Aplicações Baseadas em Tarefas, Visualização de Desempenho, Modelo de Desempenho.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CFD	Computational Fluid Dynamics
CUDA	Compute Unified Device Architecture
CPI	Cycles-Per-Instruction
CSR	Compressed Sparse Row
DAG	Directed Acyclic Graph
FDM	Finite Differences Method
FEM	Finite Element Method
FMA	Fused Multiply-Add
FVM	Finite Volume Method
FXT	Fast Kernel Tracing
GMRES	Generalized Minimal Residual Method
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
HPC	High Performance Computing
LWS	Local Work Stealing
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language
OTF2	Open Trace Format Version 2
PSNR	Peak Signal-To-Noise Ratio
RFA	Radiofrequency Ablation
RAFEM	Radiofrequency Ablation Finite Element Method

SIMD Single Instruction, Multiple Data

STF Sequential Task Flow

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

High-Performance Computing (HPC) has been an essential tool for any research field. It provides the necessary computational power to perform very complex and precise simulations, for example, which contribute to many advances in technology. But just providing this computational power through many different complex architectures is insufficient to solve the ever-increasing problems in this context. To efficiently solve them, we need applications that optimally explore the parallel capabilities of the current hardware components. To implement such big and complex applications that are fast and accurate is a real challenge. The heterogeneity over the architecture processing units builds up an extra level of concern. Also, the application workloads can be extremely irregular, all this leading to unbalanced and inefficient executions. One of the most common solutions to overcome these challenges is to follow a task-based programming approach.

Task-based programming models have been under constant evolution during the past years. There are now numerous libraries that support such a programming paradigm, focusing on developing high-performance parallel applications. Libraries like OpenMP (DAGUM; MENON, 1998), StarPU (AUGONNET et al., 2011), TBB (WILLHALM; POPOVICI, 2008), Xkaapi (GAUTIER et al., 2013), Cilk (FRIGO; LEISERSON; RANDALL, 1998), and OmpSS (DURAN et al., 2011) implement efficient and flexible task-based models. With this technique, we can describe the main application structure using a Directed Acyclic Graph (DAG), whose nodes represent different computational tasks, and their edges the data dependencies among them. This structure defines the application execution flow, which is scheduled dynamically by a runtime system following a defined scheduling policy. This approach eases the work of programmers by taking care of the scheduling step in a portable way. There are many powerful strategies like NUMA memory-aware scheduling (BROQUEDIS et al., 2010), heterogeneous data-aware scheduling considering data transfer costs (AUGONNET et al., 2010) and even performance model-based scheduling policies (AUGONNET et al., 2011). The task-based approach is an effective and portable solution to evenly distribute the workload among the computational units (DONGARRA et al., 2017), even if the workload is irregular and the architecture complex.

Although the task-based model offers some ease for developing parallel applications, analyzing the performance of that kind of application is a troublesome job. The runtime task scheduling decisions are stochastic, leading to variability in task duration

and application behavior. One strategy used for analyzing these applications is to rely on tracing systems that collect data of specific events during the application execution. This detailed data allows us to reconstruct the application behavior to analyze it over time, investigating runtime decisions, and even comparing the performance with expected results from performance models. The performance analysis is even more challenging when we face irregular workloads, where the same type of tasks have different computational weights depending on their parameters. The runtime decisions might not be aware of this characteristic of the tasks. Thus, some runtime decisions might degrade the performance, causing tasks to have a higher execution time than they should, based on a performance model. To identify these decisions and scenarios where the performance is worse than expected can help developers understand the application behavior better and improve its performance. As application runs can generate a considerable amount of tasks, we need to automatically detect such task duration anomalies because manually analyzing them can be very cumbersome. This automatic analysis can be challenging but can lead to improved performance analysis for many applications.

We propose the adoption of the `qr_mumps` application (AGULLO et al., 2013) as a case study. This application performs a task-based sparse matrix QR factorization, handling irregular task weights over multicore and heterogeneous platforms. It uses the StarPU library to handle the task-based parallelism, using 2D tiled algorithms and the concept of elimination tree (LIU, 1990) for expressing the factorization parallelism. Besides the task irregularity that can be investigated, in `qr_mumps`, we can also investigate the elimination tree structure. This algorithmic-wise investigation contributes to analyzing the application performance in a space-time view aspect, looking into how it navigates through the structure and relates it with performance. We also evaluate the use of a fully-featured parallel task-based sparse solver like `qr_mumps` in a real simulation in the RAFEM (JIANG et al., 2010) application, compared with other state of the art solvers like the `MAGMA` (TOMOV; DONGARRA; BABOULIN, 2010) library and `cuSOLVER` (NVIDIA, 2020). Besides that, we have chosen this application because our research group already has contact with the developing teams of both `qr_mumps` and StarPU, which makes the cooperation much more straightforward.

1.1 Motivation

Developing efficient large scale parallel applications is difficult, and it is a process that requires constant performance analysis methods to know what and where to improve. Since task-based programming has gained popularity over HPC applications, we also need to consider all factors that affect a task-based application in this analysis, like the runtime system and its scheduling strategy, and provide a task-wise application point of view. Providing tools and techniques for better performance analysis is an object of constant study, and they help provide the necessary understanding of the application behavior and its performance. While there are many performance analysis tools like StarVZ (PINTO et al., 2018), Vite (COULOMB et al., 2012), and Vampir (KNÜPFER et al., 2008), among many others, they fail to consider all aspects of an application like task irregularity. Furthermore, in the case of the multifrontal method, which is a widespread technique for parallel direct sparse solvers, we can align application data structures to the computational resources. This relation helps to synchronize programmers' abstractions to the performance delivered by the application, which, as far as we know, remains unexplored for the multifrontal method and the elimination tree. To further improve this kind of tools, we can provide a technique to detect anomalous tasks based on a performance model automatically. This technique can guide programmers to better analyze if there is a strange behavior over the highlighted tasks by such a strategy. We will integrate our contributions into the StarVZ tool, which represents an effort to understand and analyze the performance of irregular task-based applications.

1.2 Contributions

The present work contributions are concentrated around two main points: (1) providing enhanced performance analysis for irregular task-based applications through performance modeling using different regression models. And (2) specific visualizations for the multifrontal method through application-oriented performance visualization panels considering the algorithm data structures, in this case, the elimination tree. The principal contributions are listed as follows:

1. Provide a set of algorithm-wise visual performance analysis panels for task-based multifrontal methods in the StarVZ tool, which can be applied to any task-based

multifrontal method that provides the necessary data.

2. Create a detailed performance prediction model, considering the irregularity of tasks data and different regression model strategies, guiding the user with visual interpretations of the models, helping it choosing the adequate model for each task and run configuration.

3. Extensive set of experiments that identified some interesting scenarios regarding the impact of the many factors in the `qr_mumps` application behavior, and scenarios that caused performance degradation. Among the explored cases, we highlight:

    - A set of different sources of task anomalies in the application that were captured by our modeling.
    - The impact of different application and runtime factors in the application behavior regarding the elimination tree structure, such as scheduler, memory consumption, and task priorities.
    - Side effects of other factors that impact concurrent running tasks efficiency and reduce their data locality benefits.

4. Exploring the performance and numerical properties of different state of the art parallel sparse solvers for the real-world application RAFEM, accelerating it, and discussing further steps towards increasing speedup.

1.3 Structure of the Text

This document is organized as follows. Chapter 2 presents a background and context over the task-based programming paradigm, parallel task-based sparse solvers, scientific applications, and the process of collecting performance data for further analysis. Chapter 3 discuss related work on methods and techniques for general purpose and task-based performance analysis, and strategies and uses of performance modeling and prediction. Chapter 4 brings the methodological aspects of our contributions to the development of novel performance visualization panels related to the multifrontal method and the enhanced performance analysis of irregular task-based applications through regression models. Chapter 5 presents experimental results using the `qr_mumps` application, showing the usefulness of proposed application performance analysis techniques. Chapter 6 shows the use of `qr_mumps` in a real-world application, showing how to tune it and

discussing future improvement points. Lastly, Chapter 7 finalizes this document with the conclusions and points to future work.

2 BACKGROUND: HPC APPLICATIONS AND PERFORMANCE ANALYSIS

The present chapter provides, in a first moment, background information on the task-based programming paradigm and an important class of applications built on top of it: parallel sparse solvers. Section 2.1 presents the principal concepts of task-based programming and the view of two different APIs for supporting this paradigm. Section 2.2 presents a common target application to represent using a task-based approach, which is parallel matrix factorization algorithms and solvers. Then, in Section 2.3, we describe real scenarios where those efficient parallel solvers are needed to achieve good performance and allow computational scientific applications to simulate problems in a feasible time. Lastly, Section 2.4 provides an overview of techniques to collect performance data for application analysis.

2.1 Task-Based Programming Paradigm

HPC applications rely on hardware parallelism to accelerate computations. As the supercomputers are continually moving towards heterogeneous architectures, the complexity of creating efficient programs rises, increasing the number of concerns for parallel programs conception and the amount of work necessary to deliver such high performance. Heterogenous platforms imply distinct costs for communication, different throughput for the computational resources, creating a challenging scenario for the applications to utilize those resources efficiently. In addition, many APIs provide ways to handle the numerous accelerators and explorable types of parallelism. MPI (GROPP; LUSK; SKJELLUM, 1999) is the de facto standard for handling multi-node parallelism, coordinating intra and inter-node communications. For shared-memory platforms, the de facto standard is the OpenMP API (OpenMP, 2018). OpenMP provides directives that extend the compiler program's understanding, allowing it to partition and divide work among the machine cores automatically, supporting all sorts of parallelism types, multithreading, SIMD, task-based (started in OpenMP 3.0, improved with dependencies later in version 4.0), and heterogeneous programming offloading computation to other devices (since OpenMP 4.0). We can also use CUDA (NVIDIA, 2020), OpenACC (CAPS Enterprise, Cray, Nvidia, PGI, 2018), and OpenCL (STONE; GOHARA; SHI, 2010) to offload calculations for GPGPUs, and the latter can also control FPGAs programmability. Hence, with all available strategies that we can combine to explore parallelism over multiple platforms simul-

taneously, considering its heterogeneity, creating a correct and efficient parallel program may sound a bit overwhelming.

Task-based parallelism, also known as dataflow parallelism, is a programming approach that can ease such an overwhelming job by allowing programmers to express their applications in a more declarative way, letting the runtime system decide what, when, and where the tasks execute in the hardware (DONGARRA et al., 2017). It allows representing the application computations and communications as a Directed Acyclic Graph (DAG), transferring some of those complex responsibilities like scheduling considering, load balancing, data locality, and ensuring data coherence to the runtime system. Figure 2.1 depicts the role and the view of the runtime system in a task-based application. The runtime organizes the application tasks as a DAG like in the left part of the figure, representing the application according to the tasks data access modes. It creates the dependencies between them when we have tasks with read-after-write, write-after-read (anti-dependency) that can be avoided by making an extra copy of the data, and write-after-write conflicts.

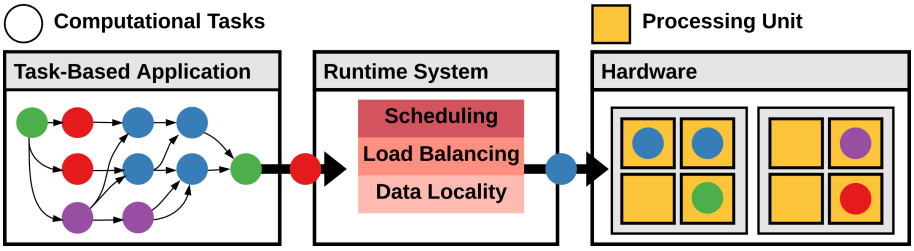
The runtime scheduler dynamically chooses how to distribute the tasks over the available computational resources. This scheduling is done very efficiently since it is a subject under constant research (TOPCUOGLU; HARIRI; WU, 2002; SINNEN, 2007; AUGONNET et al., 2011), even considering heterogeneous platforms.

As the HPC landscape is continuously shifting to bigger, heterogeneous parallel systems, the task-based approach has proved its value in simplifying programming and providing performance in some cases. Standard APIs like OpenMP now support task-based parallelism, and many others (AUGONNET et al., 2011; DURAN et al., 2011; BOSILCA et al., 2012a) have arisen to help in efficient parallel programs development. Furthermore, by decoupling the application from the computational resources, without the need to explicitly mapping them to specific workers, one can create applications with performance portability in a very natural way, as a side-effect of the task-based programming structure.

2.1.1 OpenMP Tasks

OpenMP version 3.0 introduced the concept of explicit tasks using the `task` construct, and later improved in version 4.0 with the `depend` clause, which allows defining the access mode to the task data variables that are later translated into task dependencies

Figure 2.1 – Runtime role and view in a task-based application.



Source: The Author

by the runtime. The OpenMP API already had the implicit task concept before implementing explicit tasks. According to the OpenMP 5.0 specification, it defines a task as "a specific instance of executable code and its data environment that the OpenMP implementation can schedule for execution by threads" (OpenMP, 2018). Every time that a thread reaches a `task` construct, the program creates a new task that can be immediately executed or postponed to a future moment. The `task` construct receives a series of optional OpenMP clauses for specifying task dependencies, priority, affinity, and other behavioral aspects for its data or synchronization with other tasks. A set of statements follows this construct, representing the work the task will perform asynchronously. Figure 2.2 shows three examples of task creation in OpenMP using the task construct with the `depend` clause.

Figure 2.2 – Example of code to define tasks using OpenMP.

```
1 #pragma omp task depend(inout:x,y) depend(in:a)
2 { x+=a; y*=a; }
3
4 #pragma omp task depend(in:x)
5 { foo(x) } // wait for x value from the first task
6
7 #pragma omp task depend(in:y)
8 { bar(y) } // wait for y value from the first task
```

Source: The Author.

The runtime constructs the DAG according to the task data dependencies specified in the `depend` clause using the values `in`, `out`, and `inout`, and other synchronization points like the `taskwait`, `barrier`, and groups of tasks using the `taskgroup` construct. For example, the first task in line 1 of the figure uses the value of the variable `a` as read-only defined by the `depend(in:a)` to update the variables `x` and `y` values. The other two tasks that use `x` and `y` as input must wait for this first task, and they can run concurrently. The created task can be immediately executed or postponed to a

future moment. The runtime can also preempt a task to run one with higher priority, resuming the interrupted task's execution later. OpenMP categorizes tasks into `tied` and `untied` tasks. When a thread starts executing a task, the default behavior is to tie this task to that thread, making it the only thread that can resume that task's execution if it was preempted. The OpenMP standard implements a limited preemptive strategy (SERRANO; ROYUELA; QUIÑONES, 2018), which restricts the `tied` task preemptions to specific application points, called task scheduling points in the OpenMP terminology. Those scheduling points include creation, completion, synchronization points, and manually specified points using the `taskyield` construct. In contrast, the runtime can preempt an `untied` task at any time, and any thread can continue the `untied` task execution. The API also offers ways to avoid creating a huge data environment by creating `mergeable` tasks and avoiding the overhead of creating too many tasks using the `final` clause, which can help when using recursive algorithms and nested tasks.

The OpenMP API now provides a reliable way to describe task-based parallel programs and many features that achieve better performance. It is now moving towards being the standard for shared-memory task-based parallelism, motivating popular libraries such as Plasma to move from specialized custom task schedulers (AGULLO et al., 2009), to the OpenMP general-purpose tasking model (DONGARRA et al., 2019), without losing performance. However, it still lacks some flexibility, like providing multiple platform-specific implementations of a task and deciding at runtime which implementation to use. Still, there is an interest in implementing such approaches in an OpenMP environment given its usefulness for heterogeneous platforms (MILANI, 2020). This ability to provide multi-implementations of tasks is essential to achieve peak performance in the widespread heterogeneous systems. Other runtime systems for task-based programming like Xkaapi (GAUTIER et al., 2013) and StarPU (AUGONNET et al., 2011) already consider such aspects to provide performance.

2.1.2 StarPU Runtime

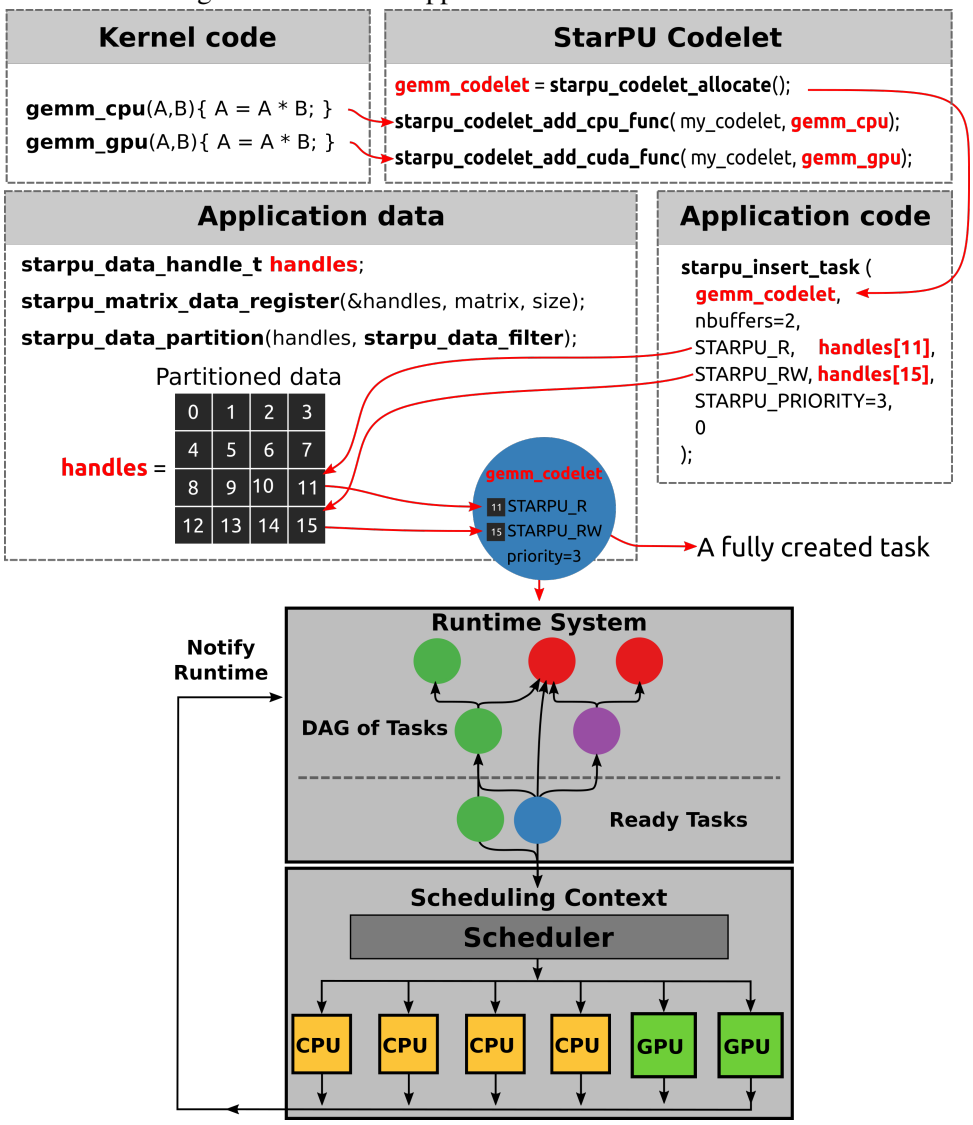
StarPU is a library for task-based programming over heterogeneous architectures with a C/C++ API and a Fortran interface to allow programmers to submit computational tasks over resources, describing the application in terms of a DAG. StarPU allows expressing task parallelism by providing a representation of the program's fundamental parts: the tasks and the data, along with its input and output dependencies. StarPU exploits paral-

lelism following the natural program flow by adopting the Sequential Task Flow (STF) programming model (AGULLO et al., 2016), submitting tasks as they appear in the sequential code, and letting the StarPU runtime schedule them asynchronously. The StarPU runtime infers the task dependencies based on their data access conflicts, building up the application DAG and managing to provide optimized heterogeneous scheduling, data transfers considering prefetching, data replication, and efficient cluster communication.

We can understand the StarPU functionality focusing on its basic concepts. The top part of Figure 2.3 presents the steps to build a task until its submission following the red arrows. The first part is implementing the computational `kernels` representing the work a task will perform considering architecture-specific code. In the figure example, we have a matrix multiplication kernel. A StarPU `codelet` relates a task operation to its kernel implementations. Notice that a `codelet` can provide multiple implementations for the same task operation using different kernels to represent the task's work, allowing the application to have one implementation for each type of devices like CPUs, CUDA GPUs, and OpenCL devices. In the StarPU codelet in the figure, we have a matrix multiplication codelet, the `gemm_codelet`, that has two different operation implementations, one for GPU and the other CPU. The `data handles` in the application data part represents a portion of data managed by StarPU. A task will then perform its kernel operations specifying the handles access mode: `read-only`, `write-only`, or `read and write`. StarPU provides structures and functions that help the user partition the registered data, creating subsets from the total data. This partitioning is configured in the `starpu_data_filter` structure, passed to the `starpu_data_partition` function. In the example, we divide the content of the variable `matrix` into 2D blocks. A `task` in the StarPU concept is the instantiation of a `codelet` over some `data handles` which executes the kernel atomically, represented by the bigger blue circle in the figure. Unlike in OpenMP, the StarPU execution model has no preemption during task execution.

From the application point of view, it is only submitting tasks in a specific order using the `starpu_insert_task` function, which incredibly simplifies the efforts to create efficient parallel programs. However, under the hood, StarPU is taking care of the many concerns to provide performance and data coherence. Figure 2.3 also provides an overview of the StarPU execution system, from the task submission to the runtime scheduler and worker's role. The data dependencies between tasks `data_handle` access modes build up the DAG, and only the ready tasks (those whose dependencies are

Figure 2.3 – StarPU application and runtime overview.



Source: The Author

already satisfied) are moved to the scheduler layer. The scheduler is responsible for assigning each task to a worker, considering aspects to provide load-balance and minimize the total execution time. The StarPU runtime provides a comprehensive set of scheduling policies that vary from simple central task-queues considering priorities (`prio`), one task-queue per worker, considering classical techniques like work stealing (`lws`), to heterogeneous data-aware strategies that consider the tasks performance models and the machine topology communication costs (`dmda`), using the expected task cost as a scheduling hint, considering where the data is currently located (`dmdasd`), and heterogeneous priorities (`heteroprio`). StarPU enables the user to define multiple scheduling contexts with different policies and a different set of workers, choosing which context to insert the tasks, and also run through distributed memory environments with its MPI module

(AUGONNET et al., 2012).

2.2 Towards Modern Parallel Task-Based Solvers

While many applications can achieve excellent performance with data-parallelism and other techniques, there is still a good portion of applications that can take advantage of the task-based parallelism (THIBAULT, 2018). Parallel solvers are one of those applications. They are essential for numerous scientific applications, as they often have to solve large and typically sparse linear equations systems, representing more than 80% of the application execution time (CAMARDA; STADTHERR, 1998). With the task-based programming delivering a compact and powerful way to express parallelism, many works considered implementing custom schedulers (AGULLO et al., 2016; HÉNON; RAMET; ROMAN, 2002) based on algorithmic knowledge, and many domain-specific runtime systems implementations arose for linear algebra applications (CHAN et al., 2008; SONG; YARKHAN; DONGARRA, 2009; YARKHAN; KURZAK; DONGARRA, 2011; BOSILCA et al., 2012b). In fact, the task-based approach was explored even in the pre-multicore era (GEIST; NG, 1989; AMESTOY; DUFF; L'EXCELLENT, 1998).

Throughout the years, the task-based approach proved to ease programming while achieving high performance for many algorithms, widely adopted for both dense and sparse linear algebra applications. Given this ease and flexibility to program, and the ability to provide performance over heterogeneous platforms, an uncountable number of projects for high-performance parallel direct solvers, if not all of them, nowadays adopt a task-based approach. We have seen projects like Plasma moving from the Quark runtime towards the OpenMP task standard as it evolved to support task-based parallelism. Fully featured libraries like `MAGMA` (TOMOV; DONGARRA; BABOULIN, 2010) and Chameleon (AGULLO et al., 2010), being developed on top of dynamic runtime systems. And other application like `qr_mumps`, and `PaStiX`, once based on hand-coded schedulers moving towards general-purpose runtime systems like StarPU and considered adopting other runtimes like Parsec.

The runtime systems' dynamism is very useful for irregular problems, helping to balance the computational load efficiently in cases of sparse or adaptative mesh refinement (AMR) problems (GANGULY; LANGE, 2017; KLINKENBERG et al., 2020). Its flexibility also allows to easily merge DAGs representing different application regions together, like the case in `PaStiX`, and `qr_mumps`, where the factorization and the solve

operations can partially execute concurrently. The runtime system plays a role in the performance as well. With many options available, studies try to evaluate different runtimes considering the same workload to answer which one provides more performance for a specific application, considering many aspects including runtime overhead, scheduling, and energy efficiency (AGULLO et al., 2017; LIMA et al., 2019; MILETTO; SCHNORR, 2019). Such comparisons are no easy job, as the different runtime systems come with their own programming interface which can extend and provide new implementations of task-based models, creating their own specific environments which complicate comparisons between different runtimes (AGULLO et al., 2017). In general, many runtimes provide similar performance in the best case scenario considering application parameters and runtime configurations. However, some of them provide more functionalities, genericity, and customizability than others, providing performance portability in exchange for a loss of fine-grain task efficiency (THIBAULT, 2018). The runtime system's choice may be guided by the different main goals that a runtime system and the application have, like being lightweight or highly portable and customizable. Despite the differences, using the task-based approach on top of modern runtime systems is definitely a good way to go for performance efficient parallel solvers (DONGARRA et al., 2017). Next, we will describe the implementation task-based parallel sparse solvers based on the multifrontal method.

2.2.1 Multifrontal Method for Sparse Factorization

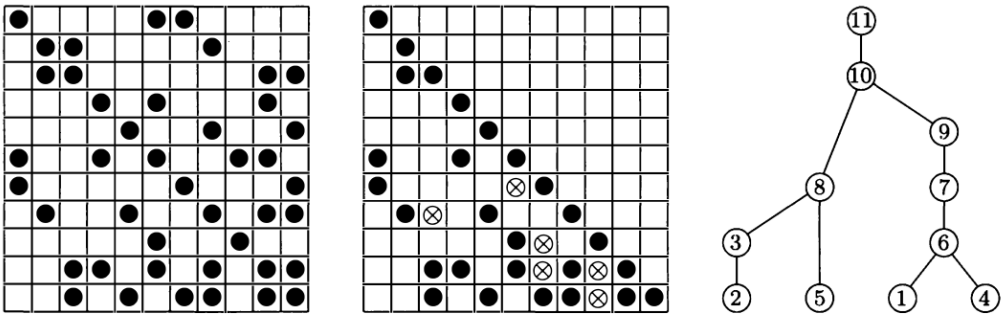
The multifrontal method (DUFF; REID, 1983) is an extension of the classical frontal method (IRONS, 1970). It was designed to factorize sparse symmetric matrices using the Cholesky decomposition. However, the method provides a structure that can be adjusted to sparse unsymmetric matrices factorization using the QR and LU decompositions, considering that the matrix holds the *Strong Hall* (i.e., fully indecomposable) property. The multifrontal approach is now a widespread technique for the parallelization of sparse direct solvers (DAVIS; RAJAMANICKAM; SID-LAKHDAR, 2016). This method reorganizes and breaks the problem of factorizing the whole sparse matrix into a set of smaller and denser subproblems called frontal matrices or simply fronts. These subproblems represent partial factorizations, like one elimination step related to a specific column j. Because of the matrix sparsity pattern, many of those elimination steps involve disjoint sets of matrix coefficients and allows us to perform many of those steps in parallel, factorizing multiple fronts at the same time, which gives the name to the method.

However, when an elimination step of a column changes the coefficients used in another step, there is a dependency between those front factorizations. This set of independent and dependent subproblems are captured by a structure named the *elimination tree* (SCHREIBER, 1982), which lies at the heart of the multifrontal method. The tree structure nodes hold the frontal matrices, and it represents the dependencies between them as a parent and child relation in the tree. The dependencies imply that a parent node can only be factorized after all its child nodes were already factorized and their contribution blocks were assembled into it. The whole matrix factorization is done by traversing the tree in the topological order, from the bottom to the top. Figure 2.4 shows an example of a given symmetric sparse matrix A (left), the computed Cholesky factor with the fill-in coefficients represented with the "$\otimes$" symbol (middle), and the matrix corresponding elimination tree (right). The elimination tree is constructed based on the symbolic pre-computed structure of the Cholesky factor L, where the relation between the parent p_j of a child node that represents the elimination of column j, is defined by Equation 2.1 as follows

$$p_j = \min\{i \mid i > j \text{ and } L_{ij} \neq 0\}. \tag{2.1}$$

The multifrontal approach provides two sources of parallelism based on the tree structure and its denser matrices, where all the tree leaves represent independent fronts.

Figure 2.4 – Example of a sparse symmetric matrix, its computed fill-in after a symbolic factorization step, and its elimination tree.



Thc matrix A A's Cholesky factor A's elimination tree

Source: (LOAN; GOLUB, 2013)

As sparse problems have to use specific matrix representations formats like Coordinate (COO), and Compressed Sparse Row (CSR), specific data structures, and need to handle the fill-in effect created during factorization, sparse solvers execution flow is commonly divided into three different phases:

- **Analysis phase**: this step handles a critical concern in sparse matrix factorization:

keeping the generation of new nonzero coefficients (fill-in effect) under control. In this phase, software libraries like COLAMD (DAVIS et al., 2004), Scotch (PELLEGRINI; ROMAN, 1996) and Metis (KARYPIS; KUMAR, 1997) use matrix/graph reordering algorithms such as Approximate Minimum Degree, Nested Dissection, and Cuthill-McKee (LOAN; GOLUB, 2013) to provide a matrix permutation that reduces the fill-in during the factorization, taking advantage of the sparse matrix structure. Applications also perform a symbolic factorization step that enables them to know where the fill-in coefficients appear to preallocate the correct necessary memory size for the final structure by calculating the matrix's final structure after the factorization. This step creates the frontal matrices and their child/parent dependencies. At the end of this phase, we have the elimination tree structure ready to be computed by the next phase.

- **Factorization phase**: this 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. Many of these front factorizations can be done in parallel. For example, the method can process all the leaves of the tree simultaneously since all of them do not have any child node. This parallelism source is called *tree parallelism*. As the computations move towards the tree root, the tree parallelism becomes more scarce because we have more parent and child dependencies and fewer nodes than in the bottom of the tree. However, the nodes upwards the elimination tree concentrate way more computations than the bottom part. As fronts get bigger, we can use techniques to parallelize the factorization of an entire tree node. Efficient implementations of the multifrontal method consider applying techniques like tiled/blocked factorization to explore the intra-node parallelism, commonly called *node parallelism*.
- **Solve phase**: lastly, the tree is traversed one more time, applying forward and backward substitutions, and a triangular solve operation for each front, grouping their results to provide the solution of the entire matrix.

2.2.2 `QR_mumps`: Fine-Grained Multifrontal QR Factorization

The `qr_mumps` application is a task-based parallel sparse solver based on the multifrontal method that uses the StarPU runtime, written in Fortran 2003. It uses the elimination tree to partition and parallelize the problem. In the first versions of `qr_mumps`

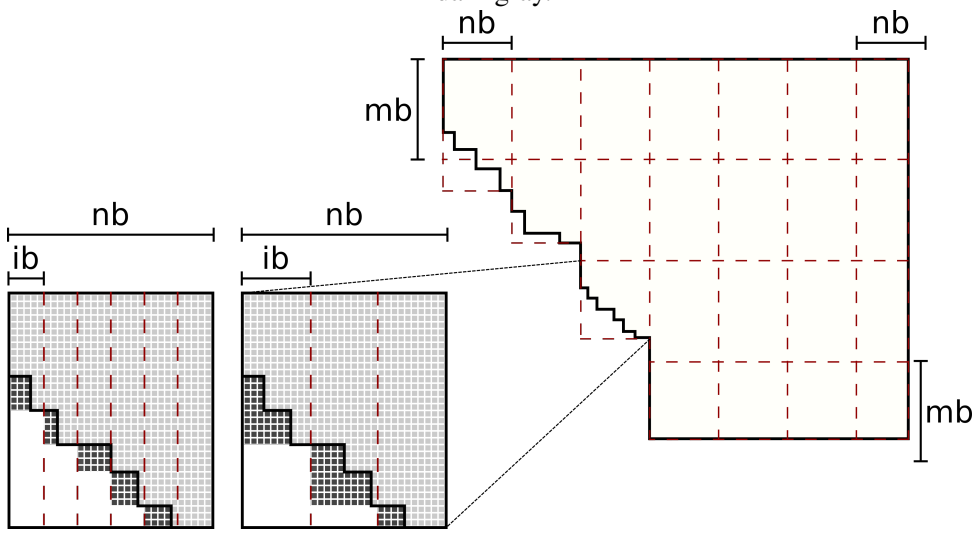
(BUTTARI, 2013), it uses a block-column partitioning, relying on an in-house scheduler to provide load-balance and memory consumption control over multicore architectures (AGULLO et al., 2013). There are some comparisons between this earlier version against a StarPU-based `qr_mumps` implementation and another QR sparse solver called SuiteSparseQR (DAVIS, 2009). Both `qr_mumps` implementations outperformed the SuiteSparseQR, but the StarPU version was slightly slower than the `qr_mumps` original version. While there is a higher overhead for the StarPU execution, typically below 10%, adopting a modern runtime system increased the code portability and allowed the programmers to better control memory usage during the factorization. This portability and ease to program allowed to improve `qr_mumps` using a finer-grained partitioning, substituting the 1D block-columns for 2D tiles on top of the StarPU runtime system (AGULLO et al., 2016). This finer-partitioning enables the application to explore more parallelism and provide heterogeneity support through the StarPU runtime system, considerably increasing the speedup against the 1D version. The fine-grained 2D tiled partitioning approach over the StarPU runtime system allows `qr_mumps` to explore a new parallelism level in the multifrontal method, referred to as *interlevel parallelism*. This new level allows to overlap computations of parent and child nodes, which was earlier blocked until the complete child node factorization. In this finer-partitioned approach, as soon as a part of the parent's node is completely assembled with its child contribution blocks, the parent node factorization can start. This overlapping brings more performance improvements but also increases the complexity of understanding the application execution in performance analysis tools.

We can divide the application into four different sets of tasks: initialization tasks, deinitialization tasks, communication, and computational tasks. The initialization and deinitialization tasks are created considering the frontal matrices and their blocks, allowing fine control of the memory usage and ensuring data coherence through the StarPU runtime system. The communication tasks represent the assembly of the contributions of a child node to a parent node. As `qr_mumps` partition the fronts in 2D, these tasks can assemble their contributions in the parent front while there are still computations in the child node. The most costly part of the algorithm consists of the computational tasks, which are calls for LAPACK kernels: `geqrt`, `gemqrt`, `tpqrt`, and `tmpqrt`.

The granularity of the matrix partitioning and LAPACK operations is controlled by three user-defined parameters: `mb`, `nb`, and `ib`. The first two define the block and task size, being `mb` the number of rows and `nb` the number of columns. The last parameter,

ib, controls the internal block size granularity of the LAPACK operations, which holds the number of extra floating-point operations needed in 2D algorithms (BUTTARI et al., 2009) while taking advantage of the cache memory hierarchy. Despite this beneficial effect from the LAPACK ib parameter, in qr_mumps, those kernels were slightly modified to help control fill-in level inside the blocks where there are zeroes in the bottom left part of the block, forming the *staircase* structure. We can observe the influence of those parameters in a frontal matrix partitioning and the fill-in level, and the task irregularity sources in Figure 2.5. The figure depicts a frontal matrix with its rows sorted by the leftmost nonzero, emphasizing the staircase structure, leading to many zero elements in its bottom left part. The red dashed lines express the front partitioning following the mb and nb parameters. The left part of the figure shows the effect of two different values for ib: $nb/6$ and $nb/3$. Observe how the fill-in is reduced with smaller ib values, but this comes at the cost of lower efficiency in the BLAS-3 operations. The task cost irregularity arises when the staircase structure traverses a block or when a task computes the bottom or right border of the matrix in cases where the front size is not perfectly divisible by mb or nb. This way, we can have tasks with the same type and size and a different computational cost because of their block content. This irregularity makes performance analysis and prediction a much harder job.

Figure 2.5 – Front partitioning considering mb and nb, ib size effect in fill-in, and sources of task irregularity. Matrix coefficients are represented as light gray squares and fill-in coefficients as dark gray.



Source: The Author

In practical implementations of the multifrontal method like in qr_mumps, we commonly observe the amalgamation of elimination tree nodes to group elimination steps forming bigger and denser frontal matrices, providing better exploiting the Level-3 BLAS

operations efficiency. In contrast with the tree presented in Figure 2.4, where each node represents the elimination of one column, leading to small fronts, in amalgamated trees (also called *assembly tree*), we have one tree node representing the simultaneous elimination of k columns. This operation grouping comes at additional fill-in costs but is controlled with a threshold value for the additional fill-in generated due to the amalgamation process. While grouping tree nodes improve performance, the multifrontal method still naturally leads to many small fronts in the elimination tree leaves. When running over a runtime system, this number of small fronts can lead to considerable overhead in the execution task due to creating too many tasks, reaching parallelism levels higher than the architecture can compute. For this reason, `qr_mumps` implements what is referred to as *logical pruning* technique (BUTTARI, 2013). This technique identifies a layer in the tree where the subtrees rooted at that level contribute to a small amount of the total factorization cost. According to a weight threshold, for example, 1% of the total factorization cost, nodes whose weight contribution is smaller than the threshold are marked as *pruned*. For each subtree marked as pruned, one unique task performs the complete factorization over that subtree, making the operations in that region more performance effective and avoiding creating too many tasks, reducing the runtime overhead.

For controlling the memory consumption, the StarPU runtime system allows stopping the task submission according to the fronts memory usage. The application calculates the memory usage peak according to a sequential traversal of the elimination tree. It defines a maximum usable memory threshold based on that value, for example, using two times the sequential memory peak. This strategy reduces the application memory footprint while maintaining performance as described by previous experiments (LOPEZ, 2015).

These optimization strategies specified in the `qr_mumps` application on top of StarPU work as an architecture-independent solution. This solution provides performance portability through heterogeneous platforms, thanks to the capabilities offered by the StarPU task-based programming model. Despite all optimizations, a detailed performance analysis of an application can reveal where we have high performance, and where the lack of it manifests, evaluating and guiding further improvements or application and runtime configuration parameters.

2.3 Computational Scientific Applications

Scientific computing provides a way to create computational simulations of any size in a controlled and reproducible way. Those simulations represent a cheaper alternative than real experiments both in cost and time and enable to study a phenomenon even when there is no access to the physical equipment that can create it (or because it is infeasible to produce such equipment). This ability to mathematically represent the behavior of a real-world event, translating it to computational algorithms, helps to significantly impel progress in many fields of research. However, this kind of application is a major source of computational demand, requiring a lot of computing power and depending on parallelization approaches to produce accurate solutions in a feasible amount of time. Many of these simulations involve describing the rate of change in time and space of a continuous variable using differential equations. While some of these equations are simple enough to find an analytical solution, in almost all cases, we end up using partial differential equations (PDEs) to express the problems, which are mostly hard to solve exactly. There are many methods for the discretization to approximate the solution of partial differential equations using a numerical model. For example, some of the most popular ones are the finite differences method (FDM), the finite volume method (FVM), and the finite element method (FEM), each one with its own characteristics and cases where they fit best. However, these different discretizations create a system of equations that need to be solved to obtain the approximate solution.

Regarding the type of the system generated, PDEs are a common source of sparse matrices (SAAD, 2003). Therefore, we cannot use basic dense factorization algorithms since the fill-in overhead would significantly degrade performance. Thus this is where high-performance parallel sparse solvers like `qr_mumps` and `PaStiX` come into play, providing an efficient solution for such problems. This Section will discuss some of the discretization approaches and present a real-world case with the RAFEM application (JIANG et al., 2010).

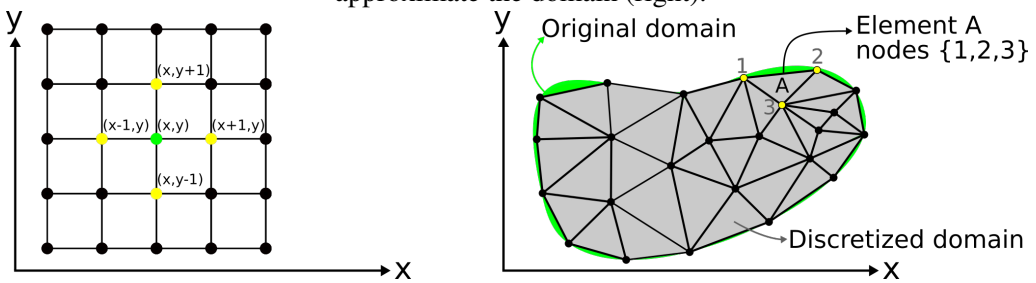
2.3.1 Scientific Applications: Mathematical Modeling, Discretization

Scientific applications start with a determined phenomenon that they want to simulate, typically describing its physical behavior over space and time, for example, in fluid dynamics problems. Many physical laws can be described by PDEs, describing

the changes in a continuous domain using more than one independent variable at a time. The mathematical modeling of those equations depend on the domain we are interested in studying the phenomena, considering, for example, the number of dimensions and physical properties we are interested in. Thus, after we determine the physical problem, its mathematical model should be aligned with our domain representation. The domain discretization consists of representing the problem in a finite way, allowing solving PDEs approximately through numerical methods.

We can discretize a physical domain in a computational environment using finite meshes. The mesh is composed of a finite set of many interconnected points in space, where the application will calculate the equations in those specific regions considering the neighboring points. The meshes provide just an approximation of the real domain shape using a combination of simple geometric forms. The more detailed our mesh (more points it has), the more precision we will be able to estimate the PDEs result. A connected set of points forms an element also called a cell. We have two different mesh types according to their connectivity patterns: *structured* and *unstructured* meshes, as presented by Figure 2.6, along with their particularities. In a structured mesh, all its points have the same number of neighbors, and typically the points are distributed equidistantly, like in a regular 2D grid where all elements have the same shape and size. This regular distribution provides an efficient way to store and locate the neighboring points based on the sum of their indices, allowing representations of such mesh types in structures like matrices. In the unstructured meshes, the connectivity is irregular, where the nodes can have an arbitrary number of neighbors. This irregularity makes the storage less efficient because we need to keep track of all node coordinates and adjacency lists, preventing simple storage strategies like the arrays and matrices used in a structured grid.

Figure 2.6 – Structured 2D mesh (left) and an unstructured mesh using triangular elements to approximate the domain (right).



Source: The Author

Structured meshes provide a memory-efficient way to represent the domain and better convergence properties due to its alignment, which implies less time waiting for

results. However, many problems have a complex geometry where using a structured mesh will lead to rough approximations. Thus we should use an unstructured mesh in those cases, which better approximates those geometries and can also represent specific regions with more precision. Furthermore, *hybrid* meshes combine both strategies and take advantage of what each mesh type offers. We can also use adaptative mesh refinement techniques to improve our solution quality and speed. As the simulation behavior changes over time, adding more points when reaching critical stages where more precision is needed or loosening the representation in areas far from our region of interest by adjusting the mesh structure.

The most extensively used numerical methods to solve PDEs under discretized meshes are the FDM, the FVM, and FEM. Each one of these methods better fits a specific meshing approach and particular problems. They represent the whole problem by combining the simpler equations we have defined for the elements and their neighbors in an equation system. The FDM can be used with a structured grid, providing a very convenient way to solve problems modeled in structured meshes. It is commonly used in cases whenever we have a large number of cells in our discretized domain, where simpler approximations help keep the computational cost under control, like some CFD simulations. The FVM is well suited for the simulation of conservation laws, and where knowing the flux property is relevant. FVM works with structured and unstructured meshes, also being widely used for CFD simulations. The FEM is the most used method for many engineering sciences due to its flexibility to discretize PDEs over complex geometry. It divides the whole domain into finite smaller elements, defining the governing equation for each element, and combining them to form a global system of equations. The flexibility of FEM also allows us to describe multiple material properties, like in cases where we have different element types.

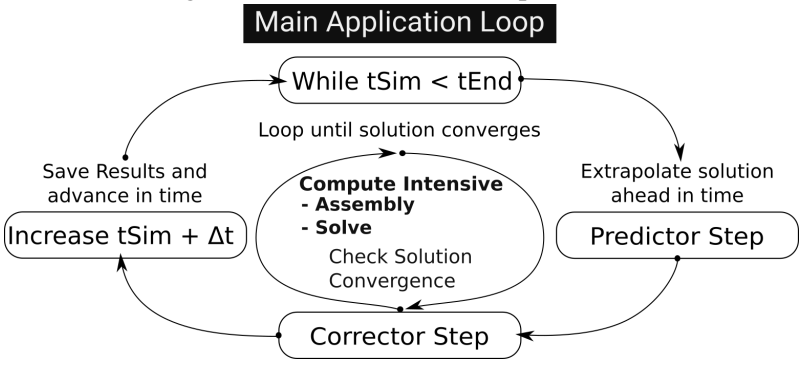
2.3.2 Radiofrequency Ablation Finite Element Method: RAFEM

The RAFEM application (JIANG et al., 2010) is a scientific application to simulate the Radiofrequency Ablation (RFA) procedure. The RFA is a minimally invasive treatment for some cases of hepatic cancer. This procedure consists of using a special electrode connected to an alternated current generator. The tip of the electrode is positioned in the center of the tumor in the patient's liver to eliminate the tumor cancer cells. It generates heat in the area around the electrode through the Joule effect, killing the can-

cerous cells because of the high temperature. The efficiency of the medical procedure depends on the total heated area and the reached temperature. If it does not eliminate all cancer cells, tumor recidivism can occur. The problem is that when the RFA starts, any medical imaging techniques like ultrasound, computational tomography, and magnetic resonance imaging are unable to provide precise visualization and measurements of what is happening inside the liver, like the area affected by the heat. This way, the simulation helps specialists adjust the RFA parameters for each patient, maximizing the success rate and avoiding resubmitting the patient to a new RFA procedure.

RAFEM uses the FEM to model the RFA procedure in a 3D tetrahedral mesh, calculating the heat and voltage distribution over time. Its original version is based on the frontal method technique for solving the resulting equation system. While this method is useful for reducing memory footprint, it is not focused on efficiency, spending too much computational time to produce the results. Recently, works have dedicated efforts to accelerate the two most costly phases in the application, which is the assembly of the equation system and obtaining its solution (SCHEPKE; MILETTO, 2020), proving the usefulness of sparse solvers and other parallel solvers in such applications. Figure 2.7 presents the main loop structure of the RAFEM application. The application principal execution flow consists of two nested loops. The outer loop controls the steps advance in time, and the inner loop controls the numerical convergence of the obtained solution. The loop is based on the predictor-corrector method, where the application extrapolates the solution further in time and corrects it until a certain precision threshold. These loops make many repetitions of the costly phases of assembling and solving the system of equations, where any improvements in their performance will propagate through these loops. We consider using this application, integrating `qr_mumps` to accelerate it, and to analyze, study, and tune the performance of `qr_mumps` in a real application scenario.

Figure 2.7 – RAFEM main loop structure.



Source: The Author

2.4 Collecting Performance Data

Performance analysis tools and techniques depend on performance data collected from an application execution. There are many methods to collect this kind of information, which depends on the level of detail we need and the objective of our analysis. We can obtain different types of data, which also implies different overheads that disturb application execution. This disturbance is called the probe effect or also intrusion.

2.4.1 Performance Data Acquiring Methods and Data Types

Instrumentation is the act of modifying a program, either actively or automatically, for an alternative purpose (ISAACS et al., 2014b). In this case, we are concerned about techniques of modifying a program to obtain performance data for its post-mortem or offline analysis. The data acquiring methods can be guided by time or by events (REED, 1994). Our analysis objective defines which type of data and method we should be using.

Profiling is a time-based instrumentation method that periodically pauses the execution in an interval defined by the user, and either it only saves the content of the instruction pointer to know which function is executing or can save the whole call stack. The profiling technique produces **sampled data** , counting how many times its measurements occurred in specific program regions. This sampled data is analyzed after the application execution to estimate the percentage of the application execution time was spent in each of its sampled regions. The precision of this estimation depends on how frequently the application collects information. The more it collects data, the better we have estimations at the cost of more overhead caused in the application execution due to the probe effect. However, as the information produced by profiling is very simple, losing temporal information, its overhead is very low, and it uses a small amount of memory. Profiling is an excellent technique to provide a straightforward and quick way to detect critical hot spots and bottlenecks in the code since it commonly does not involve any other step like manually modifying the application code.

Interception is an event-based instrumentation technique consisting of intercepting the functions calls present in a source code. The function calls are wrapped by an interception library function that executes performance measurement code, collecting performance data for that specific function. These interceptions can either be done automatically according to the present function calls by simply compiling the original application

with a performance measurement library or specified manually in the code. Compared to profiling, the interception technique can aggregate the data to provide the exact profile of the application, recording the precise time spent in each code region and how many times a specific event occurred. Furthermore, besides collecting exact **timing data**, it can register data from specific parameters passed to a given function, which provides semantic context to the analysts. Also, it allows the collection of **hardware counters data** for a determined event. The interception overhead depends on the number of events and how much information we collect for each event but is much more significant than in a profiling technique.

Tracing is possible through the interception method to create **tracing data**. A trace is made of all the non-aggregated events that occurred during the application execution, with their entry and exit times. The application traces allows the analysts to reconstruct the full application behavior along time and among the computational resources. This detailed time-related data allows one to understand in detail what happened in the execution, which can help answer questions about performance. Ideally, to fully represent the application behavior, we should record all types of events, but this is impracticable because of the considerable intrusion generated (SCHNORR, 2014). Thus, we can carefully select the essential events for our analysis to keep intrusion levels under control.

There are also different types of analysis, like online analysis, where data is collected and analyzed concurrently with the application execution, which avoids the need to save huge trace files, but limits the scalability in large-scale applications and causes more intrusion. Hence, the post-mortem analysis is the most widely used analysis type.

2.4.2 Intrusion or Probe Effect

When an application runs without taking any other measurements, it expresses its natural behavior. The methods to collect data presented earlier disturb this natural behavior causing the probe effect or intrusion, which is mostly unavoidable for performance analysis studies (SCHNORR, 2014). This effect represents the time spent using the platform resources to record and manipulate the performance data, which will possibly slow down the application by some factor.

We can characterize the intrusion level by comparing the application execution time when it is not under observation to when it is. If the impact is too high, the behavior we will analyze will probably be different from the real one. This way, we should

carefully consider our data collection strategy to avoid changing the application behavior while studying it. Besides the impact on time, memory usage can be a problem too. If the collected data size is considerable, this can represent a limiting factor in systems with reduced available resources. Thus, the data collecting system's choice and method should consider the level of intrusion that it causes. Otherwise, it can harm the analysis process.

2.4.3 Performance Data Over StarPU

The StarPU runtime system supports two ways of providing performance data for any application built on top of it: online statistics and execution traces. The online statistics consist of very simple measurements and should not degrade application performance (THIBAULT, 2018). It provides profiling data like how much time was spent in a given type of task, scheduling, idling, or waiting for data transfers. This information gives us a big picture of the application execution, which helps to understand at a high level what happened in the execution but is unable to tell where or why there were performance deviates.

Conversely, the StarPU tracing system can provide all that information, helping performance analysts to have an in-depth view of the application performance. Each task entry and exit timing is recorded, along with other data related to task parameters. StarPU uses the FxT (DANJEAN; NAMYST; WACRENIER, 2005) library to produce detailed traces of the application, saving all task-related information through its execution, relating it with the architecture, and locating events in time. The FxT tracing system was designed to collect data efficiently, minimize impact in the execution, and proved to produce low intrusion overhead (THIBAULT, 2018).

The tracing can be quickly enabled and disable using environment variables, and it produces one FxT tracing file per computational node. In this work, we rely on those FxT files, which are converted through the StarVZ workflow to the Paje/PajeNG (SCHNORR, 2012) general trace file format, processed and used as input for its many performance visualization panels. Also, a combination of techniques is considered for future works in the StarPU tracing system (THIBAULT, 2018), like a sampled tracing technique, which would help in the scalability for very long application executions in a large scale computing environment.

3 RELATED WORK: PERFORMANCE ANALYSIS

The performance analysis is an essential step to understand and improve any application. As HPC platforms become increasingly complex, exploring different heterogeneity levels with GPUs, CPUs, and FPGAs, including NUMA memory nodes, shared and distributed memory, and various communication types, analyzing applications for such platforms became more complex as well. Moreover, considering that the execution of task-based programs is stochastic, without regular computation and communication phases. It turns out they are a more challenging scenario to analyze than data-parallel applications, making traditional performance analysis strategies unfit because of lack of DAG and runtime information. Task-based applications rely on dynamic runtime decisions, which sometimes consider many task aspects such as priorities and cost estimations for the duration and even energy consumption. The representation of the application as a DAG simplifies the programming but loses track of application-specific structures by diluting them in the individual tasks and dependencies. This way, there is an interest in extending classical visualization techniques to help in task-based performance analysis, easing application and runtime developers process to understand, debug, and analyze applications. Another way to improve task-based performance analysis is to recover this application-related data, helping to relate performance through visualization in a meaningful way to application developers.

This chapter presents a set of general and task-based performance analysis tools and techniques in Section 3.1. In Section 3.2, we present works concerned with performance modeling and prediction techniques commonly used for problem partitioning and scheduling and can also help in application comprehension, performance analysis, and anomalous behavior detection.

3.1 Performance Analysis Tools and Techniques

Performance visualization is a valuable technique to earn a comprehensive understanding of HPC applications and the factors that affect their performance. This comprehension is fundamental to improve applications on many levels, helping to detect and enhance different inefficiency sources. Improving performance is the final objective of performance analysis and visualization. Therefore, the means to do so can follow different paths, focusing on many particular points of an application. According to (ISAACS

et al., 2014b), we can divide the different sub-goals of visual performance analysis into three distinct categories: global comprehension, problem detection, and diagnosis and attribution.

Global comprehension provides an overall big picture of the application, allowing users to perceive the natural application behavior, observing its phases like communication, computation, data movement, dependencies, and resource utilization. Such representation can depict common patterns of these different aspects and how they relate to each other, strengthening the understanding of an application execution performance and guiding problem detection. The **problem detection** goal is to identify performance bottlenecks, load imbalance, low resource parallelism usage, and anomalous behavior. **Diagnosis and attribution** follows the latter goal. After the detection, one must try to understand the cause of the problem. The problem origin might come from the computational environment, application parameters, bad scheduling, and many other factors. Pinpointing the cause of the detected problem can be quite challenging and commonly needs an extensive investigation, demanding the use of specific visualization and performance analysis methods.

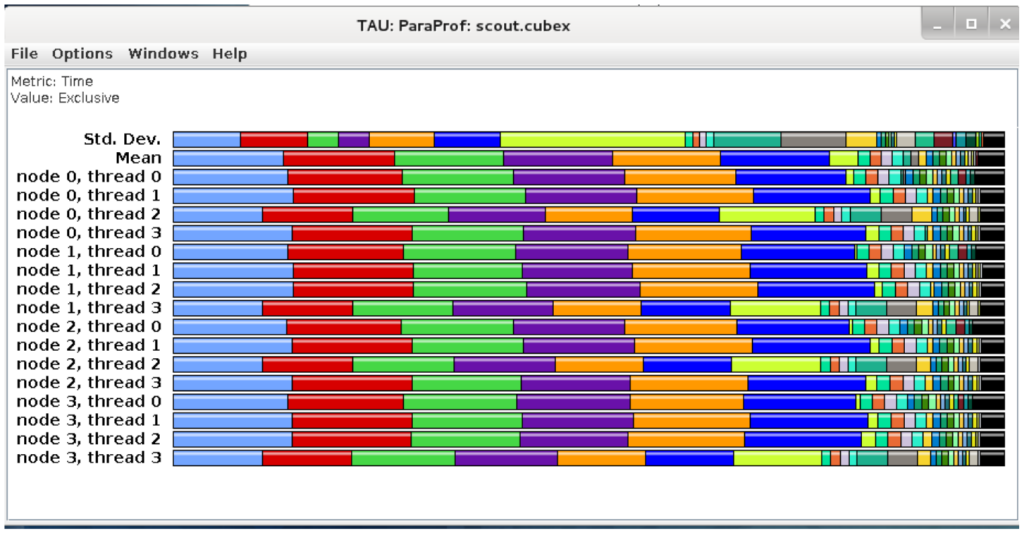
3.1.1 General HPC Performance Analysis Tools

To face the complexity of parallel programs performance analysis, the scientific community has been developing a vast number of performance analysis tools to enhance analysts' abilities for application performance comprehension, problem detection, and diagnosis. Such tools rely on profiling and tracing data to collect and derive performance metrics. Commonly, the amount of data in large-scale parallel performance analysis can be huge, making the analysis hard. Thus, it is often essential to summarize performance data (BOUDEC, 2010), transforming raw data into knowledge. The common objectives of general performance analysis tools are to highlight many types of performance problems, pinpoint their location in the program execution both in time and source code, and the system process that was running. Such applications fully support well-established parallel programming techniques like MPI/OpenMP, providing exploratory tools that help users go through application events, identify computation and communication patterns, bottlenecks, and other inefficiency sources. We present some of these classical tools in the following paragraphs. We selected popular tools that cover the analysis of traditional paradigms like the combinations of MPI, OpenMP, and CUDA, highlighting their strate-

gies and functionalities for performance analysis.

TAU (Tuning and Analysis Utilities) (SHENDE; MALONY, 2006) is a complete framework and toolset for the performance instrumentation, analysis, and visualization, of large-scale parallel programs, conceived to be robust, flexible, and portable. Its instrumentation and data conversion features provide performance data that fits for trace analysis and trace visualization. Despite this flexibility, TAU has its own trace explorer (PerfExplorer) and visualizer (ParaProf). The PerfExplorer (HUCK; MALONY, 2005) is a framework for parallel performance data mining, including features like clustering and dimensionality reduction to reduce large-scale data complexity, as well as correlation analysis and metric visualization panels. These features support single and multiple experiments analysis, allowing users to compare the results of numerous experiments quickly. ParaProf (BELL; MALONY; SHENDE, 2003) is the TAU's performance visualization tool. It can present classical views like timeline views, histograms and bar charts, call graphs, along with performance data derived metrics. It also supports OpenGL-based 3D visualizations to compare multiple operations and metrics at the same time. The 3.1 presents a simple timeline view in a thread-centric way and its mean and standard deviation metrics for the different event's duration. The tool allows to explores these thread regions in detail, displaying their performance statistics.

Figure 3.1 – ParaProf's timeline view, displaying threads time information.

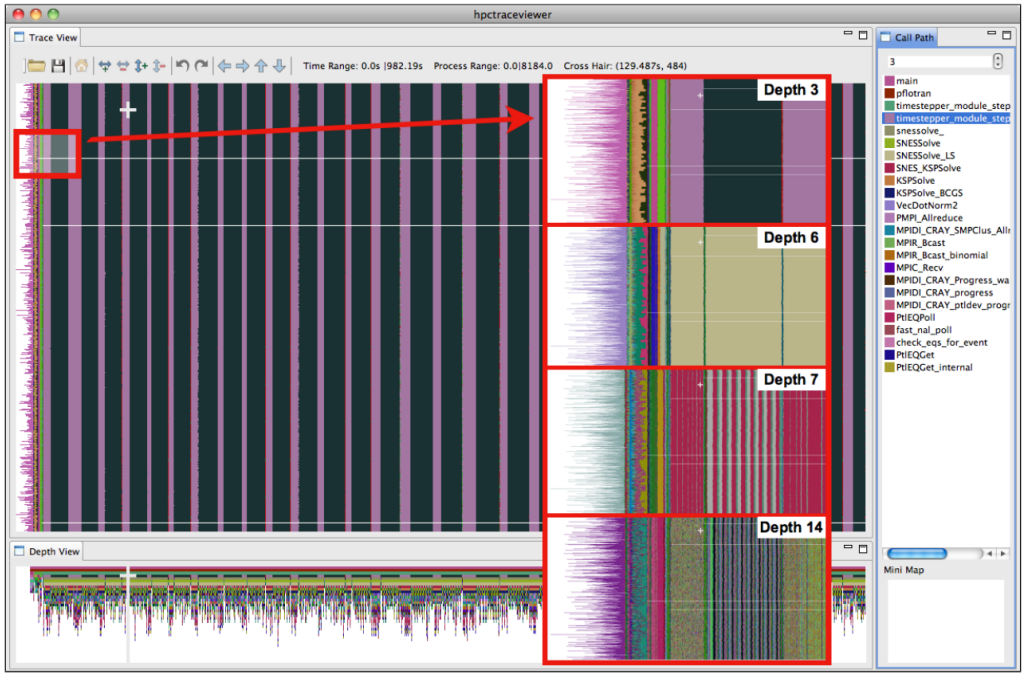


Source: (BELL; MALONY; SHENDE, 2003)

HPCToolkit (ADHIANTO et al., 2010) is a set of tools for scalable and efficient measurement and performance analysis of HPC applications. It provides a statistical sampling of region times and hardware counters, which causes a low overhead in the application (1-5%) and can trace applications to record space-time information during the

execution for a more detailed post-mortem analysis. The toolkit counts with complementary tools to analyze the profiling and traces. For example, the *hpcviewer* (ADHIANTO; MELLOR-CRUMMEY; TALLENT, 2010) is a java-based tool that allows users to explore, combine, and derive metrics from performance data, associating and organizing performance measurements by their call-path in a hierarchical tree-based structure and relating them to the source code. The *traceviewer* (TALLENT et al., 2011) provides an interactive performance trace data visualization, presenting data as a classical Gantt chart. The difference is that the user can control the Gantt chart's colors by selecting different call paths visible depth levels as presented by Figure 3.2, which can reveal execution patterns, helping to understand and pinpoint sources of inefficiency.

Figure 3.2 – The traceviewer panel with multiple examples of call-path depth selection.

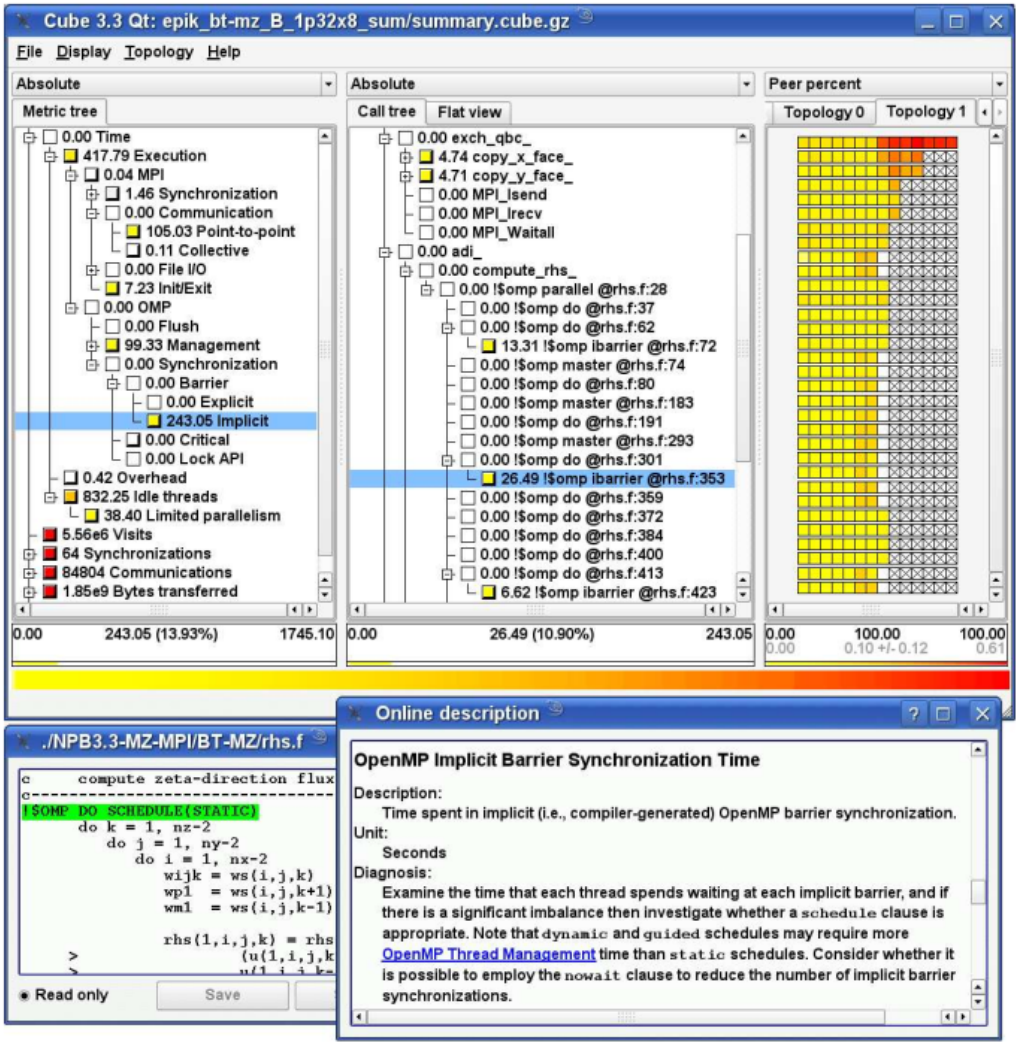


Source: (TALLENT et al., 2011)

Scalasca (GEIMER et al., 2010) is a collection of tools specially designed for large-scale parallel applications performance analysis, focusing on MPI and OpenMP based programs or hybrid approaches that combine both. It has a measurement library that uses the PMPI interface to capture the MPI events and the POMP profiling interface for tracing OpenMP. It can use the Scalasca EPIK user instrumentation API macros and Score-P manual instrumentation regions to instrument user-defined regions. Furthermore, users can use TAU source-code instrumentor to insert Scalasca measurement API calls straightforwardly. The tool automatically analyzes the trace to find performance bottlenecks, focusing on problem detection like MPI late sends and receives, wrong order

messages, and measuring MPI waiting time. It also provides these automatic analyses for OpenMP-based code, analyzing aspects like task team creation and starting overhead. Then, users can further investigate this enriched performance report data using tools like the CUBE4 trace explorer, which presents the performance metrics, the call-path, and the system resources. Figure 3.3 shows an example of the Scalasca/CUBE trace explorer use, relating the performance metrics in the left part, to the program code in the middle to the system architecture in the right.

Figure 3.3 – CUBE trace explorer using Scalasca performance report.

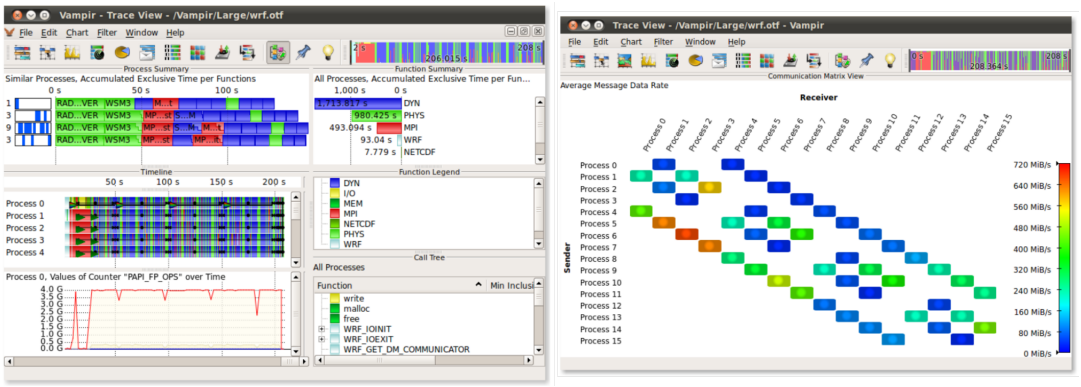


Source: (GEIMER et al., 2010)

Vampir (KNÜPFER et al., 2008) is a proprietary toolset to analyze parallel applications performance and message passing characteristics. It uses program trace events collected in OTF2 (Open Trace Format 2), provided by many tracing tools and libraries like TAU, Score-P, and VampirTrace, allowing instrumenting the application either automatically or manually by user-defined regions. The Vampir toolset can use the Vam-

pirServer system to access performance data in remote computation nodes and present it on a client-side. It shows several performance visualization panels for statistical data, an interactive timeline view, including message passing information, allowing the user to navigate and filter over the visual data, also relating it to other performance data like hardware counters. Figure 3.4 presents a group of utility charts from Vampir, with many different performance metrics, summarized data, and timeline information. It also shows another type of visualization relating total data transferred by MPI communications between processes. Users can choose the color to represent other metrics, like the total time spent in communications or the number of messages.

Figure 3.4 – Vampir trace view and communication matrix.

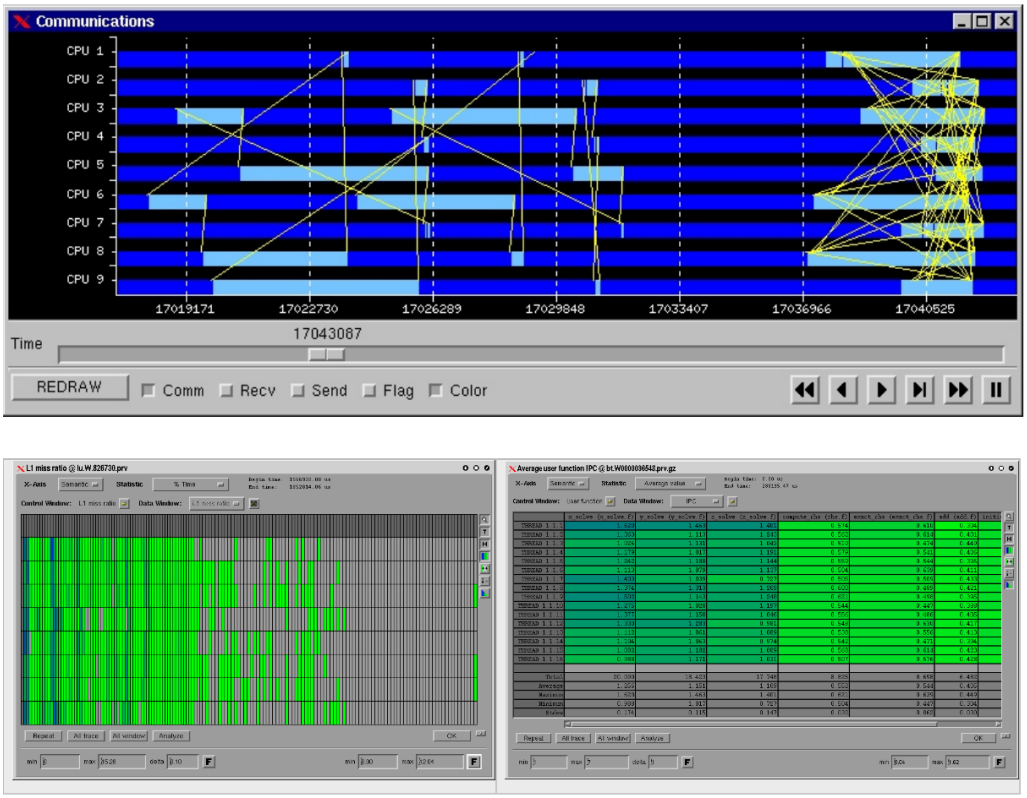


Source: (Vampir, 2019)

Paraver (PARAllel Visualization and Events Representation) (PILLET et al., 1995) focus is on being a flexible performance data browser, where the user can extend the tool's metrics by using a set of functions and operations provided by the application without the need of performing changes in the trace visualizer. It uses data from the Extrae measurement system (CENTER, 2015), which supports MPI, OpenMP, pthreads, OmpSs, and CUDA. Such flexibility provided by this tool can be noticed by recent studies using Paraver to study energy-efficiency in HPC applications (MANTOVANI; CALORE, 2018). In the visualization part, Paraver counts with a small set of views, having two main visualization panels presenting qualitatively different information types: the timeline and the statistics display. The first one represents the classical timeline view, showing the application behavior along time, with communication and computation patterns. The latter provides numerical analysis about any desired metric in a user-defined specific time window. Figure 3.5 presents these two types of views, with a timeline display on top and two different statistic displays.

ViTE (Visual Trace Explorer) (COULOMB et al., 2012) is part of a tool-chain for performance analysis, responsible for interactive, multi-format, and fast trace data vi-

Figure 3.5 – Paraver timeline and statistics displays.

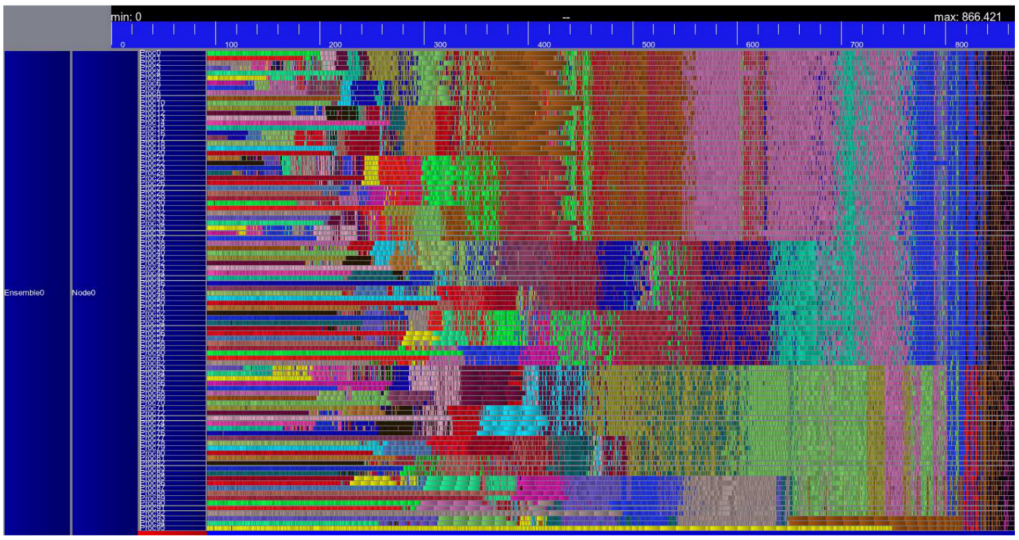


Source: (PILLET et al., 1995)

sualization. It reads Pajé (KERGOMMEAUX; STEIN, 2000) trace files and can use the EZTrace tool to collect performance data, which have pre-defined plug-ins to trace MPI, OpenMP, pthreads, and user-defined regions. EZTrace records the application events using the FxT library (DANJEAN; NAMYST; WACRENIER, 2005). This trace is then converted by the GTG (Generic Trace Generator) to the Pajé or OTF2 trace formats. ViTE can read both formats and quickly generate full trace visualization without time aggregation using OpenGL, representing the resource states and communications interactively, giving the user a fast and elementary overview of the program behavior. ViTE can easily represent visualizations of vast amounts of raw trace data, as depicted by Figure 3.6. The figure represents the hardware hierarchy in the left part, and the events are drawn through time, identified by colors.

Nowadays, plenty of tools can help with the complex task of analyzing HPC applications' performance. Other tools with similar functionalities include Pajé (KERGOMMEAUX; STEIN, 2000), JumpShoot (ZAKI et al., 1999), and Ravel (ISAACS et al., 2014a). Although many of them have overlapping functionalities, they tackle large-scale parallel performance data analysis very differently, even considering only post-mortem analysis tools. We also have many differences in the instrumentation and tracing part.

Figure 3.6 – ViTE trace view, displaying millions of events.



Source: (COULOMB et al., 2012)

However, the efforts to standardize tracing information for performance analysis tools like OTF2 and Paje/PajeNG (SCHNORR, 2012) traces are well accepted since many tools can read or convert those formats to one that suits for its own use. The tools have different focuses, like having simple but useful views like Paraver and ViTE, having a comprehensive set of visualization displays like in Vampir, and displaying specific information like in HPCToolkit. Some of them present automatic analysis to ease the analysts' work significantly, like Scalasca. Those tools can also be used in a complementary way, combining their non-overlapping functionalities to understand a program's performance fully.

As tools should present performance data in meaningful ways for the user, aligning application structures and parallel language constructs (SHENDE, 1999), which works well for MPI, OpenMP, CUDA, and other data-parallel approaches in the presented tools. We have a lack of task-based DAG-specific information in those tools. This way, we need specific task-based tools to investigate task-based applications' performance properly.

3.1.2 Task-Based Performance Analysis Tools

As the task-based programming paradigm has some specialized structures and entities like the application DAG and the scheduler, to enhance task-based application debugging, testing, and performance comprehension, we need to consider such information. The stochasticity of the scheduler decisions and tasks and resources heterogeneity makes the analysis hard and challenging to identify inefficiencies. Thus, specific tools

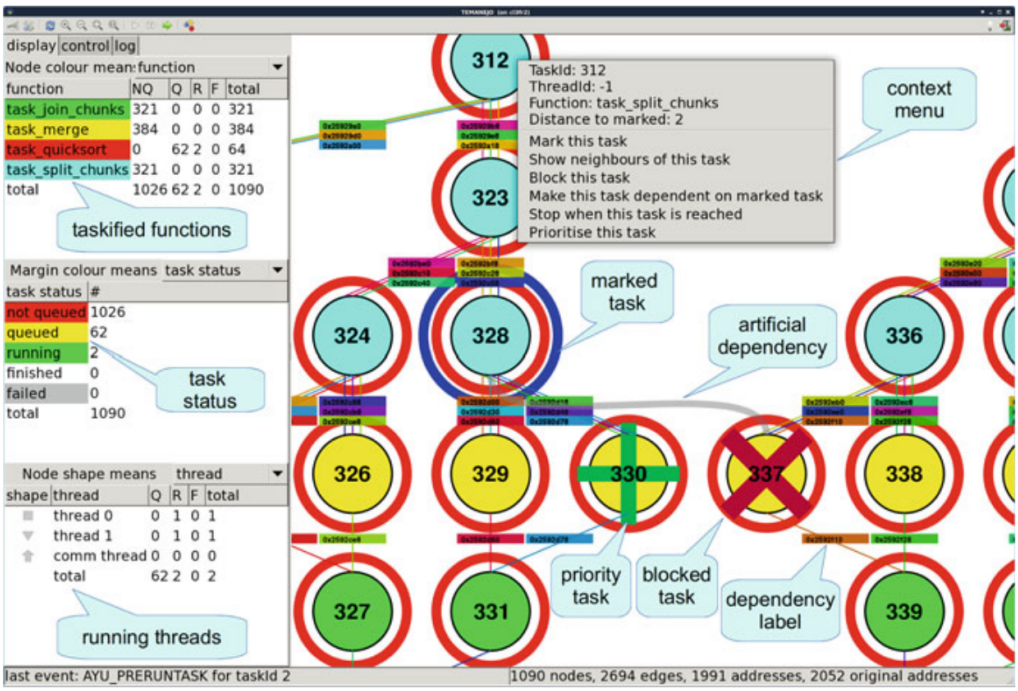
and methods for a more in-depth task-based analysis look for task dependencies, different task types and their durations, DAG-based metrics like the critical-path, trying to relate this to poor application performance in both runtime or application-level.

TaskInsight (CEBALLOS et al., 2017) is a technique that characterizes the memory behavior of task-based applications considering data reuse and tries to relate application performance with the scheduler decisions that favor or not the cache reuse between tasks. Its methodology relies on using task-wise hardware counters data such as the number of cycles, instructions, cache miss ratio, and it uses the Pin tool (LUK et al., 2005) to obtain a sample of the tasks' memory accesses. They try to evaluate scheduler decisions considering four different memory access categories: new data, last reuse, second last reuse, and older reuse, being the latest when a task access a data region referenced later than the two tasks before. As an experimental test case, the authors used a dense Cholesky factorization application on top of the OmpSs runtime (DURAN et al., 2011). They compare a naive approach that schedules tasks by their creation order according to a breadth-first search against a smart scheduler that uses a heuristic that prioritizes child tasks over other tasks in the breadth-first order. This way, the authors could verify the scheduling decisions, quantifying their impact on tasks' memory behavior through a higher miss ratio and cycles per instructions. They report that besides the temporal locality aspect that influences the miss ratio, co-running tasks can degrade their performance because of how they interact with the last-level cache.

Temanejo (BRINKMANN; GRACIA; NIETHAMMER, 2013) is a toolset for debugging purposes with an online debugger for parallel task-based applications. It has a graphical user interface that displays the running application DAG and its tasks status in the runtime system. The tool allows the user to analyze and interact with the DAG tasks by prioritizing, blocking, setting breakpoints, and even creating artificial dependencies between them. It also allows calling the `gdb` to analyze any task further. It relies on the Ayudame library, part of the toolset, to instrumentate the application and communicate with the runtime during its execution. The versatility of this library enables many runtimes to use Temanejo (DURAN et al., 2011; AUGONNET et al., 2011; BOSILCA et al., 2012b; OpenMP, 2018). Figure 3.7 presents an annotated overview of the Temanejo visual debugger.

DAGViz (HUYNH et al., 2015) is a visualization tool focused on parallel task-based programs that provide a task-centric visualization of the applications by displaying the program tasks as a DAG plus a classical timeline view of the application with the

Figure 3.7 – Temanejo's visual debugger overview with annotations.

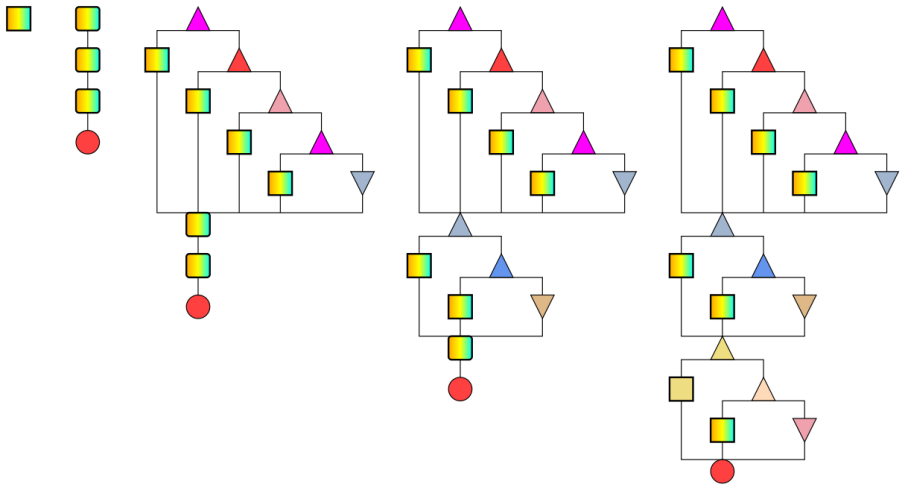


Source: (BRINKMANN; GRACIA; NIETHAMMER, 2013)

available parallelism from the DAG on its top. It allows the user to explore the DAG by expanding/contracting tasks and their dependent tasks, allowing them to focus on a given part of the DAG. The DAG representation helps compare different schedulers, as it has a consistent structure between other schedulers' executions. The DAG Recorder extracts the DAG information for tracing, and it is supported by OpenMP, Cilk Plus, Intel TBB, Qthreads, and MassiveThreads. DAGViz breaks the task behavior using three primitives: CreateTask when the current task spawns a new child task, WaitTask marking a task that waits for all others inside a section, and MakeSection, which defines a region of tasks creation until a WaitTask. Figure 3.8} presents a DAG representation with DAGviz with different contractions. The upper triangles represent the CreateTask primitive, the downward triangles are the WaitTask, and the squares depict the MakeSection primitive. The colors identify the resource that executed the task, and sections with multiple colors mean that many workers computed the DAG area. Also, the red circle marks the application ending point. Although this DAG visualization provides a very detailed view of the application, it can quickly become overwhelming depending on the size of the drawn DAG area.

Grain Graphs (MUDDUKRISHNA et al., 2016) is a performance visualization analysis tool focused on OpenMP programs with tasks and parallel-for loops. The tool constructs the visualization of grains, which are computations made by tasks or parallel-

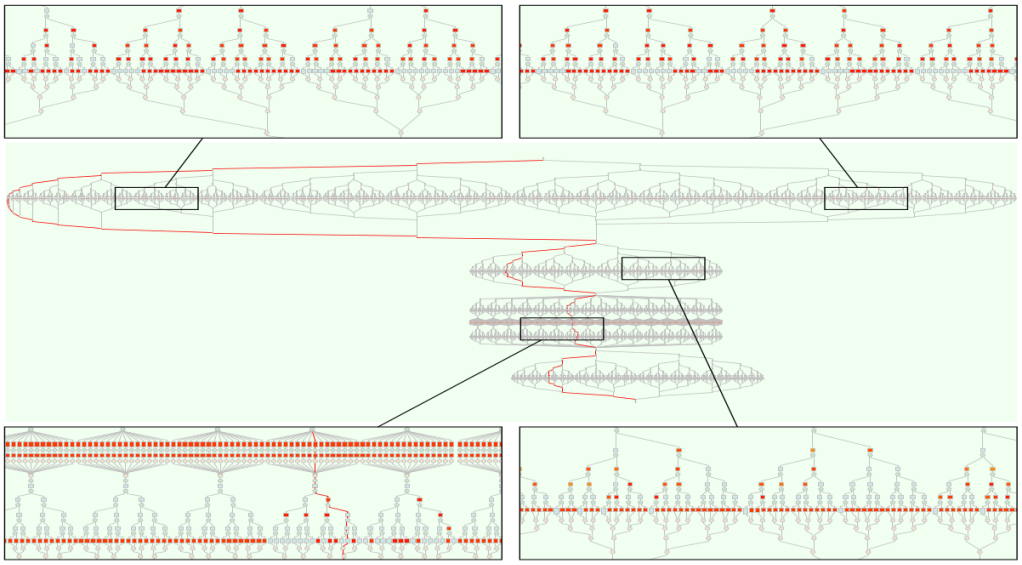
Figure 3.8 – Four different expansions of a DAG represented with DAGViz.



Source: (HUYNH et al., 2015)

for loops. It can create derived metrics considering the graph structure considering aspects like memory system behavior like cache miss ratio and memory hierarchy utilization, load imbalance, task creation and synchronization overhead, and parallelism level. The tool then uses those metrics to highlight grains that present performance problems in the DAG according to one of the derived metrics. Figure 3.9 shows the analysis for the fast Fourier transform application of the BOTS benchmark, considering memory hierarchy utilization calculated as a ratio of the cycles spent computing and cycles waiting for data. The tool uses a color gradient to depict low memory hierarchy utilization by the grains from red (low) to yellow (high) while fades grains without this problem.

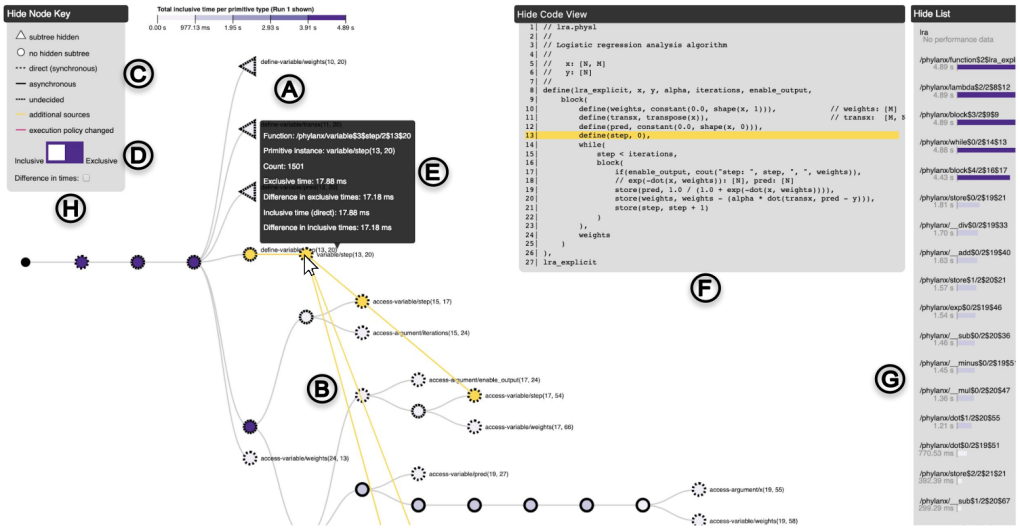
Figure 3.9 – Representation of the memory hierarchy utilization for the fast Fourier transform of BOTS benchmark with Grain Graphs.



Source: (MUDDUKRISHNA et al., 2016)

Atria (WILLIAMS; BIGELOW; ISAACS, 2019) is a task-based performance analysis tool for the Phylanx/HPX runtime (TOHID et al., 2018; KAISER et al., 2014). It does not use trace data and thus does not provide any timeline view. The tool's primary goals are to present an overview of the execution using a DAG structure, relating the DAG execution with the code, and associate performance data (hardware counters and timing) with the tasks and paths, helping to understand performance and scheduler decisions. Atria represents the DAG as an expression tree where the nodes are the operations and children are operands, which makes sense in the Phylanx programming point-of-view. The tool can compare two runs by overlapping their expression trees, and adding a color encoding to distinguish task duration differences, easing the comparison between different scheduling policies. Figure 3.10 shows an example of the tool's views.

Figure 3.10 – Atria's functionalities overview.

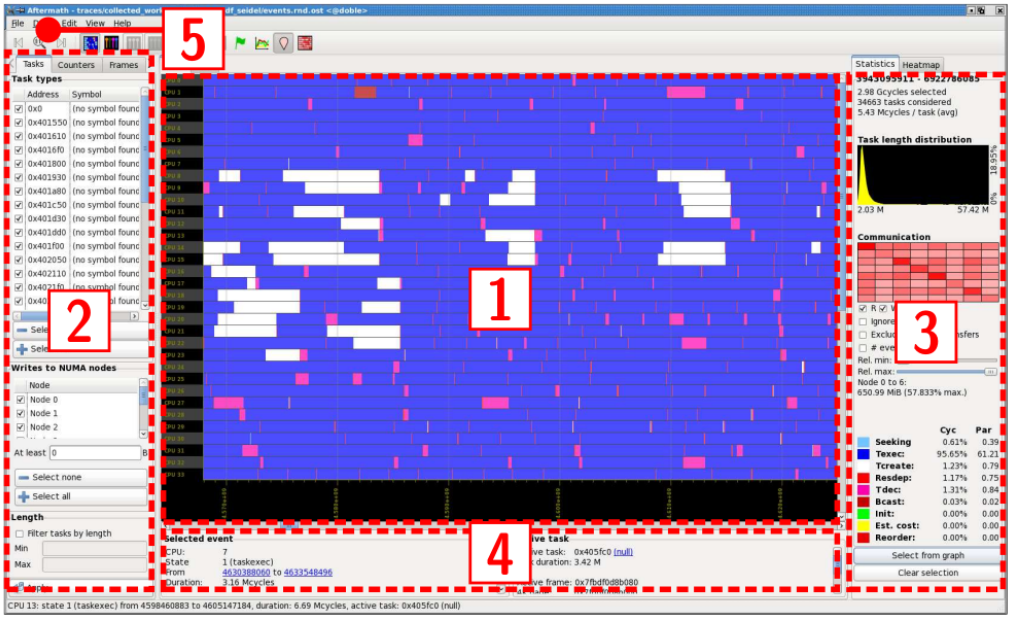


Source: (WILLIAMS; BIGELOW; ISAACS, 2019)

Aftermath (DREBES et al., 2014b) is a graphical tool for task-centric performance analysis for the OpenStream runtime, an OpenMP data-flow, stream programming extension. Currently, it only works with its own native trace format, optimized for OpenStream applications. The tool collects hardware counters performance data and runtime task managing information. It uses an interactive timeline panel to display task execution and task managing information like task creation and scheduling. The user can use a set of filters and summary statistics to explore application performance, following metrics evolution along time or inside individual tasks, and even create annotations. Despite providing individual task analysis and task management information, the tool lacks DAG dependencies information. Enhancements on Aftermath (DREBES et al., 2014a) provide support for automatic detection of performance bottlenecks using a threshold-based anal-

ysis considering the execution time and the number of processors to determine if there is enough parallelism or too much overhead in task creation or scheduling. Also, it provides a linear regression-based analysis to automatically determine which hardware counters the user should use in the tasks' performance analysis. Figure 3.11 shows an overview of the Aftermath timeline view (1), along with its filters(2), statistics (3), events(4), and menu to create derived metrics.

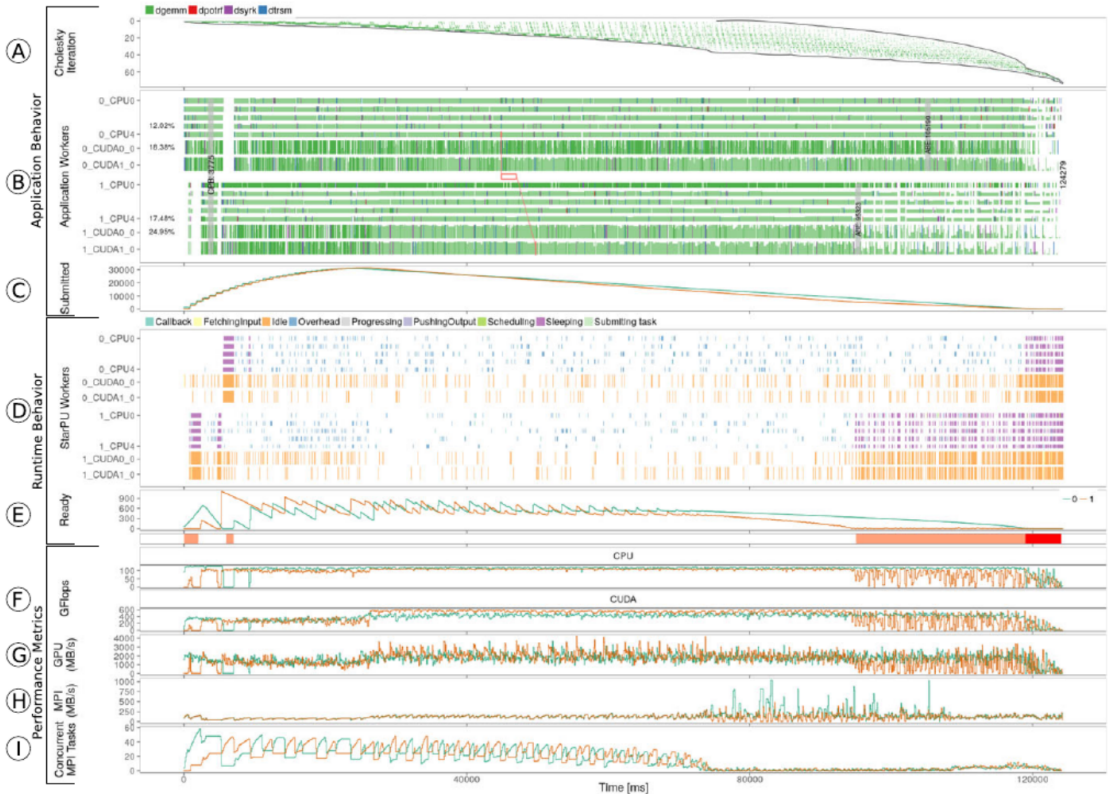
Figure 3.11 – Overview of the Aftermath graphical visualization tool.



Source: (DREBES et al., 2014b)

StarVZ (PINTO et al., 2018) is an `R` package available on CRAN, for the performance analysis of task-based applications based on the StarPU runtime system. It relies on the built-in tracing of StarPU that generates FxT traces. The tool provides a comprehensive set of visualization panels for application behavior, runtime behavior, and performance metrics visualization over the application execution time. It enriches the classical Gantt chart timeline view with load-based metrics like the area bound estimation (a lower bound for the execution time given the load) and DAG-based metrics like the critical-path bound. It highlights abnormal tasks whose duration is higher than the mean execution values given a task type and resource type and can represent dependency chains between tasks. It also implements algorithmic specific visualizations like the Cholesky iteration plot, which illustrates how the scheduler traverses the DAG considering algorithmic information. Recently, the StarVZ tool was enhanced to support memory behavior analysis at runtime level (NESI et al., 2019). Figure 3.12 presents a set of visualization panels from StarVZ.

Figure 3.12 – Overview of the StarVZ panels.



Source: (PINTO et al., 2018)

As we have seen, the task-based specific performance analysis tools also have overlapping functionalities with general-purpose ones. However, they extend the analysis by including DAG and runtime information, approximating the performance data to the real application scenario, helping in performance comprehension. The DAG information is essential to compare executions because of the non-deterministic nature of dynamic scheduling decisions. In this sense, understanding the causes of poor performance can become a very complicated job. Thus, performance analysis tools should aid the performance investigation allowing the user to explore performance data and provide automatic analysis mechanisms.

3.2 Performance Modeling and Prediction

Another facet of performance analysis that helps in many aspects throughout its process is performance modeling (JAIN, 2008). Performance models provide cost estimation for application's specific regions. This cost can represent many metrics related to performance, like duration, the number of floating-point operations, energy consumption, the volume of data transfers in communications, and the size of data used in cache. Mod-

eling generally focuses on describing performance in low-level parts of the application, like individual tasks. However, it can also combine such fine-grained modeling to predict a whole computational system's performance or one of its components. The crucial advantage of modeling performance is knowing beforehand the expected performance for a given application component. We can then use a model (1) to build faithful simulations of applications reducing the experimental cost to study many different factors, (2) to develop smart scheduling algorithms that consider the expected performance and data transfer time in heterogeneous platforms. And lastly, (3) to improve post-mortem performance analysis by automatically detecting where performance deviates from its expected value.

3.2.1 Task Performance Modeling Types and Use Cases

High-performance applications frequently use some cost metric obtained from modeling to help perform duties like problem partitioning and achieving effective load balancing. For example, the `qr_mumps` application uses the tasks' theoretical floating-point operation cost to prune the elimination tree nodes. `PaStiX` (HÉNON; RAMET; ROMAN, 2002) uses communication and BLAS models to dynamically decide the elimination tree nodes partitioning in either 1D (column-blocks) or 2D (tiled factorization) and define a static scheduling when using its native internal scheduler. There are many techniques and types of performance modeling for computational tasks, from developing analytical models, which go deep in the details of how a computational kernel works, to using statistical models based on previous executions of applications and its sampled or measured performance.

When we have scenarios like in `qr_mumps` and `PaStiX`, where we need to know the values online, a common approach is to develop analytical models for the application's basic building blocks, which are the computational tasks. Such application tasks are built upon the widely used computation kernels of HPC applications BLAS/LAPACK routines. Thus, extensive research was done to properly model its routines costs analytically (GEIJN; QUINTANA-ORTÍ, 2008; PEISE; BIENTINESI, 2012). Exploring different aspects of the performance like considering communication (DACKLAND; KÅGSTRÖM, 1996), memory access patterns (IAKYMCHUK; BIENTINESI, 2012), and even providing and using models to auto-tune linear algebra routines through the optimization of its parameters (CUENCA; GIMÉNEZ; GONZÁLEZ, 2004). There is also a concern on

modeling such routines and application performance considering sparse matrix operation (GRIGORI; LI, 2007; CICOTTI; LI; BADEN, 2010). Moreover, studies still look at how to model the task cost mathematically (STANISIC et al., 2015). This latter work focuses on modeling the block-column version of `qr_mumps` using a set of task parameters and machine-dependent memory hierarchy coefficients to correctly model its irregular task behavior.

A model conception can depend on its purpose. For example, StarPU uses performance models to improve scheduling policies. It builds the model for tasks according to their type, the type of computational resource it was executed, and the underlying architecture. Then, StarPU uses a calibration run to sample performance data to create the models. The gathered information can be as simple as the duration of the task in a given computational resource, although that works only for regular tasks. Combined with communication bandwidth models in the target architecture, these measurements help schedulers decide how to efficiently schedule tasks over heterogeneous platforms. In the simulation aspect, the SimGrid (CASANOVA et al., 2014) tool uses those model results to simulate application executions with lower computational costs, based on the DAG information and a performance model for its tasks, providing a quick and reproducible way to study resource-demanding applications. In the post-mortem performance analysis utility, estimating the theoretical task cost can enable us to classify either the task had an expected duration or an anomalous duration compared to the other tasks. (PINTO et al., 2018) provides an example model used in performance analysis, highlighting tasks whose duration was above the expected in a Gantt chart.

However, such simple models per task type are not sufficient in irregular task cases. There is a need to combine task performance influential parameters like its size and BLAS parameters to model irregular tasks' performance behavior properly. Another practical and feasible technique is to use regression models. They can summarize the expected task performance considering its different irregular costs throughout the application execution. Such regression models can provide useful and straightforward predictions and build very sophisticated simulations, as we will see in the next Section.

3.2.2 Regression Model-based Predictions

As we can model the performance of an application component through a set of factors, we need to use a regression model to consider them all to make our predictions.

Regression models provide a tool that assesses the different factors and the many levels that they can assume to predict a response variable. The usefulness of regression model predictions is noticeable in modeling applications with irregular tasks, where they can extend the purposes mentioned earlier.

The StarPU runtime (AUGONNET et al., 2011) can consider different regression models for application tasks' to help in scheduling, including non-linear and multiple regression models. One example that shows the usefulness of performance models is Simgrid (CASANOVA et al., 2014). This tool supports the StarPU performance models and uses them to provide accurate simulations of irregular applications, achieving a very close result to real executions in the `qr_mumps` simulation example (STANISIC et al., 2015). Typically, it manages to keep the difference below 3%, with some exceptions for architectures with higher core count per L3 cache level and multiple NUMA nodes, where the error rate rises to 8%. Still in the simulation goal, other works also consider modeling the whole application, runtime system, and its workload using a regression model (OZ et al., 2019). It models BOTS benchmark applications considering the scheduler and its queueing policies, cut-off policy to hold task submission, the number of threads, and input size to predict the execution time of a given configuration, reaching an error rate between 6.3% and 14%.

Lastly, besides smart scheduling algorithms and precise, low-cost simulations, regression models can be quite useful for the post-mortem performance analysis. For example, in the Simgrid scenario with many cores per the last level cache, task performance degradation occurs due to cores cache sharing. If we can model such an effect, it may come in handy to detect the regions in a space-time view where this degradation occurs when analyzing a real execution. Another example is to provide an automatic analysis of which factors are interesting to consider in performance analysis of a given workload by analyzing the correlation coefficient of linear models (DREBES et al., 2014a). Lastly, we can extend previous techniques, like automatically detecting anomalous tasks looking at their deviation from the mean task duration (PINTO et al., 2018), which works only for regular tasks, to consider irregular tasks through a regression model.

3.3 Discussion about Performance Analysis Toolset and Modeling

Many of the presented performance analysis tools focus on giving an overview of the application performance through a classical timeline view of the computational re-

sources and the application events. While this technique can be really useful for obtaining a first big picture of what happens to application performance, it needs to be aided by other techniques and tools to enrich performance analysis. For such reason, the tools commonly present a trace explorer that allows users to navigate, filter, and derive metrics for better application performance comprehension. However, as computational platforms became more complex, programming paradigms like task-based arise and represent a paradigm shift that affects both programming and program performance analysis. Thus, to align programmers' abstractions to the application performance, many tools represent the application in a DAG or task-centric way. Task-based performance analysis tools consider extra costs that the runtime system implies to the application through task creation and scheduling. The extra information allows for investigating scheduler dynamic decisions according to the actual DAG scenario regarding its tasks and dependencies. Modeling also plays a role in application performance analysis as they guide scheduler decisions, enable faithful simulations to quickly study factors effects, and can enhance post mortem performance analysis.

The use of visual performance analysis is undoubtedly useful for application analysis as it allows to quickly analyze events that depict application performance through time. Nevertheless, we can relate performance to specific application structures like DAG, which can provide different insights by aligning the programmer's point of view to performance metrics, and better suit visualization to task-based workloads (HAUGEN et al., 2015).

Based on this alignment between performance and application structures, we propose a novel application-centric visualization strategy related to the elimination tree used in the widespread multifrontal methods to enhance performance comprehension by displaying performance data over such a structure. While tree visualization is a quite popular approach in the information visualization field (JANKUN-KELLY; MA, 2003; NOBRE et al., 2018; NOBRE; STREIT; LEX, 2018), few works explore it for performance visualization. The typical approach found is related to representing the code's call-graph as expression trees (WILLIAMS; BIGELOW; ISAACS, 2019; ADHIANTO; MELLOR-CRUMMEY; TALLENT, 2010). While there is an interest to represent such a structure, as many multifrontal-based factorization applications like `qr_mumps` and `PaStiX` provide routines to output a `.dot` file representing the tree structure for debugging and analysis. The only work that provides a visual representation of elimination trees is (ALVARADO, 1990). However, it does not relate application computation over time to the tree structure.

To the best of our knowledge, this is the first work to represent computations over the elimination tree.

Another contribution of this work relies on the use of performance models for post mortem analysis. While regression models are quite useful for simulation and improving scheduling decisions, they are rarely used in enhanced visual performance analysis. The amount of application performance data can quickly become overwhelming for a user to analyze. Thus, tools have been concerned about providing some automatic analysis to guide the user to focus on interesting performance data or application regions (GEIMER et al., 2010; DREBES et al., 2014a). We provide a methodology based on the StarVZ framework (PINTO et al., 2018) that extends the automatic anomalous task detection for regular tasks employing regression models using the task's modeled cost through their theoretical GFlops count.

As performance variability is present in complex HPC systems because of the many factors that can affect task performance, works consider that performance data may come as bimodal or multimodal data. For example, in a scenario where we have a considerable number of slower tasks than expected, and we also have many expected behavior tasks. In this scenario, by considering the use of mixture models, we can better represent the application tasks performance (XU et al., 2020). We employ both a regression model and a mixture of models allowing the user to choose which ones to use for each task. This mixture of models can capture task variability, typically caused by parallel task execution interference, which can stress the L3 cache level and degrade memory-bound tasks' performance. We can then use our models to enrich the Gantt chart, highlighting areas where tasks with anomalous duration occur, leading to a guided analysis that may reveal performance problems on many levels, like system, application, or runtime level. This variability modeling might also help to develop better simulations, for example, in Simgrid. The ability to model irregular tasks, capturing cases where task behavior is strange and needs to be represented with more than one model, can enhance the post-mortem performance analysis, as we will discuss in the next chapter.

4 CONTRIBUTION: ENHANCED PERFORMANCE ANALYSIS FOR IRREGULAR TASK-BASED APPLICATIONS

This chapter presents the methodological aspects of our contributions towards enhanced performance analysis for irregular task-based applications and multifrontal method parallel solvers. Section 4.1 presents the regression models construction and validation, describing the different approaches we used. Section 4.2 goes through the multifrontal-based performance visualization panels, focusing on how and why we propose such a set of panels and what we can represent with those application-wise performance visualizations. Then, in Section 4.3 we describe the addition of those techniques in the StarVZ framework. Finally, Section 4.4 ends this chapter by discussing the proposed strategies, how they can help developers in performance tuning, and our approach's limitations.

All the strategies described in this chapter depend on tracing information provided by the StarPU FxT tracing system and organized in a specific way by the StarVZ framework as implemented in the respective `R` package (SCHNORR et al., 2020). For the regression models, we need the computational weight cost of each task that we want to analyze, their type, and the type of resource it was executed. We assume a performance model that captures the irregular aspects of the tasks like in `qr_mumps`, which considers the blocking sizes, fill-in, and the staircase structure. We need access to the elimination tree structure for the multifrontal-related panels and the tree node information each task belongs to.

4.1 Regression Model for Automatic Task Anomalies Detection

As we saw in Section 3.2, there are different uses for an application performance model, all relying on the power of predicting the expected performance. Here we are interested in the performance analysis facet of performance modeling. We use them in post-mortem performance analysis to compare the obtained performance in the real execution, comparing it to what is expected according to our model. This way, we can automatically detect those tasks that deviate too much from their expected duration and highlight them in the other visualization panels, situating them over the resources and the execution time. This classification guides a more in-depth analysis of these specific application regions where the anomalies occurred. As many applications like sparse solvers

have irregular kernels and commonly have implemented theoretical cost for partitioning purposes, we propose using a regression model based on the tasks theoretical floating-point operation count to detect anomalies automatically.

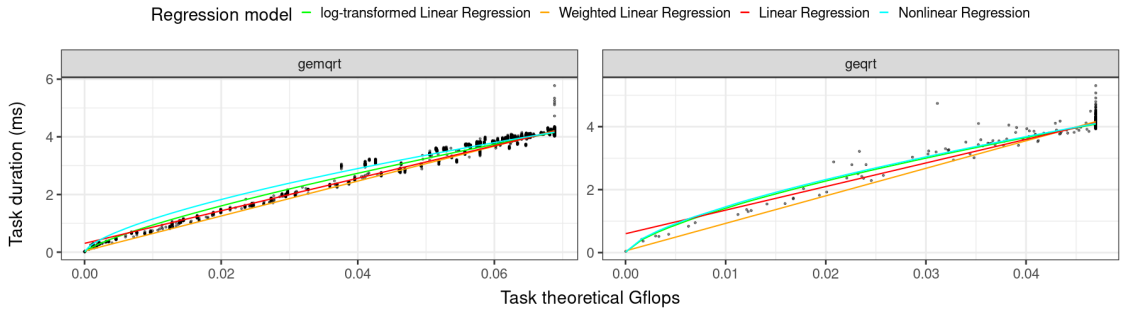
4.1.1 Building the Regression Model

To build a performance model and find what kind of regression model we should use, we need to look at the trace data with the task observations. The regression model we propose to fit application data uses the theoretical Gigaflops as the explanatory variable for the task duration in milliseconds. The more straightforward model to use is a simple linear regression model, which maps the predictor variable to the response variable. However, simple linear regression is based on a set of assumptions for which the data must hold, otherwise using linear regression is inappropriate. In our collected datasets from `qr_mumps` execution traces, we observed that although the relationship is linear, the fitted linear models do not have a normally distributed error with a constant standard error, which is one of the assumptions it should hold. The fact that the model residual values increase as the predictor variable increases, not maintaining a constant standard deviation of the errors, is called *heteroscedasticity* (JAIN, 2008).

There are ways to handle heteroscedastic data by changing the relationship using a nonlinear regression model, a weighted linear regression model, or applying transformations in the data to reduce the error spread. We have implemented these approaches to check which one provided effective results and ended up with a $log \sim log$ transformation of the data to handle heteroscedasticity and use a linear regression model over the transformed data. In Figure 4.1, we can observe the different model fits over the four computational tasks in a collected dataset from `qr_mumps` with 419 `geqrt` tasks, 3.601 `tpqrt` tasks, 9.944 `gemqrt` tasks, and 98.760 `tpmqrt` tasks. Below each model fit, we have a table with the R-squared value and other regression accuracy metrics to compare the tested models. The Mean Absolute Error (MAE) sums up the error terms using its absolute values and giving equal weights to all errors. In contrast, the Root Mean Square Error ($RMSE$) sums the squared errors, giving high penalties for bigger prediction errors. For the MAE and $RMSE$ metrics, smaller is better.

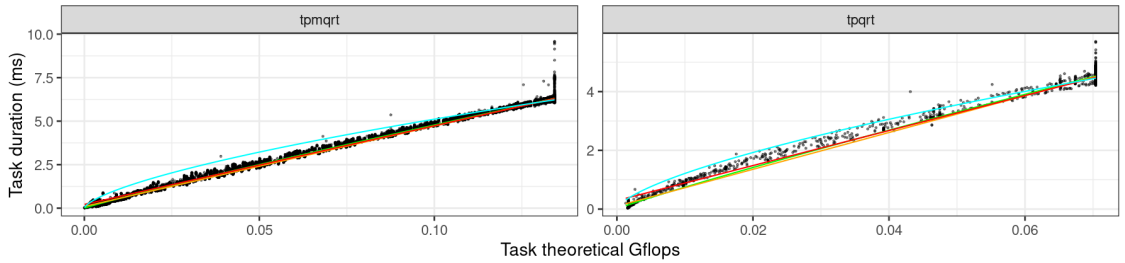
From the R-squared perspective, we can observe that all models fit very well over the data, especially the nonlinear model. However, when we look at the nonlinear model fit over the data, it does not look good because it is above many observations for smaller

Figure 4.1 – Regression models fit and accuracy metrics.



	Model	R²	MAE	RMSE
1	Linear Regression	0.9843737	**0.06134156**	0.09536623
2	Weighted LR	0.9979699	0.07290072	0.11014807
3	Nonlinear	**0.9991235**	0.06849012	0.11818004
4	log–transformed	0.9831301	0.06187557	**0.09116149**

	Model	R²	MAE	RMSE
1	Linear Regression	0.9034735	0.1427807	0.2148905
2	Weighted LR	0.9930841	0.1722731	0.2412748
3	Nonlinear	**0.9973679**	0.1279924	0.2014767
4	log–transformed	0.9661099	**0.1203297**	**0.2013978**



	Model	R²	MAE	RMSE
1	Linear Regression	0.9962546	0.06930231	0.09826259
2	Weighted LR	0.9979219	0.07276620	0.10433287
3	Nonlinear	**0.9982106**	0.14817449	0.24580012
4	log–transformed	0.9836463	**0.06509067**	**0.09212584**

	Model	R²	MAE	RMSE
1	Linear Regression	0.9779830	0.1515807	**0.1756064**
2	Weighted LR	0.9929213	0.1559634	0.1855547
3	Nonlinear	**0.9979897**	0.1636791	0.1864796
4	log–transformed	0.9681302	**0.1483706**	0.1797493

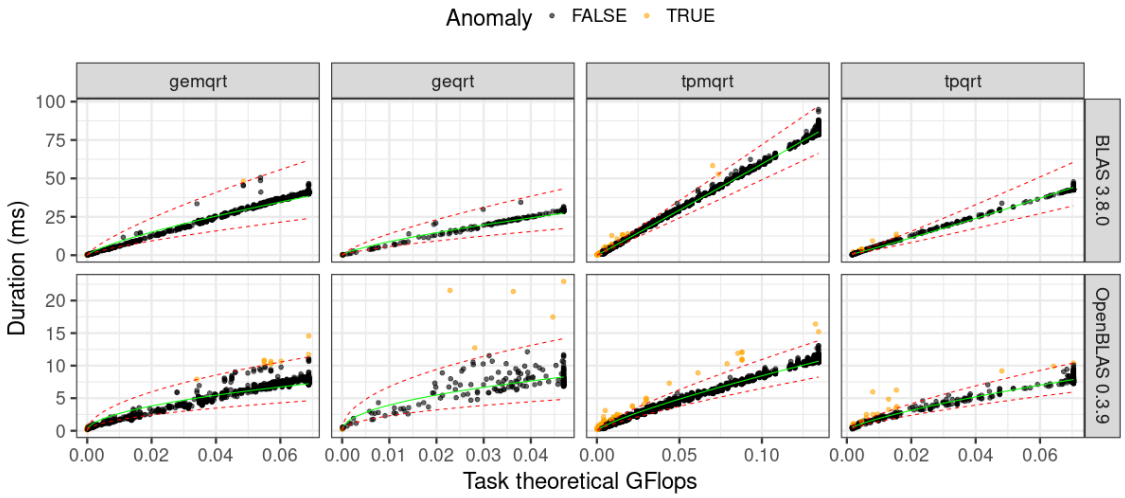
Source: The Author

GFlops values. It has a better R-squared because most of the task observations (≈78%) have the maximum GFlop value among the same type of tasks, which fits those values better. We can observe the effect of such data distribution in the linear model fit for the `geqrt` task where the red line starts with an offset from zero. This is corrected by assigning the weight $1/GFlops$ for each observation, increasing the observations' contribution with smaller GFlops for the model fit. However, looking at the MAE and RMSE accuracy metrics, we continuously observe better values for the log-transformed linear regression model in the Figure 4.1 (table cells with bold font). The linear regression model overcomes the values where the log-transformed model does not have the best results, but we should not use it since the data is heteroscedastic. While it is hard to tell which one is the right model to use, we can think of what is the useful model for us, according to our purpose. In this sense, despite having slightly better results for the accuracy metrics, we selected the log-transformed model because it handles heteroscedasticity and provided satisfactory results in the anomalous task classification process, observed throughout sev-

eral cases.

In `qr_mumps`, the performance model to estimate the floating-point operation cost is based on the reference BLAS implementation[1] and considers the fill-in generated and the `ib` parameter effect. This way, we can quickly use the theoretical GFlops count, which captures the irregularity of computational tasks, to model task duration, according to their type and the computational resource type (i.e., CPU or GPU) that executed the task. However, in practice, HPC applications use heavily optimized BLAS libraries to take advantage of data locality and architecture-specific instructions, accelerating computations. For such optimized libraries like in OpenBLAS (XIANYI, 2013), we have noticed a higher variability in the data, illustrated in the bottom of the Figure 4.2 with the log~log model. Nevertheless, despite this higher variability in the OpenBLAS version, the model captures well the reference and the optimized BLAS implementations.

Figure 4.2 – Comparing the log-transformed regression model fit for both Netlib BLAS and the optimized OpenBLAS.



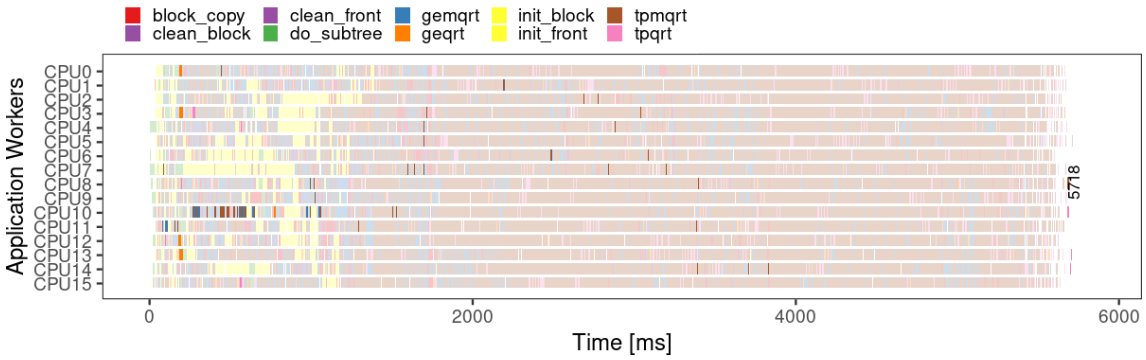
Source: The Author

By looking at the model prediction intervals represented by the dashed red lines in Figure 4.2, we can analyze its adequacy in fitting the data and detecting anomalous tasks within the task and resource types. We use the upper prediction limit, considering a confidence level of 0.95, and use the predicted values to draw this red line to determine the boundary between the expected and the anomalous tasks. We classify as anomalies, observations whose task duration lies above the line for a given value of GFlops, represented by the yellow points in the figure. Then, we use this anomalous task classification to enrich the space-time Gantt charts, differentiating the expected and anomalous tasks with

[1]Netlib reference BLAS implementation <http://www.netlib.org/blas/blas-3.8.0.tgz>

a transparency level. Tasks whose duration falls into the expected region of the model have high transparency. In contrast, the abnormal tasks have no transparency, producing a highlighting effect over those tasks with stronger colors. Figure 4.3 demonstrates this technique's use to enrich such space-time representation with task classification information.

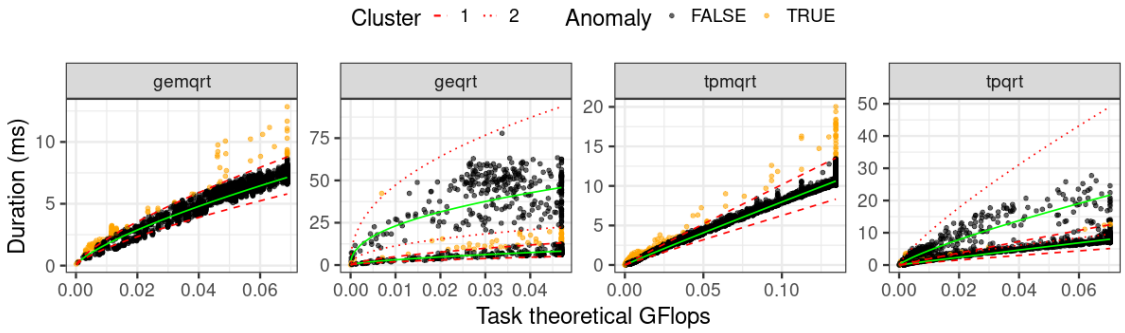
Figure 4.3 – Anomalous task highlighting in a Gantt chart.



Source: The Author

While for the case described in Figure 4.2 one model seems sufficient to represent the expected task behavior with precision. We have faced some cases during our experiments where just one model seems insufficient to represent task behavior because of high data variability. We employ a finite mixture of regression models to properly describe such data, using the log-transformed model. The finite mixture model allows us to cluster observations according to the likelihood of belonging to one of a set of multiple regression models. In cases where there is such high variability, we describe the data using the two most likely models. The Figure 4.4 represent a case where we applied this technique for the `geqrt` and `tpqrt` tasks for the OpenBLAS implementation.

Figure 4.4 – Fitting multiple models over data using finite mixture models.



Source: The Author

The fact that we have high variability in task duration of similar computational

weights is already interesting. It can be a sign of a deeper performance problem in the execution. We can use this enhanced modeling with multiple regression models the same way we use the anomalous classification with a simpler model: highlighting the tasks in the Gantt chart. The anomalous classification is done the same way as for a single model. However, we also consider anomalies the tasks that seem extreme for both models, like the ones below the lower prediction of the topmost model and above the upper prediction for the bottommost model in the `geqrt` task.

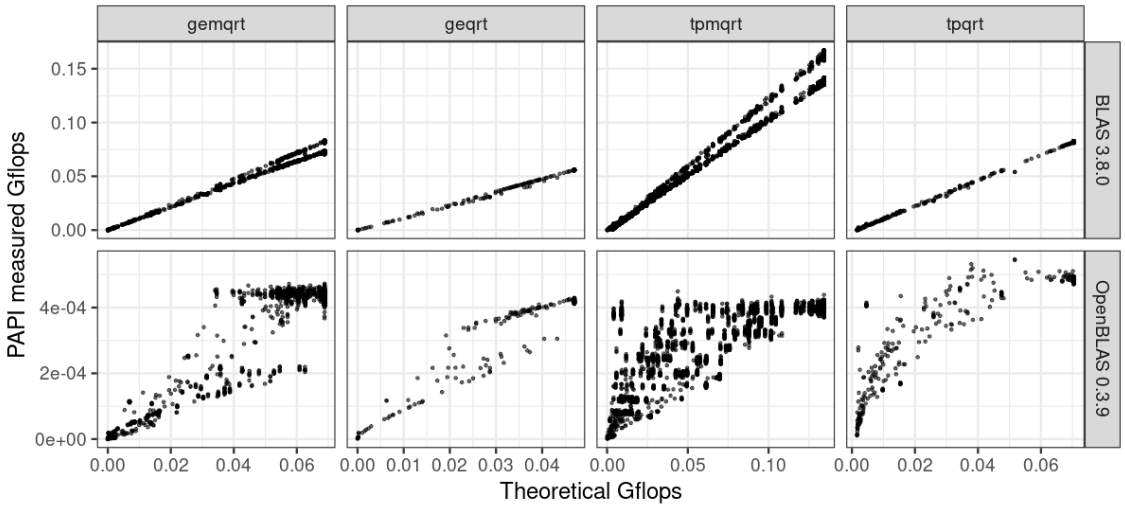
Furthermore, as we observe in Figure 4.4, the two models successfully divide tasks into what seems to be a cluster with slow tasks and an expected duration task cluster. Since there are not many anomalous tasks within the clusters, it can be useful to highlight the clusters in the Gantt chart instead of just the anomalous tasks. This cluster highlighting will reveal the temporal and spatial characteristics of those two different behaved groups of tasks.

4.1.2 Model Validation

As the theoretical GFlops values provided by `qr_mumps` are just an approximation for the tasks' real cost, we should check its accuracy in modeling the real cost. Because theoretical cost calculation in `qr_mumps` considers many task aspects, like the fill-in, `ib` size, and the staircase structure, its estimation is reasonably close to the real floating-point operation count reported by the PAPI library. Adding hardware counter values in the StarPU FxT tracing system allows us to verify the theoretical model alignment compared to the real number of floating-point operations provided by the hardware counters. Figure 4.5 shows the relation between the theoretical and hardware counter values for the tasks' GFlops cost for the reference Netlib BLAS 3.8.0 implementation and the OpenBLAS 0.3.9.

We observe the theoretical cost is very close in the magnitude of its value compared to the PAPI floating point measurements for the BLAS version, validating how `qr_mumps` computes the computational task costs. Even so, we observe a peculiar effect for the `gemqrt` and `tpmqrt` tasks in the reference BLAS plot, where the observations draw two well-defined lines. They may have arisen from the natural variability. However, their organization is very structured, which is strange. The other hypothesis we have for this is that low-level optimizations like speculative execution are causing an overcounting for the `PAPI_FP_OPS` event, which can occur according to the wiki in the PAPI

Figure 4.5 – Comparing task theoretical cost provided by `qr_mumps` with the PAPI hardware counter measured floating-point operations.



Source: The Author

repository[2].

In the case of the optimized BLAS library, the OpenBLAS, we have a very different scenario. There is a considerable mismatch between the theoretical values and the values collected from the hardware counters. This mismatch comes from the implemented optimizations in the OpenBLAS version, which explore operations like fused multiply-add (FMA) and vectorized instructions like the ones provided by Advanced Vector Extensions (AVX). However, despite this discrepancy between theoretical and real values, we can observe in Figure 4.2, built with the same data used in Figure 4.5, that there is a relationship between task duration and their theoretical cost values for OpenBLAS. Furthermore, they present a strong positive association that enables us to use these theoretical values to develop regression models for both BLAS implementations.

4.2 Multifrontal-Based Performance Visualization

The multifrontal method is a widespread technique to obtain the direct solution of a sparse system of equations, adopted by many implementations of high-performance parallel solvers. Since the first multifrontal code (DUFF; REID, 1983), many other implementations adopted this technique to provide performance (AMESTOY; DUFF; L'EXCELLENT, 1998; JOSHI et al., 1999; GUPTA, 2000; TOLEDO, 2003; DAVIS,

[2]PAPI repository Wiki <https://bitbucket.org/icl/papi/wiki/PAPI-Flops>

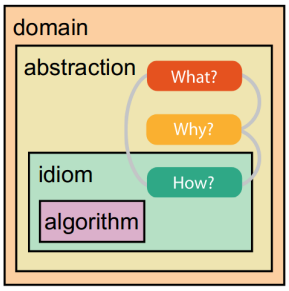
2004; DAVIS, 2011; AGULLO et al., 2013), see (DAVIS; RAJAMANICKAM; SIDLAKHDAR, 2016) for a comprehensive list of libraries and methods for direct sparse factorization. The elimination tree is the structure located at the core of the multifrontal method, shaping and guiding the computations and communication in a elegant way. Despite that, the *supernodal method*, which is another main approach for parallel numerical factorization, can also describe its computations using a tree structure, with computations on the nodes and communications on the edges, for which the visualizations we propose in this work could also help.

Visualizing software for performance optimization is a well-established technique (MATTILA et al., 2016) because of its power to understand application behavior better. We propose a set of visualization panels related to the elimination tree structure in the StarVZ tool's context to depict application performance behavior along time, relating it to this structure that plays an essential role in the factorization. We include panels depicting the elimination tree along time, relating the application operations and performance metrics to its structure, and resource utilization according to tree properties. By combining those application-specific visualizations with classical ones like Gantt charts, we can enhance the performance analysis by aligning the application data structure and the way it represents the problem to the performance.

To guide our visualization panels development, we used the `What-Why-How` (WWH) (MUNZNER, 2014) visualization analysis framework, represented in Figure 4.6, to ponder about the essential aspects to consider in the development of our panels. In this framework, the **domain** represents our target users, who are familiar with the presented matrix factorization method, or, more specifically, people who work with or use `qr_mumps` or similar applications. The **abstraction** level concerns translating the specific domain vocabulary to the visualization vocabulary. At this level, we have the **What** and **Why** questions to answer. The first one asks us about what can be visualized by the user, what kind of data we have to present, and what attributes our data has. For the second question, we need to answer why we are presenting the data we have. Thinking about this helps us define visualization tasks in their abstract form, which can be seen as a pair of actions and targets, like in our regression model classification in the Gantt chart: locate outliers. A more general and domain-specific task, for example, is to visualize the behavior of the algorithm over the tree structure over time. After defining a set of goals in the **Why** phase, the **How** concerns about determining ways to support those goals through the visualization **idiom**, which is how the visual encodings and mappings will work to

accomplish those goals.

Figure 4.6 – What-Why-How analysis framework.



Source: (MUNZNER, 2014)

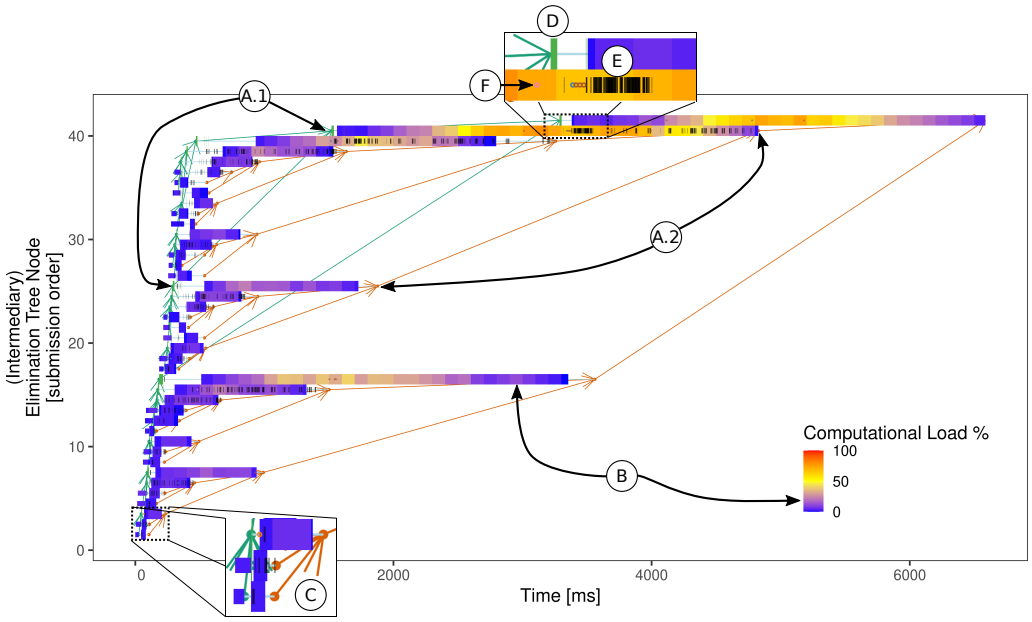
4.2.1 Elimination Tree Computation Along Time

The elimination tree's shape and size depend on many factors during the analysis phase of a direct factorization algorithm. By changing the ordering algorithm, amalgamation threshold, and pruning techniques, we can produce very different elimination trees for the same input matrix. Having a visual representation for this structure, we can provide a way to understand the effect of its parameters in performance and tune them. Figure 4.7 represents our proposed panel for visualizing the events over the elimination tree. It organizes the tree nodes by their submission order in the Y-axis, representing the execution time in the X-axis.

At its core, this panel draws the tree structure using points to represent the tree nodes and arrows to represent the parent and child relation between them. The tree with the green nodes and arrows marks each node's starting point on its first initialization task, while the orange tree delimits the last task for a tree node. The arrows represent the parent-child relation, pointing from the green child nodes to its parent starting point and connecting the orange ones to the end of its parent. In the highlighted nodes (A.1) and (A.2), we can observe this parent and child relationship, and the starting moment of the nodes in the green tree (A.1), and their ending point in orange (A.2). Each node occupies a horizontal line in the panel and is delimited by these two different points. The period between these two points, we call it the node *lifespan*. Which solely represents the node memory footprint, not meaning that we had computations during all its lifespan. For example, the distance between the two tree points (green and orange), pointed by the lower arrows in (A.1) and (A.2).

We represent the computations inside each node using a color gradient to represent

Figure 4.7 – The elimination tree panel depicting tasks over the tree structure and time for the e18 matrix.



Source: The Author

low to high resource usage intensity based on the factorization tasks only, represented in (B). This computational intensity over the structure reveals where the scheduler is concentrating the computational efforts. Alternatively, we can represent other performance metrics besides the resource utilization, like the GFlop throughput or hardware counter values. To represent the numerous pruned subtree nodes, we group all the pruned subtree roots with the same parent, reusing its Y position, aggregating their computations, and drawing them with half of the height of the non pruned nodes as depicted in (C). Besides this spatial aggregation for the pruned nodes that share the same parent, we also use a user-defined time aggregation to represent the computations in time slices, helping in cases where the task count is too numerous and when the application has a long makespan. In the Figure 4.7, we defined an aggregation step of size 100ms.

Furthermore, we also provide the representation of other events along the tree and their execution. The initialization and memory allocation tasks are represented with green rectangles, highlighted in (D). The black rectangles represent the communication tasks between the child and parent nodes using transparency to know when we have a higher concentration of these tasks as depicted in (E). Their length represents the raw task duration, and their height is also half of the node height to avoid overlapping the computations. Lastly, we can also represent the anomalous tasks location in the tree as dots over the node's computation representation, following the same colors for the Gantt chart tasks, as pointed by (F).

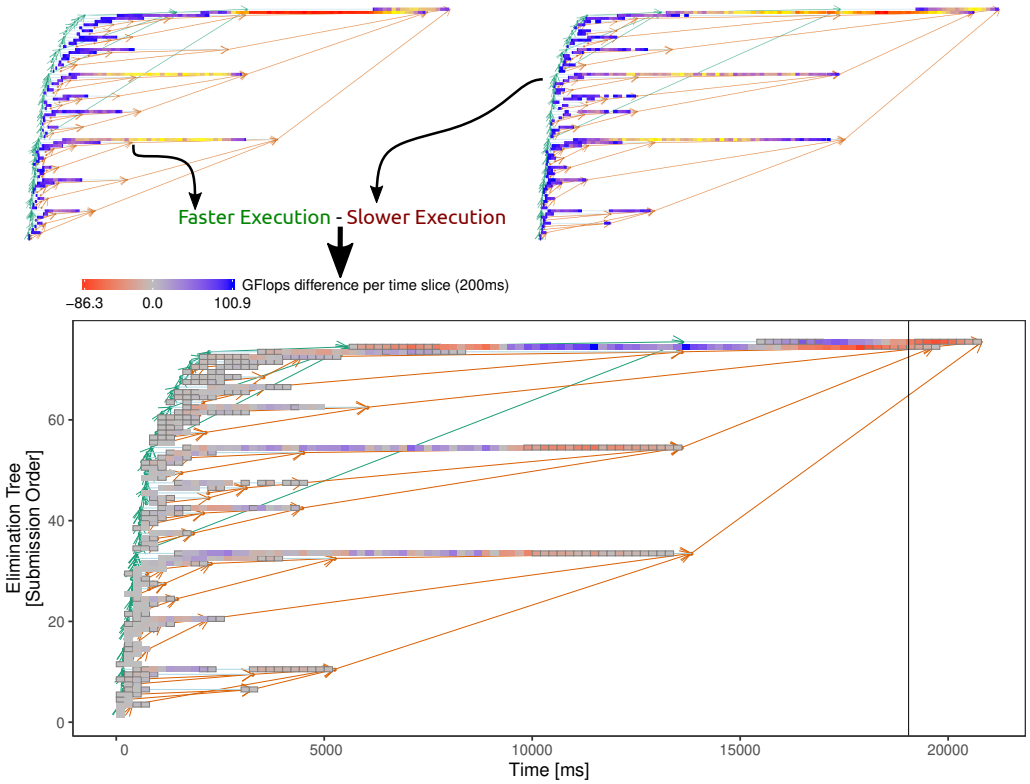
Besides promptly displaying an overview of the tree topology, with this representation of where and when the computations, initializations, communications, and anomalies occur. Our elimination tree panel depicts precisely the scheduler path to traverse the tree, revealing the prioritized and postponed paths and nodes through the computational intensity over time that we observe in the figure. This panel still reveals the parallelism sources in the application. The concurrent execution of nodes in different Y positions reveals the tree parallelism, gradient colors within a node represent the node parallelism, and the overlap in the computations of two dependent nodes the interlevel parallelism. As this tree structure guides the execution controlling the number of available tasks and memory usage, observing how the scheduler traverses it to provide performance can reveal some patterns imposed by task priorities or memory consumption restrictions, unveiling the scheduler signature to traverse the DAG, relating it to the tree structure.

Another aspect that can be interesting is to compare different application and runtime configurations for the same elimination tree. While simply juxtaposing two trees can reveal some differences, it is hard to perceive them in detail. This way, we provide a panel that uses the same visual elements to represent the tree structure and its computations, merging the two trees and deriving new data to represent their performance difference. This new data represents the difference in performance by subtracting the slower execution from the faster execution, using the desired performance metric like resource usage or GFlop throughput. Figure 4.8 shows an example of this comparison panel using the difference in the GFlops computed per time slice. In this plot, the segments with a thicker line represent moments where we have an exclusive execution by some of the trees. Segments that do not have this line means that we have a concurrent execution. The vertical line at the end marks the end moment of the faster execution.

4.2.2 Resource Utilization and Sources of Parallelism

The usefulness of the elimination tree panel is to represent details of how the multifrontal method evolves, but it lacks an aggregated view of the computational power over time. Hence, we provide an additional visualization to depict the resource utilization considering the node and depth elimination tree properties. Figure 4.9 provides a succinct view of how much computational power is dedicated to each tree node (top) and each tree depth level (bottom), and an overview of the idleness through the white area. Those panels stack the cumulative resource utilization, aggregated by a user-configurable time

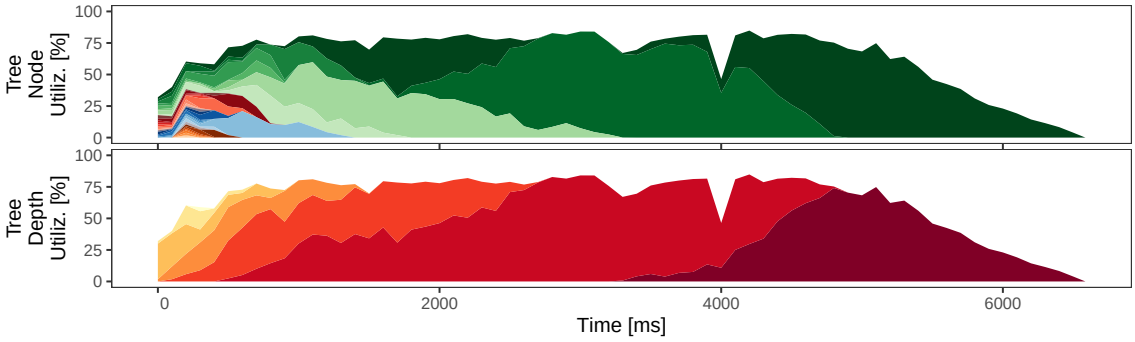
Figure 4.8 – Scheme that shows the creation of the panel to compare two trees.



Source: The Author

step (100ms in this case).

Figure 4.9 – Visualization panels representing the resource usage by elimination tree node (top) and depth (bottom).



Source: The Author

We use the colors for the nodes to purely differentiate one tree node from another, not identifying each node individually. However, for the tree depth panel, the color scale has a meaning that represents the distance from the root, with a darker color for nodes near the root and lighter colors for nodes far from the root. The node panel reuses colors to represent nodes that do not have overlapping task executions, reducing the range of used colors and increasing the user's capability to differentiate between two nodes. This color

difference can reveal how computing power tackles parallelism in its different forms. We observe the node parallelism by looking at the height of a specific color, the tree parallelism by the multiple colors stacked at a given moment, and the interlevel parallelism when we combine this visualization with the elimination tree panel.

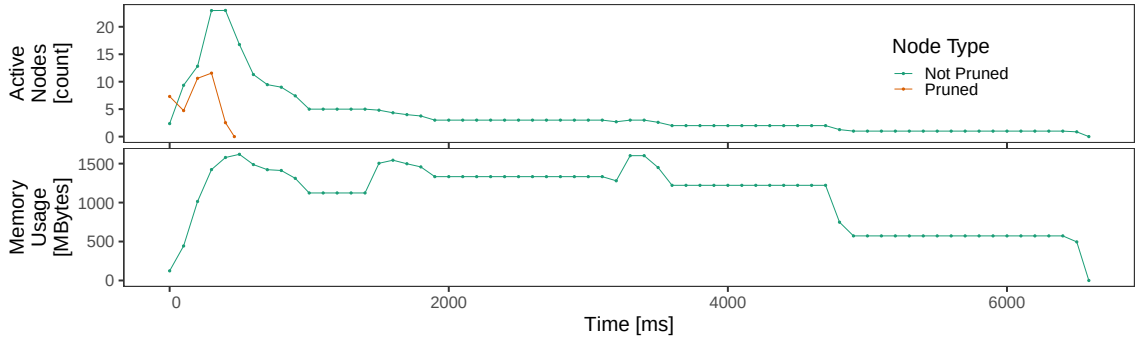
4.2.3 Active Nodes and Memory Consumption

The order the scheduler traverses the tree computing its nodes has a significant impact on the memory consumption peak. On the other side, memory consumption restrictions can have a significant impact on the tree traversal. Memory management is a real concern over the multifrontal method since it can impact the available parallelism, and if not adequately controlled, the peak consumption can overcome the available memory. Thus, memory is an important factor in such applications, and thus, considering it in visualizations is also crucial.

In the multifrontal method, as the parent nodes need to gather their child node contribution blocks, this implies that all nodes involved in this operation must be active in memory at that moment. Figure 4.10 presents the panels related to the number of active nodes in memory (top) and the total memory usage by the nodes (bottom). These two panels provide a summary to understand better how the number of in-memory active nodes and current memory usage evolves through execution time. As the `qr_mumps` application can constrain the memory usage during the execution, keeping it under control, these panels help understand how it manages the memory-related issues in scenarios with restrictions.

The two different lines in the in-memory active nodes represent their type, if they

Figure 4.10 – Number of active nodes by their type (top) and memory usage (bottom).



Source: The Author

are pruned or not. Depending on the traversal of the tree and task priorities, it is usual that the sequential nodes only exist at the beginning of the execution because they are the leaves of the tree, which release the tree parallelism. Both panels' information can also be aggregated over time by user-defined intervals, or we can choose to represent the raw data.

4.2.4 An Application-centric View of Performance Data

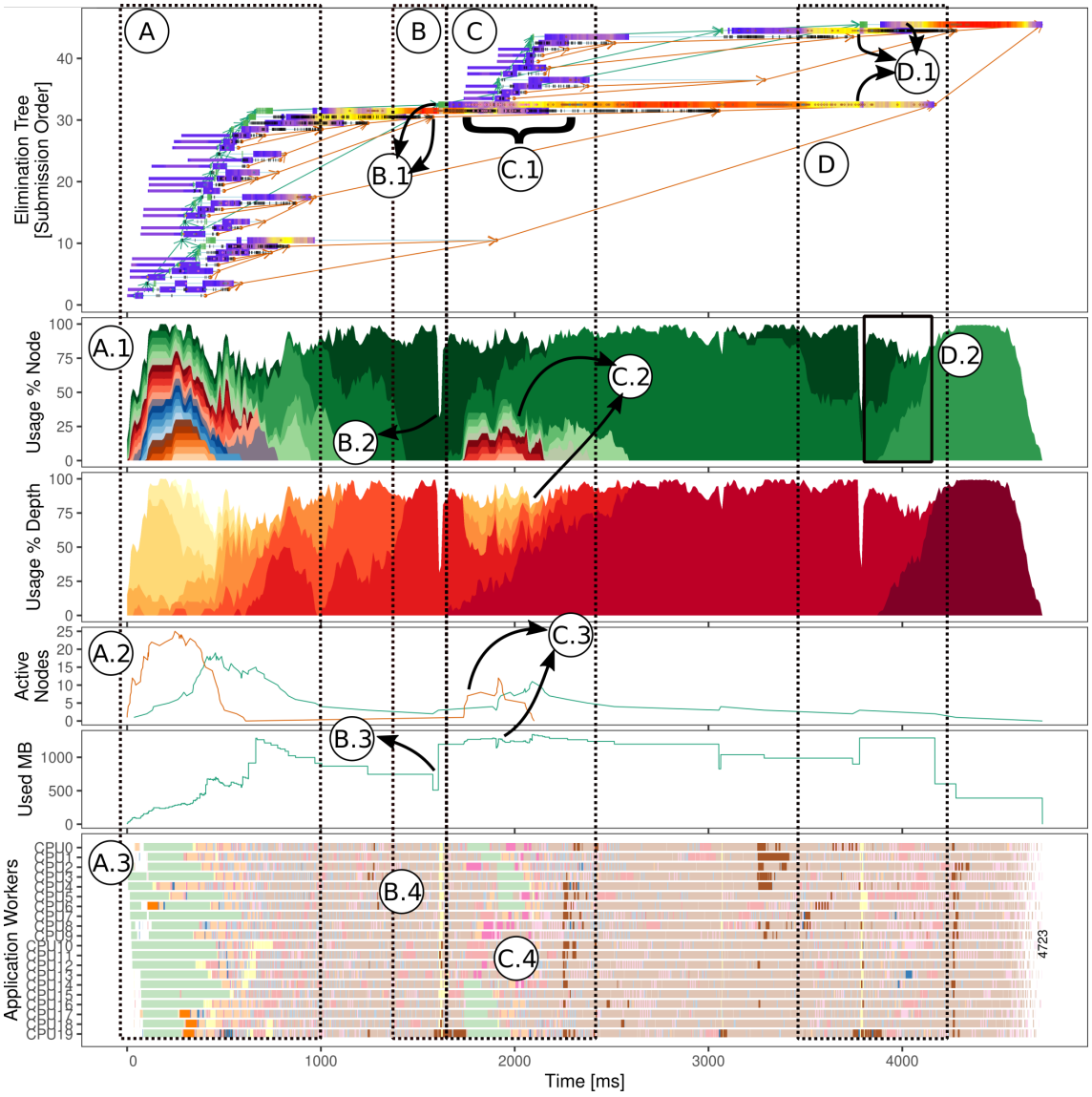
All these presented application-wise panels contribute to capture and represent the performance behavior with a focus on the algorithmic point of view and its data structures. In Figure 4.11, we present those panels in action for a real execution, pinpointing some perspectives of the application execution that such set of visualization provides.

We observe in (A.1), in the application beginning, that there is a high level of tree parallelism, depicted by the many different colors we see in the resource usage per node panel. This amount of tree-level parallelism represents that the application built the tree in a way to provide sufficient tree parallelism to occupy all workers at the beginning of the application. Furthermore, we observe in (A.2) that the vast majority of the nodes in memory are pruned nodes, and they are very small considering the amount of memory they use. We also observe all those pruned nodes in (A.3), where the green tasks are `do_subtree` computations, where we can observe a slow start in the application.

At (B), we highlight a delayed node activation imposed by the memory consumption constraint. We observe in the nodes pointed by (B.1), the moment the lower node is freed from memory, it released sufficient memory for the topmost node allocation without violating the constraint. This new node activation creates lots of memory initialization tasks (`init_front` and `init_block`), which reduces resource utilization in (B.2) because we do not consider the time taken by these tasks in the resource utilization. The memory usage plot in (B.3) details this moment. We observe that the newly allocated node would use too much memory, and so it had to wait for more available memory to respect the constraint. We point out in (B.4) that the Gantt chart represents all these initialization tasks, but it would be difficult to interpret them without the tree dependencies and the memory usage plot.

This scenario represents scheduler related decisions, which could be influenced by the submission order of the tasks and their priorities. The scheduler could have chosen to explore the subtree highlighted in (C) earlier in the execution. In (C.1), we see how that

Figure 4.11 – Example of tree-related plots enriching a Gantt chart view.



Source: The Author

entire subtree was postponed to later in the execution, while (C.2) points out that many new nodes appear in the resource utilization plot and that they are part of previously explored depths of the elimination tree. Another fact that emphasizes that this was a scheduling decision is because the new nodes allocated in (C.3) use just a few Megabytes of RAM, which will probably not exceed the memory limit if computed earlier. In (C.4), we observe that this decision led to late `do_subtree` tasks and might be related to the outliers that appear at that moment.

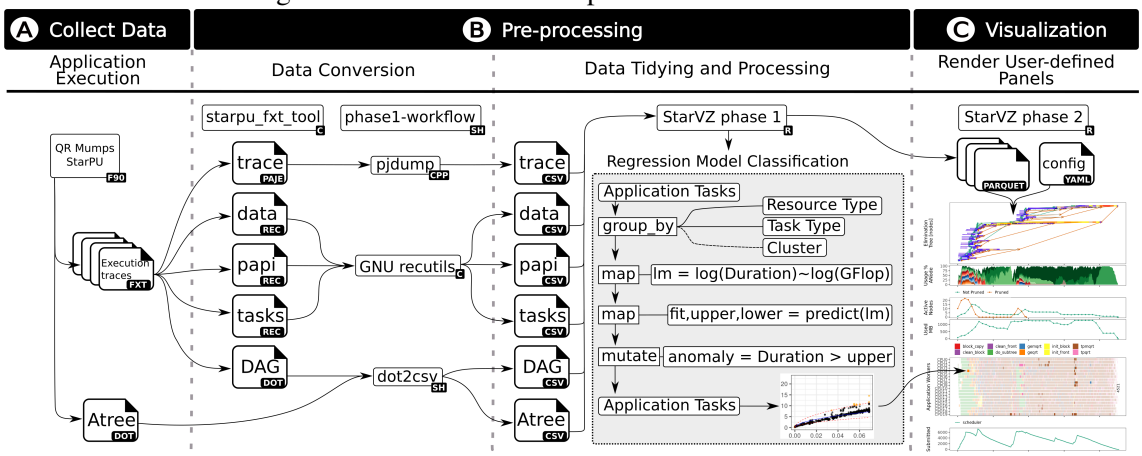
Lastly, we observe in (D) that two of the nodes pointed by (D.1) are child nodes from the last node, and even so, the three are working in parallel, also presented in the resource utilization panel in (D.2). This is the interlevel parallelism presented earlier,

which is possible because of the finer-grained partitioning of the elimination tree in a DAG.

4.3 Implementation in The StarVZ Framework

StarVZ is a performance analysis tool available on CRAN[3] that provides a set of performance visualization panels for task-based applications that use the StarPU runtime system. The tool works in two phases, where the Figure 4.12 illustrates this workflow. In Figure 4.12 (B) is the preprocessing phase or StarVZ phase one, and (C) is the visualizations construction phase analogous to the StarVZ second phase. It converts the StarPU FxT trace files to the Paje trace format in phase one, breaking them into different files, cleaning and organizing data. Then it converts those files to suitable formats to read them in a modern data analysis language like `R`. Still in phase one, it performs some operations over the data like the anomalous task classification. Finally, at the end of phase one, it saves all organized, and processed data in the efficient columnar storage format Apache Parquet (VOHRA, 2016). Phase two then reads those files and a configuration file to generate a set of fully configurable performance visualization panels, which can also be manually produced by the user through the `R` package.

Figure 4.12 – Workflow and phases of the StarVZ tool.



Source: The Author

We enhance StarVZ with new functionalities. For example, building the regression models to support the anomalous classification of applications with irregular tasks, based on the existing per-task theoretical floating-point operations cost. This step is high-

[3]StarVZ CRAN R-package <https://cran.rstudio.com/web/packages/starvz/index.html>

lighted in Figure 4.12 by the gray area. We group the computational tasks by their type, resource type, and cluster (if using multiple models). Apply the regression model, and classify the anomalous tasks. This classification will then translate in the highlighted tasks in the Gantt chart or the tree structure. The other visualization panels were built upon this preprocessed data from phase one, where we provide simple function calls to create the visualizations, using the grammar of graphics as implemented by `ggplot2` (WILKINSON, 2012). Those functions allow the users to fully configure visualizations through a set of parameters.

4.4 Discussion and Summary

We have seen that we can provide very detailed performance visualization representations of the `qr_mumps` application with all these strategies and visualizations. With all this detail, we should be able to explore and analyze many different cases to observe in practice what we can find out about performance with our techniques, which otherwise would be inaccessible by just looking at a Gantt chart. Despite the multifrontal method, the other main method for parallel sparse direct factorization is the supernodal approach, which can also express parallelism in a tree structure. This extends the usability of our contributions to other applications. They only need to provide the data.

Despite the usefulness of our proposed methodology, there are some limitations. Although the user can explore data in the trace through the StarVZ function parameters, a fully interactive display would be rather efficient than our static visualization, in terms of data exploration aspects. Also, for huge elimination trees, properly identifying the visuals aspects could be problematic without zooming. This way, a spatial aggregation like the one used for the pruned nodes should also be considered to group the pruned nodes.

5 EXPERIMENTAL RESULTS ON ENHANCED IRREGULAR TASK-BASED PERFORMANCE ANALYSIS

This chapter describes our experiments using the `qr_mumps` application over different workloads, machines, and different configurations to test our visualization strategies proposed to the StarVZ tool. Section 5.1 describes the experimental setup, involving the workload, computational platforms, runtime and application factors, and experimental design. Section 5.2 presents the results of our investigation using the anomalous task classification approach. Section 5.3 describes detailed investigations for the interesting cases we have found using our methodology. Lastly, Section 5.4 closes this chapter bringing a summary of the chapter and discussion points.

5.1 Experimental Setup

To explore our performance analysis visualization techniques in distinct scenarios, we selected different workloads from real problems registered in the SuiteSparse matrix collection[1] repository. The Table 5.1 lists the selected matrices, sorted by descending order of the total GFlops when ordered with Scotch.

Table 5.1 – Matrices used as workload for `qr_mumps`.

Name	Rows	Cols	NNZ	GFlops with Scotch
TF18	95.368	123.867	1.597.545	196200.78
sls	1.748.122	62.729	6.804.304	66382.59
TF17	38.132	48.630	586.218	12389.67
ch8-8-b3	117.600	18.816	470.400	10978.69
Rucci1	1.977.885	109.900	7.791.168	5182.91
flower_8_4	55.081	125.361	375.266	2702.14
e18	24.617	38.602	156.466	1460.89
degme	185.501	659.415	8.127.528	402.05
karted	46.502	133.115	1.770.349	246.78
lp_osa_60	10.280	243.246	1.408.073	182.08
EternityII_E	7.362	150.638	782.087	133.41
g7jac200	59.310	59.310	717.620	90.16
fxm3_16	41.340	85.575	392.252	0.322

To run the experiments, we used three machines from the PCAD[2] cluster, listed in the Table 5.2. They provide different configurations in terms of hardware, which creates

[1] SuiteSparse Matrix Collection <http://sparse.tamu.edu>
[2] PCAD UFRGS <http://gppd-hpc.inf.ufrgs.br/>

different scenarios for the `qr_mumps` application because the size and shape of the tree depend on the number of available resources and the pruning technique. Furthermore, they provide different heterogeneous scenarios for the StarPU schedulers to handle. We choose these matrices from previous works with `qr_mumps`, using a diversity of matrices structural shapes, dimensions and computational weight.

Table 5.2 – Description of the machines used in the experiments.

Machine	**CPU**	**GPU**	**CUDA cores**	**RAM**
Draco	2×8 E5-2640v2, 2.0GHz	Tesla K20	2.496, 706MHz	64GB
Hype	2×10 E5-2650v3, 2.3GHz	Tesla K80	4.992, 824MHz	128GB
Tupi	1×8 E5-2620v4, 2.1GHz	GTX 1080Ti	3.584, 1.582MHz	64GB

For the system configuration of the machines, we have Debian 10 kernel version 4.19.0-8-amd64 with exclusive access to the machines during the experiments. We used a branch of the `qr_mumps` v3.0 with our modifications to enrich the trace data with the information we needed. It was compiled using GCC 8.4.0, linked with CUDA 11.1.0, Scotch 6.0.8, and Metis 5.1.0 for matrix reordering. The BLAS implementation from OpenBLAS 0.3.9, and the StarPU library master branch with the commit hash `cc8d6e7fea87e825b90631bd9a164082cbb1ae5b`.

The `qr_mumps` application is extensively configurable through its parameters. We considered two global configurations of the application that enables or disables computations on the GPUs. For the cases using only CPUs, we fixed the block size (nb) as square blocks of size 320 and 32 for the internal block size (ib). When using GPUs, we expect more performance by creating bigger tasks to take advantage of the GPU computing power. Thus, we used $nb = 900$, and $ib = 90$. We fixed those values to enable performance comparisons between two executions with the same global configuration but different factors. These values may not provide the best performance for each tested matrix, but preliminary experiments showed that they reach reasonable performance levels.

Despite fixing nb and ib, we explored different application and runtime factors to investigate their impact over the tree structure and application behavior. We explored the impact of the **memory constraint** parameter using two levels for it: `limited` to sequential peak and `unlimited` memory usage. These parameters determine and control the maximum amount of memory that the application is allowed to use during the factorization. When limited, it restricts the memory usage to the peak reached by a sequential traversal, representing a lower bound. This memory control is important for the application behavior and directly impacts the available parallelism. Another crucial

factor considered in our analysis is the StarPU **scheduling algorithm**. We have chosen the `prio` and the `lws` schedulers for executions using only CPU, and `heteroprio`, `dmda`, and `dmdasd` for executions including GPUs. We also considered two different **ordering algorithms**, using the `Scotch` or the `Metis` library. Both libraries handle matrix reordering differently, generating unique eliminations trees and different costs of floating-point operations.

We then created a factorial design for our experiments combining all machines, factors, and `qr_mumps` configurations. In Section 5.2, we systematically executed, collected trace data, and created the visualizations to analyze the presence of anomalous behavior between tasks for some configurations. Then for Section 5.3, we performed executions without tracing information first to find interesting cases where performance between configurations is different and further analyzed with our proposed techniques.

5.2 Detecting Anomalous Tasks Within `qr_mumps`

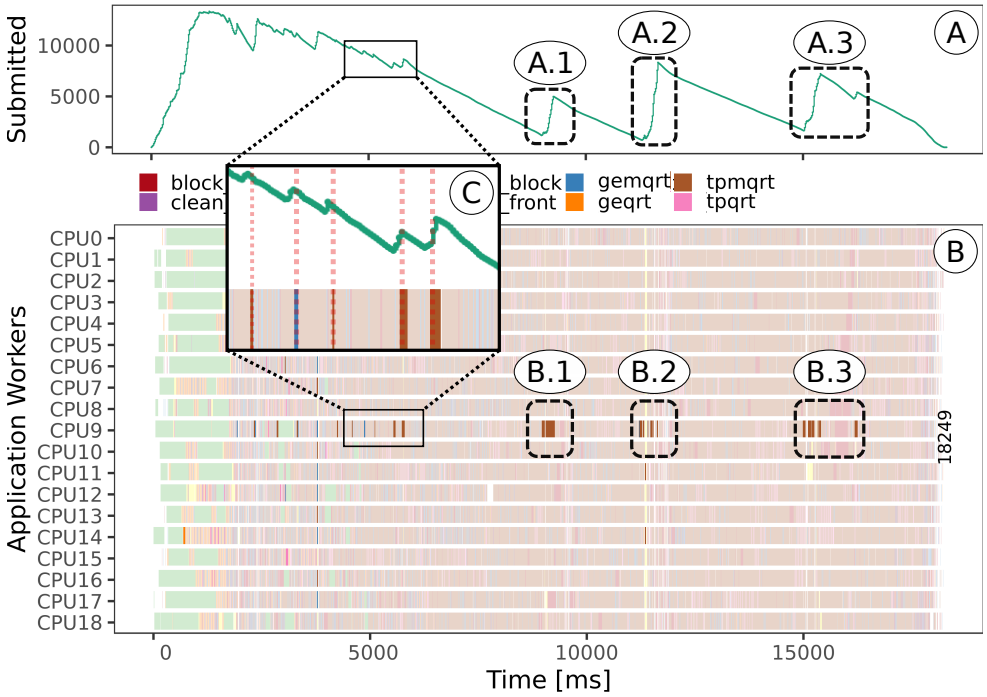
This Section presents our regression-based task anomaly classification results, explaining the causes of anomalies in four different cases. The first is related to the task submissions, the second with application tracing overhead, the third is related to the numerical content of the task data handles and compiling flags. Lastly, in the fourth case, the anomalous tasks are associated with a high L3 cache miss number.

5.2.1 Case 1: Task Submission Peak

In the StarPU programming model, the main thread is responsible for handling the STF task submission, which can occur at any moment during the application execution. Commonly, the whole DAG is unrolled and submitted early in the execution. Nevertheless, some scenarios with memory restriction, for example, may delay these submissions. Figure 5.1 depicts the Gantt chart anomalies along with the StarVZ task submission panel, showing a case where the tasks submissions are spread throughout the execution. In this figure, we observe that the Gantt chart has a set of anomalous tasks highlighted in the same worker (B.1, B.2, and B.3). Aligned with these anomalies, we observe that we have peaks of task submission related to them (A.1, A.2, and A,3), and we also observe in (C) that even a small number of task submissions can cause anomalies. In this example, we

bound the main submission thread to the `CPU9`, which causes a slightly increased duration for the tasks in that worker because of the competition for the computational resource captured by our model.

Figure 5.1 – Panel (A) depicts the number of submitted tasks over time, while (B) shows the Gantt chart enriched with the anomaly dectection.



Source: The Author

We used the environment variable `STARPU_MAIN_THREAD_BIND` to control this thread's location and investigate its impact interference in the application performance. Given a set of `N` workers, we explored three configurations:

1. using all `N` workers and enforces the submission thread to compete for a core with one of the StarPU workers,
2. dedicating an exclusive core for the submission thread, using `N-1` workers, and
3. using `N-1` workers with the submission thread competing for resources.

We used the `flower_8_4` and the `TF17` matrices in the Draco machine and considered the different orderings and memory limitation values, making 30 repetitions for each configuration. We try to measure the impact of the submission thread in the execution. As expected, configuration 1 with `N` workers provided faster results, but the interesting comparison is between configurations 2, and 3, with the same number of workers. The results we observed favors configuration 2, with the submission thread with an exclusive core. However, the difference was kept below 1% (up to 722ms in the makespan).

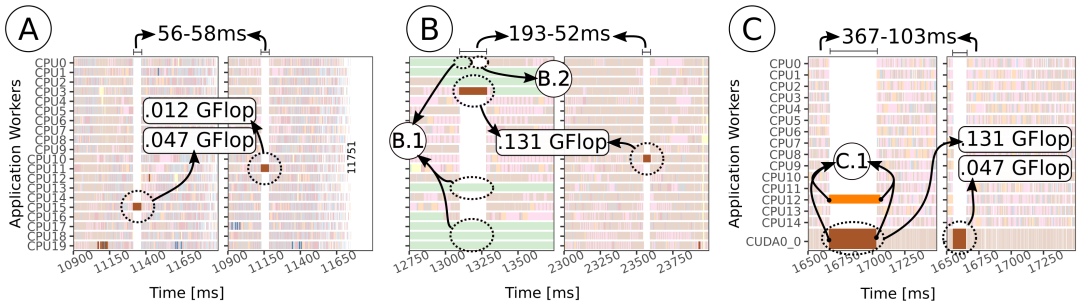
Nevertheless, when considering a 95% confidence interval to test the factorization time variation for these executions, the performance difference has no statistical significance for the cases we tested.

Hence, even that task submission is unavoidable and causes a small overhead for its assigned worker. The additional cost to do it is neglectable, making efforts to avoid it not worth it. As this effect was captured by our model and analyzed, we know we can control it to not appear in the next cases, easing the task to focus on other anomaly sources.

5.2.2 Case 2: Tracing Overhead

Another abnormal behavior we have noticed throughout the experiments represents a global idle time, spanning through all workers vertically at the same time. Figure 5.2 depict six different moments where this happened. We have observed this anomalous effect throughout many different configurations of machines, matrices, and application parameters, looking like a time-dependent phenomenon for some executions but random for others. Looking to the StarPU workers' state, when the cores are idling, they are in an overhead state (among other states such scheduling, fetching, sleeping). Frequently, only the anomalous task keeps executing while all others remain frozen, even if we have enough ready tasks to compute. In other scenarios, some other tasks that already started before this anomalous task were able to keep their execution, as depicted by the left side of graphic (B), in (B.1).

Figure 5.2 – Six examples of the vertically spread idle time, associated with an anomalous task.



Source: The Author

We detail these cases in Figure 5.2, zooming in the Gantt chart where these anomalies appear. In (A), we have the same execution with the `lws` scheduler on the left and `prio` on the right. We highlighted two tasks that occur at a similar time, which may

seem some time-dependent event. However, the duration of both anomalous region is very close (56ms and 58ms) despite their tasks having mismatching computational costs, which may indicate that this region's duration has nothing to do with the task execution itself. In Figure 5.2 (B) we illustrate two different moments of the same execution. In (B.1), we observe that tasks that started before the anomalous one in `CPU3` remain unaffected and continue running. However, at (B.2), we observe that all other tasks are delayed until the end of the anomalous task. We also noticed another anomaly later in the execution, coincidentally with the same computational cost, and yet, they imposed very different durations for the anomalous region (193ms and 52ms). This fact reinforces our point that the anomalous task duration defines such anomalous regions, but its duration does not necessarily represent that the tasks were executing. Lastly, in the rightmost plot of Figure 5.2 (C), we observe the same effect when running with GPU. Here we can notice two anomalous tasks. In (C.1), we identify that the task in the worker `CUDA0_0` started 370µs earlier than the task in `CPU12`, and it seems that it was that task that was preventing the others from starting. The CPU task ends 30ms later than the GPU task, and at this moment, the other tasks already started their executions. In (C.1), the anomalous region was responsible for up to 14% of the total worker idleness.

We hypothesize that this is not a task-dependent event but some behavior internal to StarPU. Because tracing is an invasive technique, depending on the problem structure, size, and granularity of the tasks, the amount of memory used to save the trace can be considerable. Thus, application tracing systems consider flushing the collected trace if the trace size meets some threshold during the application execution to avoid consuming too much memory. With further investigations, we discovered that those anomalous regions and tasks were related to the FxT tracing system, which was dumping the traces to the disk during execution. This happens when the trace buffer, with its limited size controlled by the environment variable `STARPU_TRACE_BUFFER_SIZE`, gets full. Hence, we now consider setting the value big enough and still fitting in RAM to avoid flushing the trace to disk during the application execution for our next experiments.

5.2.3 Case 3: Task Numerical Content

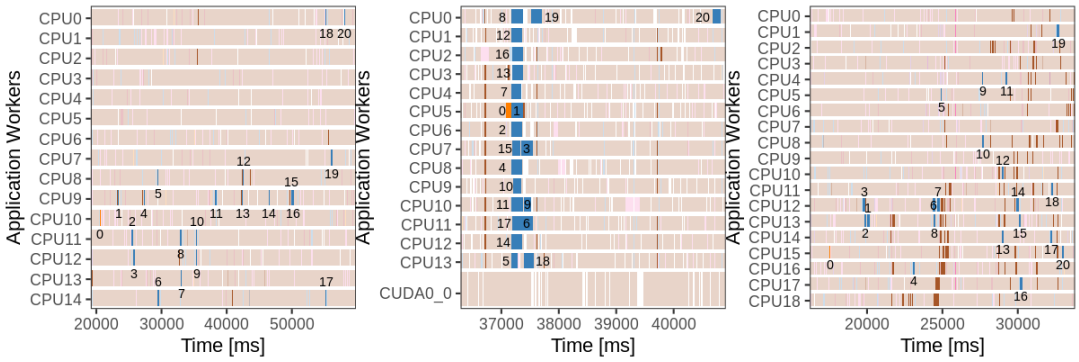
We noticed a somewhat curious effect in the experiments was the consistent presence of anomalies for certain matrix and ordering configurations, considering different machines, schedulers, and memory factors. For example, the matrix `ch8-8-b3` with the

Scotch ordering. Among the different executions, there were always specific tasks whose duration did not match the expected value according to the regression model.

The first step towards investigating those tasks was using the PAPI hardware counters to check if the cache misses explained the higher duration for those anomalous tasks compared to expected tasks with the same amount of GFlops. We found out while there is a positive correlation between the L1, L2, and L3 cache misses with the durations of the tasks, it does not explain the enormous difference of those tasks repeatedly classified as anomalies, for example, from ≈7ms to 65ms for a `geqrt` task, and from ≈9ms to 150ms for the `gemqrt` tasks. Neither the cache misses nor other hardware counters like the total floating-point operations explained their increased duration.

We also observed such an effect of repeatedly appearing anomalies for other matrices. For the execution mentioned earlier, we noticed the same 20 `gemqrt` task identifiers, preceded by an equally slow `geqrt` task, even in different machines and schedulers. To make sure, we even performed a sequential execution of the application, where again, the same tasks were classified as anomalies. Figure 5.3 represents this anomaly. The first two plots are executions on the Draco machine using the `lws` schedule. The first does not use GPU, and the anomalies appear spreading along with the execution, while in the second plot, using GPU, they appear all at the same time, despite de last `gemqrt` task. The last plot is an execution in the Hype machine that presents the same `geqrt` and `gemqrt` tasks classified as anomalies. We have also noticed that many `tpmqrt` anomalies in the figure are part of the same elimination tree node, which might be related to the same effect that occurs in those 21 tasks, but their variability is much smaller than the other anomalies. Furthermore, the quantity of `tpmqrt` anomalous tasks is not the same among the different execution as what happens for the presented `geqrt` and `gemqrt` tasks.

Figure 5.3 – Three different executions with the same 20 blue `gemqrt` tasks and the same orange `geqrt` task consistently classified as anomalies over the executions.



Source: The Author

As the cache misses failed to explain the increased duration for those tasks, our next step in the cause investigation was to analyze the blocks of the matrix those tasks use, analyzing the spatial position and even their internal content, which can enable or prevent some architecture-specific optimizations. This way, we dumped the content of the matrix blocks those tasks use to investigate their content. Our finding includes that compared to the expected task blocks, parts of the anomalous task block contains numbers really close to zero that underflows the floating-point precision and are represented as a subnormal or denormal number, according to the IEEE 754 standard for floating-point arithmetic. Those numbers exceed the smaller representable normal numbers, for which arithmetic operations can have different behaviors. When encountering a subnormal number, there is an increase in the latency of the processor operations like multiplication, divisions, and square root. However, we can control the way operations behave when denormal numbers arise. 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 this behavior (WITTMANN et al., 2015). Enabling these flags can improve application performance without losing solution quality if the application does not need denormal precision. This way, we recompiled the `qr_mumps` application enabling those flags, and those anomalies are gone.

This scenario occurred only for the Scotch reordered `ch8-8-b3` matrix. Hence, this finding can be interesting to the analysis of ordering algorithms in this numerical side effect aspect. Thus, in addition to application and runtime related anomalies, we were able to detect workload-dependent anomalies, more precisely depending on the ordering algorithm and the underlying architecture configuration.

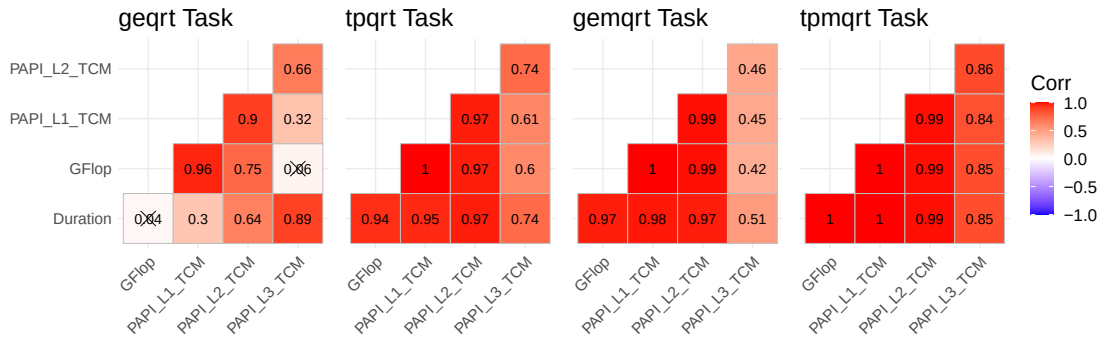
5.2.4 Case 4: High Cache Misses

Despite all the previously described cases, there are still anomalies that were not explained. Arising in different moments than submission peaks, not aligned with overhead states, and not repeatedly classified because of their specific block content. Our hypothesis is that they are caused by bad scheduling decisions that harm temporal and spatial locality, increasing the number of cache misses, rising task duration. We considered all the previously discussed configurations to isolate these anomalies in order to test our hypothesis and used the PAPI library to collect the tasks' number of L1, L2, and L3 cache misses. With this data, we try to relate the anomalous tasks explained by a higher

cache miss level to bad scheduling decisions.

Firstly, we verify if there was some correlation between the total miss number for all cache levels to the task duration increase. In general, the GFlops, L1, L2, and L3 cache misses have a strong positive correlation with task duration, with the correlation decreasing as we go to the lower cache levels. However, as we described in Section 4.1, there are cases where using the GFlop is not good enough to explain task duration, as well as the L1 and L2 misses. One example of a configuration with this behavior is with the `flower_8_4` matrix, previously illustrated by Figure 4.4 where we used the finite mixture model to fit multiple models over the data. For this example, we noticed that the huge variability for the `geqrt` tasks makes the GFfops variable not statistically significant to explain duration, considering a significance level of 0.01, as presented by Figure 5.4 in the combinations with the "✕" symbol. However, for this task and `tpqrt`, the L3 cache misses presented a high correlation value compared to other cases. Both of these tasks are memory-bound, which makes the cache misses more decisive in the task performance.

Figure 5.4 – Pearson's correlation matrix for the flower_8_4 matrix tasks.



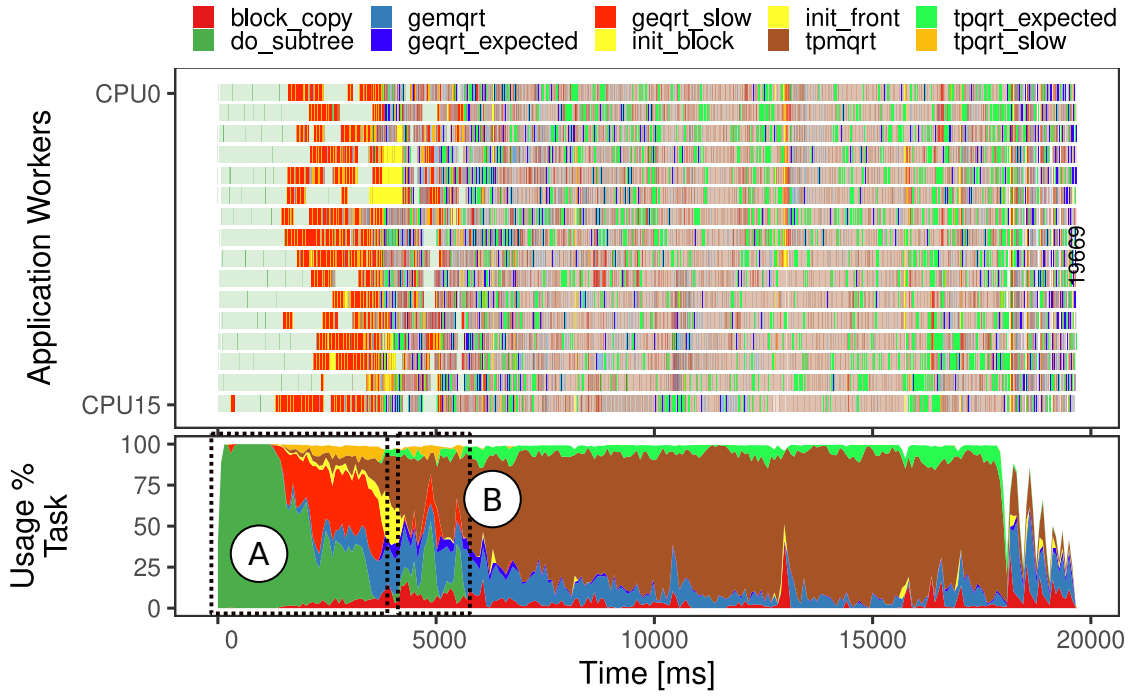
Source: The Author

We tried to investigate deeper those cases where the L3 misses explain the tasks increased duration by analyzing the scheduler behavior related to the spatial and temporal locality. We wanted to see if such tasks were scheduled far away from their last dependencies in both space and time. The trace files contain data handles of the tasks, which define the data blocks of size `mb` × `nb`, representing the matrix regions the tasks will work. Using this information, we tried to check when and where was the last use of an anomalous task data handles, considering if it was in the same CPU, same NUMA node, and how far in time it happened. This investigation led to no significant correlation between these parameters and cache misses per task. Such evaluation is hard to work because the data

handle states in the cache are uncertain, we can have parts of it in the cache and others in the main memory or other levels of cache. In addition, the cache policies interfere in how the architecture explores the temporal and spatial locality.

However, by clustering the tasks with the multiple regression model approach, separating them into slower tasks and expected ones, we used the cluster information instead of the anomalous classification to highlight tasks in the Gantt chart. By doing so, we observed that the slower tasks are related to specific moments. Figure 5.5 depicts these moments in (A) and (B), presenting the highlighted clusters in the Gantt chart on the top and the resource usage by tasks in the bottom. In both of these moments, we observe that we have many `geqrt` and `tpqrt` tasks happening simultaneously and also many `do_subtree` tasks. We can observe focusing in (B) that when we have peaks of resource usage by `do_subtree` tasks. We start observing the slow cluster tasks again. The slower tasks are related to the concurrent execution of the `do_subtree` tasks, which can reduce cache reuse of those memory-bound tasks in two ways: (1) because the `do_subtree` tasks can use a considerable amount of memory of distinct branches of the tree, flooding the cache with new data disturbing data reuse for other tasks. Moreover, (2), because `geqrt` tasks are the starting point for a node factorization, they are executed spatially far from each other. Thus they do not share matrix blocks with other `geqrt` tasks, which can also be the case for the `tpqrt` tasks.

Figure 5.5 – Cluster highlighting for the `geqrt` and `tpqrt` tasks.



Source: The Author

We analyzed this interference by running a sequential experiment using the same partitioning of the parallel execution. As consequence, the tasks do not suffer from interference from other concurrent tasks, enabling us to capture their expected behavior without the interference. We could also use our fitted regression model using the cluster with the expected tasks (i.e., the lower model in Figure 4.4 for `geqrt`) to extrapolate the results of a scenario without task perturbation, to estimate how much computational time we could save if we manage to avoid this undesired effect over tasks. Table 5.3 presents the comparison between the parallel and sequential execution for the total execution time for the tasks in the Draco machine. We observe that the `geqrt` tasks, when running in parallel, took 357% more time than the sequential version. However, the worst-case was with the `do_subtree` tasks, presenting a total increase of 12.3 times, which means that we need at least 13 CPUs working in parallel to beat the sequential execution, making the parallel execution of those tasks extremely inefficient. The reason why the `do_subtrees` are slower is the same as for `geqrt` and `tpqrt` because they are composed of the same memory-bound kernels.

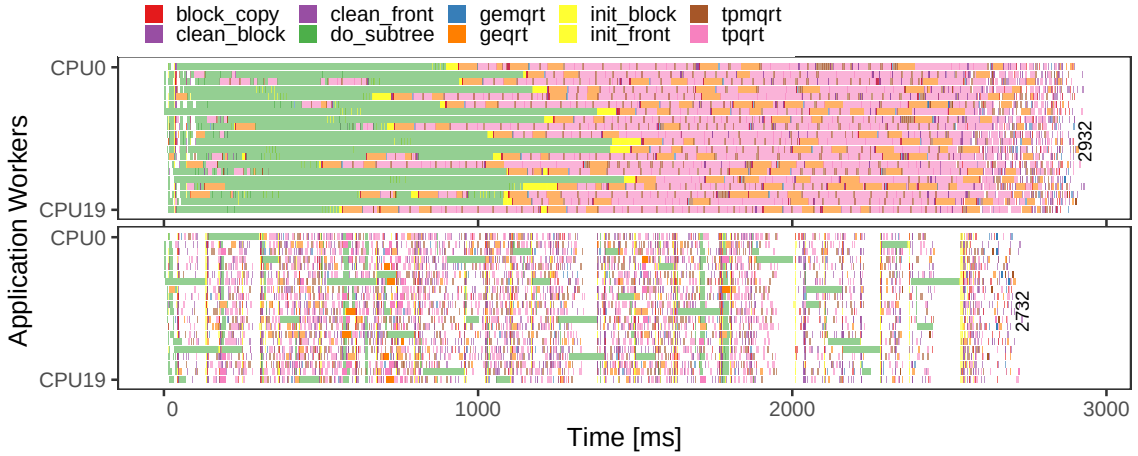
Table 5.3 – Total task time for parallel and sequential execution of flower_8_4 in the Draco Machine.

Task Type	**# of Tasks**	**Time parallel**	**Time sequential**	**slower factor**
`do_subtree`	427	48,83s	3,95s	12,35×
`geqrt`	974	17,82s	4,99s	3,57×
`tpqrt`	2.857	15,47s	11,71s	1,32×
`gemqrt`	5.338	32,38s	26,84s	1,20×
`tpmqrt`	22.973	172,23s	152,12s	1,13×
`block_copy`	6.265	11,83s	10,94s	1,08×

Another case corroborates with the later results and the L3 cache miss analysis. We noticed that for the `EternityII_E` matrix, we have a more extreme case related to the task interference and cache misses. In our experiments, the execution with memory limitation presented a smaller makespan than when running without memory limitation. In this case, a simple look at the Gantt chart does not reveal anything because the parallelism occupies all resources with little idle time, leading to no visible problems, as the Figure 5.6 represents.

However, when we look at the relation between the task duration the theoretical GFlops in Figure 5.7, we observe that there is high variability in duration and that the tasks of the unlimited memory execution (orange) are clearly above the tasks of the limited execution (green). As in the `flower_8_4` case, the `EternityII_E` execution has many concurrent `geqrt` and `tpqrt` tasks when running with no memory limit, causing

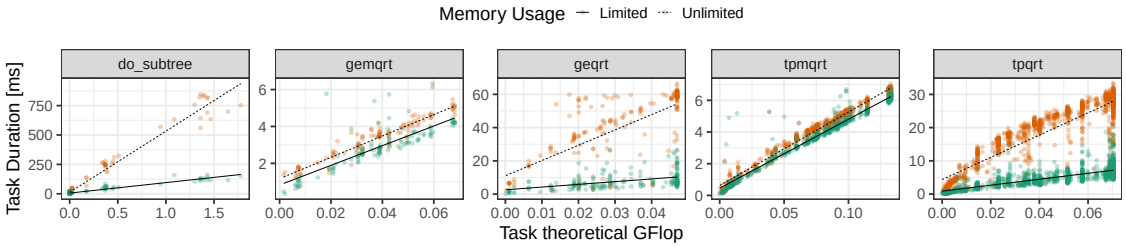
Figure 5.6 – Unlimited memory consumption (top) and limited (bottom) Gantt chart comparison for the EternityII_E matrix in the Hype machine.



Source: The Author

more L3 cache misses. Highly inflating task duration as we can compare by looking at Figure 5.7 with the L3 cache misses in Figure 5.8. The Table 5.4 reveals the differences between the two executions. This execution represents a specific scenario where the `geqrt`, `tpqrt`, and `do_subtree` task sum up more than 60% of the total computational weight for the factorization, having fewer `gemqrt` and `tpmqrt` tasks. As we mentioned before, these three task performance is more sensitive to cache misses, while for `gemqrt` and `tpmqrt`, we observe that they had a slight change in both cache misses and duration. In contrast, the other tasks have a much higher duration. Their cumulative execution time was from 4 to 5 times greater, which is explained by the higher number of cache misses, and explain why we observe an almost fully filled Gantt chart with no idle time.

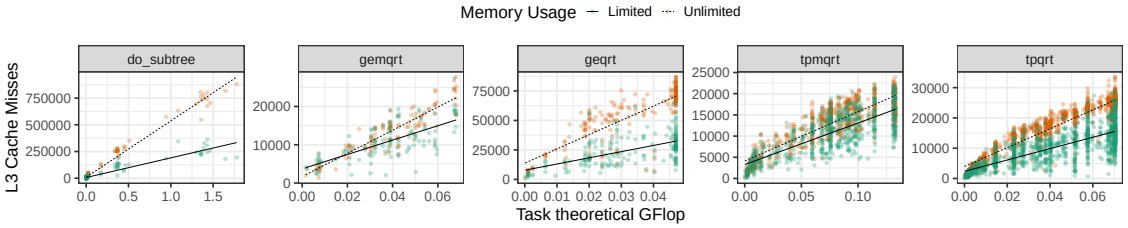
Figure 5.7 – Comparing the task duration between unlimited and limited memory consumption executions for EternityII_E.



Source: The Author

In opposition, when running with limited memory consumption, the workers are idle most of the time, but the execution time was reduced by up to 7,2% in the Hype

Figure 5.8 – Comparing the L3 cache misses between unlimited and limited memory consumption executions for EternityII_E.



Source: The Author

Table 5.4 – Total task time for limited and unlimited memory usage of EternityII_E in the Hype machine.

Task Type	# of Tasks	Weight	Time unlimited	Time limited	Slower factor
`geqrt`	199	4,63%	8,13s	1,57s	5,16×
`do_subtree`	292	20,20%	18,36s	4,25s	4,31×
`tpqrt`	1.206	37,79%	22.92s	5,77s	3,97×
`gemqrt`	116	3,18%	0,33s	0,39s	1,18×
`tpmqrt`	633	34,18%	2,60s	2,36s	1,10×

machine. This happened because the limitation constrained the number of concurrent `do_subtree`, `geqrt`, and `tpqrt` tasks. Such slowdown due to the parallelism is referenced as locality efficiency by (AGULLO et al., 2016). While such interference effect is expected for parallel task executions, some techniques could better handle it to improve overall performance. These techniques involve smart matrix partitioning and mapping to resources or some interference-based or cache-aware scheduling to keep this inefficiency under control.

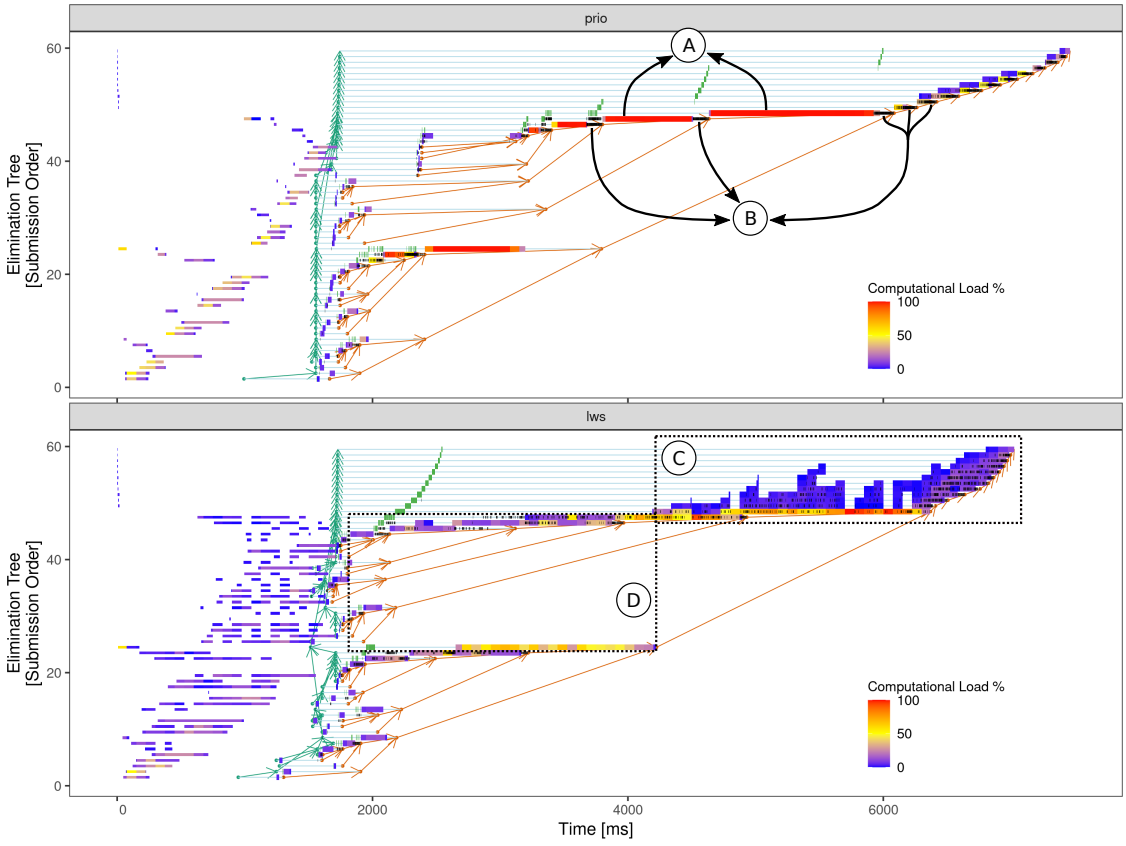
5.3 Analyzing Performance Considering Runtime and Application Factors

This Section evaluates our performance visualization techniques analyzing different cases for the parallel sparse factorization application `qr_mumps`. We consider factors at the runtime level like the scheduling policy and application-level parameters such as the fill-reducing ordering, memory constraint, and elimination tree pruning and amalgamation techniques. Those case studies represent frequently applied analysis in task-based applications and the multifrontal method for sparse factorization. They also serve as a validation process of our techniques when combined, enabling performance analysis beyond the classical Gantt charts, aligning performance data to application data structures.

5.3.1 Comparing Different Schedulers Behavior

We can use the visualization of the elimination tree structure over time presented and described in Section 4.2.1 to study scheduler behavior in the tree traversal, considering task priorities and heterogeneous platforms. For example, the Figure 5.9 presents a comparison between the `lws` and the `prio` schedulers for the `degme` matrix reordered with Metis and unlimited memory usage in the Tupi machine, using the resource utilization as the gradient colors. We can notice that the `lws` execution (7.02s) is faster than the `prio` execution (7.46s).

Figure 5.9 – Comparison between prio and lws schedulers for the degme matrix in the Tupi machine.



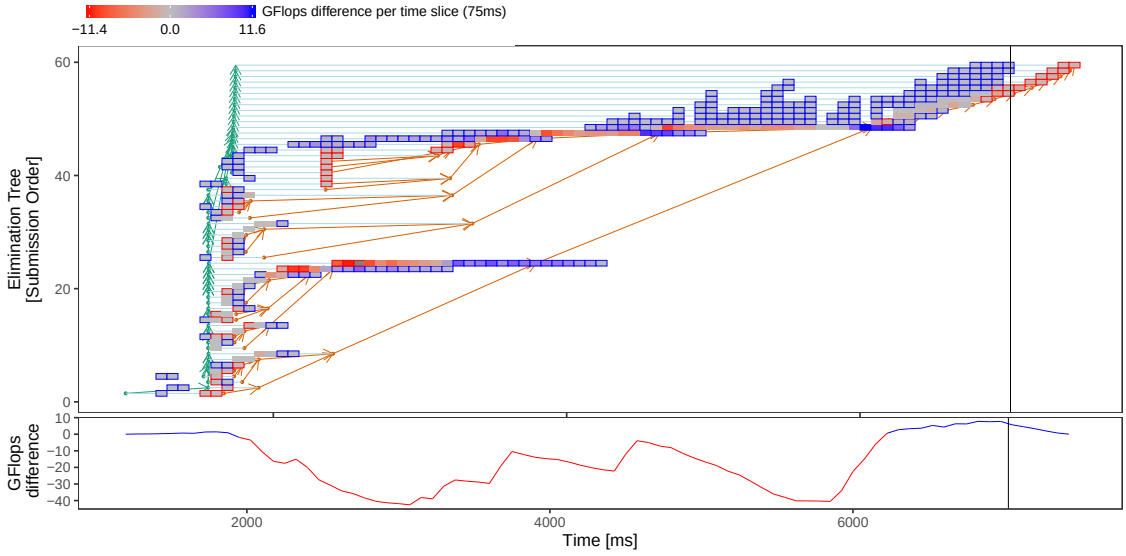
Source: The Author

We observe that the `prio` scheduler's behavior is to focus on one elimination tree node at a time, as pointed in (A). This behavior occurs because `qr_mumps` assigns decreasing priorities for later submitted nodes, forcing the `prio` scheduler to consider this submission order and sort the tasks in its unique central ready task queue, creating this effect of lower tree parallelism and higher node parallelism. Another particular behavior that we notice for `prio` is that it has clear computation and communication patterns as

pointed by (B). The `block_copy` tasks are postponed to the end of a tree node after all its computations ended because of its lower priority, delaying parent and child communication.

Restricting computations to only one or few tree nodes simultaneously, like in the `prio` case, may improve spatial data locality. However, its restriction in the communication reduces the availability of the tree and interlevel parallelism, compared to the interlevel parallelism achieved by `lws` in (C) and the tree level parallelism in (D). A more direct comparison of both executions using the compare tree panel presented by Figure 5.10, where we observe the non pruned nodes GFlops throughput difference of the faster execution with `lws` in blue, and the `prio` slower execution in red. We observe that the focus on only one node computation at a time indeed provided more performance when looking at the computed GFlops difference in the bottom of Figure 5.10. Nevertheless, the communications at the end of the nodes abruptly slow the execution (B).

Figure 5.10 – Using the compare tree plot to visualize where the GFlops throughput differ.

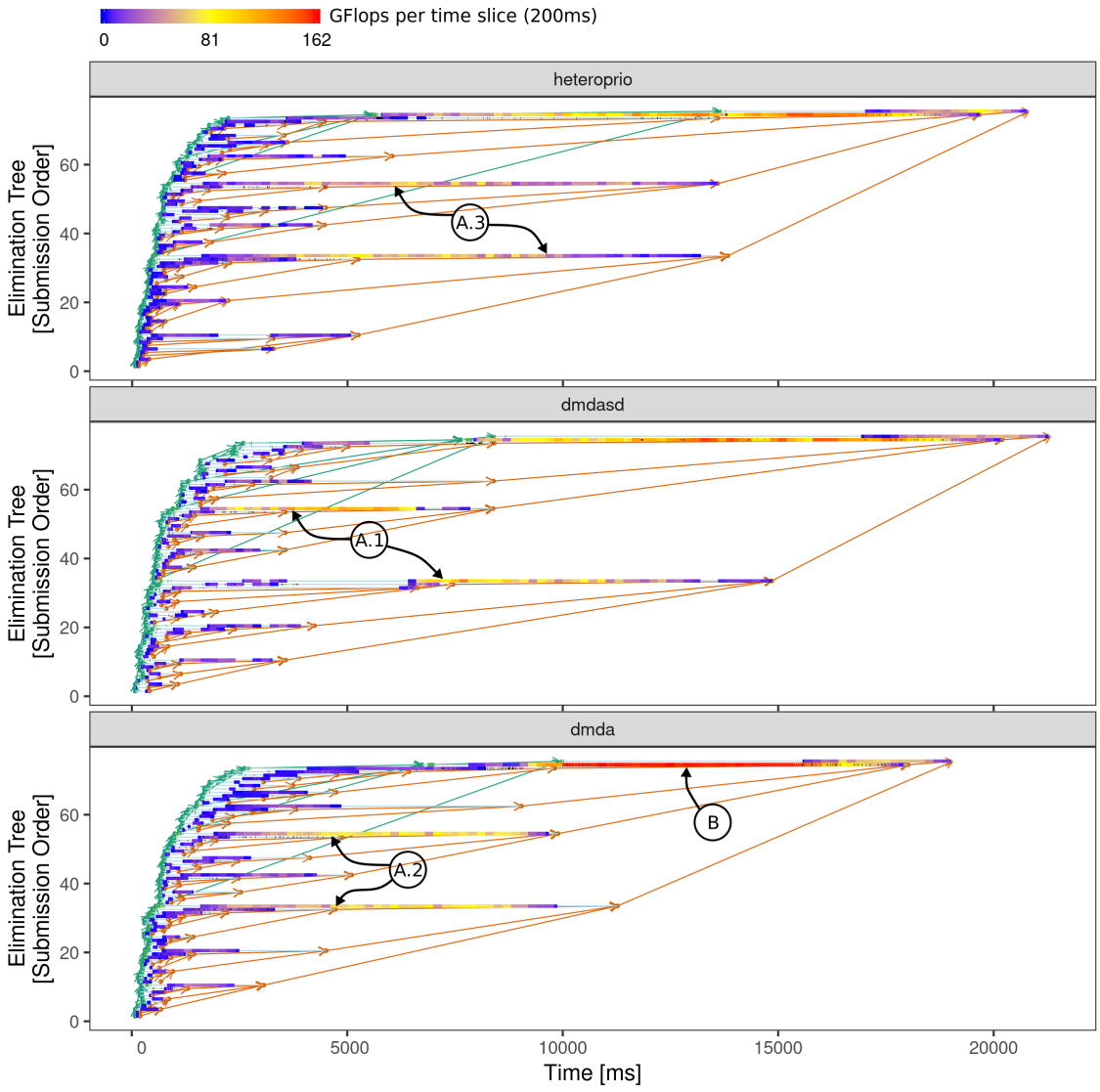


Source: The Author

However, the fact that the `lws` scheduler explored more tree and interlevel parallelism, making node computations last longer, as pointed in (A), was beneficial to performance. This was good in this scenario because the lack of tree parallelism at the end of the application is compensated by the interlevel parallelism, which does not happen for `prio`, as pointed in (C). This case depicts an excellent example of the alignment of the application data structures with performance visualization, which clearly illustrates what is happening and can help developers evaluate the impact of modifications towards improving application performance.

Comparing an execution using GPU, we have the `TF17` matrix reordered with `Scotch` and memory limitation in the Hype machine. Figure 5.11 represents the elimination tree plot using the GFlops throughput. We used the GFlops metric because the resource time utilization does not fit well when we have heterogeneous resources as the CPUs and GPUs have different GFlops throughput. With this, we can compare the three different schedulers. In the nodes pointed by (A.1) and (A.2) for `dmdasd` and `dmda`, we observe a consistent increase in the GFlops throughput, gradually increasing and reducing while the computations reach the node end. For the `heteroprio` scheduler on (A.3), we can notice that the node computations spread for longer periods. This way of exploring the tree, focusing computations in different areas, is the scheduler's signature and how it computes the tree.

Figure 5.11 – Comparing heteroprio, dmdasd, and dmda schedulers for the TF17 matrix in the Hype machine.



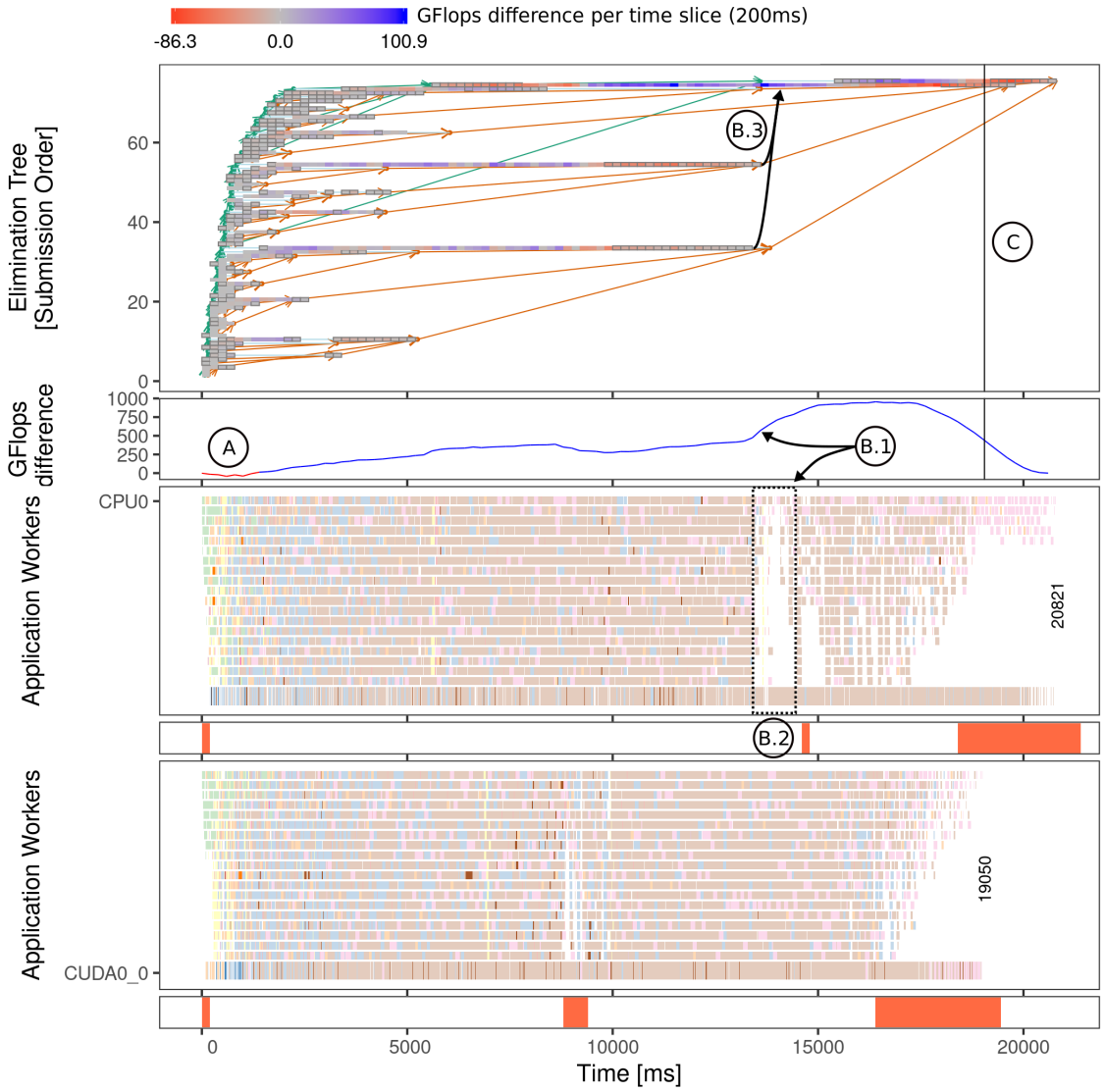
Source: The Author

In (B), we notice that the dmda scheduler was able to keep the peak performance represented in the color gradient for an extended period, which could have led to its reduced makespan, being 10,65% faster than dmda, and 8,5% faster than heteroprio. However, looking at the cumulative time spent for all tasks, the heteroprio and the dmdasd reduced the total computing time by 3,78%, and 12,45% compared to dmda. This difference reflects the focus on choosing the faster resource for each task, which all three schedulers consider, but considering different aspects, causing different idleness over the resources: 11,8% for dmda, 22,4% for heteroprio, and 31% for dmdasd. Nonetheless, the application final makespan is what matters in the end.

Despite these differences, we can also merge two different trees for comparison, highlighting their differences where performance deviates. We choose the heteroprio and dmda schedulers to compare for this execution in the Figure 5.12 because they provided better performance than dmdasd. We can observe in (A) that the heteroprio scheduler has a slightly better start than dmda. However, this advantage quickly changes for dmda, increasing slow and consistently throughout the execution until the moment represented in (B.1), where there is a steeper increase in the computed GFlops difference for dmda. We relate this increase to the increase in the idleness depicted by the area pointed by (B.1). However, the cause of this idleness was not due to lack of ready tasks to compute, as we can observe by the lack of ready tasks panel in (B.2), which highlights moments with fewer ready tasks than the number of workers in orange. The idleness in heteroprio was caused by its decisions to schedule all those available tasks to the GPU because of the higher priority the tpmqrt tasks have for GPU. This heteroprio decision not resulted in better performance than what dmda provided, as pointed by (B.3). By observing the tree structure, we can see that the dmda scheduler better handles the nodes' transition in (B.3) to the other node in terms of data prefetching and resource utilization. Lastly, the vertical line in (C) delimits the end time of the faster execution.

In the experiments, we also have cases where heteroprio provided better performance than dmda or dmdasd. By checking these cases, we noticed that the reason the performance was better for heteroprio is the same that reduced its performance in the last presented case. It only scheduled all gemqrt and tpmqrt tasks in the GPU, which somehow not created communication delays due to data transfers and provided better performance.

Figure 5.12 – Comparing the tree computation difference (top) and Gantt chart for heteroprio (center) and dmda (bottom) schedulers for matrix TF17.



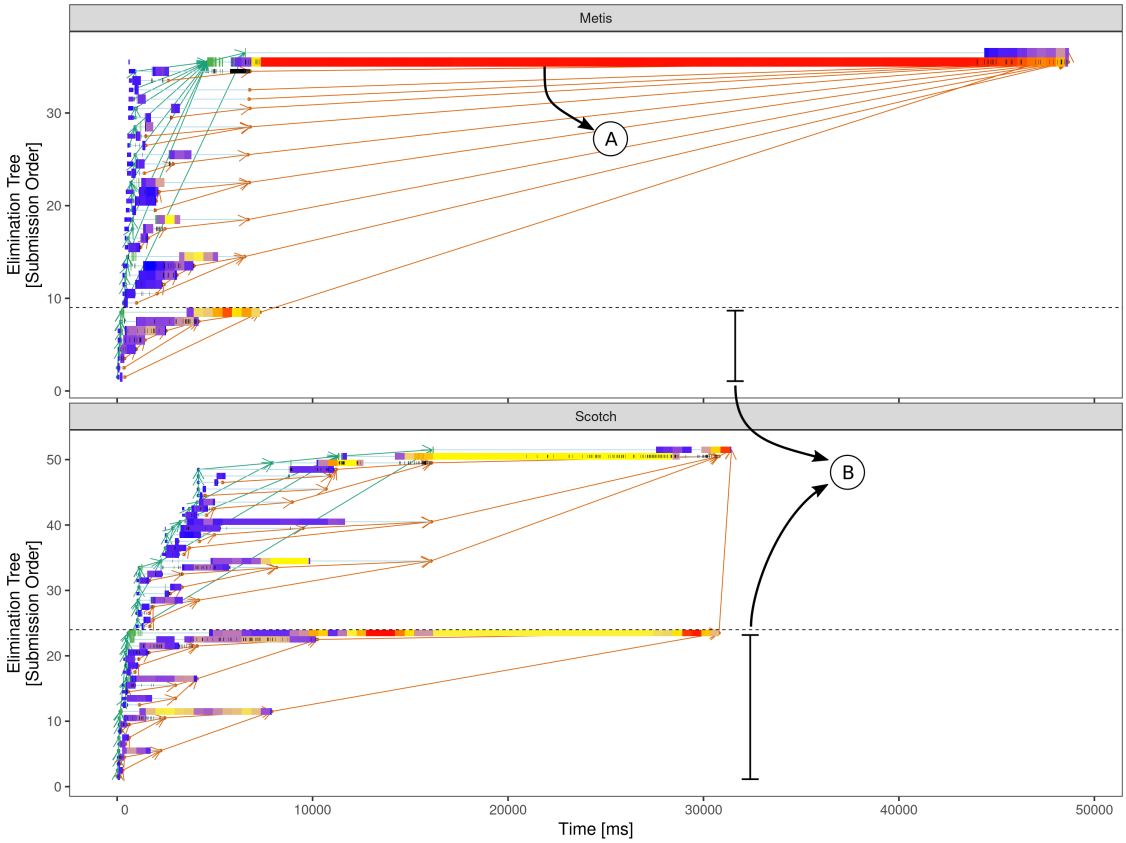
Source: The Author

5.3.2 Comparing Fill-Reducing Orderings

A determinant factor in the execution of sparse factorization methods that define the tree structure is the fill-reducing ordering operation performed in the analysis phase. In our experiments, we explored the `Scotch` and `Metis` fill-reducing ordering packages. The Figure 5.13 presents the tree structure plot for two executions with the same parameter configurations, except for the ordering. In general, the computation time difference between executions is caused by a difference in the total amount of GFlops for the factorization imposed by the fill-in and the reordering algorithm. Despite the difference in the total computational cost, we can observe that the elimination tree structure

is significantly different. We can observe that the tree organization for `Metis` produced a poorly balanced tree, with few small nodes and a huge node that concentrates most of the computations, pointed by (A). Comparing to the `Scotch` produced tree, we have a more balanced tree, having more nodes, which better divided the computational load. Also, as highlighted in (B), where the dashed lines cut the elimination tree at level two, we observe the different sizes of subtrees, with a very small subtree for `Metis` and two subtrees of very similar size for `Scotch`. Beyond these differences, they have a similar number of pruned nodes, producing few `do_subtree` tasks and a small node at the root. Such visualization can help application developers analyze performance relating to the tree structure provided with a specific ordering algorithm.

Figure 5.13 – Comparing Metis and Scotch ordering for ch8-8-b3 matrix in hype Machine.



Source: The Author

However, the fact that a specific ordering was able to reduce the total number of floating-operations does not necessarily mean that the application's performance will automatically be better. Because the tree structure combined with other application parameters like memory limitation can dictate how the execution unfolds, respecting the tree dependencies and memory threshold, different trees may have a more restrictive sequential memory peak. Furthermore, the analysis phase cost can sometimes exceed the

performance that was gained in the factorization.

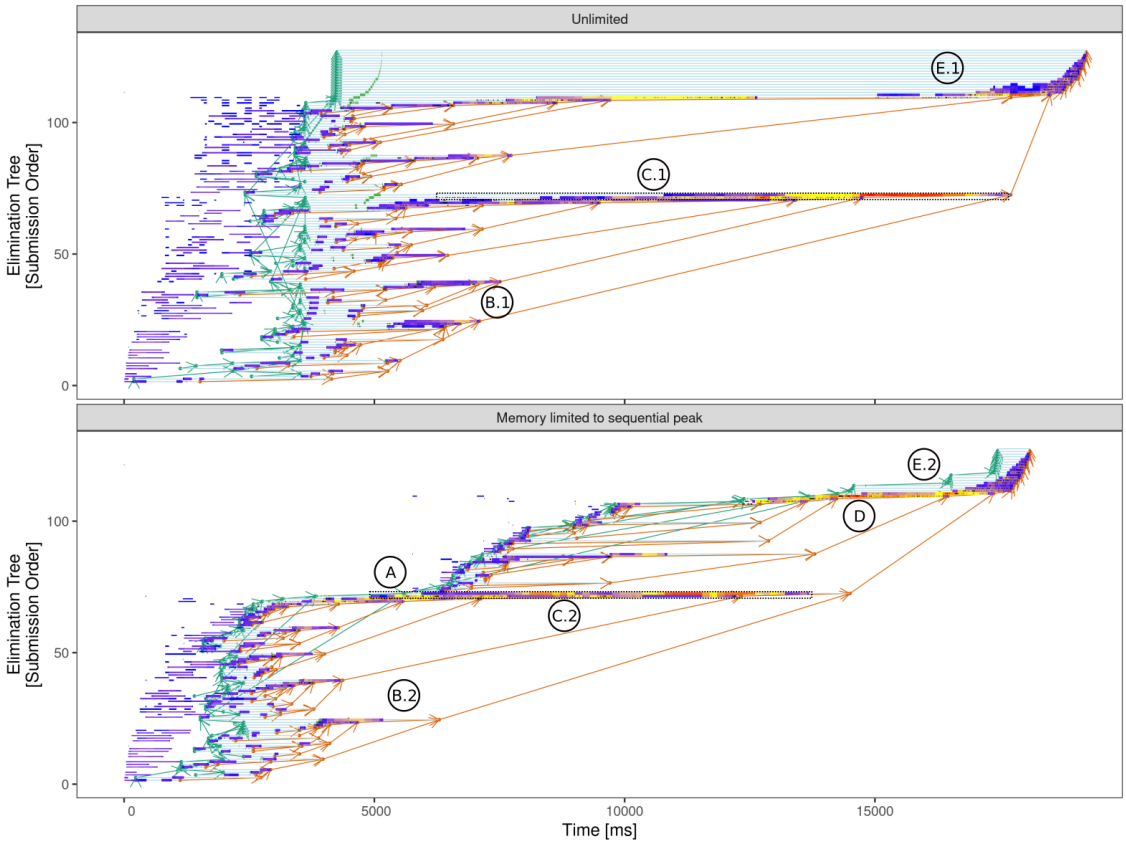
5.3.3 Comparing Memory Usage Threshold

For all experiments, we have noticed that the `qr_mumps` application was able to keep performance while reducing the memory footprint, except for those cases where the memory threshold was too limiting. Furthermore, there were also cases where the memory limitation improved the application performance because it reduced interference between tasks. The elimination tree panel reveals how the memory limitation impacts in terms of the tree nodes exploring by the scheduler.

Figure 5.14 shows a case where performance was slightly improved by limiting the memory consumption to the sequential peak for the `flower_8_4` matrix using `Metis` in the Hype machine using only CPUs and the `lws` scheduler. While in the plot with unlimited memory usage, we see that the entire tree is allocated early in the execution. For the limited case, the allocations were postponed to the last moments of the factorization. The allocation of different portions of the tree in different moments impacts the available tree-parallelism. In (A), we can notice that by around 5 seconds of execution, half of the tree intermediary nodes were not touched yet by the execution with memory constraint, while the other has started and even finished computing many other nodes.

The higher tree-parallelism available provided by the higher memory consumption causes the nodes to have a longer computation span, as we observe n (B.1) and (B.2), where the highlighted nodes computations end later for the unlimited memory configuration. This was caused by the smaller computational focus on those nodes, as the scheduler had many other tree nodes to explore, as highlighted by (C.1) and (C.2). Also, the moment in (D) is different from the unlimited memory case because the tree nodes' exploration order differs due to memory availability and their earlier or late availability. Of course, this also depends on the scheduler used, but we compare an execution with the same scheduler in this case. Finally, in (E.1) and (E.2), we can observe that the last tree nodes' computation started earlier for the unlimited memory case. However, even with this, the application performance with memory limitation was not degraded. In fact, it was faster because of the lower competition between the `do_subtree` tasks at the start of the unlimited memory case. However, even if the application performance does not change, this kind of visualization helps to understand the impact of such a parameter in the application execution regarding this tree structure.

Figure 5.14 – Limited and unlimited memory consumption for Metis ordering in flower_8_4 matrix on the Hype machine.



Source: The Author

5.3.4 Improving Performance by Reducing Task Interference

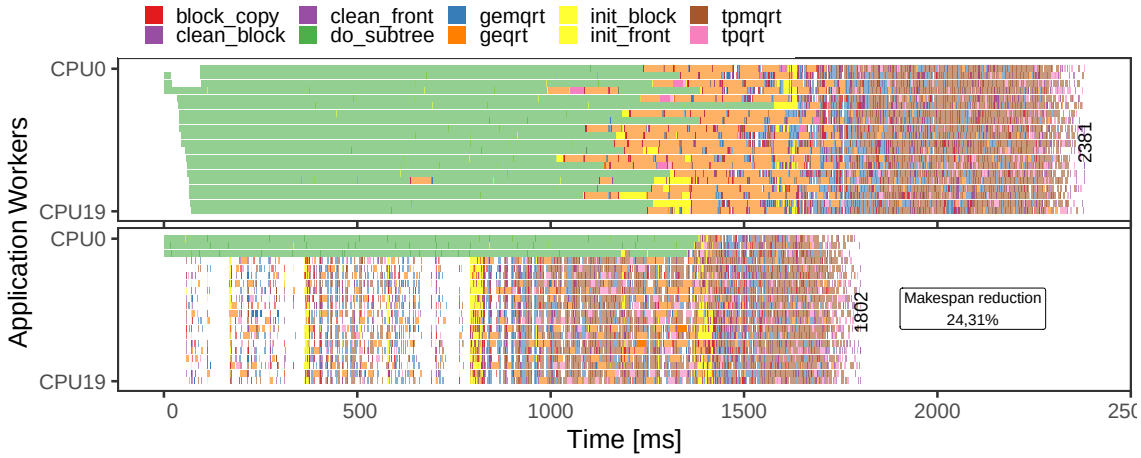
As we have been observing throughout our experiments a constant presence of performance degradation due to running certain tasks in parallel as in the `flower_8_4` matrix case presented in Figure 5.5. Furthermore, we noticed cases where memory restriction reduced the parallelism but improved task efficiency in a way that the execution ended up being faster than without memory limitation as presented in the end of Section 5.2.4 by Table 5.4 for the `EternityII_E` matrix.

Based on our observations, we propose a naive solution to mimic the beneficial effect of spreading the `do_subtree` tasks along with the application execution. We created a different scheduling context for the `do_subtree` tasks, reducing the number of available workers to these tasks. This controlled context helps to reduce the `do_subtree` tasks interference between themselves and the other memory-bound tasks. We provided two environment variables, one to control the tree pruning according to the number of workers, and the other to control the number of workers the `do_subtree`

scheduling context will have.

From another set of experiments considering this scheduling context difference we noticed cases where performance was improved by up to 24% using this technique. The Figure 5.15 shows the case of the `karted` matrix in the Hype machine using the `Scotch` ordering. We observe that half of the makespan of the execution that does not use the scheduling context is for `do_subtree` tasks, and other good portion is for `geqrt` tasks.

Figure 5.15 – Gant chart comparison for the karted matrix in the Hype machine not using the do_subtree scheduling context(top), and using the restricted scheduling context (bottom).



Source: The Author

While this simple strategy improves performance in scenarios where we have many concurrent memory-bound tasks, there are more sophisticated ways of controlling the interference effect. For example, for the Hype machine, where we have two NUMA nodes, we can map different branches of the tree to reside on only one NUMA node, as some distributed memory versions of the multifrontal method consider (GUPTA et al., 2016). Thus, we improve locality and better utilization of the shared cache within a NUMA node. Other strategies at the runtime level can also be explored, like considering inter-task interference and cache-aware scheduling strategies (GUO et al., 2020), which could be implemented in the StarPU library.

5.4 Discussion and Summary

In this chapter, we have seen that the experiments revealed some particular behaviors of the application and the runtime. We analyzed the presence of anomalous tasks in

different scenarios related to runtime, tracing, hardware configurations, and application tasks. Although we can avoid most anomaly sources, some of them remain hard to fix, like in the case of task submissions. However, in the case of task submission, the impact was neglectable, so the efforts to reduce or avoid this cost would provide small or no performance gain. In contrast, some other anomalous behavior significantly impacted the tasks' performance, affecting the overall execution performance. We identified this case while trying to model the task's behavior, using multiple models and clustering the tasks observations.

The use of multiple models for task representation revealed a more profound performance problem. We observed that the tasks that were part of the slower model happened in specific moments in the execution and certain task types. Using the PAPI library, we later related the increase in duration to an increase in the L3 cache misses for those tasks caused by concurrent task interference. Based on some experiments' side effects, like cases where reducing the available memory improved performance, we tried to enhance performance by spreading the `do_subtree` tasks along with the execution (see Section 5.3.4), which proved to be useful for cases where we have many `do_subtree`, `geqrt`, and `tpqrt` tasks. However, this is a naive approach. More robust techniques can be used in future works to improve the application, like efficiently mapping the tree structure to computational resources and considering scheduling policies that account for task interference.

We also presented use cases of the elimination tree computation panel captured the application and runtime behavior, which helped us analyze and understand the performance of different configurations in detail. Those proposed panels, mainly the elimination tree one, bridges the gap between the application data structures, tasks, and performance. With those panels, we can study the impact of task priorities, platforms, and other application parameters in a very detailed way, providing the necessary understanding and possibly insights into improving performance. Even the smaller performance gains can represent expressive improvements in real applications, where the factorization process is called repeatedly. In the next chapter, we will present the use and performance analysis of `qr_mumps` in a real application.

6 OPTIMIZING AND USING SPARSE SOLVERS IN RAFEM

This chapter provides a study of the RAFEM application towards its acceleration using different sparse solvers, including the previously studied task-based `qr_mumps` solver and numerical analysis of the solution. Section 6.1 presents the steps towards the application optimization, focusing on the two most costly steps: assembling the equation system and solving it. Section 6.2 describes an evaluation study for the numerical solution quality provided by the different solvers adopted in the optimization. Section 6.3 shows a detailed analysis of the application performance and the `qr_mumps` solver performance in the context of RAFEM. Section 6.4 closes the chapter discussing possible techniques to improve the analysis and other performance optimizations.

6.1 RAFEM Application Optimization Strategies

The RAFEM application, earlier described in Section 2.3.2, uses the widely adopted FEM strategy to represent the RFA procedure computationally. Such a standard method like FEM is widely studied, focusing on improving its performance. There are common steps in a FEM application that we should look for when we are concerned about performance. The two steps contributing more to the computational weight are the **assembly** and **solve** steps. During the first one occurs the assembling of all mesh element equations, which are later combined into the global system of equations. This system represents the whole problem, and it is the unknown values in this system of equations that we need to solve, representing the other most costly step. The equations assembly commonly produce large and very sparse matrices. Therefore we need to consider specialized algorithms like sparse or iterative solvers. Otherwise, the fill-in cost overhead would be catastrophic for performance, generating new coefficients over all the matrix. As the simulation in RAFEM calculates the changes over time, it repeatedly executes these two steps, advancing in the simulated time by small time-steps until it reaches the desired final simulation time. Thus, by improving these steps, the benefit of performance gain spreads along the application execution time.

We use the sequential application version as the baseline comparison for both speedup and numerical validation experiments. The machines that we used in the RAFEM experiments are the same described in Section 5.1 by Table 5.2. Table 6.1 describes the two workloads used in the RAFEM application, representing two different finite element

mesh resolutions and the matrices they generate. The test cases generate two square and sparse matrices represented in double precision.

Table 6.1 – FEM meshes used in the RAFEM experiments.

Mesh	# Nodes	# Elements	Matrix size	Nonzeroes	Sparsity
A	3.548	18.363	7.096×7.096	189.462	0.0037%
B	8.364	44.811	16.728×16.728	450.451	0.0016%

To collect performance data, we manually instrumented the code using the ScoreP 6.0 library (KNÜPFER et al., 2012). This user-defined tracing allows us to collect data only from code regions that we judge interesting, collecting simple measurements without generating too much overhead in application execution time caused by tracing intrusion (MEY et al., 2011). We focused on three main events in the application, composed of smaller, more specific regions. Those main events are the assembly phase, the solve phase, and the memory copies between CPU and GPU. The application tracing enable the characterization of the application's computational cost in these different regions, helping decide which part we should optimize and estimate how much we expect to reduce the application time. The ScoreP tool records application trace in the OTF2 format. We used a conversion tool named `otf22csv`[1] to transform the OTF2 trace data into the CSV format, enabling the analysis with modern data analysis tools like the `R` language. To obtain the performance metrics to calculate mean execution time and speedup considering the total computation time on a sequential system. We used a full factorial design considering three factors machine (Draco, Hype, Tupi), workload (mesh A, mesh B), and solver(Original, `MAGMA`, `cuSOLVER`, `qr_mumps`), making 30 repetitions for the smaller workload A and 10 for the bigger one B. In total, we performed 480 executions to obtain the performance metrics. Application-specific parameters like the total simulation time which was fixed to 15 minutes, and the RFA generator input power as 15.

6.1.1 The Assembly Step Parallelization

The assembly step parallelization was already studied in a previous work (SCHEPKE; MILETTO, 2020), which includes GPU support to accelerate the assembly step. A set of CUDA kernels is responsible for assembling the individual element equations and later grouping them in the Compressed Sparse Row (CSR) format, considering the bound-

[1] otf22csv tool <https://github.com/schnorr/otf2utils>

ary equations. The assembly can be seen in two main phases: assembling the element equations and combining them. The first phase is an embarrassingly parallel problem, and each GPU thread was responsible for one element. In the second phase, we need to solve the race conditions of combining all these equations in the global matrix in parallel. Hence, GPU threads were mapped to assemble specific matrix regions, similar to a graph partitioning approach.

At the end of the assembly step, it provides the equation system matrix in the CSR format, commonly what we need to use as input for parallel sparse solver libraries. However, this assembly solution represents a naive approach. Other works consider designing specialized algorithms and data structures aligned to GPU programming specific concepts like dynamic GPU memory allocation, thread-block shared memory (SANFUI; SHARMA, 2017; KIRAN; SHARMA; GAUTAM, 2019). Thus, there is a potential improvement to explore in the assembly part that we can later consider throughout the optimization process. However, our focus on acceleration was in the most costly part: solving the generated system of equations.

6.1.2 Incorporating and Tuning Parallel Solvers

This same previous work (SCHEPKE; MILETTO, 2020) that parallelized the assembly step also used a parallel solver to speed up the application. It used the `MAGMA` library for the solution phase, using an iterative solver based on the Restarted Generalized Minimal Residual Method (GMRES(m)), but did not explore different solvers and computational environments. In this work, besides the `MAGMA` solution, we also consider the sparse QR factorization from `cuSOLVER`, handled entirely in GPU, and the task-based sparse QR factorization from `qr_mumps`. All of them use CUDA version 11.1.0. The version of the `MAGMA` library used is 2.5.2, and the `qr_mumps` version was compiled with the same libraries versions previously described in Section 5.1, with small changes in its C-Fortran interface to get the computed column permutation.

The CSR generated matrix fits as input for the `MAGMA` and `cuSOLVER`, but the matrix type for `qr_mumps` needs to be in the Coordinate format (COO). We then adapted the assembly process to create a COO matrix as well. Compared to the GPU-only solvers, we need to perform an extra memory copy in the `qr_mumps` case to bring the global matrix assembled in the device memory (GPU memory) to the host memory (RAM) for every corrector step iteration (innermost loop in Figure 2.7). After obtaining the solu-

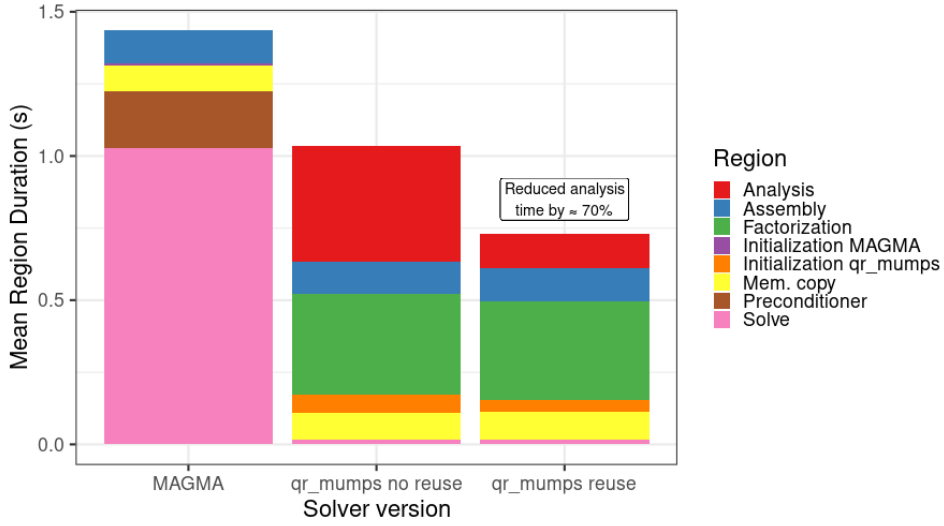
tion, it needs to be both in the host and device memory for the convergence check of the corrector step in CPU and the GPU's next matrix assembly, making the same number of memory transfers for this data in all three solver configurations.

Having the three parallel solvers working, we started to study the parameters that affect their performance, tuning them to provide better performance. The matrix reordering algorithm is crucial for direct sparse solvers to enhance parallelism and control the fill-in effect. Thus, for `qr_mumps` and `cuSOLVER`, we explored the `Scotch` and `Metis` reordering algorithms, where `Metis` provided better acceleration for the matrix factorization. For `qr_mumps`, as it performs a 2D tiled factorization, we need to specify the `mb`, `nb`, and `ib` parameters, which we set to square blocks of size 320 because it provided slightly better performance than using the default value 256, and the internal block size default value was used, which is 32. We explored different values for these parameters, but none provided better performance in the GPU for `qr_mumps` because the workload was too small. Thus, we only consider using CPUs in `qr_mumps` along with the `lws` scheduler.

Studying the problem, we noticed that the matrix structure remains the same throughout the simulation. This structural steadiness allows us to reuse the fill-reduction column permutation completed in a previous step, computing it only once. This reuse reduces the time spent in the analysis phase that precedes the factorization, reducing the total time for the application solve step. The `qr_mumps` application enables users to pass a user-defined column permutation array to rearrange the matrix instead of calling the `Metis` library again. Unfortunately, the `cuSOLVER` function call does not enable passing a user-defined reordering as `qr_mumps` do. Figure 6.1 demonstrates the impact of this optimization according to our trace data by reducing the total iteration time for the inner corrector loop, avoiding recomputing the column permutation in the analysis phase. However, for the `cuSOLVER` library, the solver function call does not allow the user to pass an already computed column permutation. In an iterative solver such as the `MAGMA` GMRES(m), the analogous step of the analysis is the matrix preconditioner. This step that precedes the solution improves the matrix convergence properties but can take a considerable time in the iteration, as observed in the figure for the `MAGMA` solver. Thus, using techniques that allow reusing preconditioners (SINGH; AHUJA, 2020) between problems with similar structure would benefit performance in this case.

For the `MAGMA` library, we used the iterative method GMRES(m). This method has the `m` parameter that impacts the solver convergence rate, directly affecting its per-

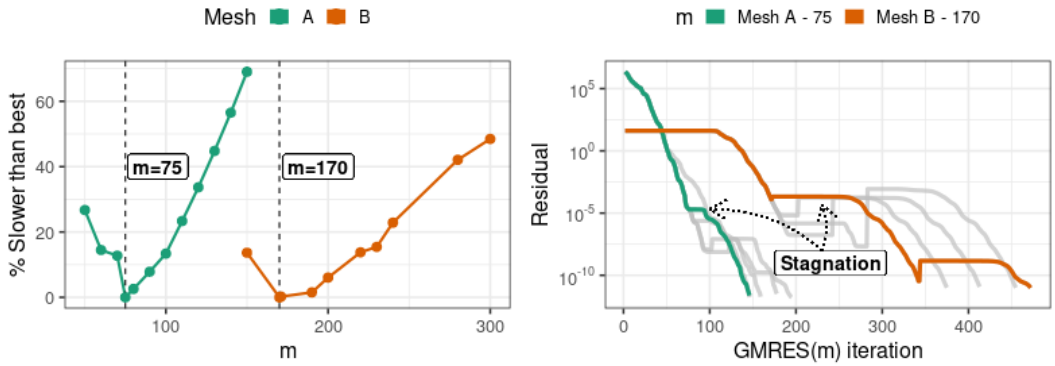
Figure 6.1 – Average time for different code regions and solver versions for mesh A.



Source: The Author

formance by controlling the number of iterations inside the solver to reach the desired tolerance for the solution's residual value. Considering this effect, we explored different values for `m` in previous experiments to find a value that provides adequate performance for the mesh workloads as presented by Figure 6.2. In the left part of the figure, we have the different `m` values for the two meshes in the X-axis, and by how many times it was slower than the faster execution time obtained for a given `m` value in the Y-axis. Another aspect of the GMRES(m) iterative solver is that different values for `m` impact the solver stagnation effect, represented in the right part of Figure 6.2. We could achieve better performance results with the `MAGMA` GMRES(m) if it incorporates some ideas like using an adaptative `m` value throughout the simulation execution for each solution (CABRAL; SCHAERER; BHAYA, 2020). Another configuration of the GMRES(m) is the tolerance value for the solution residual error, for which we set its value to 10^{-10}.

Figure 6.2 – Best value for the parameter m for each mesh (left), and stagnation effect (right)



Source: The Author

We adjusted all these solver configurations used, like the reordering algorithms, task and matrix partitioning granularity, the `m` value, preconditioner, and tolerance for the iterative solver, trying to extract the best performance for these solvers. Thus, we could have a fair comparison between them in the real application scenario.

6.2 Solvers Numerical Validation

Besides parallelizing and accelerating the application, a crucial step is checking if the application parallel versions are still producing valid numerical results. Because there is no such thing as infinite precision, we are bound to the finite precision expressed in floating-point numbers and limited to the machine epsilon or machine precision. This limitation can cause rounding errors that can be propagated and magnified throughout the application execution. Furthermore, by running operations in parallel, the order of the rounding operations are different and can thus produce slightly different results. Other architecture-specific factors like the Fused Multiply-Add (FMA), present in NVIDIA GPUs, perform operations of the type $X \times Y + Z$ using a single rounding step, which improves the accuracy of the results (WHITEHEAD; FIT-FLOREA, 2011).

To evaluate and quantify the quality of the numerical solution provided by the multiple solvers, we propose the use of the Peak Signal-To-Noise Ratio (PSNR) (Korhonen; You, 2012), which is a common metric to estimate quality involving images. In our case, we used this metric to estimate how far the parallel solutions with the three solvers are from the original sequential solution, using it as the solution that remains unaffected by noise. Using the RAFEM results for temperature and voltage written in binary files for each simulation time step, we combine the results of different executions to compare the solution given for each time step, using the following formula for calculating the PSNR metric:

$$\text{PSNR}_i = 20 \times \log_{10} \left(\frac{\text{MAX}(A)}{\sqrt{\text{MSE}_i}} \right) \tag{6.1}$$

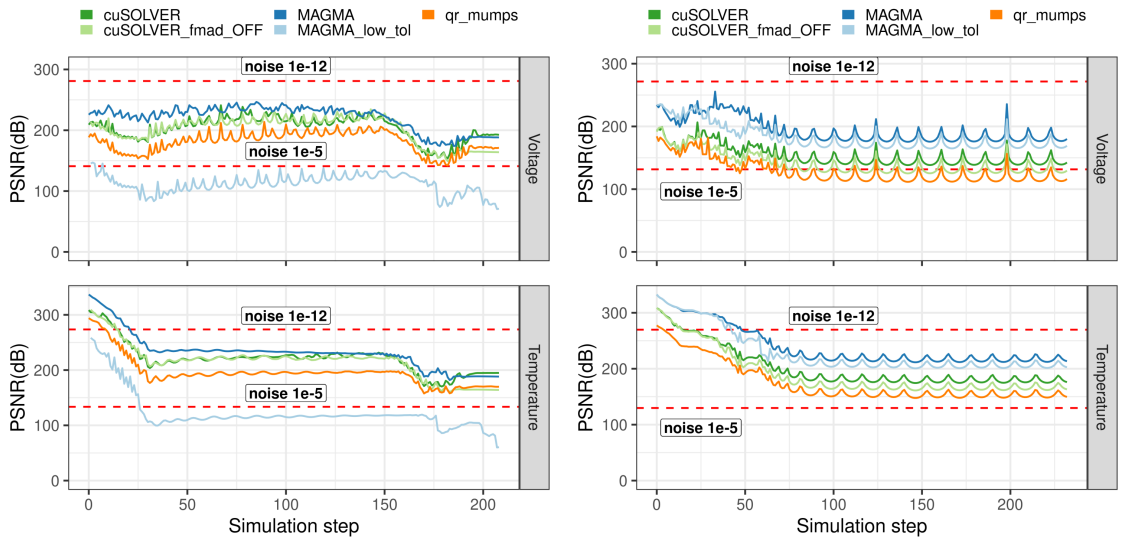
where the Mean Squared Error (MSE) is

$$MSE_i = \frac{1}{n} \times \sum_{j=1}^{n} [A_{ij} - B_{ij}]^2 \tag{6.2}$$

We calculate the PSNR value for each simulation step, using the $MAX(A)$ as

the maximum value for temperature or voltage in the current time step. The $PSNR_i$ represents the PSNR value for the simulation step i, where n is the number of nodes in the mesh, A is the original solution, and B the solution of a parallel version. Using these values in the formula, we produced the results presented in Figure 6.3. The figure presents two interesting cases for the PSNR values for the tested solvers using the two meshes (Mesh A, left, and Mesh B, right), including different configurations of the solvers like the `MAGMA` iterative solver with a lower tolerance of 10^-6 (`MAGMA_low_tol`), and the `cuSOLVER` compiled with the option `--fmad=false` (`cuSOLVER_fmad_OFF`), disabling the GPU FMA operation. Lastly, the red dashed lines in both figures represent control values for the PSNR metric to know what PSNR value we have when using the original solution summed with a static noise of 10^{-12} and 10^{-5} in each mesh node.

Figure 6.3 – PSNR value for temperature and voltage for a simulation with mesh A (left) and mesh B (right).



Source: The Author

In the Mesh A figure (left), we observe that the `MAGMA` solver with lower tolerance provided poorer results than all the other solvers, while for the bigger mesh it kept close to the `MAGMA` with a tolerance of 10^{-10}, presenting a higher PSNR value than the other solvers while accelerating the solution for `MAGMA`. In the Mesh A case, we also noticed that the main loop corrector step not converged in less than the maximum configured iterations (50) for step number 207. When this occurs, the RAFEM application does not accept the solution for that step and reduces the time step size in half, continuing the simulation from the same moment as when the solution was not accepted, increasing the application execution time. Thus we have a mismatch in the simulation time that

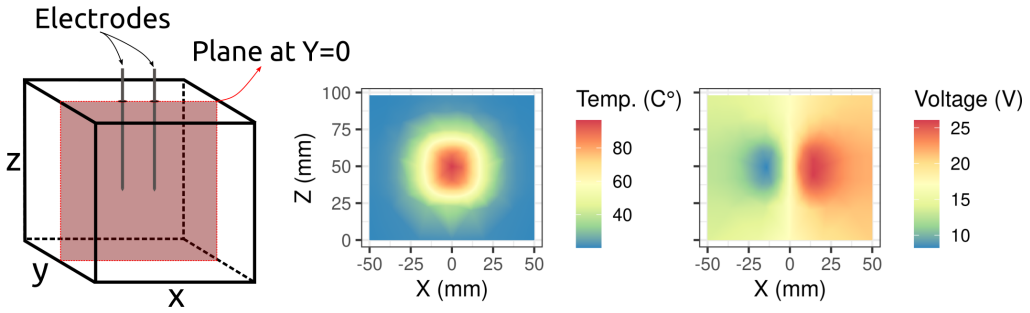
this step represents, which decreases the PSNR value at the end. Hence, depending on the input problem and parameters, the simulation stability may be compromised when relaxing the tolerance of an iterative solver, which may cause the need for smaller time steps to advance in the simulation. This impact of precision in application performance is discussed in the next Section, using this same configuration for the `MAGMA` solver.

Despite this particular case, comparing the other solvers, we observe that the `qr_mumps` solver presented a lower value for PSNR in both cases, indicating higher differences in its provided solution. In opposition, the `cuSOLVER`, which also performs a sparse QR factorization, presented a higher value for PSNR than `qr_mumps`, even when we disabled the FMA operations (`cusolver_FMAD_OFF`), except in the last steps for Mesh A (left). Another interesting observation is that we can see how the FMA affects the precision of `cuSOLVER` principally in the Mesh B. Overall, the PSNR metric reveals that those different solvers and configurations provide slightly different numerical results. However, we observe the same behavior for PSNR values with their ups and downs, except for Mesh A with the `MAGMA_low_tol` configuration. This metric can represent simulation-specific characteristics, like numerical stable and unstable regions during the simulation through the variation of PSNR value.

One drawback of using the PSNR metric to evaluate the numerical solution is that it hides the magnitude of the differences and their spatial distribution. Observing the summarized PSNR data for a time step, we are unable to tell if we have many small differences or fewer more significant differences and how they are spread over the mesh space. This way, we compared the binary files used to calculate the PSNR values to generate 2D surface plots of the steps that presented the lower PSNR value like step 177 for Mesh A, visually representing the differences of the solutions from the original sequential code for temperature and voltage. Figure 6.4 shows the original values of temperature and voltage with the Y-axis fixed to 0, displaying the electrodes side-by-side as the diagram in the left shows. We observe that the distribution of the spatial values for temperature and voltage is very symmetrical, except that for voltage, we have lower values on one side to create the current flow between the electrodes.

We used the same plane to compare the numerical differences between solvers as presented in Figure 6.5. The figure's left part shows the temperature and the right the voltage, faceting plots by their solver configuration. Note that the scale for the differences in temperature and voltage is substantially different. From these plots, we can observe that the higher numerical differences are contained in the area between $x[-25, 25]$ and

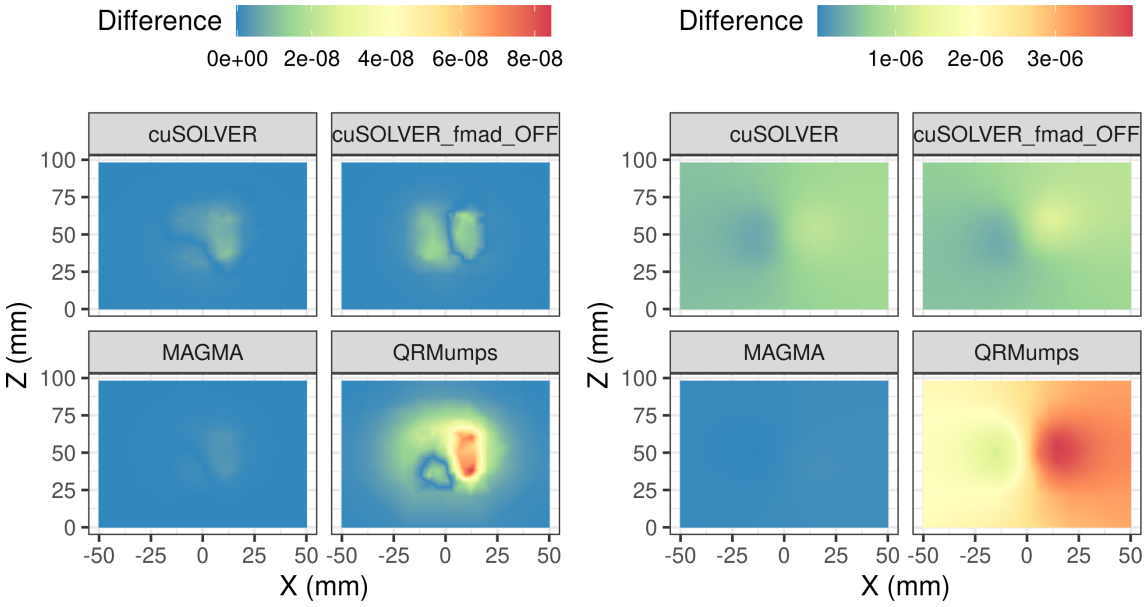
Figure 6.4 – Diagram of the Y plane facing the electrodes side-by-side (left), and the original version values for temperature (center) and voltage (right) at simulation step 177.



Source: The Author

$z[25, 75]$, around the electrodes' position. These figures help us observe that no significant numerical differences arise considering a spatial sense, being contained only around that area and not spreading in regions like the edges of the image, or following the pattern of the original values like in the `qr_mumps` case for voltage.

Figure 6.5 – Numerical differences for temperature (left) and voltage (right) for the step 177 of Mesh A.

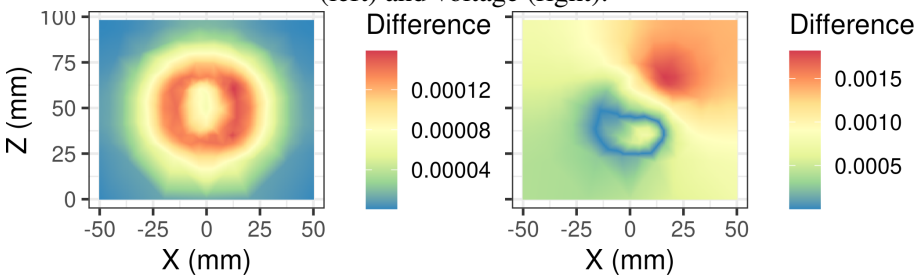


Source: The Author

We can observe that the solution provided by `MAGMA` is much closer to the original values than the others for both temperature and voltage because of the smaller differences, as the PSNR value illustrated. There is an increase in the difference when we go through `cuSOLVER` to `qr_mumps`, including between the `cuSOLVER` and its version with FMA disabled, but all the solvers presented differences around the same area. The maximum absolute differences remain low, for example, $4e-6$ for `qr_mumps` voltage, and $8e-$

08 for temperature. However, looking at the `MAGMA_low_tol` case in Figure 6.6 the difference values scale is much larger than the other configurations. Besides that, the area where the differences occurred is wider than for the other solvers. In this execution configuration of GMRES(m), we may start visually perceiving the differences in the final simulation results, which can harm the analysis process. Furthermore, as we stated earlier, these numerical differences may also impact the overall application performance because of this numerical divergence. These differences can impact the number of time steps to reach the simulation end and iterations within a corrector step to reach convergence.

Figure 6.6 – Numerical differences for MAGMA solver with lower tolerance for temperature (left) and voltage (right).



Source: The Author

6.3 Trace Performance Analysis of `qr_mumps` and RAFEM

Our performance analysis experiments within the RAFEM application context have two main objectives. Firstly we analyze and characterize the different solvers' performance in the most elementary program part: the corrector step iterations. Secondly, we look for the whole application performance behavior throughout the simulation, looking for behavioral changes within these iterations and time steps, characterizing the application with the different solvers. Furthermore, we also evaluate our previously described optimization to reduce task interference in `qr_mumps` to this real application scenario.

6.3.1 Detailed Performance Analysis in RAFEM

The performance experiments considered the different machines and solvers collecting the total execution time and different executions to obtain trace performance. Table 6.2 presents the mean execution time considering a confidence level of 99,7% for Mesh A and B for all solvers and machines. We observe that the `qr_mumps` solver was

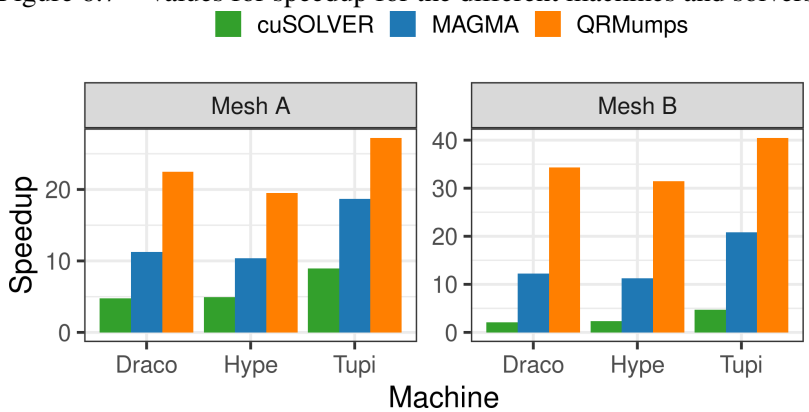
faster for all machines and workload combinations, but the time difference between the GPU solvers and `qr_mumps` shrinks when we have a more powerful GPU in the Tupi machine. The original sequential version is also faster for the Hype and Tupi machines due to the faster CPUs.

Table 6.2 – Execution time in minutes for Mesh A and Mesh B for each code version.

Machine	Original	`cuSOLVER`	`MAGMA`	`qr_mumps`
Mesh A				
Draco	111,76 ± 0,079	23,57 ± 0,226	9,94 ± 0,148	4,97 ± 0,104
Hype	83,57 ± 0,918	16,98 ± 0,084	8,06 ± 0,086	4,28 ± 0,018
Tupi	82,99 ± 0,111	9,3 ± 0,029	4,36 ± 0,024	3,05 ± 0,011
Mesh B				
Draco	1324,1 ± 2,03	621,47 ± 10,2	107.87 ± 0,67	38.51 ± 0,86
Hype	970,36 ± 3,53	411,84 ± 3,3	86.14 ± 0,133	30.82 ± 0,14
Tupi	958,34 ± 3,81	201,37 ± 2,1	45.96 ± 0,68	23.66 ± 0,09

We also present the speedup metric in the Figure 6.7 where the acceleration effect of more powerful GPUs is better observed in the application acceleration, mainly in the GPU solvers, where we have better speedup values for the Tupi machine. However, looking at the different workloads, we observe a lower speedup for `cuSOLVER` for the bigger Mesh B, suggesting that its QR solver does not scale very well for bigger problems. The `MAGMA` and `qr_mumps` solvers presented more significant speedups for Mesh B, but the increase for `MAGMA` compared to Mesh A was only about one or two times. In contrast, the increases in speedup for `qr_mumps` are as high as 13 times for Mesh B, reaching 40× of speedup compared to the 27× for mesh A.

Figure 6.7 – Values for speedup for the different machines and solvers.

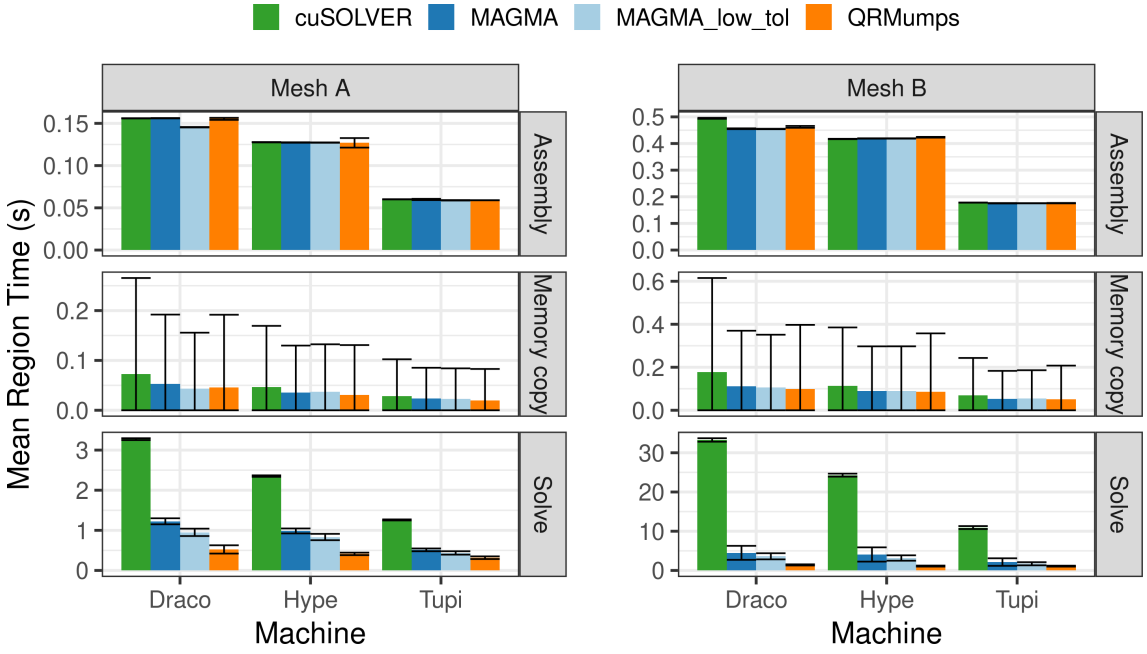


Source: The Author

A part of the application speedup for all cases comes from the assembly step, where the faster GPUs will provide better speedups because it is a very well scalable

problem. Figure 6.8 presents the average duration for the assembly, memory copies, and solve traced regions between all step iterations. Note that we also included the performance of the `MAGMA_low_tol` version for these regions to represent how relaxing the GMRES(m) tolerance affects the solve step duration. Nonetheless, we report that this relaxation in the tolerance can reduce the total computational time by up to 20% for the Mesh B case. However, there are cases with less performance increase like the execution used in Figure 6.3 for Mesh A, where the simulation needed more correction steps because one step not converged. In addition, this reduced tolerance can cause undesirable effects in the numerical results, as presented by Figure 6.6. For the larger mesh, however, the solution with lower tolerance had no problems, performing the same number of steps and iterations as the solution with the more restrict tolerance, which might be related to the better discretization of Mesh B, creating a more stable problem. We also point out that since we accelerated the solve step time, and with the assembly step being fast for powerful GPUs, now the memory copies take a significant amount of time of the iteration, as much as the entire assembly step, like for the Mesh A for Tupi machine. This way, efforts to reduce this transfer time should now be considered in a future work.

Figure 6.8 – Trace iteration time for different application regions.

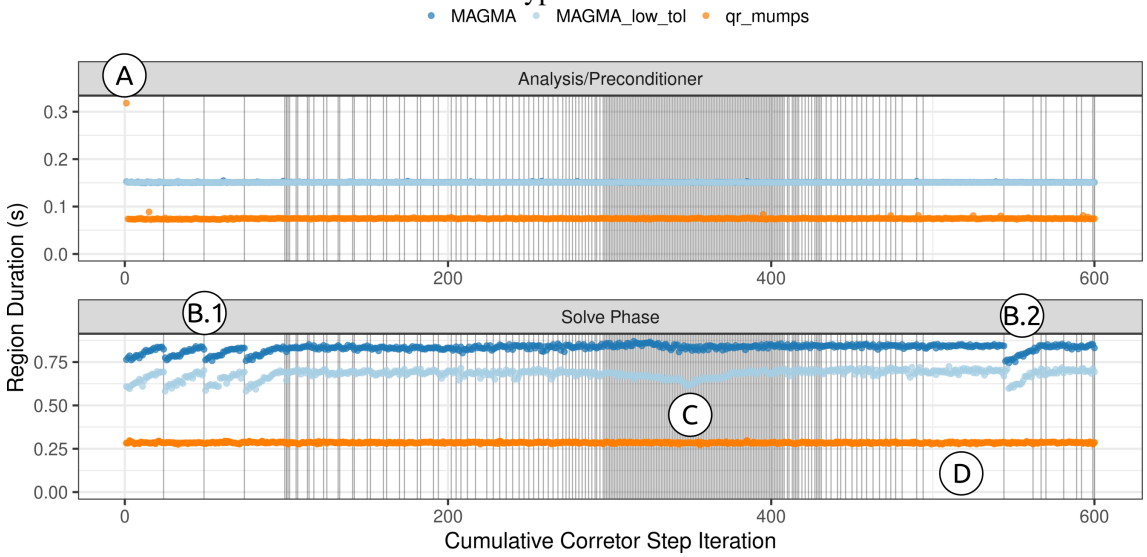


Source: The Author

Another aspect that can be observed in Figure 6.8 is that the `MAGMA` average time for the solve step presents a higher variability, which motivated a detailed investigation of its behavior along with the application iterations. Figure 6.9 depicts these regions

execution duration over the application iterations with vertical lines to indicate the end of a corrector step. In this figure, we can observe the difference of the first `qr_mumps` execution for the analysis step in (A), where the actual column permutation calculation is done. The subsequent analysis steps just reuse this calculated permutation. In (B.1) and (B.2), we observe an interesting behavior for the `MAGMA` solver with both configurations for tolerance, where we have an increase in the cost to obtain the solution within the iterations of a corrector step. In (C), we observe a stable moment in the simulation where the corrector steps converge in one or a few iterations, and we can notice a reduction in the time to solution for the `MAGMA_low_tol`. Lastly, we report that the step demarked in (D) not converged for all configurations, and it is after this step that we notice the same effect of (B.1) in (B.2), while the `qr_mumps` time for solution remained stable during all the execution. The `cuSOLVER`, which is absent in the figure, also presents a stable behavior like `qr_mumps`, but with a higher duration (≈2.3s).

Figure 6.9 – Detailed trace performance data for the analysis/preconditioning phase and system solution in the Hype machine for Mesh A.

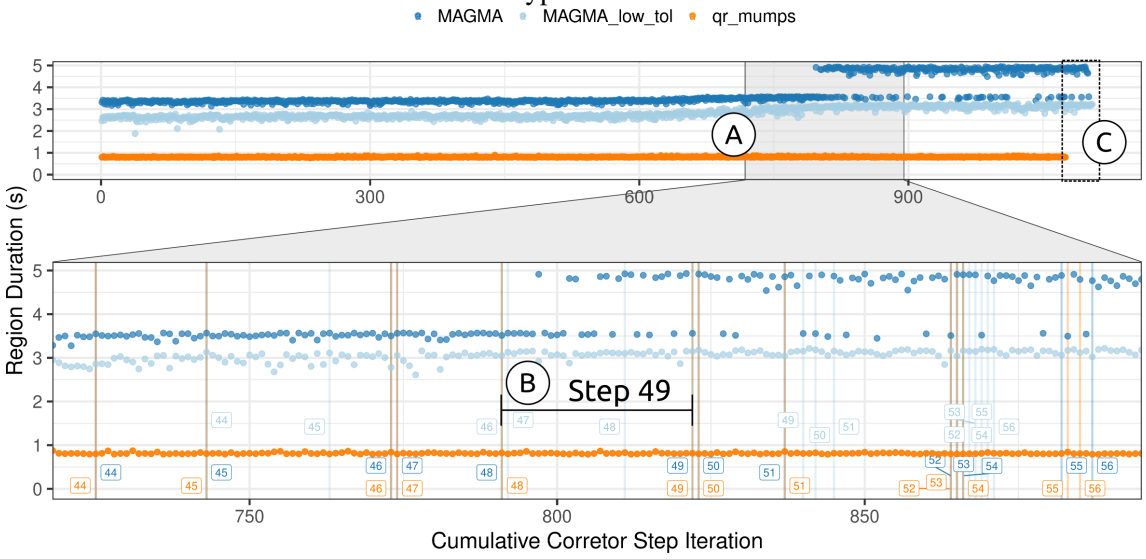


Source: The Author

The `MAGMA` variability is also present in the Mesh B case. Figure 6.10 depicts the solve phase duration over the application iterations for the Mesh B. We observe the same behavior through time: `qr_mumps` is stable and faster than `MAGMA`, and the `MAGMA` with the $1e^{-6}$ tolerance is faster than using $1e^{-10}$. However, in the highlighted area in (A), we observe a change in behavior for `MAGMA`. The faster execution starts slightly increasing the time to obtain the solution, spending more than three seconds. For the slower `MAGMA` configuration, we notice a considerable change in performance during the

simulation step 49 (B), which remains until the simulation end. This happens because the GMRES(m) iterative method with its fixed m parameter needs more iterations to converge for specific cases throughout the simulation, which reinforces the need for some adaptative technique for the m value. Lastly, we also observe in the zoomed plot that the different configurations have a different amount of iterations per step, thus causing a mismatch throughout the simulation and affects the total number of iterations performed, as we observe at the end of the topmost plot in Figure 6.10, pointed by (C).

Figure 6.10 – Detailed trace performance data for the analysis/preconditioning phase and system solution in the Hype machine for mesh B.



Source: The Author

6.3.2 Improving `qr_mumps` for RAFEM

We have noticed that, despite the sequential execution for Hype (970min) and Tupi (958 min) were close, the speedup value for the Tupi machine using `qr_mumps` is higher than in Hype, even though Hype has more than twice the number of CPU cores than Tupi. As the most costly part is in the solve step, this difference suggests that we may not have enough parallelism in the matrix factorization because of `qr_mumps` parameters and matrix structure, or because it might be some lack of task efficiency related to the matrix partitioning, similar to the case presented in Section 5.3.4. Thus, further optimizations can explore these aspects to improve application performance in this specific case, as we investigate in this Section.

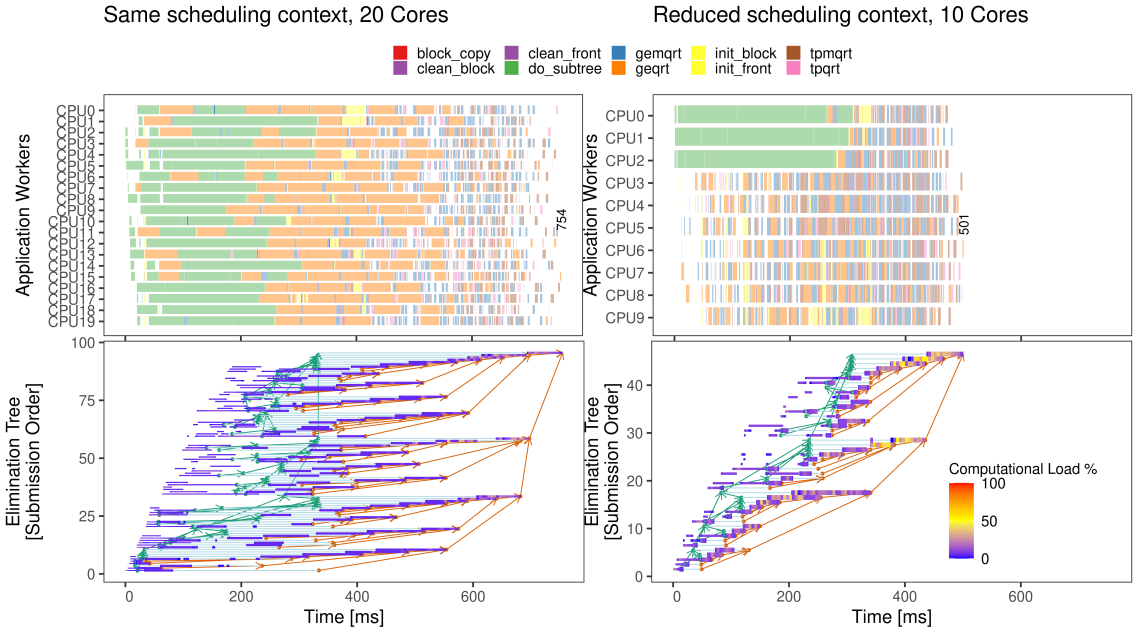
Despite reusing the column permutation, which significantly improves `qr_mumps`

performance, we can use our proposed techniques in Chapter 4 to understand and optimize the performance for this specific case of RAFEM. For this, we wrote the matrices generated by the meshes A and B to an external file in the matrix market file format to run it outside the RAFEM application and study the performance with StarVZ. By running these two cases from the meshes and analyzing their tasks, we noticed that the factorization is composed of many `do_subtree`, `geqrt`, and `tpqrt` tasks, which are more cache-sensitive. Furthermore, besides the best overall makespan for the Tupi machine, we also observed that the solve phase for `qr_mumps` was faster in this machine with fewer cores than the others and with only one NUMA node.

Reducing task interference, lowering L3 cache misses: Based on these assumptions, we used the same technique as before (see Section 5.3.4) by adopting a smaller scheduling context for the `do_subtree` tasks and reduced the number of cores to use only the same NUMA node. We ran a set of experiments for the three different machines varying the number of cores and the `do_subtree` scheduling context size, making thirty repetitions for each. Those experiments revealed that for those matrices, the best configuration for the machines that have NUMA nodes (Hype and Draco) is to reduce the number of cores to use only one NUMA node and reduce the `do_subtree` concurrency by limiting the scheduling context to three cores, allowing a factorization acceleration of Mesh B by up to 35%. For the Tupi machine with only 1 CPU socket, using the `do_subtree` interference reduction technique provided up to 12% improvement in performance for the factorization. Figure 6.11 shows the results for the Hype machine. In the topmost plots, we have the configuration used for RAFEM, and in the bottom, the best configuration. The two Gantt charts reveal how different the `do_subtree` and `geqrt` task duration is because of higher task interference and L3 cache misses.

The elimination tree panel in Figure 6.11 reveals the differences between the two trees created and how they were traversed. We observe that the tree constructed for the 20 workers is very wide, made up of many small nodes and lots of pruned nodes. The other elimination tree is smaller, with fewer nodes and more pruning because we have fewer workers to compute it. However, the total cost for the factorization in both trees is the same. The problem was the increase L3 cache miss number imposed by many cache-sensitive concurrent tasks and the use of two NUMA nodes without mapping the tree structure to the architecture. While StarPU has configuration options for considering different NUMA nodes by using the environment variable `STARPU_USE_NUMA` combined with changing the weight of communications with `STARPU_BETA`, the NUMA-aware

Figure 6.11 – Gantt chart and elimination tree for the configuration used in RAFEM (left) and the optimized configuration (right) for Hype using Mesh B matrix.



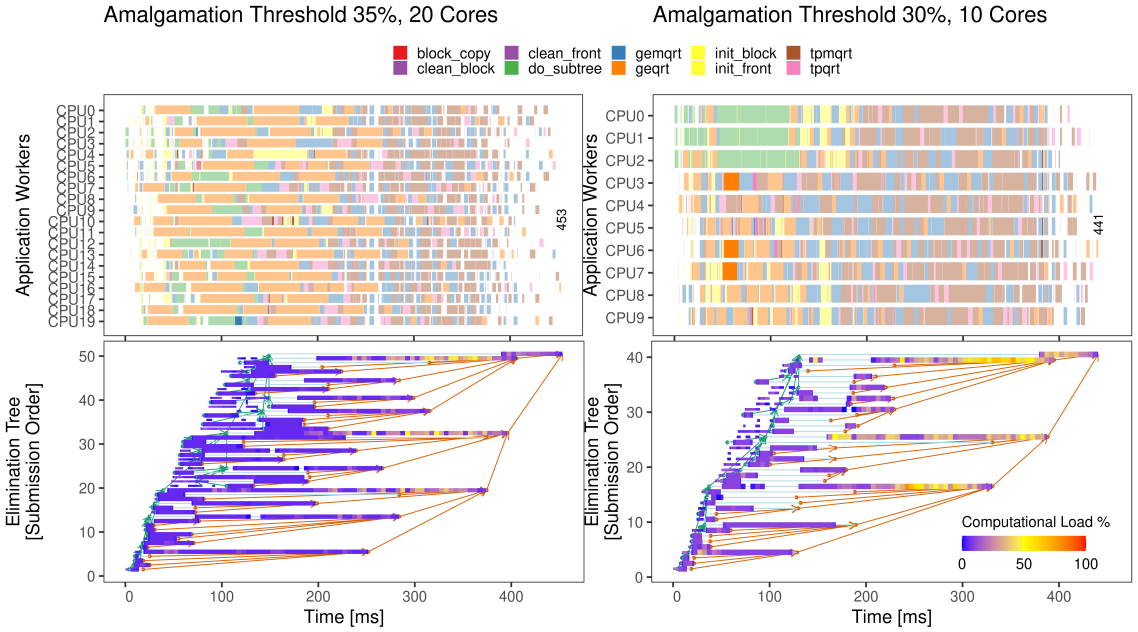
Source: The Author

scheduling still in an experimental phase and did not provide better performance for our tested cases. In this case, a better strategy would be mapping the tree to the architecture processors considering the different NUMA nodes (FAVERGE; RAMET, 2009).

Tuning amalgamation threshold, improving performance with more computations: Despite these architectural aspects, the fact that the elimination tree comprises many small nodes with few computations may not be suitable for performance. We can change these nodes by tuning the node `amalgamation threshold` parameter in `qr_mumps`, used to better exploit the level-3 BLAS routines at the cost of additional fill-in. This parameter controls the fusion of the child nodes into their father node, forming a supernode. It represents a percentage, and if the additional fill-in to amalgamate a node to its parent is less than this percentage, the node is amalgamated, otherwise not. We improved a little by exploring different values for this threshold and tuning the number of workers in the RAFEM matrices. However, now, the total cost of factorization compared to the executions in Figure 6.11 increased by 1.4 and 1.54 times because of the additional fill-in of amalgamating more nodes. Even though we could improve performance a little as presented by Figure 6.12, compared to the results in Figure 6.11.

Considering these optimizations, we can tune `qr_mumps` and rerun the RAFEM simulation to measure the impact in application acceleration. This parameter adjusts increased RAFEM's speedup from 31.4 to 39.6 times for the Mesh B in the Hype machine,

Figure 6.12 – Gantt chart and elimination tree with different configurations for amalgamation threshold for Mesh B matrix in Hype.



Source: The Author

showing the power of a fully configurable solver on top of a runtime system. Another optimization could be done by overlapping the factorization and solve phase by running it asynchronously. From the RAFEM traces, the `qr_mumps` solve step, after the factorization takes up from 2.1% to 2.8% of the total execution time, which is already done in parallel, also following a task-based approach. This solve parallelization is important because some cases can have the time spent in solve comparable to the time used in factorization (CAYROLS; DUFF; LOPEZ, 2020). However, the asynchronous routines for factorization and solve are unavailable for the `qr_mumps` C-Fortran interface, but the acceleration with this technique would be limited to the total time used in this solve step as an upper bound.

6.4 Discussion and Summary

This chapter provided a detailed analysis of different solvers in accelerating a computational simulation application and a numerical analysis of the results. The overall performance of the RAFEM application also depends on the quality of those solutions because it influences the total number of simulation iterations that will be needed. We observed this for the `MAGMA` solver varying its tolerance, for which relaxing the tolerance

provided better results for the bigger mesh case, while the need for more iterations reduced the gains for the smaller mesh. While iterative solvers like the `MAGMA` GMRES(m) naturally fit for sparse equation systems, we observed that its performance changes throughout the execution because of the numerical properties of the matrix change, and a single m value does not provide the best results to reduce the number of iterations to converge for GMRES during all simulation.

Since `qr_mumps` provided the best performance between the tested solvers, we explored different parameters and techniques using the proposed visualization panels to optimize it. The application has many configurable parameters that can affect the final matrix and tree structure, total number of tasks, and those tasks efficiency. However, we showed that it provided the best performance within the solvers, even only using CPUs and its default configuration parameters. Furthermore, we show how a fine-tuning of these parameters can improve performance even more. Besides its internal configurations, the architecture also affects the performance, which also depends on the input problem. This way, combining its lots of parameters, the architectural aspects, and the problem input, finding the optimal configuration for a given problem is very difficult. Thus, our visualization techniques support this optimization process by providing the task modeling and the elimination tree structure overview. Another tool that would be great to find the optimal parameter set for a given input problem would be a simulation of the application, enabling the exploration of a comprehensive parameter set with a low cost (TESSER et al., 2017). However, for the simulation, effects like task interference and NUMA architectures must be considered as we saw how they affect performance. Lastly, an ultimate approach would be providing an autotuning technique considering the application, runtime, and architectural aspects (Bruel et al., 2019). However, finding the optimal values for every given problem is very hard since they depend on many factors (LOPEZ, 2015).

7 CONCLUSION

With the ease of programming and flexibility for performance portability over heterogeneous systems provided by the task-based paradigm, its use nowadays is widely adopted for high-performance applications. With the runtime system helping in many responsibilities like memory management, data coherence, and dynamic scheduling, it takes a crucial role in application performance and execution behavior, where many other aspects can impact performance. Thus, a process of continuous performance analysis throughout the development of a parallel application is necessary to attain high performance. The visual performance analysis is a well-established technique for guiding such a process. However, it needs to keep evolving to fulfill analysts' requirements considering complex architectures and equally complex applications, aligning the developer's application point of view to performance metrics. Along with the task-based abstractions like the DAG representation of an application, other algorithmic-specific characteristics like the irregularity of tasks and structures like the elimination tree for parallel direct sparse factorization methods should also be considered when analyzing performance. Sparse solvers encompass such a variety of complex and extensively configurable applications, for which the performance analysis represents a vital step for its development and for tuning the configurations for specific architectures and set of parameters.

This work presents two main contributions to improve task-based application performance analysis, a set of application-specific visualization panels with the novel elimination tree panel among them, and the automatic anomalous task classification for irregular tasks. This tree-based visualization enhances the performance information with the algorithmic structure used in the application, aligning them to the performance metrics, improving the understanding of application behavior. Furthermore, it includes a technique for enhancing the post-mortem analysis by providing an automatic classification of anomalous tasks considering their irregular theoretical cost, task type, and computational resource type to predict the expected duration. All these contributions are part now of the StarVZ and are publicly available.

We validate the usefulness of the proposed techniques within the task-based sparse solver `qr_mumps` using a set of test workload matrices and included `qr_mumps` in a real simulation application scenario. These strategies were evaluated considering different architectures, runtime, and application configurations. We could observe the impact of different parameters in the application relating them to the tree structure, like the memory

consumption threshold, scheduling algorithms, task priorities, and ordering algorithms. The anomalous task detection mechanism revealed four different sources of task anomalies: (i) peaks of task submissions, (ii) tracing system overhead, (iii) numerical content of task's data block generating denormal numbers, and (iv) the task L3 cache misses. We investigated a simple technique to mitigate the last case effect from these experiments, which we found out to be caused by cache interference when running cache-sensitive tasks in parallel. We proposed reducing the concurrency for the `do_subtree` tasks by restricting their scheduling context to a few cores. This interference reduction improved the execution time by 24% just by doing this, accelerating the tasks execution. Given this anomalous task behavior source and the possibility of performance gain, more robust techniques can be explored to gain performance in these cases.

These strategies were also applied along with a study for accelerating the RAFEM application by using parallel solvers. This study included solver tuning parameters, a detailed performance analysis over the application point of view using tracing techniques, and a numerical analysis to assess the solution provided by the different solvers. The `qr_mumps` application outperformed the iterative GMRES(m) solver from the `MAGMA` library and another sparse QR solver from the `cuSOLVER` library. While all solvers were able to accelerate the application, the `MAGMA` solver presented unstable behavior throughout the simulation iterations. It increased the time-to-solution, revealing that the starting configuration, initially good, no longer provided the best performance, implying the use of an adaptative m parameter for the GMRES(m) solver. While `MAGMA` was able to keep the same speedup value for the bigger and the smaller test case, the `cuSOLVER` reduced the speedup value for the bigger mesh case from 9× to 5×, suggesting that its QR solver does not scale well for bigger problems. However, for `qr_mumps`, the speedup was increased for the bigger mesh case, reaching values up to 40×. We used our performance analysis techniques for `qr_mumps` for the bigger mesh in the Hype machine, where we improved the speedup from 31.4× to 39.6×, proving the usefulness of `qr_mumps` and how adjusting its parameters can significantly improve performance, even when the total cost for the factorization rises but it is composed of more performance efficient tasks.

Future work includes refining and proposing other visualization panels and performance models, improving the scalability of the elimination tree visualization by grouping tree nodes as did for the pruned nodes, and expanding a tree node within the tree, showing its computational tasks. Also, one could apply the presented techniques to analyze other elimination-tree task-based applications such as pastix. Furthermore, an interactive

visual approach can also be beneficial for the visual performance analysis process. In the anomalous detection behavior, studies to automatize and explore the relation between the tasks duration, GFlops, and cache miss, for example, can be explored with techniques like a multiple regression model. Lastly, a way to ease or automatize finding optimal parameters for `qr_mumps` would be quite impressive, along with the ability to consider task interference in the scheduling plus mapping the tree structure over the underlying architecture.

7.1 Publications

The following papers were produced throughout this dissertation, representing the description of our performance analysis techniques and the study over RAFEM application:

- MILETTO, Marcelo Cogo; SCHEPKE, Claudio; SCHNORR, Lucas. Optimization of a Radiofrequency Ablation FEM Application Using Parallel Sparse Solvers. In: 2020 18th International Conference on High Performance Computing & Simulation (HPCS). Barcelona, Spain (Virtual/Online event). (accepted)
- MILETTO, Marcelo Cogo; NESI, Lucas Leandro; SCHNORR, Lucas; LEGRAND Arnaud. Enhanced Performance Analysis of Irregular Task-Based Applications on Hybrid Platforms. In: Future Generation Computer Systems. (to be submitted)

Other publications made during the master course are:

- MILETTO, Marcelo Cogo; SCHNORR, Lucas. OpenMP and StarPU Abreast: the Impact of Runtime in Task-Based Block QR Factorization Performance. In: SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (WSCAD), 20. , 2019, Campo Grande. Porto Alegre: Sociedade Brasileira de Computação, 2019 . p. 25-36. DOI: <https://doi.org/10.5753/wscad.2019.8654>.
- MILETTO, Marcelo Cogo; SCHNORR, Lucas Mello. Mecanismo de Detecção de Tarefas Anômalas Para a Análise de Desempenho de Aplicações com Tarefas Irregulares. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD-RS), 20. , 2020, Santa Maria. Porto Alegre: Sociedade Brasileira de Computação, 2020 . p. 143-144. ISSN 2595-4164. DOI: <https://doi.org/10.5753/eradrs.2020.10778>.

- DA SILVA, Henrique C. P.; MILETTO, Marcelo Cogo; PINTO, Vinicius Garcia; SCHNORR, Lucas Mello. Análise da Influência do Runtime OpenMP no Desempenho de Aplicação com Tarefas. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD-RS), 20. , 2020, Santa Maria. Porto Alegre: Sociedade Brasileira de Computação, 2020 . p. 133-136. ISSN 2595-4164. DOI: <https://doi.org/10.5753/eradrs.2020.10774>.

- SCHEPKE, Claudio; MILETTO, Marcelo Cogo. Acceleration of Radiofrequency Ablation Process for Liver Cancer Using GPU. In: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE, 2020. p. 385-389. DOI: <https://doi.org/10.1109/PDP50117.2020.00065>

- NESI, Lucas Leandro; PINTO, Vinicius Garcia; MILETTO, Marcelo; SCHNORR, Lucas Mello; THIBAULT, Samuel. Introdução ao Desenvolvimento de Aplicações Paralelas com o Paradigma Orientado a Tarefas e o runtime StarPU. In: XX ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD-RS), 2020. DOI: <https://doi.org/10.5753/sbc.4400.9.4>.

- MILETTO, Marcelo Cogo; SCHEPKE, Claudio; SCHNORR, Lucas. Aprimorando a Análise de Desempenho de Aplicações Baseadas em Tarefas Irregulares e Árvores de Eliminação. In: XXI ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD-RS). (submitted)

- NESI, Lucas Leandro; MILETTO, Marcelo; PINTO, Vinicius Garcia; SCHNORR, Lucas Mello. Desenvolvimento de Aplicações Baseadas em Tarefas comOpenMP Tasks. In: XXI ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD-RS), 2021. (submitted)

REFERENCES

ADHIANTO, L. et al. Hpctoolkit: Tools for performance analysis of optimized parallel programs. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 22, n. 6, p. 685–701, 2010.

ADHIANTO, L.; MELLOR-CRUMMEY, J.; TALLENT, N. R. Effectively presenting call path profiles of application performance. In: IEEE. **2010 39th International Conference on Parallel Processing Workshops**. [S.l.], 2010. p. 179–188.

AGULLO, E. et al. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In: HWU, W. mei W. (Ed.). **GPU Computing Gems**. [S.l.]: Morgan Kaufmann, 2010. v. 2.

AGULLO, E. et al. Bridging the gap between openmp and task-based runtime systems for the fast multipole method. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 28, n. 10, p. 2794–2807, 2017.

AGULLO, E. et al. Multifrontal qr factorization for multicore architectures over runtime systems. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 2013. p. 521–532.

AGULLO, E. et al. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. **Acm transactions on mathematical software (toms)**, ACM New York, NY, USA, v. 43, n. 2, p. 1–22, 2016.

AGULLO, E. et al. Numerical linear algebra on emerging architectures: The plasma and magma projects. In: IOP PUBLISHING. **Journal of Physics: Conference Series**. [S.l.], 2009. v. 180, n. 1, p. 012037.

ALVARADO, F. L. Visualizing sparse matrix computations. In: IEEE. **IEEE International Symposium on Circuits and Systems**. [S.l.], 1990. p. 1268–1171.

AMESTOY, P. R.; DUFF, I. S.; L'EXCELLENT, J.-Y. Mumps multifrontal massively parallel solver version 2.0. Citeseer, 1998.

AUGONNET, C. et al. StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In: **Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface**. Berlin, Heidelberg: Springer-Verlag, 2012. (EuroMPI'12), p. 298–299. ISBN 978-3-642-33517-4. Available from Internet: <http://dx.doi.org/10.1007/978-3-642-33518-1_40>.

AUGONNET, C. et al. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In: **16th Intl. Conference on Parallel and Distributed Systems**. Shangai: [s.n.], 2010.

AUGONNET, C. et al. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 23, n. 2, p. 187–198, 2011.

BELL, R.; MALONY, A. D.; SHENDE, S. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 2003. p. 17–26.

BOSILCA, G. et al. Dague: A generic distributed dag engine for high performance computing. **Parallel Computing**, Elsevier, v. 38, n. 1-2, p. 37–51, 2012.

BOSILCA, G. et al. Dague: A generic distributed dag engine for high performance computing. **Parallel Computing**, Elsevier, v. 38, n. 1-2, p. 37–51, 2012.

BOUDEC, J.-Y. L. Performance evaluation of computer and communication systems. EPFL Press, Lausanne, 2010. Available from Internet: <http://infoscience.epfl.ch/record/146812>.

BRINKMANN, S.; GRACIA, J.; NIETHAMMER, C. Task debugging with temanejo. In: **Tools for High Performance Computing 2012**. [S.l.]: Springer, 2013. p. 13–21.

BROQUEDIS, F. et al. Forestgomp: an efficient openmp environment for numa architectures. **International Journal of Parallel Programming**, Springer, v. 38, n. 5-6, p. 418–439, 2010.

Bruel, P. et al. Autotuning under tight budget constraints: A transparent design of experiments approach. In: **2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)**. [S.l.: s.n.], 2019. p. 147–156.

BUTTARI, A. Fine-grained multithreading for the multifrontal qr factorization of sparse matrices. **SIAM Journal on Scientific Computing**, SIAM, v. 35, n. 4, p. C323–C345, 2013.

BUTTARI, A. et al. A class of parallel tiled linear algebra algorithms for multicore architectures. **Parallel Computing**, Elsevier, v. 35, n. 1, p. 38–53, 2009.

CABRAL, J. C.; SCHAERER, C. E.; BHAYA, A. Improving gmres (m) using an adaptive switching controller. **Numerical Linear Algebra with Applications**, Wiley Online Library, v. 27, n. 5, p. e2305, 2020.

CAMARDA, K. V.; STADTHERR, M. A. Frontal solvers for process engineering: local row ordering strategies. **Computers & chemical engineering**, Elsevier, v. 22, n. 3, p. 333–341, 1998.

CAPS Enterprise, Cray, Nvidia, PGI. **The OPenACC Application Programmin Interface v2.7**. Santa Clara, CA, USA: [s.n.], 2018. Available from Internet: <https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf>. Accessed in: 2020-12-26.

CASANOVA, H. et al. Versatile, scalable, and accurate simulation of distributed applications and platforms. **Journal of Parallel and Distributed Computing**, Elsevier, v. 74, n. 10, p. 2899–2917, 2014.

CAYROLS, S.; DUFF, I. S.; LOPEZ, F. Parallelization of the solve phase in a task-based cholesky solver using a sequential task flow model. **The International Journal of High Performance Computing Applications**, SAGE Publications Sage UK: London, England, v. 34, n. 3, p. 340–356, 2020.

CEBALLOS, G. et al. 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**. [S.l.: s.n.], 2017. p. 11–20.

CENTER, B. S. **Extrae User guide manual for version 3.1. 0**. [S.l.]: May, 2015.

CHAN, E. et al. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In: **Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming**. [S.l.: s.n.], 2008. p. 123–132.

CICOTTI, P.; LI, S.; BADEN, S. Performance modeling tools for parallel sparse linear algebra computations. **Advances in Parallel Computing**, v. 19, 01 2010.

COULOMB, K. et al. An open-source tool-chain for performance analysis. In: **Tools for High Performance Computing 2011**. [S.l.]: Springer, 2012. p. 37–48.

CUENCA, J.; GIMÉNEZ, D.; GONZÁLEZ, J. Architecture of an automatically tuned linear algebra library. **Parallel computing**, Elsevier, v. 30, n. 2, p. 187–210, 2004.

DACKLAND, K.; KÅGSTRÖM, B. An hierarchical approach for performance analysis of scalapack-based routines using the distributed linear algebra machine. In: SPRINGER. **International Workshop on Applied Parallel Computing**. [S.l.], 1996. p. 186–195.

DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. **IEEE computational science and engineering**, IEEE, v. 5, n. 1, p. 46–55, 1998.

DANJEAN, V.; NAMYST, R.; WACRENIER, P.-A. An efficient multi-level trace toolkit for multi-threaded applications. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 2005. p. 166–175.

DAVIS, T. Multifrontral multithreaded rank-revealing sparse qr factorization. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FÜR INFORMATIK. **Dagstuhl Seminar Proceedings**. [S.l.], 2009.

DAVIS, T. A. Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method. **ACM Transactions on Mathematical Software (TOMS)**, ACM New York, NY, USA, v. 30, n. 2, p. 196–199, 2004.

DAVIS, T. A. Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse qr factorization. **ACM Transactions on Mathematical Software (TOMS)**, ACM New York, NY, USA, v. 38, n. 1, p. 1–22, 2011.

DAVIS, T. A. et al. Algorithm 836: Colamd, a column approximate minimum degree ordering algorithm. **ACM Transactions on Mathematical Software (TOMS)**, ACM New York, NY, USA, v. 30, n. 3, p. 377–380, 2004.

DAVIS, T. A.; RAJAMANICKAM, S.; SID-LAKHDAR, W. M. A survey of direct methods for sparse linear systems. **Acta Numerica**, Cambridge University Press, v. 25, p. 383–566, 2016.

DONGARRA, J. et al. Plasma: Parallel linear algebra software for multicore using openmp. **ACM Trans. Math. Softw.**, Association for Computing Machinery, New York, NY, USA, v. 45, n. 2, may 2019. ISSN 0098-3500. Available from Internet: <https://doi.org/10.1145/3264491>.

DONGARRA, J. et al. With extreme computing, the rules have changed. **Computing in Science & Engineering**, IEEE Computer Society, v. 19, n. 3, p. 52–62, 2017.

DREBES, A. et al. **Automatic Detection of Performance Anomalies in Task-Parallel Programs**. 2014.

DREBES, A. et al. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In: . [S.l.: s.n.], 2014.

DUFF, I. S.; REID, J. K. The multifrontal solution of indefinite sparse symmetric linear. **ACM Transactions on Mathematical Software (TOMS)**, ACM New York, NY, USA, v. 9, n. 3, p. 302–325, 1983.

DURAN, A. et al. Ompss: a proposal for programming heterogeneous multi-core architectures. **Parallel processing letters**, World Scientific, v. 21, n. 02, p. 173–193, 2011.

FAVERGE, M.; RAMET, P. A numa aware scheduler for a parallel sparse direct solver. In: **Workshop on Massively Multiprocessor and Multicore Computers**. [S.l.: s.n.], 2009. p. 5p.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In: **Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation**. [S.l.: s.n.], 1998. p. 212–223.

GANGULY, D.; LANGE, J. R. The effect of asymmetric performance on asynchronous task based runtimes. In: **Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017**. [S.l.: s.n.], 2017. p. 1–8.

GAUTIER, T. et al. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: IEEE. **2013 IEEE 27th International Symposium on Parallel and Distributed Processing**. [S.l.], 2013. p. 1299–1308.

GEIJN, R. A. V. D.; QUINTANA-ORTÍ, E. S. The science of programming matrix computations. Citeseer, 2008.

GEIMER, M. et al. The scalasca performance toolset architecture. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 22, n. 6, p. 702–719, 2010.

GEIST, G.; NG, E. Task scheduling for parallel sparse cholesky factorization. **International Journal of Parallel Programming**, Springer, v. 18, n. 4, p. 291–314, 1989.

GRIGORI, L.; LI, X. S. Towards an accurate performance modeling of parallel sparse factorization. **Applicable Algebra in Engineering, Communication and Computing**, Springer, v. 18, n. 3, p. 241–261, 2007.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI: portable parallel programming with the message-passing interface**. [S.l.]: MIT press, 1999.

GUO, Z. et al. Inter-task cache interference aware partitioned real-time scheduling. In: **Proceedings of the 35th Annual ACM Symposium on Applied Computing**. [S.l.: s.n.], 2020. p. 218–226.

GUPTA, A. Wsmp: Watson sparse matrix package (part-i: direct solution of symmetric sparse systems). **IBM TJ Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC**, Citeseer, v. 21886, 2000.

GUPTA, A. et al. Effective minimally-invasive gpu acceleration of distributed sparse matrix factorization. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 2016. p. 672–683.

HAUGEN, B. et al. Visualizing execution traces with task dependencies. In: **Proceedings of the 2nd Workshop on Visual Performance Analysis**. [S.l.: s.n.], 2015. p. 1–8.

HÉNON, P.; RAMET, P.; ROMAN, J. Pastix: a high-performance parallel direct solver for sparse symmetric positive definite systems. **Parallel Computing**, Elsevier, v. 28, n. 2, p. 301–321, 2002.

HUCK, K. A.; MALONY, A. D. Perfexplorer: A performance data mining framework for large-scale parallel computing. In: IEEE. **SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing**. [S.l.], 2005. p. 41–41.

HUYNH, A. et al. Dagviz: a dag visualization tool for analyzing task-parallel program traces. In: **Proceedings of the 2nd Workshop on Visual Performance Analysis**. [S.l.: s.n.], 2015. p. 1–8.

IAKYMCHUK, R.; BIENTINESI, P. Modeling performance through memory-stalls. **ACM SIGMETRICS Performance Evaluation Review**, ACM New York, NY, USA, v. 40, n. 2, p. 86–91, 2012.

IRONS, B. M. A frontal solution program for finite element analysis. **International Journal for Numerical Methods in Engineering**, Wiley Online Library, v. 2, n. 1, p. 5–32, 1970.

ISAACS, K. E. et al. Combing the communication hairball: Visualizing parallel execution traces using logical time. **IEEE transactions on visualization and computer graphics**, IEEE, v. 20, n. 12, p. 2349–2358, 2014.

ISAACS, K. E. et al. State of the Art of Performance Visualization. In: BORGO, R.; MACIEJEWSKI, R.; VIOLA, I. (Ed.). **EuroVis - STARs**. [S.l.]: The Eurographics Association, 2014. ISBN 978-3-03868-028-4.

JAIN, R. **The art of computer systems performance analysis**. [S.l.]: John Wiley & Sons, 2008.

JANKUN-KELLY, T.; MA, K.-L. Moiregraphs: Radial focus+ context visualization and interaction for graphs with visual nodes. In: IEEE. **IEEE Symposium on Information Visualization 2003 (IEEE Cat. No. 03TH8714)**. [S.l.], 2003. p. 59–66.

JIANG, Y. et al. Formulation of 3d finite elements for hepatic radiofrequency ablation. **International Journal of Modelling, Identification and Control**, Inderscience Publishers, v. 9, n. 3, p. 225–235, 2010.

JOSHI, M. V. et al. Pspases: An efficient and scalable parallel sparse direct solver. In: CITESEER. **PPSC**. [S.l.], 1999.

KAISER, H. et al. Hpx: A task based programming model in a global address space. In: **Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models**. [S.l.: s.n.], 2014. p. 1–11.

KARYPIS, G.; KUMAR, V. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

KERGOMMEAUX, J. C. D.; STEIN, B. de O. Pajé: an extensible environment for visualizing multi-threaded programs executions. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 2000. p. 133–140.

KIRAN, U.; SHARMA, D.; GAUTAM, S. S. Gpu-warp based finite element matrices generation and assembly using coloring method. **Journal of Computational Design and Engineering**, Oxford University Press, v. 6, n. 4, p. 705–718, 2019.

KLINKENBERG, J. et al. Chameleon: reactive load balancing for hybrid mpi+ openmp task-parallel applications. **Journal of Parallel and Distributed Computing**, Elsevier, v. 138, p. 55–64, 2020.

KNÜPFER, A. et al. The vampir performance analysis tool-set. In: **Tools for high performance computing**. [S.l.]: Springer, 2008. p. 139–155.

KNÜPFER, A. et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In: **Tools for High Performance Computing 2011**. [S.l.]: Springer, 2012. p. 79–91.

Korhonen, J.; You, J. Peak signal-to-noise ratio revisited: Is simple beautiful? In: **Int. WS on Quality of Multimedia Exp.** [S.l.: s.n.], 2012. p. 37–38.

LIMA, J. V. F. et al. Performance and energy analysis of openmp runtime systems with dense linear algebra algorithms. **The International Journal of High Performance Computing Applications**, SAGE Publications Sage UK: London, England, v. 33, n. 3, p. 431–443, 2019.

LIU, J. W. The role of elimination trees in sparse factorization. **SIAM journal on matrix analysis and applications**, SIAM, v. 11, n. 1, p. 134–172, 1990.

LOAN, C. F. V.; GOLUB, G. H. **Matrix computations fourth edition**. [S.l.]: Johns Hopkins University Press Baltimore, 2013.

LOPEZ, F. **Task-based multifrontal QR solver for heterogeneous architectures**. Thesis (PhD) — Université de Toulouse, Université Toulouse III-Paul Sabatier, 2015.

LUK, C.-K. et al. Pin: Building customized program analysis tools with dynamic instrumentation. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 40, n. 6, p. 190–200, jun. 2005. ISSN 0362-1340. Available from Internet: <https://doi.org/10.1145/1064978.1065034>.

MANTOVANI, F.; CALORE, E. Multi-node advanced performance and power analysis with paraver. In: IOS PRESS. **Parallel Computing is Everywhere (serie: Advances in Parallel Computing)**. [S.l.], 2018. v. 32, p. 723–732.

MATTILA, A.-L. et al. Software visualization today: Systematic literature review. In: **Proceedings of the 20th International Academic Mindtrek Conference**. [S.l.: s.n.], 2016. p. 262–271.

MEY, D. an et al. Score-p: A unified performance measurement system for petascale applications. In: **Competence in High Performance Computing 2010**. [S.l.]: Springer, 2011. p. 85–97.

MILANI, L. F. G. **Autotuning with Machine Learning of OpenMP Task Applications**. Thesis (Theses) — Université Grenoble Alpes, 2020.

MILETTO, M. C.; SCHNORR, L. Openmp and starpu abreast: the impact of runtime in task-based block qr factorization performance. In: SBC. **Anais Principais do XX Simpósio em Sistemas Computacionais de Alto Desempenho**. [S.l.], 2019. p. 25–36.

MUDDUKRISHNA, A. et al. Grain graphs: Openmp performance analysis made easy. In: **Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. [S.l.: s.n.], 2016. p. 1–13.

MUNZNER, T. **Visualization analysis and design**. [S.l.]: CRC press, 2014.

NESI, L. L. et al. 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)**. [S.l.: s.n.], 2019. p. 142–151.

NOBRE, C. et al. Lineage: Visualizing multivariate clinical data in genealogy graphs. **IEEE transactions on visualization and computer graphics**, IEEE, v. 25, n. 3, p. 1543–1558, 2018.

NOBRE, C.; STREIT, M.; LEX, A. Juniper: A tree+ table approach to multivariate graph visualization. **IEEE transactions on visualization and computer graphics**, IEEE, v. 25, n. 1, p. 544–554, 2018.

NVIDIA. **CUDA Toolkit Documentation v11.2.0**. Santa Clara, CA, USA: NVIDIA Corporation, 2020. Available from Internet: <https://docs.nvidia.com/cuda/>. Accessed in: 2020-12-26.

NVIDIA. **cuSOLVER**. 2020. <https://docs.nvidia.com/cuda/cusolver>. Accessed in: 2021-01-12.

OpenMP. **OpenMP Application Program Interface Version 5**. OpenMP Architecture Review Board, 2018. Available from Internet: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>.

OZ, I. et al. Regression-based prediction for task-based program performance. **Journal of Circuits, Systems and Computers**, World Scientific, v. 28, n. 04, p. 1950060, 05 2019.

PEISE, E.; BIENTINESI, P. Performance modeling for dense linear algebra. In: IEEE. **2012 SC Companion: High Performance Computing, Networking Storage and Analysis**. [S.l.], 2012. p. 406–416.

PELLEGRINI, F.; ROMAN, J. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In: SPRINGER. **International Conference on High-Performance Computing and Networking**. [S.l.], 1996. p. 493–498.

PILLET, V. et al. Paraver: A tool to visualize and analyze parallel code. In: CITESEER. **Proceedings of WoTUG-18: transputer and occam developments**. [S.l.], 1995. v. 44, n. 1, p. 17–31.

PINTO, V. G. et al. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 30, n. 18, p. e4472, 2018.

REED, D. A. Experimental analysis of parallel systems: techniques and open problems. In: SPRINGER. **International Conference on Modelling Techniques and Tools for Computer Performance Evaluation**. [S.l.], 1994. p. 25–51.

SAAD, Y. **Iterative methods for sparse linear systems**. [S.l.]: SIAM, 2003.

SANFUI, S.; SHARMA, D. A two-kernel based strategy for performing assembly in fea on the graphics processing unit. In: IEEE. **2017 international conference on advances in mechanical, industrial, automation and management systems (AMIAMS)**. [S.l.], 2017. p. 1–9.

SCHEPKE, C.; MILETTO, M. C. Acceleration of radiofrequency ablation process for liver cancer using gpu. In: IEEE. **2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. [S.l.], 2020. p. 385–389.

SCHNORR, L. Pajeng–trace visualization tool. **GitHub Repository. https://github.com/schnorr/pajeng**, 2012.

SCHNORR, L. M. Análise de desempenho de programas paralelos. **Anais da ERAD/RS**, v. 2014, p. 57–81, 2014.

SCHNORR, L. M. et al. **starvz: R-Based Visualization Techniques for Task-Based Applications**. [S.l.], 2020. R package version 0.4.1. Available from Internet: <https://github.com/schnorr/starvz>.

SCHREIBER, R. A new implementation of sparse gaussian elimination. **ACM Transactions on Mathematical Software (TOMS)**, ACM New York, NY, USA, v. 8, n. 3, p. 256–276, 1982.

SERRANO, M. A.; ROYUELA, S.; QUIÑONES, E. Towards an openmp specification for critical real-time systems. In: SPRINGER. **International Workshop on OpenMP**. [S.l.], 2018. p. 143–159.

SHENDE, S. Profiling and tracing in linux. In: CITESEER. **Proceedings of the Extreme Linux Workshop**. [S.l.], 1999. v. 2.

SHENDE, S. S.; MALONY, A. D. The tau parallel performance system. **The International Journal of High Performance Computing Applications**, Sage Publications Sage CA: Thousand Oaks, CA, v. 20, n. 2, p. 287–311, 2006.

SINGH, N. P.; AHUJA, K. Reusing preconditioners in projection based model order reduction algorithms. **IEEE Access**, IEEE, v. 8, p. 133233–133247, 2020.

SINNEN, O. **Task scheduling for parallel systems**. [S.l.]: John Wiley & Sons, 2007.

SONG, F.; YARKHAN, A.; DONGARRA, J. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In: IEEE. **Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis**. [S.l.], 2009. p. 1–11.

STANISIC, L. et al. 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. p. 481–490. Available from Internet: <https://doi.org/10.1109/ICPADS.2015.67>.

STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. **Comp. in sci. & eng.**, IEEE, v. 12, n. 3, p. 66–73, 2010.

TALLENT, N. R. et al. Scalable fine-grained call path tracing. In: **Proceedings of the international conference on Supercomputing**. [S.l.: s.n.], 2011. p. 63–74.

TESSER, R. K. et al. Using simulation to evaluate and tune the performance of dynamic load balancing of an over-decomposed geophysics application. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 2017. p. 192–205.

THIBAULT, S. **On Runtime Systems for Task-based Programming on Heterogeneous Platforms**. Thesis (Habilitation à diriger des recherches) — Université de Bordeaux, dec. 2018. Available from Internet: <https://hal.inria.fr/tel-01959127>.

TOHID, R. et al. Asynchronous execution of python code on task-based runtime systems. In: IEEE. **2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)**. [S.l.], 2018. p. 37–45.

TOLEDO, S. Taucs: A library of sparse linear solvers. **http://www. tau. ac. il/˜ stoledo/taucs/**, 2003.

TOMOV, S.; DONGARRA, J.; BABOULIN, M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. **Parallel Computing**, v. 36, n. 5-6, p. 232–240, jun. 2010. ISSN 0167-8191.

TOPCUOGLU, H.; HARIRI, S.; WU, M.-y. Performance-effective and low-complexity task scheduling for heterogeneous computing. **IEEE transactions on parallel and distributed systems**, IEEE, v. 13, n. 3, p. 260–274, 2002.

Vampir. **Vampir 9.7 User Manual**. Dresden, Germany: GWT, 2019. Available from Internet: <https://tu-dresden.de/zih/forschung/ressourcen/dateien/projekte/vampir/dateien/Vampir-User-Manual.pdf?lang=en>. Accessed in: 2020-12-18.

VOHRA, D. Apache parquet. In: **Practical Hadoop Ecosystem**. [S.l.]: Springer, 2016. p. 325–335.

WHITEHEAD, N.; FIT-FLOREA, A. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. **rn (A+ B)**, v. 21, n. 1, p. 18749–19424, 2011.

WILKINSON, L. The grammar of graphics. In: **Handbook of computational statistics**. [S.l.]: Springer, 2012. p. 375–414.

WILLHALM, T.; POPOVICI, N. Putting intel® threading building blocks to work. In: ACM. **Proceedings of the 1st international workshop on Multicore software engineering**. [S.l.], 2008. p. 3–4.

WILLIAMS, K.; BIGELOW, A.; ISAACS, K. Visualizing a moving target: A design study on task parallel programs in the presence of evolving data and concerns. **IEEE transactions on visualization and computer graphics**, IEEE, v. 26, n. 1, p. 1118–1128, 2019.

WITTMANN, M. et al. Short note on costs of floating point operations on current x86-64 architectures: Denormals, overflow, underflow, and division by zero. **arXiv preprint arXiv:1506.03997**, 2015.

XIANYI, Z. **OpenBLAS: An optimized BLAS library**. 2013. Available from Internet: <http://www.openblas.net/>.

XU, L. et al. Modeling i/o performance variability in high-performance computing systems using mixture distributions. **Journal of Parallel and Distributed Computing**, Elsevier, v. 139, p. 87–98, 2020.

YARKHAN, A.; KURZAK, J.; DONGARRA, J. Quark users' guide: Queueing and runtime for kernels. **University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02**, 2011.

ZAKI, O. et al. Toward scalable performance visualization with jumpshot. **The International Journal of High Performance Computing Applications**, v. 13, n. 3, p. 277–288, 1999. Available from Internet: <https://doi.org/10.1177/109434209901300310>.

APPENDIX A — RESUMO EXPANDIDO EM PORTUGUÊS

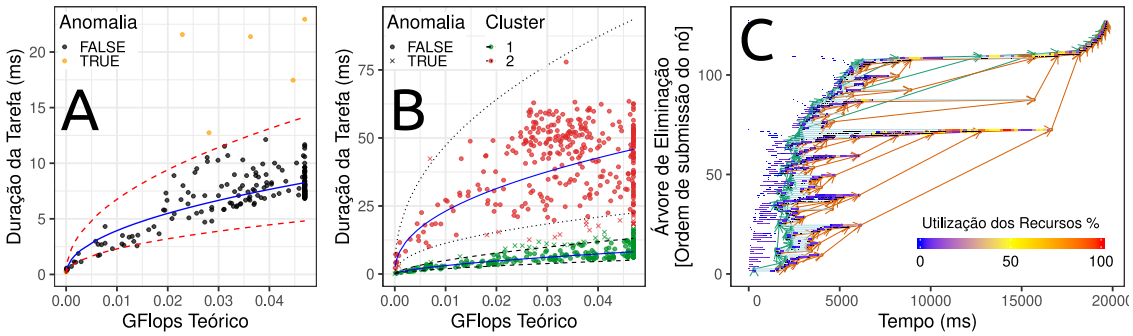
A adoção do paradigma de programação baseada em tarefas está sendo amplamente utilizada no contexto de computação de alto desempenho devido à sua capacidade de simplificação na programação, e pela sua habilidade de prover portabilidade de desempenho. Isso graças ao sistema de *runtime*, que assume responsabilidades como o gerenciamento dos dados e o escalonamento das tarefas. Este paradigma é flexível e se adapta a arquiteturas heterogêneas, que são hoje onipresentes em supercomputadores, sendo suportado por diversas bibliotecas e interfaces de programação paralela (DAGUM; MENON, 1998; AUGONNET et al., 2011; WILLHALM; POPOVICI, 2008; GAUTIER et al., 2013; FRIGO; LEISERSON; RANDALL, 1998; DURAN et al., 2011). A análise de desempenho de tais aplicações é uma etapa fundamental e complexa, que deve considerar os aspectos específicos deste paradigma, como a representação da aplicação como um grafo acíclico dirigido, para guiar uma análise voltada para as tarefas. Assim como para análises envolvendo o paradigma baseado em tarefas, também devemos considerar aplicações que usam estruturas específicas que definem e moldam a sua execução, como o caso da estrutura da árvore de eliminação usada no método multifrontal. Considerando estes aspectos, conseguimos alinhar as abstrações e estruturas usadas no desenvolvimento de um método ou modelo de programação com as informações obtidas sobre o desempenho de uma aplicação.

Este trabalho apresenta três principais contribuições. A primeira consiste em estratégias voltadas para a melhoria do processo de análise de desempenho de aplicações baseadas em tarefas com comportamento irregular. A segunda é um ocnjunto de técnicas de visualização de desempenho específicas considerando a estrutura de árvores de eliminação, que são o alicerce de métodos diretos paralelos para a solução de sistemas de equações lineares. Estes métodos de análise foram desenvolvidos na ferramenta StarVZ (PINTO et al., 2018) e avaliados usando a aplicação `qr_mumps` (AGULLO et al., 2016), que implementa o método de fatoração QR multifrontal. Já a terceira contribuição consiste em um estudo sobre o uso de diferentes *solvers* esparsos em uma aplicação de simulação.

Mecanismo de Detecção de Tarefas Anômalas: A primeira contribuição é um mecanismo para detecção automática de anomalias de desempenho entre as tarefas da aplicação. Para isto, foram utilizados e avaliados diferentes técnicas e modelos de regressão, baseando-se no custo teórico de operações de ponto flututante destas tarefas para explicar

a sua duração, possibilitando assim capturar a irregularidade do custo das tarefas em um modelo. Usamos um modelo de regressão sobre uma transformação usando a função `log` para a variável preditora e a variável resposta, a fim de contornar o efeito de heteroscedasticidade dos dados. Um exemplo deste modelo é representado na Figura A.1 (A), onde as linhas pontilhadas vermelhas representam os limites superior e inferior de predição considerando um intervalo de confiança de 95%. As tarefas marcadas como anomalias são aquelas em que a duração excede o intervalo de predição superior. Durante os experimentos, notamos que para alguns casos o uso de somente um modelo se mostrou insuficiente para representar o comportamento das tarefas. Assim, consideramos o uso de uma mistura finita de modelos de regressão, representado em (B), onde usamos dois modelos para agrupar e representar as tarefas de uma execução.

Figura A.1 – Modelo de regressão com transformação logarítmica utilizado para a classificação de tarefas anômalas (A), uso de múltiplos modelos para a representação do comportamento de tarefas (B), e a representação da computação da árvore de eliminação ao longo do tempo (C).



Fonte: O Autor

Com a técnica de detecção de anomalias pudemos detectar quatro diferentes fontes de anomalias dentre as tarefas: (1) durante picos de submissões de tarefas, (2) sobrecarga do sistema de rastreamento, (3) conteúdo numérico dos blocos de dados de uma tarefa, gerando números desnormalizados, e (4), o aumento do número de cache *misses* no nível L3 causado pela interferência de tarefas concorrentes. Neste último caso, pudemos detectar o momento em que essas anomalias ocorriam e exploramos uma abordagem para reduzir a interferência causada por tarefas sensíveis a *misses* na cache L3. Para isto, restringimos a execução de um tipo de tarefa a um número menor de CPUs, o que reduziu o tempo de execução de um dos casos de teste em até 24%, onde havia uma interferência significativa.

Métodos de Visualização para Análise de Desempenho: A segunda contribuição na parte de análise de desempenho é ilustrada na Figura A.1 em (C), onde podemos observar

o painel de visualização da computação da estrutura da árvore de eliminação ao longo do tempo. As linhas representam a estrutura hierárquica da árvore e as cores em gradiente a taxa de uso dos recursos computacionais dedicados à computação de um nó específico da árvore. Com esta técnica, é possível visualizar as decisões de escalonamento em relação à árvore de eliminação, possibilitando investigar o impacto de diferentes parâmetros a nível do sistema de *runtime* e de aplicação na computação de sua estrutura. Exploramos aspectos como a restrição no consumo de memória, diferentes algoritmos de escalonamento e reordenamento, alinhando a estrutura principal do método *multifrontal* ao desempenho da aplicação, o que aproxima a análise de desempenho do ponto de vista de especialistas e desenvolvedores.

Estudo de *Solvers* Paralelos na Aceleração de uma Simulação: A terceira contribuição deste trabalho foi o uso e análise de *solvers* paralelos na aceleração de uma aplicação de simulação computacional, o RAFEM (JIANG et al., 2010). Essa aplicação usa o método dos elementos finitos para a simulação de um procedimento médico usado no tratamento de alguns casos de câncer de fígado. A simulação consiste num laço principal que avança no tempo de simulação. Onde para cada passo de simulação, se monta e resolve um sistema de equações representado por uma matriz esparsa, constituindo a parte mais custosa do programa. A parte de montagem do sistema já foi paralelizada usando GPUs em um trabalho anterior (SCHEPKE; MILETTO, 2020). O presente trabalho explora o impacto no desempenho e na qualidade da solução numérica ao se usar diferentes bibliotecas para obter a solução do sistema de equações.

Comparamos as bibliotecas `cuSOLVER` e `MAGMA` que fornecem *solvers* acelerados por GPUs, e também incluímos a biblioteca `qr_mumps`. Foram exploradas diferentes configurações para estas bibliotecas, a fim de termos uma comparação mais justa entre o desempenho das diferentes bibliotecas ao usar a configuração com o melhor desempenho para cada uma. Realizamos uma série de experimentos usando duas malhas de elementos finitos de tamanhos diferentes. Usando técnicas de rastreamento com a biblioteca Score-P, caracterizamos o desempenho da aplicação em três diferentes regiões: montagem, cópias de memória, e a obtenção da solução do sistema. Observamos um comportamento estável para o tempo de solução ao longo da execução da simulação para as bibliotecas `cuSOLVER` e `qr_mumps`, e um comportamento variável para a biblioteca `MAGMA` usando o *solver* iterativo GMRES(m), revelando a necessidade de se recalibrar os parâmetros do método ao longo da execução. Para a biblioteca `cuSOLVER`, foi detectada uma falta de escalabilidade, pois o nível de aceleração da biblioteca comparado com a

versão sequencial da aplicação foi menor para o caso com a malha maior. A biblioteca `qr_mumps` apresentou os maiores níveis de aceleração, chegando a ser até 40 vezes mais rápida que a simulação sequencial, reduzindo o tempo de espera por resultados de 16 horas para 23 minutos, mantendo a qualidade da solução numérica.

Ainda realizamos uma análise detalhada do desempenho do `qr_mumps` dentro da aplicação RAFEM, onde, usando as técnicas de visualização e análise, pudemos identificar fontes de ineficiência e contorná-las através de configurações na plataforma e na biblioteca, aumentando a aceleração de um dos casos estudados de 31.4 vezes para 39.6 vezes. Essa aceleração atingida mostra o valor de se ter uma biblioteca construída sobre um sistema de *runtime*, altamente configurável, e do uso de técnicas específicas de análise de desempenho para encontrar a configuração mais adequada da aplicação, melhorando o seu desempenho.

Conclusão: A série de experimentos realizados neste trabalho demonstram a usabilidade das técnicas propostas, e permitiram investigar e identificar cenários onde o desempenho é prejudicado. Como trabalhos futuros, destacamos melhorias na escalabilidade do painel de visualização da árvore de eliminação para o tratameto de casos com árvores formadas por milhares de nós, e o uso de estratégias mais sofisticadas para a redução de interferência entre tarefas sensíveis a cache *misses*, como o uso de escalonadores que consideram este efeito (GUO et al., 2020) e o mapeamento da estrutura da árvore sobre a arquitetura alvo. Além disso, explorar o uso das técnicas propostas em outras aplicações como o `PaStiX` podem gerar resultados interessantes.