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

VINÍCIUS GARCIA PINTO

Performance Analysis Strategies for Task-based Applications on Hybrid Platforms

Thesis prepared under a *co-tutelle* agreement
and presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science
at the Federal University of Rio Grande do Sul
and the University Grenoble Alpes

Advisor (UFRGS): Prof. Dr. Nicolas Maillard
Advisor (UGA): Prof. Dr. Arnaud Legrand

Porto Alegre
October 2018

CIP — CATALOGING-IN-PUBLICATION

Pinto, Vinícius Garcia

Performance Analysis Strategies for Task-based Applications on Hybrid Platforms / Vinícius Garcia Pinto. – Porto Alegre: PPGC da UFRGS, 2018.

178 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor (UFRGS): Nicolas Maillard; Advisor (UGA): Arnaud Legrand.

1. Performance analysis. 2. Task-based applications. 3. Hybrid platforms. 4. Trace visualization. I. Maillard, Nicolas. II. Legrand, Arnaud. III. Título.

ACKNOWLEDGMENTS

First I would like to thank CAPES (Brazilian Federal Agency for Support and Evaluation of Graduate Education) for the financial support during these years. I also would like to thank Inria (French National Research Institute for the Digital Sciences) for giving me access to their incredible technical, physical and personnel infrastructure. Additionally, I would like to thank the Institute of Informatics at the Federal University of Rio Grande do Sul and the Grenoble Informatics Laboratory (LIG) at the University Grenoble Alpes.

I would like to thank my advisors Arnaud Legrand and Nicolas Maillard. Nicolas, thank you for opening me the doors of the academic world and for encouraging me to conduct this Ph.D. in an international cooperation context. Arnaud, I am so grateful for having the chance to work with you. I really appreciate your ability to formulate new hipothesis and solutions even after we show you the same result for the tenth time, and, of course, always taking care of the reproducibility aspects. You are a source of inspiration and motivation in my incipient career as a researcher. Finally, I would like to thank Lucas Mello Schnorr. Lucas, even if it has not been possible to include you as my official co-advisor due to bureaucratic formalities, you have spared no efforts to help me, and I am very thankful for your all support with very nice insights and encouragements. I am sure that this work would not be the same without your endless list of suggestions and improvements.

I would like to thank Samuel Thibault, Luka Stanisic and Vincent Danjean who are my co-authors on several papers in the subject of this thesis. Vincent was also my first contact in France, and I am thankful not only for your technical support but also with your academic support with the formalities and paperwork for administrative stuff.

I would like to thank all my colleagues from GPPD and POLARIS/DATAMOVE teams. I really appreciate your company during all these years, not only in the technical discussions but also in the lunch (or sometimes procrastination) time. I would like to thank, in particular, my colleagues from offices 205 (INF), B116 (INRIA) and 434 (IMAG) for tolerating me during these years.

I would like to thank the members of the jury committee: Alfredo Goldman Vel Lejbman, Gaël Thomas, Mathieu Faverge, Gerson Geraldo Homrich Cavalheiro, Bernd Mohr and Philippe Olivier Alexandre Navaux for accepting the invitation, for the nice discussion and their valuable suggestions.

I would like to warmly thank my parents for all their support and patience. I also would like to thank Leandro and my sister, Larissa, who was my inspiration to embrace this crazy and challenging adventure called the academic career.

Finally, I would like to thank my lovely fiancee, Franciele Cordeiro, for being beside me (on both sides of the Atlantic) during all this long journey. I couldn't have done this work without you, your support, your suggestions, your courage, and your patience. Thank you for sharing your life and your knowledge with me.

ABSTRACT

Programming paradigms in High-Performance Computing have been shifting toward task-based models that are capable of adapting readily to heterogeneous and scalable supercomputers. The performance of task-based applications heavily depends on the runtime scheduling heuristics and on its ability to exploit computing and communication resources.

Unfortunately, the traditional performance analysis strategies are unfit to fully understand task-based runtime systems and applications: they expect a regular behavior with communication and computation phases, while task-based applications demonstrate no clear phases. Moreover, the finer granularity of task-based applications typically induces a stochastic behavior that leads to irregular structures that are difficult to analyze.

In this thesis, we propose performance analysis strategies that exploit the combination of application structure, scheduler, and hardware information. We show how our strategies can help to understand performance issues of task-based applications running on hybrid platforms. Our performance analysis strategies are built on top of modern data analysis tools, enabling the creation of custom visualization panels that allow understanding and pinpointing performance problems incurred by bad scheduling decisions and incorrect runtime system and platform configuration. By combining simulation and debugging, we are also able to build a visual representation of the internal state and the estimations computed by the scheduler when scheduling a new task.

We validate our proposal by analyzing traces from a Cholesky decomposition implemented with the StarPU task-based runtime system and running on hybrid (CPU/GPU) platforms. Our case studies show how to enhance the task partitioning among the multi-(GPU, core) to get closer to theoretical lower bounds, how to improve MPI pipelining in multi-(node, core, GPU) to reduce the slow start in distributed nodes and how to upgrade the runtime system to increase MPI bandwidth. By employing simulation and debugging strategies, we also provide a workflow to investigate, in depth, assumptions concerning the scheduler decisions. This allows us to suggest changes to improve the runtime system scheduling and prefetch mechanisms.

Keywords: Performance analysis. task-based applications. hybrid platforms. trace visualization.

Estratégias de Análise de Desempenho para Aplicações baseadas em Tarefas em Plataformas Híbridas

RESUMO

No contexto da Computação de Alto Desempenho, a utilização de modelos de programação baseados em paralelismo de tarefas é cada vez mais frequente uma vez que estes são capazes de se adaptar mais facilmente à supercomputadores que utilizam arquiteturas híbridas. O desempenho das aplicações baseadas em tarefas depende fortemente de heurísticas de escalonamento dinâmicas e da habilidade destas em explorar os recursos de computação e comunicação.

Infelizmente, as estratégias de análise de desempenho tradicionais não são adequadas à análise de ambientes de execução dinâmicos e de aplicações baseadas em tarefas. Estas estratégias esperam, em geral, um comportamento relativamente regular com alternância de fases de computação e comunicação enquanto as aplicações baseadas em tarefas não apresentam estruturas tão bem definidas. Além disso, o assincronismo e a granularidade mais fina das aplicações baseadas em tarefas induz comportamentos estocásticos que levam à estruturas naturalmente irregulares que são difíceis de analisar.

Nesta tese, nós propomos estratégias de análise de desempenho que exploram simultaneamente a estrutura da aplicação, características do escalonador e informações da plataforma. Nós mostramos como nossas estratégias podem ajudar à compreender e resolver problemas de desempenho não triviais em aplicações baseadas em tarefas executadasm em plataformas híbridas. Nossas estratégias de análise de desempenho são construídas sobre ferramentas de análise de dados modernas e genéricas, o que possibilita a criação de visualizações específicas e adaptadas. Estas visualizações permitem a compreensão e a identificação de problemas de desempenho ocasionados por decisões de escalonamento impróprias bem como por configurações incorretas do ambiente de execução e da plataforma. Por meio da combinação de técnicas de simulação e depuração com visualizações especificamente desenvolvidas para representar o estado interno do escalonador e suas estimativas, nós mostramos como é possível avaliar certas hipóteses sobre a pertinência das decisões de escalonamento.

Nós validamos nossa propostas por meio da análise de rastros de execução de uma fatorização de Cholesky implementada com o ambiente de execução StarPU e executada com diferentes plataformas híbridas (CPU/GPU). Nossos estudos de caso mostram como

melhorar a partição das tarefas entre diferentes núcleos CPU e GPUs para se aproximar dos limites inferiores teóricos, como melhorar o pipeline das operações MPI entre diferentes nós (*multicore* e multi-GPUs) para acelerar o início da aplicação, e como melhorar o ambiente de execução para aumentar a largura de banda MPI. O emprego de estratégias de simulação e depuração, nos fornece uma abordagem que permite examinar, em detalhes, as decisões de escalonamento bem como de propor melhorias nos mecanismos de escalonamento e *prefetch* do ambiente de execução.

Palavras-chave: análise de desempenho, aplicações baseadas em tarefas, plataformas híbridas, visualização de rastros.

Stratégies d'analyse de performance pour les applications basées sur tâches sur plates-formes hybrides

ABSTRACT

Dans le cadre du calcul haute performance, l'utilisation d'un modèle de programmation basé sur un parallélisme de tâche est de plus en plus courant. Cette approche permet de s'adapter plus facilement aux super-ordinateurs utilisant des architectures hybrides. La performance d'applications à bases de tâches dépend fortement des heuristiques d'ordonnancement dynamiques sous-jacente et de leur capacité à exploiter les ressources de calcul et de communication.

Malheureusement, les stratégies d'analyse de performance traditionnelles ne sont pas adaptées à l'analyse de supports d'exécution dynamiques et d'applications basées sur des tâches. Ces stratégies supposent en général un comportement relativement régulier avec des alternances de phases de calcul et de communication alors que les applications basées sur des tâches ne présentent pas de structures aussi précises. Par ailleurs, l'asynchronisme et la granularité plus fine des applications basées sur des tâches induit des comportements stochastiques qui donnent lieu à des structures naturellement irrégulières qui sont difficiles à analyser.

Dans cette thèse, nous proposons des stratégies d'analyse de performance qui exploitent à la fois la structure de l'application, des caractéristiques de l'ordonnancement et des informations de la plate-forme. Nous présentons comment nos stratégies peuvent aider à comprendre et résoudre des problèmes de performance non triviaux dans des applications basée sur des tâches qui s'exécutent sur des plates-formes hybrides. Nos stratégies d'analyse de performance sont construites à l'aide d'outils d'analyse de données génériques et modernes, ce qui permet de créer des vues spécifiques et adaptées. Ces vues permettent la compréhension et l'identification de problèmes de performances occasionnés par de mauvaises décisions d'ordonnancement ou bien des configurations incorrectes du support d'exécution et de la plate-forme. En combinant des techniques de simulation et de débogage à des vues spécifiquement développées pour représenter l'état interne de l'ordonnançeur et qui le conduisent à prendre ses décisions, nous avons montré qu'il était possible d'évaluer certaines hypothèses sur la pertinence de ses choix.

Nous validons notre proposition en analysant de nombreuses traces d'exécutions d'une factorisation de Cholesky implémenté avec le support d'exécution StarPU et exécutée

sur différentes plates-formes hybrides (CPU/GPU). Nos études de cas montrent comment améliorer la partition des tâches entre les différents cœurs et GPUS pour s'approcher des bornes inférieures théoriques, comment améliorer le pipeline des opérations MPI entre les différents nœuds (multi-coeurs et multi-GPUs) pour accélérer le démarrage de l'application, et comment améliorer le support d'exécution pour augmenter la bande passante MPI. L'emploi des stratégies de simulation et débogage, nous fournissons une approche permettant d'examiner, en détail, les décisions d'ordonnancement et ainsi de proposer des améliorations des mécanismes d'ordonnancement et de prefetch du support d'exécution.

Keywords: analyse de performance, parallélisme de tâche, plates-formes hybrides, visualisation de trace.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ABE	Area Bound Estimation
BSP	Bulk-Synchronous Parallel
CFD	Computational Fluid Dynamics
CPE	Critical Path Estimation
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DBF	Distributed Breadth First
DSL	Domain-Specific Language
DMDA	Deque Model Data Aware
DMDAS	Deque Model Data Aware Sorted
DOI	Digital Object Identifier
FLOPS	Floating Point Operations Per Second
GPU	Graphical Processing Unit
HEFT	Heterogeneous Earliest Finish Time
HPC	High-Performance Computing
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
IQR	InterQuartile Range
ISA	Instruction Set Architecture
LML	Lightweight Markup Language
LWS	Locality Work Stealing
MIC	Many Integrated Core
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access

OTF	Open Trace Format
PPE	PowerPC Processor Element
PTG	Parameterized Task Graph
SPE	Synergistic Processor Element
WF	Work First
WS	Work Stealing

LIST OF ALGORITHMS

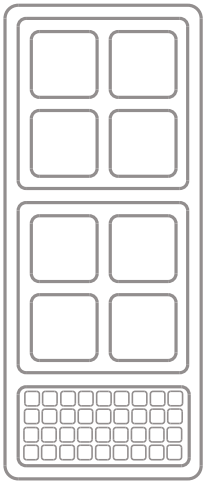
CONTENTS

1 INTRODUCTION

Computer architectures have experienced a paradigm shift in the last years. The stagnation in the processor frequency has led the adoption of other ways to fulfill the ever-growing need for computation power. The combination of multicore processors with accelerator devices has been a prevalent approach to fulfill these performance requirements. Figure 1.1 represents a common layout of current parallel computers including multiple cores associated with an accelerator device disclosing a multilevel and heterogeneous parallelism.

For a long time, the applicability of parallel computers had been limited to restricted domains, mainly concerning High-Performance Computing (HPC) research and scientific applications, e.g., climate modeling, energy research, data analysis, and simulations. However, the recent paradigm shift has put the parallelism in the essence of almost all computing devices. This way, taking only a layout of the architecture as the one of Figure 1.1, it is no longer possible to distinguish a TOP500-ranked supercomputing node from a high-end smartphone.

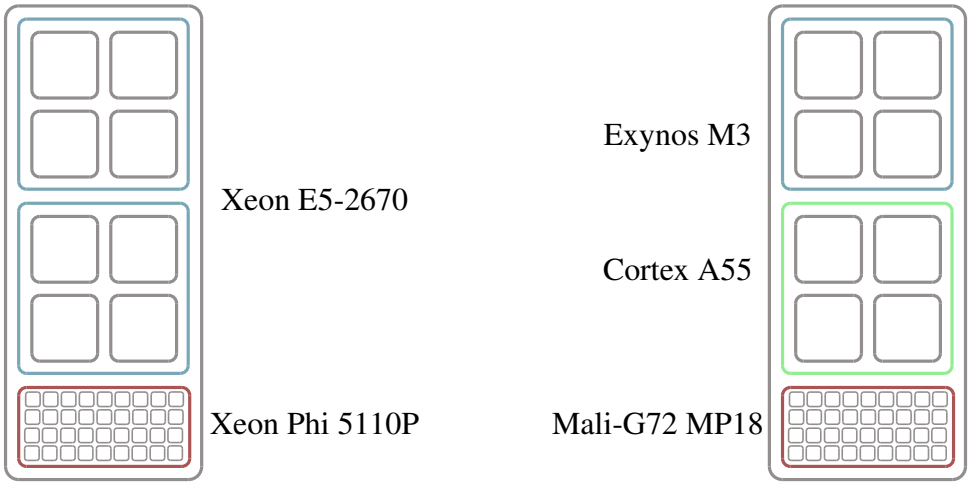
Figure 1.1: The layout of a current parallel computer with a multicore processor and a manycore accelerator device. The larger boxes represent multicore processors with its powerful cores based on sophisticated control logic and complex cache hierarchy. The smaller ones represent accelerator cores which are simpler but more numerous.



Source: The Author

Figure 1.2 illustrates the architecture of two current parallel computers. Both are octa-core systems enhanced with accelerator devices. However, despite the conceptually similar architecture, they have extremely contrasting purposes. The first (right) is a node

Figure 1.2: Two current parallel computers. On the left, a node of the *Cascade* system, ranked in the position 69 in the June 2018 Top500 list. This system has an octa-core Intel Xeon E5-2670 at 2.6GHz with an Intel Xeon Phi 5110P as an accelerator. On the right side, a Samsung Galaxy S9 smartphone using a heterogeneous ARM processor with four M3 cores at 2.7GHz and four Cortex-A55 cores at 1.8GHz with a GPU Mali-G72 as an accelerator.



Source: The Author

of the #69 most powerful system in the last Top500 list [1], while the second one (left) is a Galaxy S9[2] smartphone produced by Samsung.

Although parallel computers have become mainstream, with hybrid platforms widespread from supercomputing centers to ordinary citizens' pockets, the software stack is not fully prepared to efficiently exploit this multilevel parallelism. The paradigm shift in the hardware has exposed the limitations of traditional tools for programming and analyzing applications running on parallel platforms. Indeed, these limitations come from the context on which these tools were designed, i.e., homogeneous parallel computers used in HPC and scientific domains.

Programming hybrid platforms is complex. The common strategy based on the use of traditional tools (e.g., MPI, pthreads, OpenMP) to program parallel applications in a hybrid hardware is becoming unfeasible and potentially problematic. Efficiently programming such machines achieving portable and scalable performance has become extremely challenging since the use of explicit programming models demands a huge effort to develop and maintain the application (ASANOVIC et al., 2009). Even when successful, this development flow results in a code that is tightly coupled to the target

[1] `https://www.top500.org/system/178250`
[2] `https://www.gsmarena.com/samsung_galaxy_s9-8966.php`

hardware, which is not desirable in the hybrid scenario where the accelerator's technology is changing fast, e.g., Cell Broadband Engine Architecture (2006), Graphics Processing Unit (2007), Xeon Phi (2013) (DONGARRA et al., 2017).

The pressure on parallel programming tools has contributed to the popularization of the task-based programming model. While the traditional parallel programming paradigm relies on low-level abstractions like threads and explicit synchronizations, the task-based model describes the application in terms of dependent sequential (or parallel) tasks. Explicit synchronizations are replaced by tasks dependencies that can be, in several cases, inferred automatically from data access. The task-based model is implemented by several programming models: OpenMP 4 (OPENMP..., 2013), OmpSs (DURAN; AYGUADÉ, et al., 2011), PaRSEC (BOSILCA, George et al., 2012), XKaapi (GAUTIER, T. et al., 2013), StarPU (AUGONNET, Cédric et al., 2011).

Despite the growing availability of tools to program and execute task-based applications on hybrid platforms, there are very few analysis tools with a focus on such kind of parallel applications. One of the main reasons is that the burden of getting the maximum performance from the machine shifts from the application to the runtime developer, who implements a given scheduling heuristic. Usually, developers rely on already-established analysis tools since highly-summarized metrics, e.g., makespan, speedup, and efficiency, are too simplistic to allow understanding performance issues in such scenario. However, existing tools that are capable of giving insights to developers, such as Paraver (PILLET et al., 1995), Vite (COULOMB et al., 2009), and others (KNÜPFER et al., 2008; ISAACS, K. E. et al., 2014; BRENDEL et al., 2016; PAGANO et al., 2013), are unsuited to runtime developers because assumptions and performance bottlenecks are different from classical parallel programs and because the traces of task-based applications can be much richer (e.g., with task-dependencies or information about the state of the scheduler) than the ones of standard MPI or classical OpenMP applications.

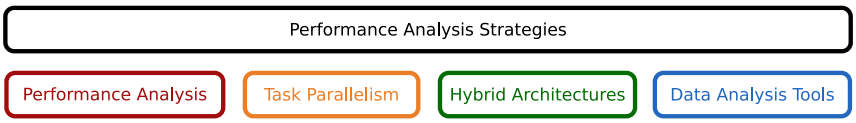
1.1 Objectives and Contributions

Considering the previously discussed scenario, the main objective of this thesis is to **provide appropriate performance analysis strategies for task-based applications executing on hybrid platforms supported by dynamic runtime systems**. We believe that, with a complex hardware/software stack, a meaningful analysis strategy should take into account:

- **application aspects** (e.g., algorithm structure, finer granularity, task dependencies)
- **platform specification** (e.g., heterogeneous processing and communication capabilities)
- **runtime system state** (e.g., dynamic scheduling, internal metrics)

In order to fulfill these requirements, our approach is anchored on the four topics shown in Figure 1.3. First, we build on well-established **Performance Analysis** concepts, such as execution traces and timeline views. Second, we exploit both characteristics of **Task Parallelism**, such as data dependencies and dynamic scheduling and of **Hybrid Architectures** such as heterogeneous processing power. Finally, we rely on modern **Data Analysis Tools**, such as R language and its libraries, to read, combine and process execution traces in order to obtain visual representations of the application execution.

Figure 1.3: The basis of the proposed analysis strategies.



Source: The Author

Our main contributions are the following:

- a set of performance analysis strategies to meet the requirements of task-based applications supported by dynamic runtime systems;
- an incremental approach where enrichments to standard visualizations and synchronized extra panels are developed and incorporated on-demand;
- case-study analyses demonstrating how to achieve performance improvements on already optimized task-based applications.

The complex hardware/software stack also emphasizes the need to employ reproducible approaches. Therefore, we also employ a reliable experimental methodology based on Git, Org-mode, Spack, and Zenodo. This methodology enables us to make available a companion dataset containing the source-code and the results data of our experiments. Our extensions to the Git and Org-mode workflow proposed by Stanisic, Legrand, and Danjean (2015) and the availability of experiments data can be considered as additional contributions of this thesis.

1.2 Research Context

The present work is conducted under the context of a joint Ph.D. between the Graduate Program in Computer Science (PPGC) of the Institute of Informatics at the Federal University of Rio Grande do Sul (UFRGS) in Brazil and the Doctoral School on Mathematics, Information Sciences and Technologies, and Computer Science (ED-MSTII) of the University Grenoble Alpes (UGA) in France. Besides, this work is involved in the context of the International Associated Laboratory in High Performance and Ubiquitous Computing (LICIA).

At UFRGS, this research is developed under the supervision of Nicolas Maillard in the Research Group on *Parallel and Distributed Processing* (GPPD) and at UGA under the supervision of Arnaud Legrand in the team on *Performance Evaluation and Optimization of Large Infrastructures and Systems* (POLARIS) of the Grenoble Informatics Laboratory (LIG). In addition to my official advisors, this work is developed in a strong cooperation with:

- Lucas Mello Schnorr from UFRGS about performance analysis techniques and StarPU-MPI applications. Lucas has been, in practice, a co-supervisor of this thesis;
- Samuel Thibault from University of Bordeaux about the StarPU runtime system;
- Luka Stanisic from Max Planck Computing and Data Facility about performance analysis of StarPU applications;
- Vincent Danjean from University Grenoble Alpes about task-based programming.

1.3 Thesis Outline

The remainder of this text is structured as follows:

- **Chapter 2** presents background concepts of the hardware and software stack of current HPC platforms. First, we present an overview about the multi-level parallelism of the architecture. Second, we discuss programming strategies for hybrid architectures with emphasis on task-based programming.
- In **Chapter 3** we present the state of the art of performance analysis tools. We discuss both traditional BSP-based approaches and Task-oriented ones.
- **Chapter 4** presents some steps to make this work more transparent and repro-

ducible. This comprises the use of literate programming to write this report and how we keep track of experiments data.

- In **Chapter 5** we detail the proposed visualization strategies to analyze task-based applications.
- **Chapter 6** presents our results when analyzing applications with the proposed strategies. We present two case studies, the first one on a single hybrid node and the second on a hybrid cluster.
- In **Chapter 7** we detail our debug-based performance analysis strategies. These strategies are useful for understanding specific scheduler decisions that cannot be totally explained using only the strategies presented in Chapter 5.
- **Chapter 8** presents our results when analyzing executions with the proposed debug strategies. We present an investigation of the StarPU scheduler state for a set of tasks that seem to be delayed due to potential scheduling mistakes.
- **Chapter 9** presents the conclusions and perspectives of this thesis.

2 CONTEXT

For a long time, parallelism has been a major strategy to increase the processing power of computing systems. Low-level techniques such as the Instruction Level Parallelism (HENNESSY; PATTERSON, 2017) were incorporated into the hardware design, and are now widespread in almost all CPU architectures. These hardware-managed techniques have delivered the benefits of parallelism with no cost in terms of software development. In contrast, the exploitation of parallelism in other levels of the hardware/-software stack was restricted to HPC researchers and scientific applications developers.

This community has developed successful programming tools such as the Message Passing Interface (MPI) to handle the communication among computing nodes and OpenMP to handle shared memory platforms. MPI (GROPP; LUSK; SKJELLUM, 2014) is a *de facto* standard for parallel programming in distributed memory. Using MPI, processes can communicate by exchanging messages, e.g., paired send/receive messages or collective communications. OpenMP (CHAPMAN et al., 2008) is a collection of compiler directives and library functions to enable shared-memory multi-thread programming. Its directives were initially designed to parallelize loop-based algorithms, but recent versions also include support for other designs (e.g., sections, tasks). Classical parallel programs are commonly designed following the concepts of SPMD (Single Program Multiple Data) and BSP (Bulk-Synchronous Parallel) models. SPMD programs run the same code on a set of computing units; each computing unit uses its unique identifier to know on which part of the problem it should work (FOSTER, 1995; SOTTILE; MATTSON; RASMUSSEN, 2009). BSP programs are designed as a series of super-steps (VALIANT; G., 1990), i.e., each computing unit performs some local computation, these computations are followed by a communication phase and finally by a synchronization one. Despite the popularity of these approaches, the emergence of computing nodes with massive multicore processors and enhanced with accelerators has exposed the limitations of such strategies.

In this chapter, we discuss the hardware and software stack of current HPC hybrid platforms. First, in Section 2.1 we address the parallelism of HPC platforms, from the low-level ILP techniques to the multicore processors and accelerators used by the powerful computing systems today. Section 2.2 presents the principles of task-based programming. Finally, Section 2.3 presents some runtime systems that enables task-based parallel programming on hybrid platforms.

2.1 Hybrid HPC Architectures

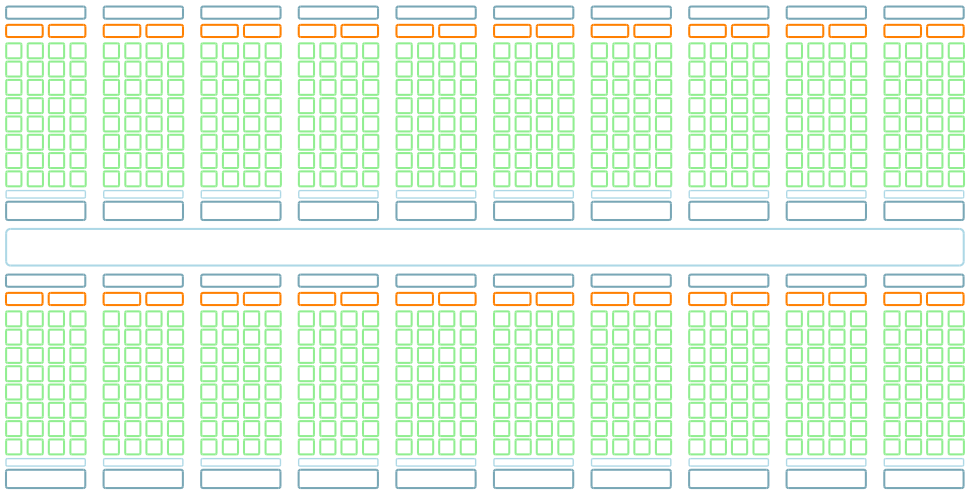
Parallelism has been exploited for a long time in computer architectures to achieve performance gains. Several parallelism techniques are employed inside the processor architecture to overlap the execution of instructions and decrease the time required to perform the whole computation, what is known as Instruction-Level Parallelism (ILP). Most of these techniques such as pipelining, register renaming and use of multiple redundant functional units are transparent to programmers since to their point of view the results are exactly the same of a sequential execution (HENNESSY; PATTERSON, 2017). Current processors also support another low-level parallelism features like vectorial instructions (*SIMD* extensions) which are more or less transparent to programmer since they are exploited by the compiler, automatic (YAZDANPANAH, 2017) or guided by directives (INTEL, 2017a), or included in high-performance libraries (WANG et al., 2014).

The paradigm shift from sequential processors to multicore chips has disclosed another parallelism layer to programmers. In contrast to the ILP where the parallelism was almost entirely handled by the hardware, in the multicore model, the software must be upgraded to take advantage of the computational power provided by multicore architecture (ASANOVIC et al., 2009). Since the number of cores continues to increase significantly in new hardware generations, the programming tools should provide a way to express the parallelism independently of the actual number of cores. This way, the software can still scale in future processors with higher core counts. Programming tools for multicore processors will be discussed in Section 2.2.

To fulfill the ever-growing need for computation power of High-Performance Computing applications, current HPC platforms are hybrid machines that rely not only on mainstream multicore processors but also in specialized accelerators. These devices are attached to the main system providing massive computational power with a reduced cost. For this reason, accelerated nodes are the main technology used by the powerful HPC platforms, e.g., 110 of the fastest supercomputers listed in the June 2018 Top500 list (MEUER et al., 2014) have nodes enhanced with some accelerator device, as shown in Figure 2.2.

Despite the computational power provided by hybrid machines, efficiently exploit both multicore CPUs and accelerators is challenging. Accelerator devices have very different design and requirements. Usually, they provide massive parallelism through a higher number of simplified cores. In comparison with a CPU, the design of an accel-

Figure 2.1: The layout of a hypothetical accelerator. Green boxes represent computing units (cores), orange ones the control system and blue ones the memory hierarchy.



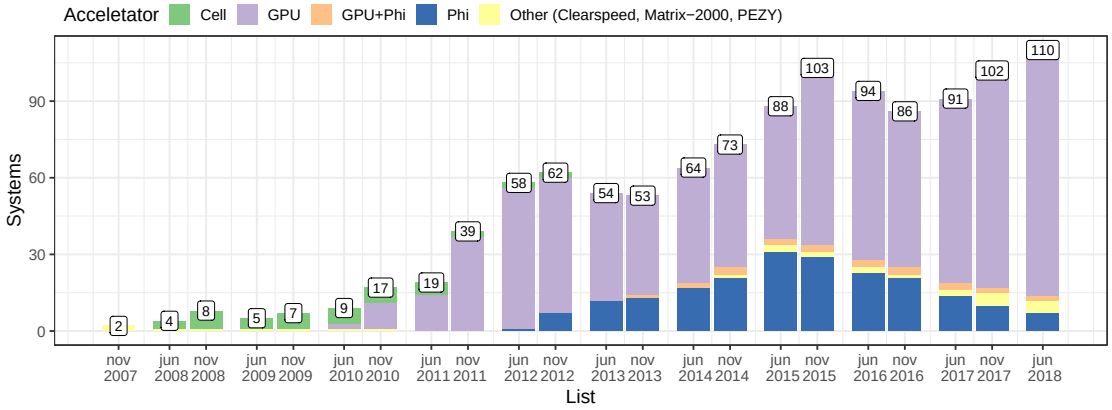
Source: The Author

erator favors the computing units rather than control and memory systems as illustrated by the hypothetical accelerator layout of Figure 2.1. Some accelerator technologies are based on specialized instruction sets, which means that they require proper compiler and programming tools (NVIDIA, 2017b). The particular memory hierarchy requires some low-level operations and/or optimizations that normally are transparent in CPU architectures (NVIDIA, 2017a; JEFFERS; REINDERS, J., 2013).

The Figure 2.2 shows the growing use of accelerators devices in the systems ranked in the Top500 list. On the one hand, this series shows that accelerated nodes are becoming commonplace in the HPC landscape, but, it also shows that accelerators technology changes fast. A decade ago, the dominant technology was the Cell Broadband Engine Architecture (CHEN et al., 2007), a heterogeneous multicore chip composed of one PowerPC core (PPE) and eight simple cores (SPE) specialized for SIMD instructions. Then, the advent of CUDA (SANDERS; KANDROT, 2010) and OpenCL (STONE; GOHARA; SHI, 2010; GASTER, 2012) technologies has facilitated the use of GPU cards for general purpose processing. More recently, Intel has introduced the Many Integrated Core architecture (MIC) (DURAN; KLEMM, 2012), an accelerator technology based on the x86 instruction set, commonly referred by its product name: Xeon Phi (JEFFERS; REINDERS, J., 2013).

Early generations of accelerator devices have presented some limitations that are being addressed in the recent versions. The performance gap between single and double precision was one of these issues. Single precision operations were from 8 up to 14 times

Figure 2.2: Evolution of the use of accelerators in TOP500 systems.



Source: The Author

faster than double precision ones in first generations of Cell and Tesla GPUs. In newer GPUs, this gap was reduced by a factor between 2 and 3 (DONGARRA et al., 2017). The device memory size and bandwidth were also limitations for several applications, but these issues were reduced either by hardware improvements or by software solutions. The bidirectional ring interconnection of Xeon Phi devices was also a source of performance disturbances that were reduced in the new Knights Landing version that uses a 2D mesh interconnection (JEFFERS; REINDERS, J.; SODANI, 2016).

The evolution of accelerators technology has promoted them from a trend to a solid approach for HPC platforms. This was achieved by reducing limitations while keeping their good ratio FLOP per money and increasing performance within the same power budget.

2.2 Programming Hybrid Architectures

The development of a software stack that allows to efficiently exploit hybrid architectures has become a major challenge in HPC research. The multiple levels of parallelism in the same system, each level comprising generally different memory and processor technologies make the use of classical programming tools ineffective. Traditional parallel programming tools were designed for homogeneous environments and normally target one level of parallelism. Using them in a hybrid platform implies combining several tools, for example, MPI to communicate the nodes, OpenMP to exploit the parallelism of multicore processors and CUDA/OpenCL to manage the GPUs used as accelerators.

The use of different models with no native integration frequently results in non-

portable applications, readability problems in the code and hand-tuned optimizations. Achieving good performance with this strategy is challenging, and comes with a considerable programming effort. Despite that, a minor platform update, such as adding a new accelerator device, may require redoing all the tuning and optimization steps. In the near future, the advent of platforms with higher core counts and complex memory hierarchies will make intractable the use of such low-level strategies.

One promising approach to manage the heterogeneity of the hardware is the use of an abstraction layer. This layer acts as a middleware between the user application and the low-levels libraries that manage the hardware. This way, it is possible to design the algorithm without considering the hardware aspects. The algorithm is described in terms of tasks and its iterations. These tasks could have dependencies that are someway provided by the programmer. All other operations, such as synchronizations, resource allocation, memory transfers, and load balancing are delegated to a runtime system. To handle the heterogeneity, each task can have multiple implementations to target the different hardware of each processing unit.

In the remainder of this section, we will first present a discussion about the concept of a task and its support on parallel programming tools and then how state-of-the-art tools are using this model to handle hybrid HPC platforms.

2.2.1 Task-Based Programming

The concept of divide a sequential problem in some independent portions of work, which can be executed in parallel, was initially proposed in the 1990s. One of the first parallel programming tools to support this task-based parallelism was Cilk (BLUMOFE, Robert D et al., 1996), originally developed as an open source software at MIT (GROUP, 2017) and now a deprecated feature of Intel C and C++ compilers (INTEL, 2017b).

In this model of parallelism, each task can perform an asymmetrical amount of sequential work and the total number of tasks is, in general, much bigger than the number of physical processing resources. A huge amount of tasks with small granularity provides more flexibility to the scheduler, resulting in a parallel program that is not tightly coupled to the target hardware. The tasks dependencies are expressed in a *fully strict* design, which means that a parent task waits for its direct successors (BLUMOFE, Robert D et al., 1996).

Since tasks are defined without considering the target hardware, task-based tools

rely on an intermediary software layer to provide good performance. This layer consists of a dynamic runtime system which is responsible for handling task scheduling, load balancing, and tasks synchronization. In Cilk, the runtime system is based on a work-stealing scheduling policy, a distributed strategy where an idle processor can steal tasks from another one (BLUMOFE, R. D.; LEISERSON, C. E., 1994).

With the advent of multicore processors in the 2000s, the task-based parallelism was adopted by other programming models such as OpenMP, with **pragma omp task** and **pragma omp taskwait** directives of version 3.0 (OPENMP..., 2008), and Intel Threading Building Blocks (TBB) (REINDERS, James, 2007). The task-based paradigm was also implemented by several experimental tools focusing on different hardware platforms: distributed memory (GAUTIER, Thierry; BESSERON; PIGEON, 2007; DINAN et al., 2009; MOR; MAILLARD, 2011), heterogeneous processors (BENDER; RABIN, 2000), GPUs (CEDERMAN; TSIGAS, 2011; TOSS; GAUTIER, Thierry, 2012), Hybrid Nodes(AUGONNET, Cédric et al., 2011; PINTO, Vinicius Garcia; MAILLARD, 2012; PINTO, Vinicius Garcia, 2013; GAUTIER, Thierry; FERREIRA LIMA, et al., 2013)

The synchronization primitive **sync** introduced by Cilk to express dependency relationships among tasks, as well its OpenMP equivalent **pragma omp taskwait**[1], are appropriate to algorithms with a recursive structure. This is the case of several classical divide and conquer applications used to illustrate how Cilk works such as the *Fast Fourier Transform*, the *Merge Sort* algorithm or the *Strassen* method for matrix multiplications. Figure 2.3 illustrates an example of a recursive algorithm in Cilk. This structure rapidly generates a huge number of tasks, and the Cilk steal semantics quickly distribute the workload among the workers. On the other hand, for other algorithm structures such as loop-based ones this strategy may incur in performance bottlenecks. Figure 2.4 illustrates different ways of declaring dependencies in Cilk and OpenMP. The **sync** (Fig. 2.4(a)) and **taskwait** (Fig. 2.4(b)) synchronization primitives create a dependency with all the previously submitted tasks. In this example, the DSYRK task of the first iteration of for loop on line 11 actually depends only on the first DTRSM in the previous loop (line 5). Ideally, the first DSYRK could be executed at the same time of all the remaining DTRSM ($i >= 2$). Nevertheless, the synchronization point of line 10 forces it to wait for all the DTRSM tasks. This kind of bottleneck is avoided when the task dependencies are

[1] Although similar, the Cilk **spawn**-**sync** and OpenMP *task-taskwait* primitives have different behavior. In general terms both **spawn** and **omp task** primitives are used to submit new tasks. However, in Cilk, a **spawn** call starts the execution of the new task and pushes the continuation of the parent task to a queue. For more information, see concepts of *fast/slow* clones and *work-first* discussed by Frigo, Charles E. Leiserson, and Randall (1998) and Guo et al. (2009).

Figure 2.3: An example of recursive tasks in Cilk.

```
1  cilk void OptimizedStrassenMult(REAL *C, REAL *A, REAL *B,
2                                  uint MatSize, uint RowWidthC,
3                                  uint RowWidthA, uint RowWidthB){
4      uint QuadSize = MatSize >> 1;
5      ...
6      spawn OptimizedStrassenMult(M2, A11, B11, QuadSize,
                                   QuadSize, RowWidthA, RowWidthB);
7      spawn OptimizedStrassenMult(M5, S1, S5, QuadSize, QuadSize,
                                   QuadSize, QuadSize);
8      spawn OptimizedStrassenMult(T1sMULT, S2, S6, QuadSize,
                                   QuadSize, QuadSize, QuadSize);
9      spawn OptimizedStrassenMult(C22, S3, S7, QuadSize,
                                   RowWidthC, QuadSize, QuadSize);
10     spawn OptimizedStrassenMult(C11, A12, B21, QuadSize,
                                   RowWidthC, RowWidthA,RowWidthB);
11     spawn OptimizedStrassenMult(C12, S4, B22, QuadSize,
                                   RowWidthC, QuadSize, RowWidthB);
12     spawn OptimizedStrassenMult(C21, A22, S8, QuadSize,
                                   RowWidthC, RowWidthA, QuadSize);
13     sync;
14     ...
15 }
```

Source: Amended from (GROUP, 2017)

expressed in function of the data access.

Since the use of global task dependencies (**spawn-sync** model) can lead to performance bottlenecks, current HPC task-based tools rely on data dependency strategies. This design consists in declaring the access mode (**read**, **write**, **read-write**) for each piece of data accessed by the task. Then, the tasks are submitted in a sequential way. From this information (submission order and access mode), a runtime system can build a task dependency graph. Data dependency models are referred by different names in the literature, such as *data-flow* model (GAUTIER, T. et al., 2013), *sequential task flow* (STF) (AGULLO, Emmanuel et al., 2016) and *superscalar* (DONGARRA et al., 2017). The last one comes from its similarity with superscalar processors which analyzes the sequential instructions flow to identify which ones do not depend on others and, consequently, can be executed in parallel.

In the data dependency model, tasks are created with non-blocking calls. In this design, a task is possibly not ready for execution at the moment of its creation. Task dependencies are automatically inferred and managed by the runtime system. Once all the dependencies are solved, the task is marked as ready for execution. This is differ-

Figure 2.4: A pseudo-code illustrating how to express dependencies in Cilk and OpenMP

(a) Cilk

```
for (k = 0; k < N; k++) {

  spawn  dpotrf(A[k, k], ...);
  sync;
  for (i = k+1; i < N; i++) {

    spawn dtrsm(A[i, k],
                A[k, k], ...);
  }
  sync;
  for (i = k+1; i < N; i++) {

    spawn dsyrk(A[i, i],
                A[i, k], ...);
    for (j = k+1; j < i; j++) {

      spawn dgemm(A[i, j],
                  A[i, k],
                  A[j, k], ...);
    }
  }
  sync;
}
```

(b) OpenMP Tasks

```
for (k = 0; k < N; k++) {
  #pragma omp task
  dpotrf(A[k, k], ...);
  #pragma omp taskwait
  for (i = k+1; i < N; i++) {
    #pragma omp task
    dtrsm(A[i, k],
          A[k, k], ...);
  }
  #pragma omp taskwait
  for (i = k+1; i < N; i++) {
    #pragma omp task
    dsyrk(A[i, i],
          A[i, k], ...);
    for (j = k+1; j < i; j++) {
      #pragma omp task
      dgemm(A[i, j],
            A[i, k],
            A[j, k], ...);
    }
  }
  #pragma omp taskwait
}
```

(c) OpenMP Tasks with Data Dependencies

```
for (k = 0; k < N; k++) {
  #pragma omp task depend(inout:A[k, k])
  dpotrf(A[k, k], ...);
  for (i = k+1; i < N; i++) {
    #pragma omp task depend(in:A[k, k]) depend(inout:A[i, k])
    dtrsm(A[i, k],
          A[k, k], ...);
  }
  for (i = k+1; i < N; i++) {
    #pragma omp task depend(in:A[i, k]) depend(inout:A[i, i])
    dsyrk(A[i, i],
          A[i, k], ...);
    for (j = k+1; j < i; j++) {
      #pragma omp task depend(in:A[i, k], A[j, k])
                       depend(inout:A[i, j])
      dgemm(A[i, j],
            A[i, k],
            A[j, k], ...);
    }
  }
}
```

Source: The Author

ent from the global task dependencies model used by Cilk and OpenMP where a task is always ready for execution at its creation time. Version 4 of the OpenMP standard has extended the task model of version 3 to support data dependencies. The code of Figure 2.4(c) illustrates how these dependencies are declared in an OpenMP program. Since task creation calls are non-blocking, this code will generate a pool of tasks. When a given task is completed, the runtime enables its successors in the dependency graph which allows finer synchronizations. This also gives more flexibility to the scheduler thanks to the availability of a greater number of tasks ready for execution. In the code example of Figure 2.4(c), the DPOTRF task of second iteration ($k = 2$) can be executed just after the end of three tasks from the previous iteration (one DPOTRF, one DTRSM and one DSYRK), while in the **spawn**-**sync** model, it should wait for all the tasks of the previous iteration.

2.3 Runtime Systems for Task-Based Parallel Programming

One of the major features of the task parallelism model is the possibility of designing an algorithm without considering architecture implementation details. Since the tasks are consistent, e.g., with no access to global variables, and their dependencies are correctly described, it is possible to postpone their implementation. This abstraction is also very useful to handle the heterogeneity of the hardware. For example, a programmer can provide multiple implementations for the same task in order to target different hardware architectures. Once all these implementations respect a common interface they can be switched without major consequences.

In this context, programmers have no control over which implementation of the task will be actually executed; they just provide the algorithm (the tasks and its dependencies) and the implementations. During the execution, an intermediary layer, the runtime system, will choose which implementation execute depending on several factors, such as the resource's availability and the provided task implementations.

Several parallel programming tools have been developed in recent years following the concepts of task-parallelism and runtime-taken decisions. In the rest of this section, we will present some of them.

2.3.1 PaRSEC

PaRSEC (Parallel Runtime Scheduling and Execution Controller), formerly know as DAGuE (BOSILCA, George et al., 2012), is a runtime system for task-based programming in distributed hybrid architectures. This tool provides automatic data transfers among workers by analyzing the access mode of each piece of data used by the task. PaRSEC has a simple distributed scheduler that implements a basic work stealing strategy.

PaRSEC uses a domain specific language (DSL) to express parallelism in linear algebra applications (BOSILCA, G. et al., 2011). This DSL allows describing a Parameterized Task Graph (PTG) (DANALIS et al., 2014) to represent a collection of tasks. In this approach, each task has a name, some parameters, the range of each parameter, data affinity, parameterized data dependencies, and a body. The parameterized data dependencies depict all the inputs and the outputs of each task with their respective origin or destination. The PTG representation is proportional to the number of task types and independent of the total number of tasks and allows to directly access the ascendants and the descendants of a task, which means that PaRSEC does not unroll the entire DAG in memory.

The code snippet of Figure 2.5 illustrates an example of a task in ParSEC. Lines 4-6 describe the parameterized data dependencies of the task POTRF. Each instance of this task will read and write on a data T. In line 4, the left arrow $<-$ indicates that data T will be read either from memory (A(k,k), if k is 0) or from another task (HERK(k$-$1, k)); on line 5, the right arrow ($->$) indicates that the data T written by the task POTRF will be consumed by a set of TRSM(k+1 .. max, k) tasks; line 6 indicates that the POTRF task will be the last writer of data A(k, k). Body region on lines 7-11 specifies the source code of the task, additional body regions can be included to target another hardware, e.g., GPUs. Recently, ParSEC has included support for a conventional data dependency description similar to OpenMP 4 (HOQUE et al., 2017).

2.3.2 OmpSs

OmpSs (DURAN; AYGUADÉ, et al., 2011) is the current version of the "Ss" family of parallel programming tools. Previous versions were named in according to the target hardware, e.g., GridSs (BADIA; LABARTA, Jesús, et al., 2003), CellSs (BELLENS et al., 2006), GPUSs (AYGUADÉ et al., 2009) and SMPSs (BADIA; HERRERO,

Figure 2.5: An example of task description in ParSEC.

```
1  POTRF(k)
2  k = 0 .. max
3  : A( k, k )
4  RW <- (k == 0) ? A(k, k) : T HERK(k-1, k)
5     -> T TRSM(k+1 .. max, k)
6     -> A(k, k)
7  BODY
8  {
9     /* C code */
10 }
11 END
```

Source: Hoque et al. (2017)

et al., 2009). The OmpSs model relies on compiler directives to support asynchronous parallelism and accelerators. The reference implementation uses the Mercurium (BSC, 2018a) source-to-source compiler to transform the directive annotated code into a real parallel code and Nanos++ (BSC, 2018b) as a runtime system.

Tasks are defined with annotations on sequential code like in OpenMP. Dependency relationships between tasks can be expressed providing the access mode ($copy_{in}$, $copy_{out}$, $copy_{inout}$) of each piece of data, which allows fine-grain synchronizations in a very similar way to the data dependency model of OpenMP 4 discussed in Section 2.2.1. In fact, this and several other OmpSs concepts have been incorporated in the versions 3 and 4 of the OpenMP specification (BSC, 2017b).

OmpSs also supports multi-version tasks, which enables the use of accelerator devices. Using the **implements** directive, it is possible to define CUDA or OpenCL replacements for a CPU task. At execution, the OmpSs runtime can choose one of these alternate implementations instead of the original one. Figure 2.6 illustrates how multi-version tasks are defined in OmpSs. The CPU task calculate_forces (defined at line 3) can be replaced at runtime by the equivalent calculate_forces_cuda , since the definition of this one (line 5) specifies that it implements a CUDA alternative to the main code of calculate_forces .

The Nanos++ runtime system of OmpSs provides several task scheduling policies. These policies vary from simple strategies like FIFO/LIFO global shared queues to sophisticated ones combining profiling or work stealing designs. The DBF (*Distributed Breadth First*) and WF (*Work First*) policies are based on work stealing; the main difference is that the first implements the help-first design while the second is equivalent to the Cilk work-first behavior. The *Socket-Aware Scheduler* implements a variation of

Figure 2.6: A pseudo-code of tasks definition in OmpSs. The **device** and **implements** keywords are used to provide an alternative implementation to calculate_forces task.

```
1  #pragma omp target device(smp) copy_deps
2  #pragma omp task out([size] pout) in([npart] part)
3  void calculate_forces(int size, float time, int npart, Part*
       part, Particle* pout, int gid){
4      /* CPU implementation */
5  }
6
7  #pragma omp target device(cuda) ndrange(1,size,128) copy_deps
       implements (calculate_forces)
8  #pragma omp task out([size] pout) in ([npart] part)
9  __global__ void calculate_forces_cuda(int size, float time, int
       npar, Part* part, Particle* pout, int gid){
10     /* CUDA implementation */
11 }
12
13 void Particle_array_calculate_forces(Particle* input, Particle*
       output, int npart, float time) {
14   for (int i = 0; i < npart; i += BS ){
15     calculate_forces(BS, time, npart, input, &output[i], i);
16   }
17 }
```

Source: Jesús Labarta (2015)

work stealing to target NUMA aspects. The *Bottom level-aware Scheduler* is also a work-stealing variation to target machines with heterogeneous cores, e.g., ARM big.LITTLE. In this algorithm, idle fast cores steal tasks from the slow ones. The *Versioning* algorithm is designed to handle tasks with multiple implementations. This strategy computes performance models for each task implementation and then use this information to schedule the task on the worker where it will be finished first (BSC, 2017a).

2.3.3 XKaapi

XKaapi (GAUTIER, T. et al., 2013) is a runtime system for task-based programming on hybrid architectures. The current implementation focus on platforms with multi-core processors and multiple GPUs. This tool has introduced the concept of multi-version tasks to target hybrid CPUs and GPUs architectures (HERMANN et al., 2010). XKaapi provides dynamic task creation and uses work stealing as a basis for its scheduling policies.

XKaapi offers a C++ API to express the parallelism. An XKaapi task has a single signature and one or more implementations. The code snippet of Figure 2.7 illustrates an example of task definition in XKaapi. The *task signature* (lines 1-4) provides the number of parameters and their access mode. From these access modes (*read*, *write*, *readwrite*) the runtime can compute the dependencies among the tasks. In this example, the task **TaskExp** has two implementations, one for CPU and other for GPUs. In the last one, the access mode is also used by the runtime to perform automatic data transfers before and/or after the task execution. XKaapi tasks are created with **ka**::**Spawn** calls. Despite this API, the tool is also able to run OpenMP code either using the KStar compiler (INRIA, 2018) or replacing the standard OpenMP runtime by an XKaapi based one.

The XKaapi runtime is inspired by several concepts of Cilk. One of these concepts is the capability to dynamically create children tasks from running ones. Another relevant influence of Cilk is the use of work stealing based scheduling policies. In XKaapi, the computation of task dependencies is integrated into the execution of a steal operation. When a task is finished, the worker starts the execution of the children ones in a FIFO order without computing any dependency, which is valid thanks to sequential consistency. If a child task was stolen, the worker switches to a work stealing strategy. Dependencies are computed by idle workers searching for ready tasks using the data access modes provided by the programmer. This strategy moves the cost of managing dependencies from

Figure 2.7: An example of XKaapi task with multiple implementations.

```
struct TaskExp: public ka::Task<2>::Signature<
    ka::R<ka::range2d<double>>,
    ka::RW<ka::range2d<double>>
>{};

template<> struct TaskBodyCPU<TaskExp> {
  void operator (ka::range2d_r<double> A, ka::range2d_rw<double>
      C){
    /* CPU implementation */
  }
};

template<> struct TaskBodyGPU<TaskExp> {
  void operator (ka::gpuStream stream, ka::range2d_r<double> A,
     ka::range2d_rw<double> C){
    /* CUDA implementation */
  }
};
```

Source: The Author

a busy worker (during the task creation) to an idle one (when searching for a ready task) (GAUTIER, T. et al., 2013; LIMA et al., 2015).

The work-stealing scheduler of XKaapi implements some additional optimizations to target hybrid architectures. One of these optimizations is the *data-aware work stealing* that aims to reduce data transfers between host and devices. Another policy, the *locality-aware work stealing* aims to reduce the cache invalidations by considering the data access mode during the task assignment. XKaapi also has a HEFT-like policy based on precalibrated performance models (LIMA et al., 2015).

2.3.4 StarPU

StarPU (AUGONNET, Cédric et al., 2011) is a runtime system for task-based programming on hybrid architectures. The runtime was initially designed to handle single-node hybrid platforms composed of multicore processors (CPUs) and accelerators (GPUs, Intel Xeon Phi). To efficiently exploit the parallelism of the platform, StarPU relies on multiple implementations of the same tasks, e.g., with CPU and/or GPU versions. The runtime scheduler decides on-the-fly where to execute the tasks considering the available processing resources, their type, the current locations of data, and the provided task implementations.

Figure 2.8: An example of StarPU *codelet*.

```
1  static struct starpu_codelet cl = {
2      .where = STARPU_CPU | STARPU_CUDA | STARPU_OPENCL,
3      /* CPU implementation of the codelet */
4      .cpu_funcs = { scal_cpu_func, scal_sse_func },
5      .cpu_funcs_name = { "scal_cpu_func", "scal_sse_func" },
6  #ifdef STARPU_USE_CUDA
7      /* CUDA implementation of the codelet */
8      .cuda_funcs = { scal_cuda_func },
9  #endif
10 #ifdef STARPU_USE_OPENCL
11     /* OpenCL implementation of the codelet */
12     .opencl_funcs = { scal_opencl_func },
13 #endif
14     .nbuffers = 1,
15     .modes = { STARPU_RW }
16 };
```

Source: (INRIA; CNRS; UNIVERSITÉ DE BORDEAUX, 2017)

Tasks are defined as *codelets* in StarPU idiom. A *codelet* is a data structure used to represent a computational kernel. Each *codelet* may contain several task attributes such as multiple implementations of the computational kernel, data buffers accessed by the codelet, access mode of each data buffer, a task identifier, and performance and energy models. The code snippet of Figure 2.8 presents an example of a *codelet* definition. This *codelet* has implementations for three hardware architectures (CPU, CUDA, and OpenCL). There are also two implementations for a CPU hardware (scal_cpu_func and a *SIMD* variation scal_sse_func). During the execution, the runtime will choose one of these four implementations according to the hardware available and the scheduling decisions.

StarPU codelets can be directly submitted as tasks using the C API which provides functions to submit new tasks (starpu_task_submit()) and to wait for all previous tasks (starpu_task_wait_for_all()) in a similar way to omp task/ taskwait directives from OpenMP. It is also possible to use a high-level pragma interface building StarPU as a GCC plugin or translate from standard OpenMP code using the KSTAR source-to-source compiler (INRIA, 2018).

StarPU offers several task scheduling policies. The DMDA and DMDAS policies are based on precalibrated performance models, while the WS and LWS use a work-stealing design, stealing tasks from the most loaded worker (original) or the most loaded neighbor worker (locality version). The PRIO policy only relies on priority hints specified

by the application programmer.

The DMDA (*Deque Model Data Aware*) and DMDAS (*Deque Model Data Aware Sorted*) algorithms are members of a family of StarPU schedulers that take the predicted task duration and data transfer duration into account when performing the task scheduling. These strategies are based on *list scheduling*, i.e., every time a resource is idle, if a task is ready, it will be scheduled on this particular resource. Such a scheduler therefore never leaves a resource idle on purpose, which ensures the well-known $(2 - 1/p)$ competitive ratio for homogeneous machines(GRAHAM, 1966). Deciding which ready task to select has a major influence in practice, and the classical heuristic consists of prioritizing tasks based on the critical path. However, the critical path notion is dynamic and obtaining a proper estimation can be quite challenging. With heterogeneous computing resources, such prioritization is generally done with variants of the HEFT (Heterogeneous Earliest Finish Time) strategy (TOPCUOGLU; HARIRI; WU, 2002). The DMDA and DMDAS algorithms are greedy heuristics that schedule tasks in the order they become available, taking into account the predicted task duration, the estimated data transfer duration between CPUs and GPUs and the relative performance of resources on each computation kernel when making its decision. The DMDAS algorithm improves DMDA decisions by sorting tasks on per-worker lists by the number of data-slices already transferred and by priority, which can be expensive when the number of tasks is large. It is therefore rather close to the original HEFT algorithm by respecting priorities and taking past scheduling decisions into account.

The WS (*Work Stealing*) and LWS (*Locality Work Stealing*) algorithms use one list per worker; new tasks are kept local by default. When a worker is idle, in the WS policy, it steals tasks from the most loaded worker. The LWS policy, on the other hand, imposes that the worker must steal first from the most loaded neighbor worker. This victim's choice differs from the classical work-stealing algorithm(BLUMOFE, Robert D.; LEISERSON, Charles E., 1999) where the victim is chosen using a random strategy.

The PRIO algorithm uses a mere centralized list that is shared by all the workers. This list keeps the tasks sorted respecting the *priorities* specified by the application programmer.

StarPU was designed to support single-node hybrid architectures. In order to target distributed architectures, the StarPU-MPI extension (AUGONNET, Cédric et al., 2012; AGULLO, E.; AUMAGE, et al., 2017) was proposed. This extension relies on the MPI specification as a mean of communication among the nodes of the network. The main

characteristic of the extension is that it runs one independent StarPU runtime instance per MPI node rather than a globally distributed one over the entire platform. For that reason, the scheduler of each runtime instance is responsible for processing a part of the whole application DAG. Existing dependencies among nodes are satisfied through MPI send/receive point-to-point operations and managed like any other task handled by the runtime. Since the domain decomposition is static, these send/receive operations are automatically queued along tasks within the DAG. One specific thread is created to handle MPI communications on each node. On completion of a task whose output is needed by a task on another MPI node, this MPI thread posts the corresponding send operation. On the other node, on reception of the data from MPI, the corresponding MPI thread releases the execution of the corresponding task.

Although StarPU is not yet a widely-used tool, there exist several real applications that are now fully ported or designed on top of this runtime system. These applications target different domains ranging from linear algebra solvers to Computational Fluid Dynamics (CFD) applications. The PaStiX (Parallel Sparse matriX package) (HENON; RAMET; ROMAN, J., 2002; FAVERGE et al., 2018) and the `qr_mumps` (AGULLO, Emmanuel et al., 2015, 2018) libraries provide sparse linear algebra routines. The Chameleon library (AGULLO, E.; BOSILCA, G., et al., 2012; INRIA; UTK; UCD; KAUST, 2018) implements several dense linear algebra routines such as Cholesky, LU and QR factorizations. ExaGeoStat (ABDULAH et al., 2018) is a machine learning framework for weather prediction applications built on top of StarPU and Chameleon. The FLUSEPA aerodynamic solver (COUTEYEN CARPAYE; ROMAN, Jean; BRENNER, 2017) is an industrial application of numerical simulation, developed by Airbus Defence and Space to simulate wave propagations during take-off of Ariane launchers.

2.3.5 Discussion

The aforementioned task-based runtime systems, in particular, the last three, are conceptually very similar. OmpSs, XKaapi, and StarPU present a set of features that made them very useful tools to exploit the parallelism of current HPC platforms. These features comprise multi-version tasks allowing different implementations for each type of hardware resource, data dependencies enabling finer synchronizations, dynamic scheduling guided by several policies (e.g., work stealing, performance models, etc), load balancing and automatic data transfers which is crucial when using platforms with separated

memory address spaces.

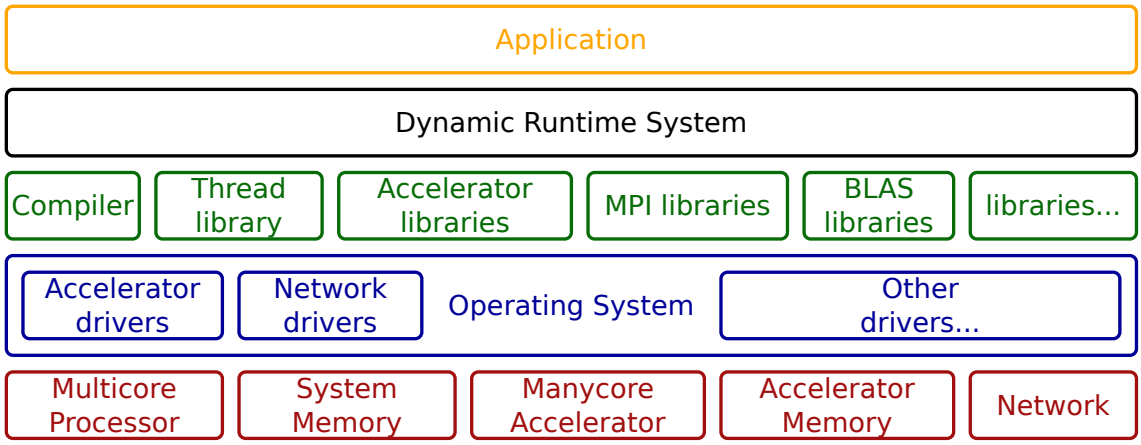
In the remainder of this work, we will consider the StarPU runtime system when designing and building our performance analysis strategies. The choice of this tool instead of others was taken due to technical and practical factors. The StarPU source code is developed and maintained by an active and attentive community which allows us to clarify and discuss the runtime internals in a very productive interaction. Although recent, there is a considerable set of applications implemented on top of StarPU, which has made easier our choice of a representative application (e.g., Cholesky factorization) to demonstrate and evaluate our strategies. Additionally, our previous knowledge about StarPU concepts and the expressive amount of previous work in the literature reinforces its representativeness within current HPC research. Anyway, the similarity of StarPU with other runtime systems suggests our approach should apply to other tools as well.

2.4 Chapter Summary

Parallelism has been a widespread strategy to increase the processing power of computing systems. However, in recent years, the hardware and software stack of HPC systems is experiencing a paradigm shift. The adoption of hybrid architectures discloses several levels of both parallelism and memory spaces exposing the limitation of traditional programming models and motivating the use of alternative ones.

In this chapter, we discuss the hardware and software context involved with this work. Previous Sections provide a basic background about the characteristics of the current hybrid HPC architectures and about the strategies to program them. As illustrated in Figure 2.9, at the hardware layer (in red), multicore processors are associated with manycore accelerators, which have a separated memory address space. At the software level, an intermediary layer (in black) between the application code and the operating system acts as a middleware providing an abstraction to the hardware heterogeneity. This abstraction layer relies on task-based programming with the support of dynamic runtime systems to produce applications that are less tightly coupled to the hardware and that, in consequence, are easier to scale on other/newer platforms.

Figure 2.9: An example of the current hardware and software stack



Source: The Author

3 STATE OF THE ART

The analysis of execution traces is a very common strategy to understand and evaluate the performance of a parallel application. Since these traces contains application-relevant events with its respective timestamps, it is possible to create a visual representation of the real execution. This visual representation can be expressed in different ways such as space-time diagrams, statistical summaries or reproducing the application structure.

Many tools exist to visualize execution traces from HPC applications. Usually, these tools focus on applications implemented with widely-used programming models, such as *pthreads*, OpenMP and MPI. However, the emerging task-based paradigm has some different requirements that are unmet by the already established analysis tools.

In this chapter, we present the state of the art of visualization strategies for HPC execution traces. We do not intend to do a comprehensive review of all performance visualization tools. Instead, we selected those that appear to be emblematic, well-established or very recent. First, in Section 3.1 we detail well-established trace analysis tools for BSP-applications. Afterward, in Section 3.2 we discuss some recent strategies to visualize traces from task-based applications. More comprehensive, historical and systematic reviews about performance visualization tools can be found in (MELLO SCHNORR, 2009) and (ISAACS, Katherine E. et al., 2014).

3.1 Traditional BSP-based Visualization

For a long time, HPC applications have been designed following the traditional Bulk-Synchronous Parallel (BSP) model. In this model, the computation consists of a sequence of *supersteps*: concurrent computations, communications, and global synchronization (VALIANT; G., 1990). This design is suitable for homogeneous platforms where regularity in computing and communication resources is expected.

Since the BSP-model was the dominant design in the HPC landscape, most performance analysis tools were developed to fit its characteristics. These tools expect homogeneity, highlighting irregular behaviors, such as longer computations or delayed communications. They usually support distributed architecture, targeting the widely known MPI interface.

A very common visualization technique is the use of space-time plots, which get

inspiration from the traditional Gantt charts (WILSON, 2003)[1]. In this design, computing resources, even physical (processors, cores) or logical (processes, threads), are arranged vertically, sometimes hierarchically organized (KERGOMMEAUX; OLIVEIRA STEIN, 2000), while the application states (functions, kernels, actions) are laid out horizontally along time. Colors are extensively used to depict different states, e.g., an MPI operation on a given computing resource. Interactions between application components are depicted as arrows whose width may correlate with the amount of transferred data. Such technique has been implemented multiples times with different technologies to improve human perception and scalability.

3.1.1 *ViTE*

ViTE (COULOMB et al., 2009) is an open-source tool to visualize execution traces. It relies on OpenGL and hardware acceleration to enable the visualization of large inputs. Since *ViTE* relies on the semantic-free Paje language (SCHNORR, Lucas Mello et al., 2016) as trace input, it can depict virtually any kind of traces.

ViTE provides traditional space-time views as shown in Figure 3.1. This Figure shows an execution of the Cholesky factorization implemented with StarPU for a matrix of 49×49 tiles of size 960×960. It uses the DMDAS scheduler in a platform with 2 nodes, each one with 4 cores and 4 GPUs. As illustrated in this example, *ViTE* depicts the computing resources in a hierarchical structure, grouping CPU cores and GPU cards in their respective memory spaces. When visualizing distributed application, it is also possible to include arrows to represent data transfers. Colors encode states, which in this example, are StarPU tasks.

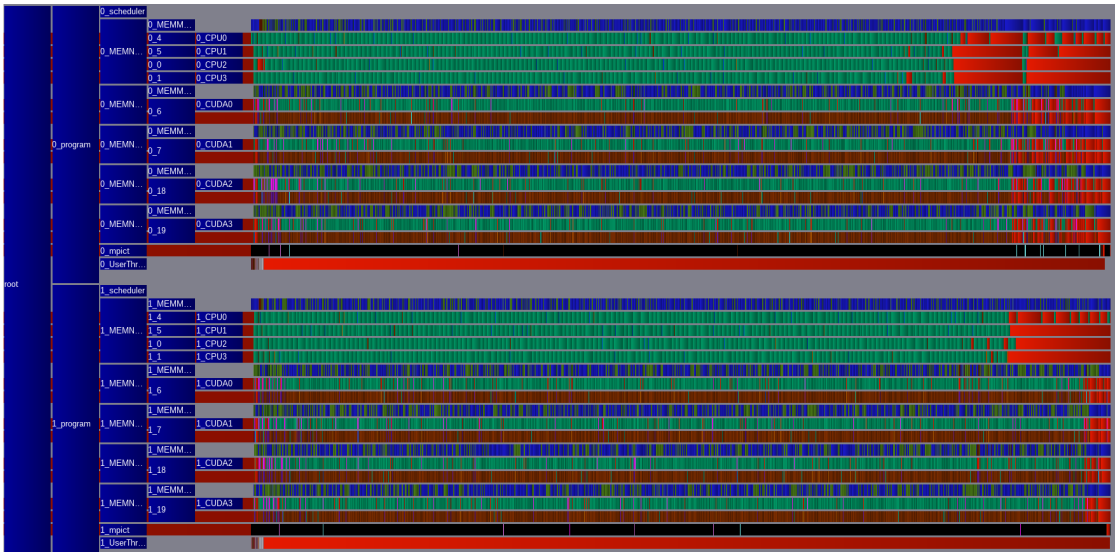
3.1.2 Paraver

Paraver (PILLET et al., 1995) is another open-source tool to visualize and analyze execution traces. Its trace file format supports many HPC programming models and can be generated by the Extrae tracing tool[2]. Scalability is tackled by implementing trace aggregation before the visualization. This way, instead of sending all trace data to the

[1] Gantt chart is a really popular kind of bar chart used in Operational Research to illustrate the schedule of the steps of a project.

[2] `https://tools.bsc.es/extrae`

Figure 3.1: A StarPU-MPI trace visualized with *ViTE*.



Source: The Author

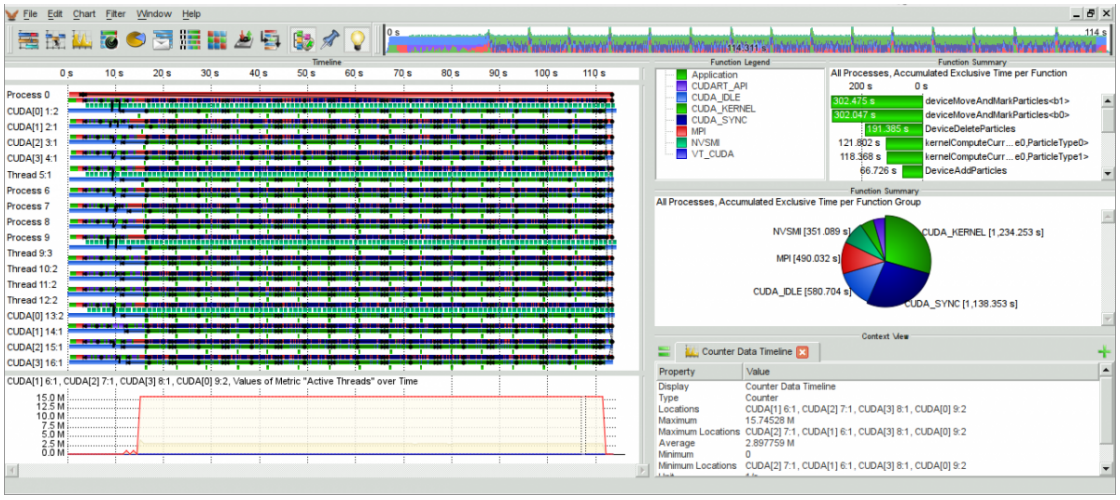
rendering driver, it decides (based on user configuration) which element to draw in a specific location on the screen.

3.1.3 Vampir

Vampir (KNÜPFER et al., 2008) is a closed-source tracing and visualization toolset. Vampir is implemented with a distributed client-server approach. The server can be executed in the experimentation hardware, e.g., several nodes of the cluster, while the client can run on the analyst's remote desktop. This strategy keeps raw traces on the platform where they were created, minimizing data transfers. On the client side, it limits the amount of data to be processed by the client to a value that is independent of the amount of data in the raw trace. For these reasons, Vampir is considered to be more scalable than similar tools, allowing larger trace files to be browsed and visualized interactively.

Vampir provides multiple views for OTF2 (Open Trace Format, version 2) trace files (ESCHWEILER et al., 2011). The main ones are based on space-time charts and statistics summaries. Figure 3.2 shows an example of an OTF2 trace visualization. The left side of the image shows the space-time plot. On the right side, statistics plots depict the time spent in each function (green horizontal bars) and per function group (pie chart).

Figure 3.2: Visualization of an OTF2 trace in Vampir.



Source: NVIDIA (2018)

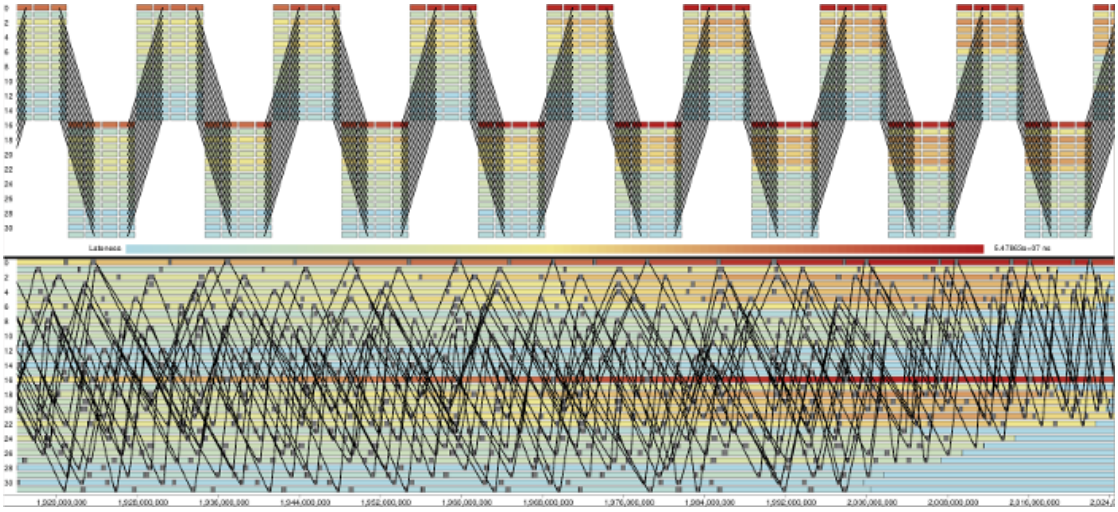
3.1.4 Ravel

Ravel (ISAACS, K. E. et al., 2014) is a visualization tool for MPI execution traces. This tool fully replaces the notion of physical time, widely used in traditional space-time view, by the order and simplicity of logical clocks. This way it is possible to emphasize the code structure and the communication patterns.

Figure 3.3 shows a comparison between logical (top) and physical (bottom) timelines in Ravel. MPI messages are represented by lines connecting events. The logical view reveals the structured communication pattern of the application which is invisible in the classical physical-time view. Besides, even in the logical timeline, the notion of physical time is kept by coloring the states. In this example, the *lateness* metric is used to depict the difference in event end-time comparing to the earliest event in the same timestep. Other metrics like start-time or event duration can also be used.

Ravel logical-time views avoid the frequent problem where lines representing communications overrides the events. Additionally, since phases are clearly identified, the tool is capable of clustering thread behavior by similarity. The approach enables one to focus on the causal relationship between processes and, at the same time, have a perception of bad performance with colors.

Figure 3.3: A comparison between logical and physical views in Ravel. Colors encode physical time. Black lines represent MPI message exchange between processes.



Source: Kate Isaacs (2018)

3.1.5 Edge Bundling extension for Vampir

Brendel et al. (2016) present some views based on edge bundling to handle numerous communication arrows. Their techniques focus both time-based and summary visualizations, combining individual messages into dominant communication flows.

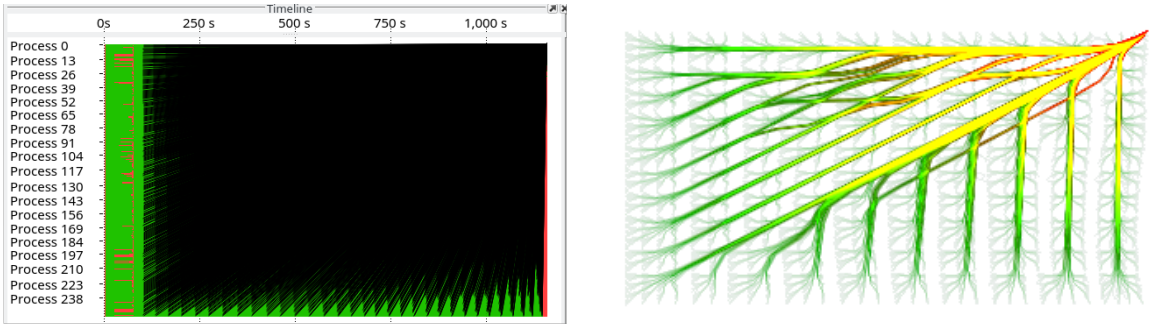
The time-based strategy reorganizes the communication arrows preserving the time information in the x-axis. However, this strategy suffers from applicability issues since it requires manual parameters tuning, which is highly individual for each trace. Figure 3.4 shows a comparison between Vampir and the proposed timed-based edge bundling views. Both visualizations use the same trace from a parallel application where all processes repeatedly communicate its intermediate results to process 0. The Vampir view (Fig. 3.4(a)) suffers from visual cluttering, i.e., communication arrows are overlapping almost all application events, which is minimized in the proposed approach (Fig. 3.4(b)).

The summary approach aims to offer an alternative to the communication matrix view from Vampir. These summary views are useful for traces from larger application runs since they reduce the number of message arrows to be drawn. This is done by grouping messages in sender/receiver pairs. Figure 3.5 shows summary visualizations for an application where each process only communicates with its neighbors. Figure 3.5(b) presents the proposed sender/receiver diagram. Although it highlights the communication pattern, this strategy also relies on manual parameter tuning.

Figure 3.4: Visualization of communication arrows in Vampir and in Brendel's Edge Bundling Approach.

(a) Vampir visualization. Communication arrows (black) are overwriting events (red and green).

(b) Proposed timed-based edge bundling visualization. Yellower lines indicate hot communication paths.

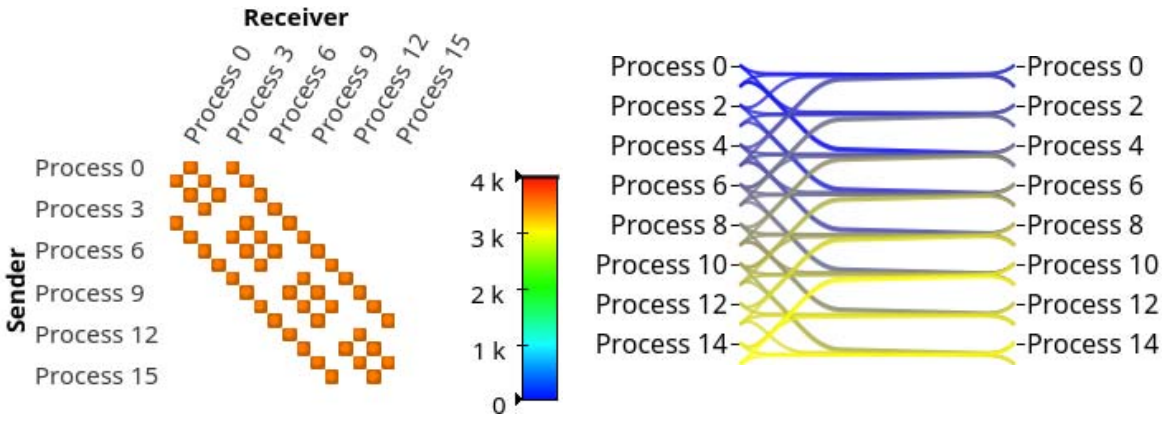


Source: Brendel et al. (2016)

Figure 3.5: Summarized views of communications in Vampir and Brendel's proposed strategy.

(a) Vampir Communication Matrix view. Colors encode the number of sent messages.

(b) Proposed summary edge bundling visualization. Colors encode the sending process.



Source: Brendel et al. (2016)

3.1.6 FrameSoc/Ocelotl

FrameSoc (PAGANO et al., 2013) is a performance analysis tool with features to handle large MPI traces. This goal is achieved with two complementary approaches; the first one handles the trace storage while the second one provides scalable aggregated visualizations.

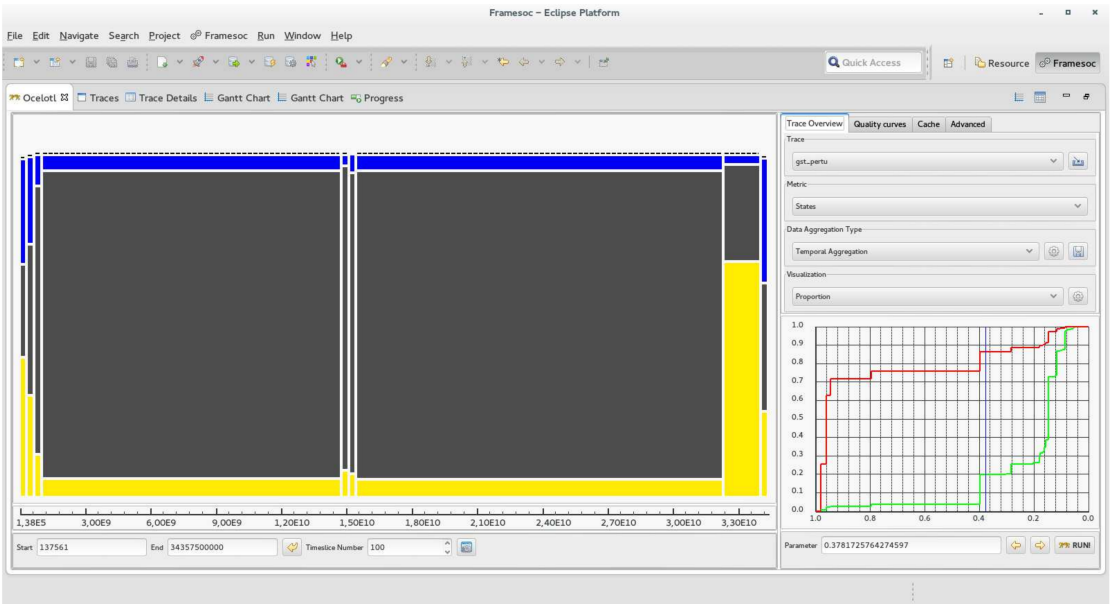
FrameSoc is able to import and store traces in several formats such as Pajé, CTF, Paraver, and OTF2. Raw traces are imported and converted to a generic data-model, which gets inspiration from Paje language (SCHNORR, Lucas Mello et al., 2016). In addition to the raw trace, this model can also store complementary information like traces metadata, annotations, and trace analysis results. The compiled data structure is stored in a relational database. This approach enables faster accessing, filtering and searching operations which is especially useful for continuous and incremental analysis since it avoids to reload and parse the same raw trace several times.

Visualization of large traces frequently suffers from graphical problems such as cluttered views. FrameSoc relies on the Ocelotl (DOSIMONT; LAMARCHE-PERRIN, et al., 2014) additional module to provide meaningful visualizations even when dealing with large-scale traces with millions of events. Ocelotl provides a user-configurable aggregation technique. Its aggregation algorithm looks for homogeneous sections in the trace to be aggregated respecting a threshold between the information-loss and the complexity of the visualization.

Figure 3.6 shows an example of the aggregated visualization proposed by FrameSoc. The *space-time panel* depicts a spatiotemporal aggregation, which means it reduces trace complexity in both temporal (states duration in time) and spatial (resource in which the state was executed) dimensions. In this view, three different states, represented by colors blue, gray and yellow, were aggregated. There are two big homogeneous areas that are separated by a period of instability, which means that something has disturbed the execution at this time. Other instabilities periods appear at the beginning and end of the execution, which is usually expected due to initialization and finishing procedures where the workload is not sufficient to fill the platform. The *red-green quality curves* (bottom right) allows the user to control the threshold ($p-$value) between information-loss and visualization complexity. In this example, p is ≈ 0.37. A $p-$value equals to 0 minimizes the information-loss, which results in the original non-aggregated view. On the other hand, a $p-$value equals to 1 minimizes the visualization complexity which results in a

full-aggregated view.

Figure 3.6: An Ocelotl aggregated view in FrameSoc. The aggregated view (big panel on the left) is dynamically generated to respect the user-defined threshold of the quality curves (red and green lines on the right).



Source: Dosimont, Corre, et al. (2015)

3.2 Task-oriented Visualization

Parallel applications following the task-based programming model (Section 2.2.1) have different performance analysis requirements from the ones implemented in the BSP design. Since the execution of these applications is supported by dynamic runtime systems, several classical steps like task scheduling and dependencies management are performed automatically with no control on the part of the application programmer.

Runtime systems rely on sophisticated scheduling heuristics to minimize the idleness of the computing resources. These heuristics enable fine-grained synchronizations executing each task as soon as possible without waiting in global barriers. This means that a task-based execution does not present well-structured phases like in the BSP model.

On the other hand, even if the task scheduling is naturally stochastic, the task dependencies impose a certain order on task execution. This order is relevant especially in the beginning and end of the execution when the number of tasks is not sufficient to occupy all computing resources. Such kind of perception is nonexistent in BSP-based performance analysis tools. As a consequence, there are very few tools that are truly oriented

towards task-based applications and runtime performance analysis. Very frequently, these tools inspire from the already established field of BSP-based trace visualization, using the intuitive view of space-time charts supplemented with interactions, i.e., mouse pointer.

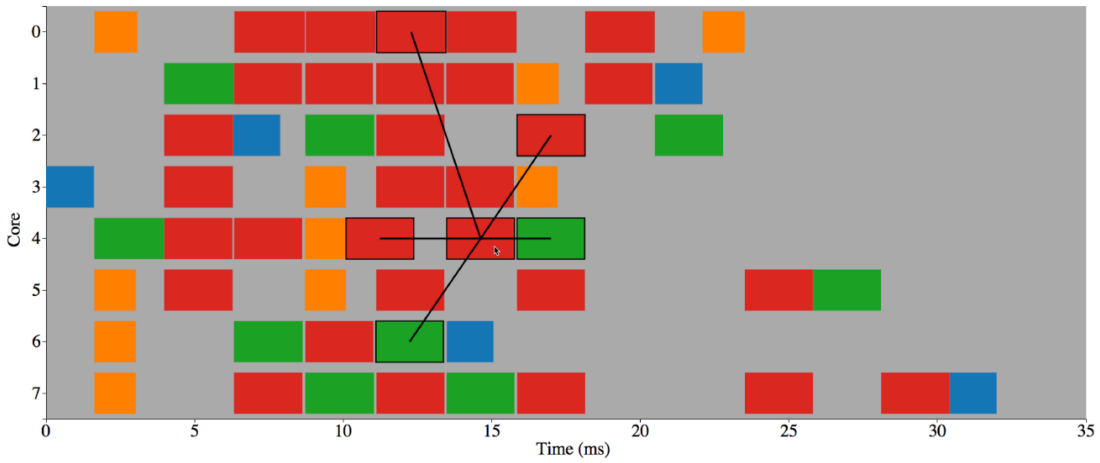
In the remainder of this section, we present some visualization tools that include at least one feature specially designed for task-based applications. This includes application structure (e.g., task dependencies, DAG), runtime-related information (e.g., delayed tasks, lack of parallelism, hardware counters) and task debugging (e.g., check/fix dependencies or priorities).

3.2.1 Execution Traces with Dependencies

Haugen et al. (2015) present an interactive tool to visualize execution traces with task dependencies. This tool relies on a combination of two separated data sets: the execution trace and the DAG, to build a single visual representation.

Task dependencies are represented by lines connecting the boxes representing tasks. Since drawing all the dependencies of all tasks quickly overwhelms the visualization, the tool transfer to the user the decision of which task select to check the dependencies. This way, the user should use mouseclicks to interactively select a task and then the tool draws dependency lines connecting tasks for which it is waiting and those that are waiting for it to finish. Figure 3.7 shows an example of a space-time view with the dependencies of a task. When the mouse is hovering on the selected red task, three lines are drawn to indicate its antecedents and the other two ones to indicate its descendants.

Figure 3.7: An example of the trace visualization with dependencies proposed by Haugen et al. (2015)



Source: Haugen et al. (2015)

This tool targets the PaRSEC runtime system (discussed in Section 2.3.1) and is implemented with a client-server design in Python and Javascript. However, the lack of a reference to the source code prevents us to check some of the claimed features. We believe this approach suffers from three issues. First, in terms of scalability, since (e.g., in tiled cholesky) tasks typically have many dependencies (up to `N` outgoing dependencies for DTRSM and DPOTRF tasks, i.e., a total of $\Theta(N^3)$), drawing everything and finding *interesting* tasks and dependencies only through mouse interaction can be very tedious. In practice, only tasks close to the critical path are important. Second, only one-level dependencies are depicted, while several levels are required to understand the history leading to the scheduling problem. Third, this tool does not really account for the heterogeneity of resources.

3.2.2 DAGViz

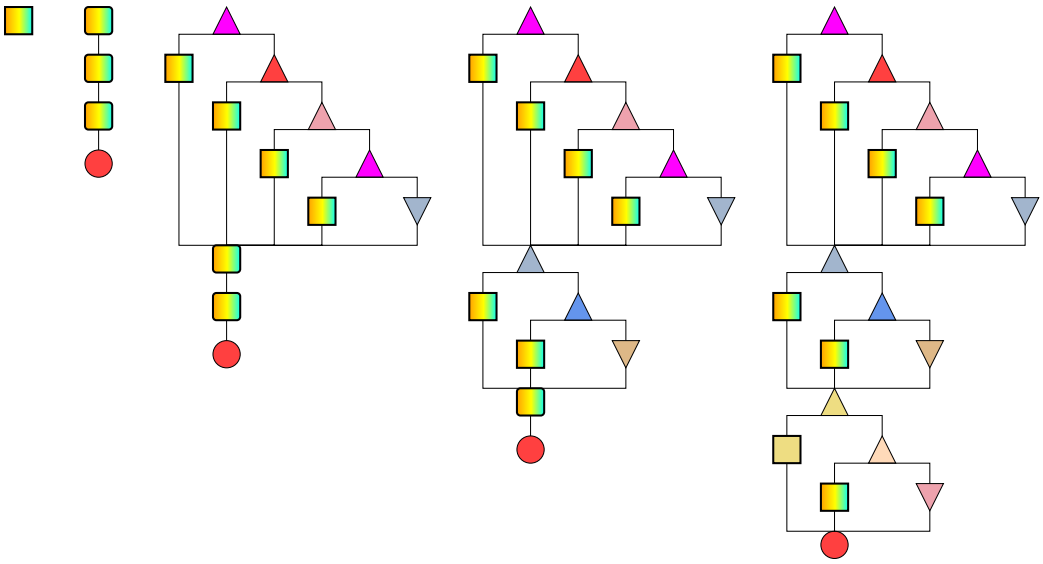
DAGViz (HUYNH, An et al., 2015) offers an alternative visualization that does not rely on space-time charts but in the DAG structure of the application. It is composed of two tools, one to extract the DAG from a task parallel execution and another to visualize it in a hierarchical way.

DAGViz uses a simple task model to extract the DAG from a parallel code. This model consists of tree primitives to create a new child task (CreateTask), to wait for all tasks in the current section (WaitTasks) and to create a new section inside a task or another section (MakeSection). These primitives are translated to real parallel code in other tools like OpenMP, Cilk or Intel TBB and into an instrumented code to track the begin and the end of each primitive.

The extracted DAG is drawn in a hierarchical visual representation that can be interactively folded/unfolded on-demand to show more or fewer details. Figure 3.8 illustrates an example of such visualization where the same DAG is (partially) unfolded in five different levels. In the initial step, the DAG is totally folded (the first graph with one node), in the next one, the DAG is unfolded into three sections. These sections are unfolded in the next steps, i.e., the third graph with only the first section unfolded until the last one with three sections unfolded. This last graph still presents folded sections. For this particular example, a totally unfolded visualization reaches thousands of nodes.

Despite the innovative approach, there is no way to retrieve the time dimension and task duration, which can make performance analysis difficult. The clear problem

Figure 3.8: An example of the DAG-based visualization proposed by DAGViz. A square represents a section (folded part of the graph), a triangle represents a task creation and an upside-down triangle represents a wait point. The fill color of each shape indicates the worker that has executed the task, the mixed colors indicate that the subgraph was executed by several multiple workers.



Source: An Huynh et al. (2015)

with such approach is the scalability: very often DAG with large inputs may be composed of millions of tasks. Even with an artificial hierarchical organization, exploring the application structure (blocks, tiles), the representation will be very hard to understand.

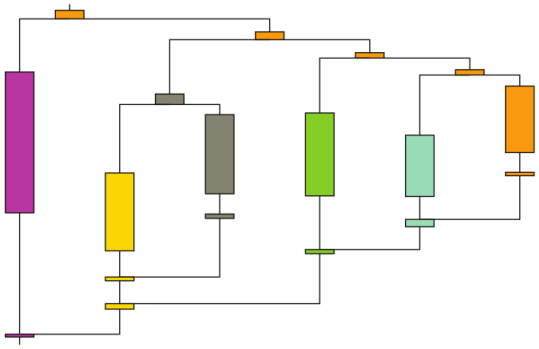
3.2.3 Delay Spotter

Delay Spotter (HUYNH, A.; TAURA, 2017) is a tool to detect delays in the runtime system decisions and represent their occurrences in terms of logical and time-based views. This approach divides the worker state into three possibilities: `work` when the worker is executing the application code, `delay` when the worker is not working on the application code and there are ready tasks available for execution, and `no-work` when there are no ready tasks available for execution. Such `no-work` periods can be due to either application (`no-work-app`) or runtime scheduler (`no-work-sched`) issues.

This tool is built on top of DAGViz and is able to analyze applications using the **spawn-sync** model in Cilk, TBB, and OpenMP. Figure 3.9 illustrates the visualization of delays in the DAG. For hybrid scenarios, this approach is ineffective since variation on task duration is expected due to resources heterogeneity. In contrast, the concepts of

`delay`, `no-work-sched` and `no-work-app` can be useful to quantify the origin of resource idleness and are pertinent to hybrid scenarios.

Figure 3.9: The Delay Spotter representation of delays in the DAG.



Source: A. Huynh and Taura (2017)

3.2.4 Temanejo

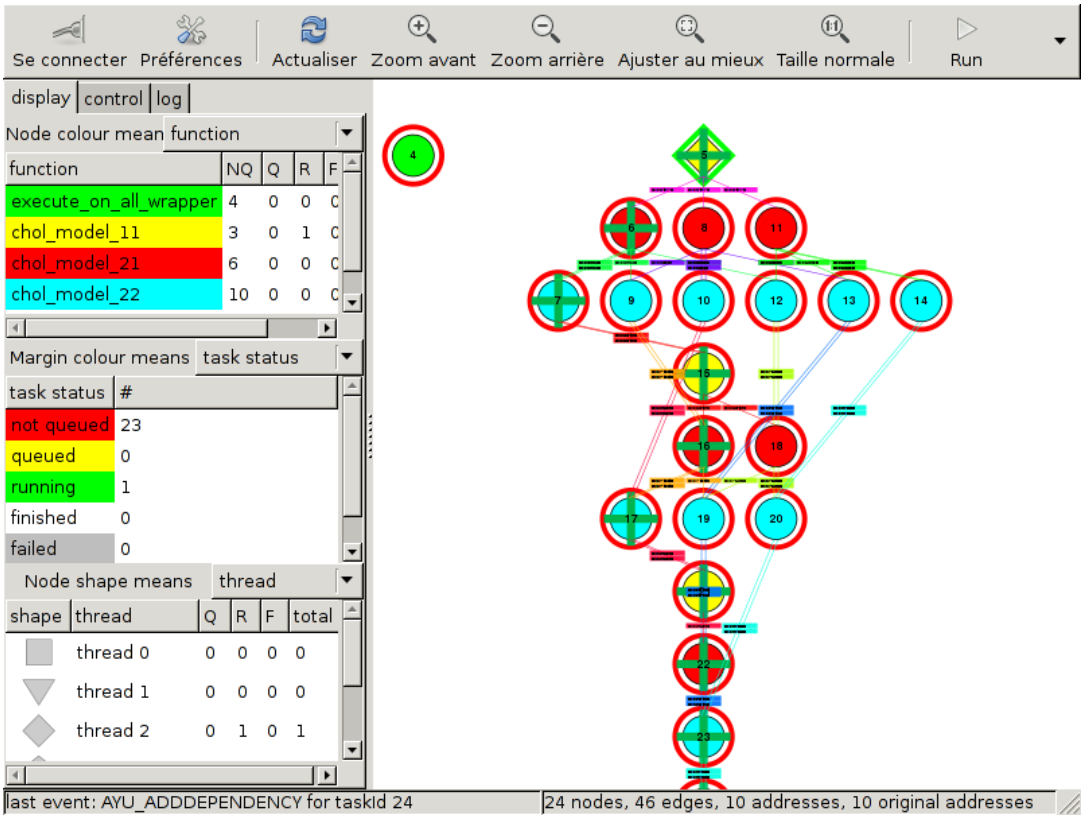
Temanejo (KELLER et al., 2012; BRINKMANN, Steffen; GRACIA; NIETHAMMER, 2013) is a visual debugger for task-based programming. It was initially designed to debug applications running in the SMPSs runtime system, presented in Section 2.3.2. Currently, it supports more tools including OmpSs, StarPU, PaRSEC, and OpenMP (BRINKMANN, Steen et al., 2017).

Temanejo relies on the Ayudame auxiliary library to interact with the supported runtime systems. This library is used to retrieve events information and to send control commands to the runtime system.

Figure 3.10 shows an example of the timeless DAG interactive view proposed by Temanejo. Although visually similar to the interactive DAG view offered by DAGViz, the Temanejo one does not use the interactive capabilities only for scaling purposes. Temanejo views can be enriched with information about the resource that executes the task, its scheduling status or its duration. Thanks to the support of Ayudame library, the user can also modify the execution on-the-fly by blocking single tasks, changing task dependencies, stopping execution when reaching a task, changing tasks priority or launching a gdb section.

The Temanejo capabilities are very useful during algorithm design on small-scale enabling the user to check and fix several task parameters such as dependencies or priorities. However, its visualization features are focused on debugging steps, being unsuited

Figure 3.10: A Temanejo view of a task-based program running with the StarPU runtime system. Node color is used to indicate the task type while node margin indicates the task scheduling status. These settings are customizable.



Source: INRIA, CNRS, and Université de Bordeaux (2017)

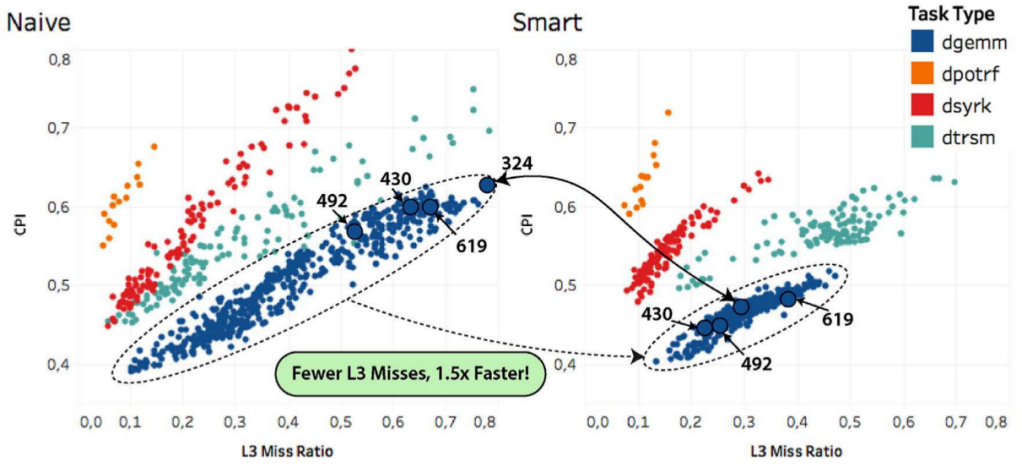
for performance analysis.

3.2.5 TaskInsight

TaskInsight (CEBALLOS et al., 2018) provides a low-level analysis focusing on the impact of memory behavior on tasks performance when using different schedulers. This tool extends the OmpSs runtime system to intercept task scheduling calls and collect hardware performance counters at the start and end of each task. Their approach is based on per-task data access patterns, which is a static information, and on hardware performance metrics, such as CPI (cycles-per-instruction), L2/L3 cache misses, which are execution-dependent.

The view of Figure 3.11 shows the correlation between task performance (CPI) and L3 cache misses grouped by task type of a Cholesky factorization (execution with 15×15 tiles of size 256×256 in an i5-3550 quad-core processor). Tasks present better performance when using the `smart` scheduler, which attempts to explore data-locality, as can be checked by the lower ratio of L3 misses. It is not clear how this fine-grain strategy will scale when dealing with the complex memory hierarchy of hybrid nodes. In small cases, as the one with 4 cores presented by the authors, the analysis is already complex since it should consider the effects of shared caches.

Figure 3.11: Per-task Performance vs L3 Miss Ratio correlation computed by TaskInsight when running a Cholesky factorization with two OmpSs schedulers (naive and smart).



Source: Ceballos et al. (2018)

3.3 Discussion

The following topics summarize a set of performance analysis features provided by the previously presented tools. Table 3.1 correlates these topics with each analysis tool.

Space-time plot (`ST`) – A common technique used to represent the application behavior along the execution duration. Each application state is represented by a different color. The vertical axis depicts the computing resource where the state happens (e.g., cores, processes, etc) while the horizontal one represents the time when it starts and finishes.

Application Structure (`AS`) – The structure of a parallel application is intrinsically related to its performance. In general, the structure can be inherited from the code structure (e.g., loop-based or recursive calls) or, in case of parallel applications, from the communication and synchronization patterns.

Information Reduction (`IR`) – Several times, the amount of raw data provided to the analysis tool is too large to be presented in its original format. The most common strategies to handle this issue are aggregation and filtering. The first one looks for similarity and uniformity, grouping data that represent a similar state or behavior while the second one looks for significance and singularity, keeping only data that is more important according to given criteria.

Execution Metrics and Statistics (`EMS`) – Data collected from the platform during the application execution can be very useful to understand and to explain the application performance. This kind of data can be collected from several sources, e.g., operating system, hardware counters, programming library or network interface. The representation can be either time-based, representing the evolution of a value along the time or statistically summarized depicting the overall behavior.

Anomalies Detection (`AD`) – Unexpected behavior can also be one of the main sources of performance disturbances. A delayed communication or computation on a given resource can impact all the following steps of the application. In addition, the occurrence of synchronized anomalies on several workers can be an evidence of a disturbance in the platform.

Debug Capabilities (`DC`) – In general, the common analysis workflow consists of executing the application, collect data about its execution, analyze this gathered information and, if necessary, re-run the application with some changes and restart the workflow.

However, sometimes these steps are very long or the data collected is not enough to understand the behavior. In this case, debug capabilities can help since the analyst can verify and fix application or runtime parameters during the execution.

Table 3.1: Visualization tools and common features

	ST	AS	IR	EMS	AD	DC
ViTE	✓	commun. arrows	filtering	✓		
Paraver	✓	commun. arrows	aggregation & filtering	✓		
Vampir	✓	commun. arrows	aggreg. (**Edge Bundling** ext.)	✓		
Ravel	✓	commun. arrows	aggregation & filtering		✓	
FrameSoc	✓	commun. arrows	aggregation & filtering	stat. only		
Traces w/ Depend.	✓	task depend.		stat. only	✓	
DAGViz		graph of tasks	folding & unfolding	stat. only	✓ (**Delay Spotter** ext.)	
Temanejo		graph of tasks				✓
TaskInsight				✓	✓	

Source: The Author

Table 3.1 shows a summary of the features present in the tools discussed in this Chapter. Many tools provide Space-Time plots (**ST**), Application Structure (**AS**) representations and some Information Reduction (**IR**) technique. The **ST** feature is generic enough to be used in a similar way in both BSP and task-based scenarios. However, this is not valid for **AS** and **IR** techniques which are more complex. In BSP-oriented tools, **AS** is represented by drawing communication arrows. Sometimes the huge amount of these arrows can overlap the **ST** view, but thanks the regular communication pattern this issue can be solved using **IR** strategies such as aggregation and logical-time reordering. These **IR** techniques enable highlighting the application structure without degrading the **ST** plot. On the other hand, in task-oriented scenarios, the finer-synchronizations does not favor the use of aggregation techniques for this purpose. Considering the DAG structure of task-based applications, effective **IR** strategies can rely either on folding/unfolding to take advantage of the hierarchical structure or filtering to select only relevant edges (e.g., considering the critical path).

Another common feature is the visualization of Execution Metrics and Statistics

(**EMS**). In BSP-oriented tools, this feature is used mainly to represent the bandwidth usage along the time or the percentage of time spent in each state. Since task-based applications execute over dynamic runtime systems, their traces can be much richer including also metrics such as the number of ready tasks or statistic summaries associated with the scheduler or the task type.

The detection of anomalies (**AD**) is a less common feature. In BSP tools, it can highlight delayed states or communications when compared to others in their groups/phases. However, most part of tools does not provide this as an explicit feature, letting the user empirically check delayed phases by visually comparing them in the **ST** plot. When considering task-oriented tools, anomalies are identified by comparing each task with others of the same type. Debug Capabilities (**DC**) are provided only by one task-oriented tool (Temanejo). This kind of feature is very useful in this scenario since the runtime libraries and schedulers are much more complex than traditional MPI or *pthreads* ones, with additional control parameters (e.g., priorities, affinity) and scheduling policies.

3.4 Chapter Summary

In this Chapter, we present some performance analysis tools that can be classified into two main groups: tools for analyzing the performance of applications designed with the BSP-model and analysis tools providing task-oriented features.

The first group of tools is designed to analyze applications implemented with classical HPC programming models, e.g., *pthreads*, OpenMP and MPI. Such applications are executed on platforms with homogeneous processing and communication capabilities and, in general, present well-structured phases, grouping computations, communications, and synchronization operations. The second group encompasses recent tools designed to analyze applications which are implemented with task-based programming. We include in this group tools that offer at least one task-oriented analysis feature.

To conclude, we summarize the common analysis features provided by the discussed tools and how these differ in the BSP and task-based domains. We take as inspiration the strengths and limitations of these features to build our performance analysis strategies that are further presented in Chapters 5 and 7.

4 METHODOLOGY

Two of the key concepts of science are demonstrability and reproducibility. Scientists usually rely on previously accumulated knowledge to build, prove or refute their scientific hypothesis. However, the success in this process of building on previous discoveries depends on how transparent and accessible is the research workflow[1] of other scientists (MUNAFÒ et al., 2017).

In the computer science context, the availability of the source code and of the input and output data are major procedures to provide reproducibility (STODDEN; LEISCH; PENG, 2014). The use of open source tools, appropriate file formats, and accessible storage platforms are technical aspects that could help to put in practice these guidelines.

Since the reproducibility traditionally seems to be lower than is desirable, in this chapter we present our steps to make this work more transparent and reproducible. The remainder of this chapter is organized in two main sections; the first one discusses the use of a literate programming approach to writing this thesis while the second one shows our strategy to generate input data to perform our analysis.

4.1 A Reproducible Report

This document is written using the Org-mode (DOMINIK, 2010) extension of the GNU Emacs editor (STALLMAN et al., 2017). Org-mode was initially designed as a tool to organize and manage tasks in plain text format, but, currently, it provides very useful authoring features. Org-mode includes a lightweight markup language (LML) to structure a document in a hierarchical way with support for labels and references. Documents written in an LML have a simple syntax and therefore are both easy to read by humans, even in raw format, and also easily editable with a basic plain text editor. This means that even without an Emacs editor, it is possible to read and modify an org document.

There are another markup languages that are simpler or more widespread, as *reStructuredText* (rST) and markdown or LaTeX, respectively. The main reason to use Org-mode as an authoring tool is its literate programming and reproducible research capabilities.

The *Babel*[2] feature of Org-mode allows defining active source code and data blocks

[1] methodology, input and output data, data analysis and interpretation

[2] `https://orgmode.org/worg/org-contrib/babel/`

inside org documents. Active code blocks can be evaluated within the org document and its output is captured and included in the document content. Source code blocks can be written and evaluated in several languages. The output of a code block (a data block) can be used as the input of another one, even if they are not coded in the same language. In this thesis, we embed code written in *R*, *shell*, *emacs-lisp*, *LaTeX*, *BibTex,* and *C*. Most part of the figures of this report are generated inside the org document and can be modified by anyone with access to its source code. The data used to generate these graphics are also available, embedded as data blocks within this document (e.g., Figure 2.2) or within the repository of this report (e.g., all the space-time views of Chapters 5, 6). This companion material is **available** at the **thesis proposal** git **repository**[3] which contains a `thesis.org` file with all this text as well as all the instructions to download our experiments data. Generated figures are also committed, but the availability of the analysis code and of the experiment's data makes possible to regenerate all them from the org document.

Org-mode implements the *weave* and *tangle* literate programming operations defined by Knuth (1984) to convert the source document into different representations: a human-readable format and to computer-executable format, respectively. Since the org document is written in a natural language mixed with several code or data blocks, the *tangle* operation allows us to easily extract the code snippets to a single source code file that can be compiled or interpreted in according with the used programming language. On the other hand, the *export* feature implements the org *weave* equivalent. This *export* operation provides a way to obtain a file in a final format such as HTML, ODT, LaTeXor PDF. During the exporting, org is able to evaluate the embedded code blocks and include its results inside the final document (SCHULTE et al., 2012).

In this thesis, whenever possible we made available data treatment, analysis, and plotting code embed into the org document. This makes possible to reproduce the generation of all graphics and helps to better understand our steps to go from a raw input data to a rich graphical representation. Moreover, in a constructive approach, one can rely on this companion material to extend our analysis even applying it to another dataset or exploiting the same dataset in a different way.

[3]git clone `https://gitlab+deploy-token-3:nKd4QDy6XNAs7boC78sh@gitlab.inria.fr/vgarciap/thesisRepository.git`

4.2 Generating Input Data

As discussed in Chapter 2, the hardware and software stack of current HPC platforms is complex. At the same time as the hardware includes new processing technologies and sophisticated memory hierarchy, the software is redesigned to handle this heterogeneity. Figure 4.2 presents the full software stack of the Chameleon solver used in this work. It illustrates how complex is the software organization of current HPC applications. In this scenario, even a hypothetical small change in a package of the dependency chain may impact the results. For that reason, it is essential to track as much information as possible about the experimentation environment. This comprises not only the source code used but also the platform status, which could help to identify if a hypothetical performance issue is related or not to some change in the experimentation environment.

There is a growing number of initiatives towards reproducible research in computer science, e.g., Vistrails (CALLAHAN et al., 2006), VCR (GAVISH; DONOHO, 2011), Sumatra (A.P. et al., 2014), Reprozip (CHIRIGATI et al., 2016). In this work, we rely on some Git and Org-mode strategies proposed by Stanisic, Legrand, and Danjean (2015) with some additional extensions to improving the management of complex software stacks and to provide an appropriate workflow to the storage of large results files.

In this work, we use execution traces as the input data to our analysis. These traces are obtained from previously traced applications as the ones provided by the Chameleon solver when running over the StarPU runtime system. This way, it is important to keep track of not only the traces but also the software stack used to generate them as well as the state of the machine where the application was executed.

4.2.1 Git and Org-mode Strategy

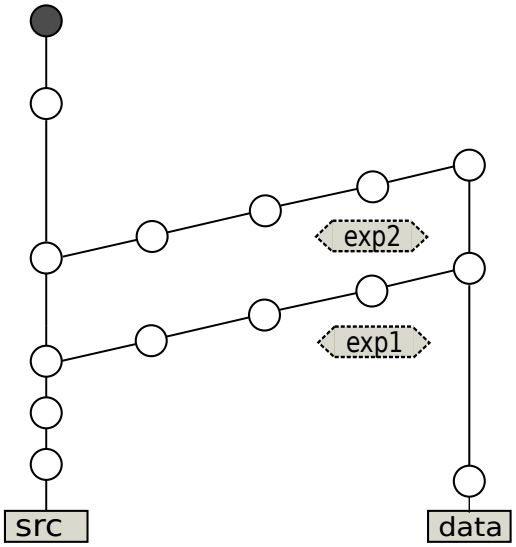
This Git and Org-mode strategy (STANISIC; LEGRAND; DANJEAN, 2015) relies on a combination of Git and Org-mode to execute and track experiments. Git[4] is a distributed version control system that supports non-linear workflows throughout flexible mechanisms of branching and merging.

The Git branching and merging scheme is employed in this approach to (a) ensure a clean experimental environment and (b) define a link between the source code and its produced data. To achieve this, a multi-branch design is used. The source code and the

[4] `https://git-scm.com/`

scripts used to execute the experiments are stored in the *src* branch. The *exp*eriments branches are created to perform experiments, a branch of this type is always created as a fork of the *src* branch. This rule provides a clean environment to perform a new experiment, without results or temporary files from previous ones. This design also reduces the time and the storage space required to start a new experiment since it is not necessary to checkout the results of previous executions. All generated results data are committed into the *exp* branch. Once the experiment is concluded, its respective *exp* branch is merged with the *data* one. The *data* branch contains all the source code, all the results data generated by the experiments and, possibly, some analysis about the obtained results. In summary, as shown in Figure 4.1, the Git repository has two main parallel branches (*src* and *data*), interconnected by several transversal *exp* branches going from *src* to *data*.

Figure 4.1: Git branching scheme proposed by Stanisic, Legrand, and Danjean (2015)



Source: Amended from (STANISIC; LEGRAND; DANJEAN, 2015)

The previously discussed Git branching approach ensures the provenance tracking of the source code used in a given experiment. However, collecting platform-related information, such as configuration and status, is also relevant and must be included in the provenance tracking procedures. Useful platform details are gathered by *shell* scripts and include: *machine state* (logged users during the experiment, environment variables, linux kernel, gcc version and GPU driver), *hardware configuration* (memory hierarchy, processor model, frequency governor, GPU model), *code* (git or svn revision), and *compilation* (build output and libraries linked to the binary file). The gathered content is recorded in a log text file in org format. This file is committed together with the results data, keeping the link among the source code, the platform status, and the produced results. In cases

where the experiment results are also in text format, they can be recorded into the log file as a specific *results* section. This is the case of Chameleon executions, where we record in the log file all the program output, including standard and error outputs, the execution trace, and the used calibration values.

Our research has a different focus from the one for which the Git and Org-mode approach was proposed. In the original use case, the authors are the main developers of the experimental software used to produce the results data. In our case, the focus is on the analysis procedures. Then, the results data generated by the experimental software are used as the input for our analysis techniques.

Despite that, we decide to keep their original strategy, tracking the code of the experimental software on the *master* branch. Our analysis code was entirely developed and tracked on the *data* branch since it intrinsically depends on the results data. In addition, we **implement extensions** to target some specific requirements of our use case.

Our extensions have two main goals: (a) improving the management of complex software stacks and (b) providing an appropriate workflow to the storage of large results files. To meet the first goal, we rely on the Spack package manager (GAMBLIN et al., 2015). The second one is addressed by a combination of **git-annex** and Zenodo.

4.2.2 Handling Complex Software Stacks with Spack

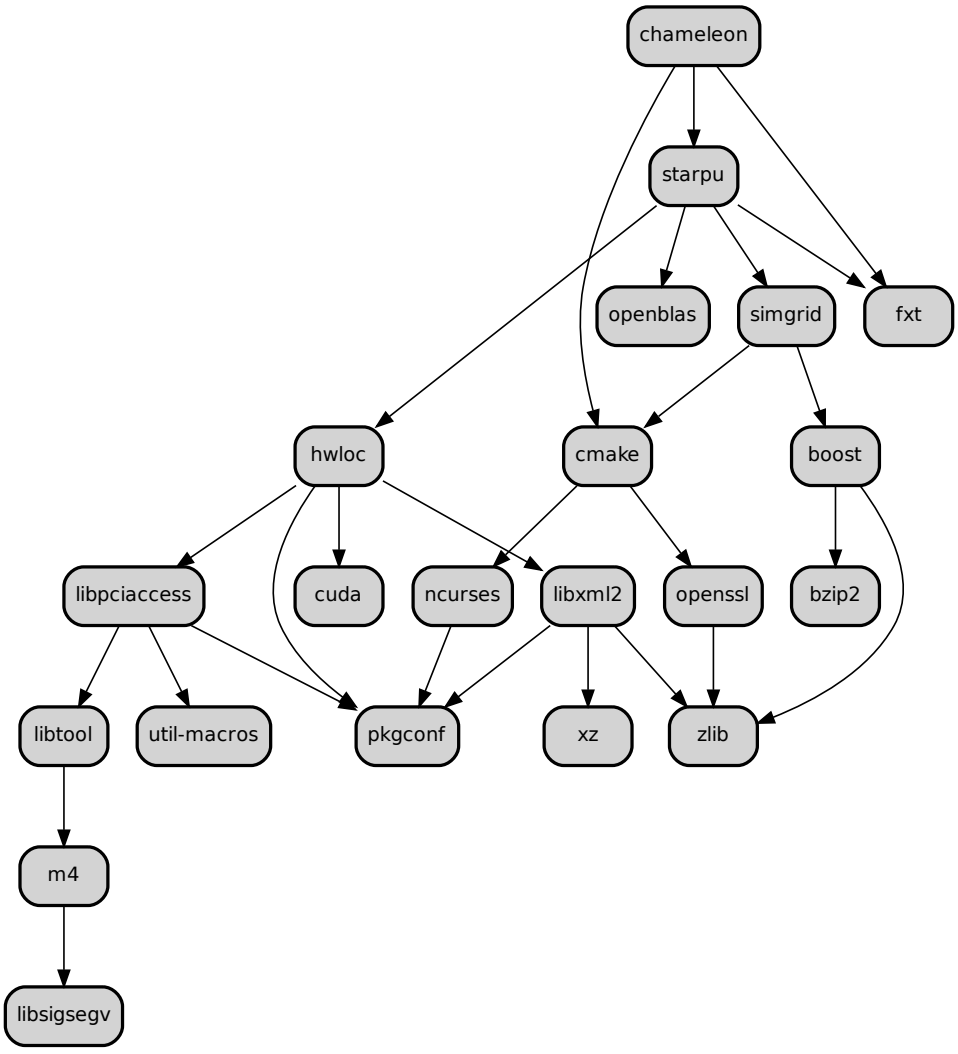
Current HPC applications usually present a sophisticated software stack comprising several levels of dependencies and optional parameters as shown in Figure 4.2. Handling this stack by hand is unfeasible since it involves download, compile and link dozens of libraries before build the main experimental software.

For this reason, we include the Spack package manager inside the Git and Org-mode approach. This tool enables us to track not only the target experimental software but also the major part of its dependencies.

Spack (GAMBLIN et al., 2015) is a Supercomputing PACKage management tool. Spack, as another widely spread package managers like homebrew (HOWELL, 2018), allows users to fetch, compile and install software in their own user directory without relying on administrator privileges and/or OS-specific commands.

In order to target HPC systems, Spacks includes several additional features to address frequent requirements of HPC community (GAMBLIN, 2018):

Figure 4.2: A graph with software dependencies of the Chameleon Solver



Source: The Author

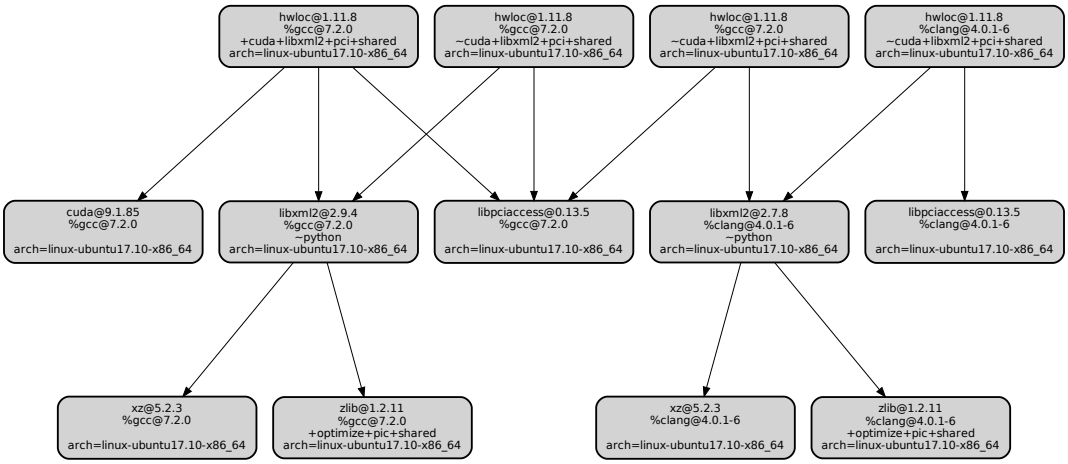
- *customized configurations*: users can specify package version, build compiler, compile-time options and cross-compile;
- *customized dependencies*: package dependencies of a particular install can also be customized, e.g., compiling a package with one compiler and a dependency with another one;
- *non-destructive installs*: a new version does not break existing ones. Since each install is identified by a combination of keys (i.e., package version, compiler, architecture, and variants), change one of this keys will generate a new install and will not override existing ones.
- *multiple installs can coexist*: Spack relies on *run-time search path* (*rpath)* to hard-code the path to shared libraries into the executable header. The user can install an older or newer version of a package P without breaking other packages depending on previous existing versions of P;
- *custom repositories*: a new package can be easily created which allows developers to make available their own packages as an additional repository.

The dependency graph of Figure 4.3 illustrates how different installs of the same package can coexist in a single Spack instance. In this example, the *hwloc* package was installed with four different combinations of options. The first three *hwloc* installs were compiled with gcc, but the first has the cuda variant enabled. The first two shares the same dependency chain, while the third one was compiled with gcc and linked with some dependencies compiled with clang. The last one was entirely compiled with clang.

After compiling and installing a package, Spack stores the build logs containing the output and error messages and the complete combination of install options, e.g., package version, compiler, architecture, and variants. These files can be used to check the build status of a previous install or to recompile the package using exactly the same options of the original build.

In order to support Spack in the workflow, we have included in the *master* branch a *script* to install and configure Spack. The default install is modified to preserve the source code of the packages after their build. When starting a new experiment, one can use Spack, either a new instance or an existing one, to install the experimental software as well as its dependencies. The install *script* is also able to apply patches to the source code when making a package. All build log files generated by Spack during the package compilation are committed as well as the packaged source code and its dependencies. The

Figure 4.3: Example of multiple installs of a package in Spack.



Source: The Author

state of the Spack instance is recorded in a section of the experiment log file. This state comprises the spack version, the configuration of all installed packages and the package state of the target experimental software. The use of Spack facilitates the install of the experimental software as well as allows us to track the not only source code of the target package but also of its dependency chain.

4.2.3 Rebuilding the Software Stack

In several cases, experiments are executed using the development version of the source code. Despite unstable, using the last available version of the code enable testing new features and benefit from recent bug fixes. Unlike stable releases, under development code usually has no version number which might be a reproducibility problem. To address this issue we have extended the approach presented in Section 4.2.1 by storing the state of the Spack package into the execution log, as illustrated in Figure 4.4. This state comprises the package install options and the code used in the compilation, including optional patches. Since package dependencies can be also compiled from unstable versions, we also keep track of their details.

We have extended Spack code with small changes to rebuild the entire software stack of a previous experiment. By default, all unstable target packages and its dependencies are recompiled using the stored source code. Stable dependencies are fetched again

Figure 4.4: Fragment of an execution log containing Spack package state.

```
-- linux-ubuntu16.04-x86_64 / gcc@5.4.0 -------------------------
g5vqqkb    chameleon@master%gcc@5.4.0 ~cuda+examples+fxt~mpi~quark+shared~simgrid+starpu
y5ddruz        ^cmake@3.10.0%gcc@5.4.0~doc+ncurses+openssl+ownlibs~qt
nxkp7ks            ^ncurses@6.0%gcc@5.4.0 patches=4110a40,f84b270 ~symlinks~termlib
sk7ayvj            ^openssl@1.0.2n%gcc@5.4.0
5nus6kn                ^zlib@1.2.11%gcc@5.4.0+optimize+pic+shared
u5xoja5        ^fxt@0.3.5%gcc@5.4.0~moreparams
nhc5doq        ^intel-mkl@2018.1.163%gcc@5.4.0~ilp64+shared threads=none
meeo7tb        ^starpu@svn-trunk%gcc@5.4.0~blas~cuda+examples+fast~fortran+fxt+mlr~mpi~
    nmad~opencl~openmp+shared~simgrid~simgridmc~verbose
hro43jw            ^hwloc@1.11.8%gcc@5.4.0~cuda+libxml2+pci+shared
5urc6tc                ^libpciaccess@0.13.5%gcc@5.4.0
sxk64lv                ^libxml2@2.9.4%gcc@5.4.0~python
htnq7wq                    ^xz@5.2.3%gcc@5.4.0
```

Source: The Author

from their official providers and then are also rebuild using exactly the same version and compilation options.

4.2.4 Storage of Large Files in GIT

In the approach discussed in Section 4.2.1, not only the source code but also the results files produced by an experiment are stored in the Git repository. This strategy is feasible for experiments producing small data files in plain text format. Since Git uses a very efficient algorithm to compress text files, this probably will not lead to performance or storage problems. However, some experiments produce large data files, sometimes in binary formats, that are not well managed by Git repositories. The storage of large files may slow down some git operations and incurs in storage quota issues. To address this question, we propose an extension based on a combination of **git-annex** and Zenodo.

git-annex (HESS, 2018) is a Git extension to manage only the metadata of files stored in Git without tracking its content. The extension uses an additional branch to track the information about the annexed files. All commits and merges on this branch are automatically handled by **git-annex**. Files managed by **git-annex** are stored even in a special directory inside the $./git$ directory tree or in an external server. When a new file is added, **git-annex** will create a symlink that points to the real content. This way, it is possible to access annexed files in their original place, in the same way of normal versioned ones. The symlinks are committed and versioned as normal Git files. **git-annex** can work without support on the server side, however, this way it is not possible to store annexed file content on this server. Lack of compression in annexed files can be

another limitation. Since it was designed to store the files in its original state[5], add files in text format with **git-annex** will use more disk space than if they were added with Git.

Our approach based on **git-annex** can significantly reduce the Git overhead when managing large files, however, it is not scalable in terms of storage. Git servers normally define a storage quota per user or project, then using a pure **git-annex** solution to store our large results data is not viable. For that reason, we design a solution based on the *special remotes*[6] feature of **git-annex**. This feature allows storing annexed files in a wide range of external (non-git) servers. It offers internal support for upload and download content from several commercial cloud services, e.g., Amazon S3, Google Drive, Dropbox, and OneDrive. Since these are commercial and non-academic services, they are constrained by storage quota, license issues and lack of persistence.

Zenodo (CERN; OPENAIRE; COMMISSION, 2018) is a web storage service designed to acts as an open access research data repository. Initially implemented as a storage infrastructure for research data from CERN's Large Hadron Collider, Zenodo is now open for any institution or research field. Researchers can upload their research outputs with no storage quota, the only limitation is the file size (up to 50 GB). An upload can include multiple files and receives a Digital Object Identifier (DOI), which ensures that each register is unique and citable. There are no restriction to file formats and a wide range of licenses is available (NIELSEN, 2017).

While Zenodo seems to fit the technical requirements to be used as our storage server, there is no native support for it in **git-annex**. To integrate Zenodo in our approach we implemented a deposit script that moves the files from the annex storage on the Git server to Zenodo using the Zenodo REST API[7]. To ensure that data was not corrupted during the transfer we perform a checksum verification between the local file and the uploaded one. After that, we add the Zenodo URL pointing to the uploaded file as an additional source to the annexed file and we ask **git-annex** for removing the copy on the Git server. This removal is successful only if **git-annex** can ensure that the annexed file has at least one valid source by testing the availability of the Zenodo copy.

The idea is to use the main Git server as temporary storage for ongoing experiments. Once an experiment is concluded, we consolidate the results pushing them to Zenodo, avoiding compromising the storage quota on the Git server. When the main Git server has no support for **git-annex**, we can use the local Git client as storage, and then

[5] which is desirable when working with binary files or to ensure access without rely on Git and **git-annex**.
[6] `https://git-annex.branchable.com/special_remotes/`
[7] `http://developers.zenodo.org/`

move the results to annex as soon as the experiment is concluded. In our approach, we use **git-annex** and Zenodo only to store data files produced as a result of an experiment. All other contents of the repository, including source code files, are stored as regularly versioned files.

The inverse operation to retrieve data from Zenodo requires a simpler strategy since we can use the web as a special remote to **git-annex**[8]. Since all our Zenodo uploads are stored with public access, **git-annex** can directly download the copy without depends on tokens or login steps.

The benefits of our **git-annex** and Zenodo strategy are twofold, at the same time we ensure a reliable place to store experiments results, we also provide public direct access to these results without depending on Git or **git-annex**.

4.3 Chapter Summary

In this Chapter, we present our approach to improving the reproducibility of this work. In order to make it more transparent and accessible to other scientists, we rely on two steps: a reproducible report and a workflow to generate input data.

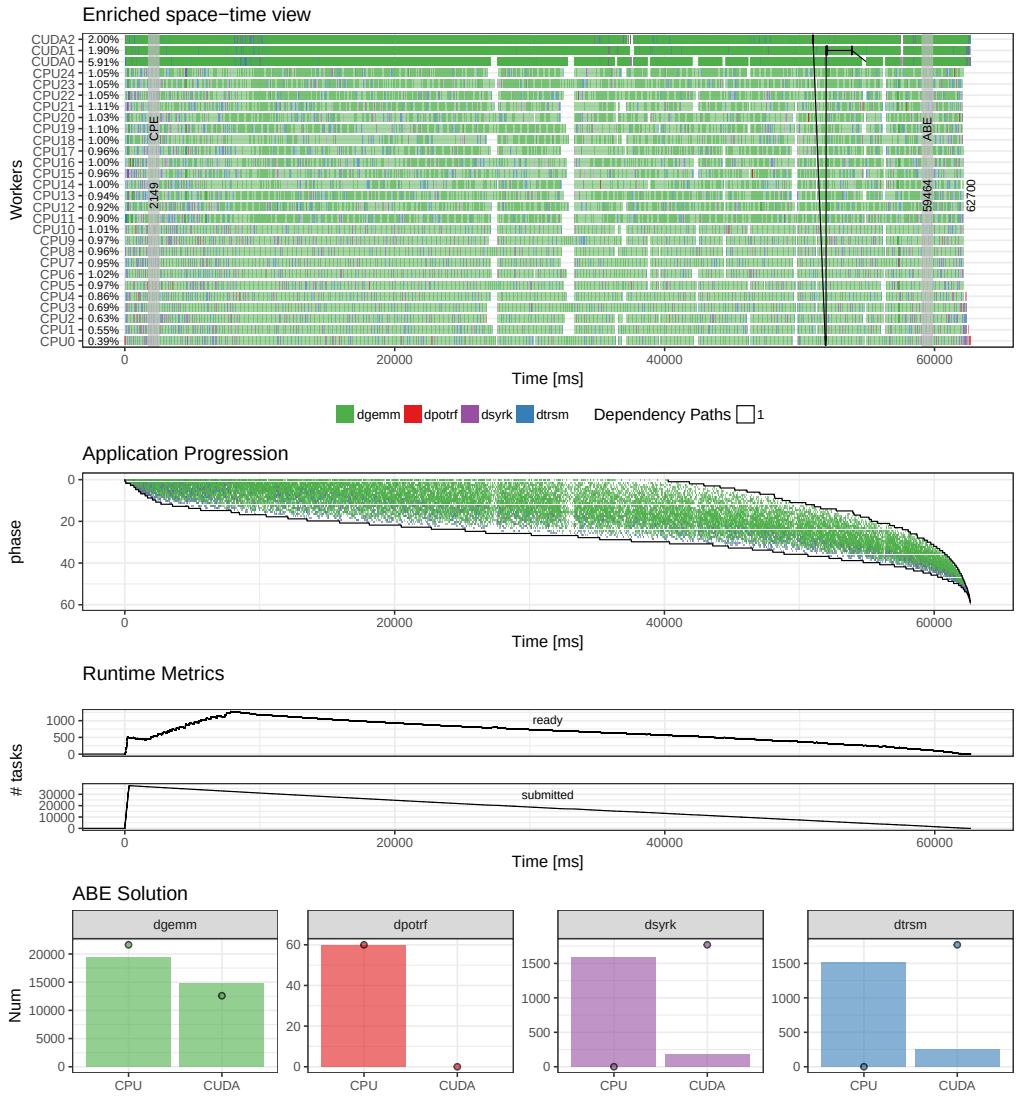
The reproducible report step relies on Org-mode to include and mix source code blocks within the text. This way, from the source file used to generate the present document it is possible to retrieve the experiments raw data as well as the steps to go from this raw data to the final visualizations presented in the figures. Our second step concerns the generation of the input data used to perform the analysis. We apply and extend a Git and Org-mode strategy that enables us to keep track of not only the experiment output (e.g., traces) but also the software stack and the machine state. We believe that this approach combining instructions inside the report with the results companion (i.e., data and hardware/software context) will be useful to both understand this work and build on it to extend its results.

[8] `https://git-annex.branchable.com/tips/using_the_web_as_a_special_remote/`

5 PROPOSED VISUALIZATIONS STRATEGIES

In this chapter, we detail our proposed visualizations strategies designed to meet the requirements and the specific characteristics of HPC task-based applications running over hybrid platforms. As discussed in Section 3.1, space-time plots, which get inspiration from traditional Gantt charts (WILSON, 2003), are a frequently-used visualization strategy employed by several HPC analysis tools to describe the application states along the space (computing resources) and the time (application duration). Since HPC analysts are familiar with this kind of view, we decide to build on it to propose new alternative visualization strategies that can both enrich the basic space-time representation and extend it by adding time-synchronized extra panels, as shown in Figure 5.1.

Figure 5.1: An overview of the proposed visualization strategies



Source: The Author

To implement our proposed visualizations, we build on top of modern data analytics tools, combining `pj_dump`, the *R* programming language (R CORE TEAM, 2018), its *ggplot2* library (WICKHAM, 2016), the *lpSolve* library (BERKELAAR et al., 2015), and the data manipulation functions provided by the *tidyverse* meta-package (WICKHAM, 2017), Org-mode (DOMINIK, 2010), and *plotly* (SIEVERT et al., 2017). Thanks to the expressiveness of the R language and this rich software stack, we can not only visualize the data available in the traces but also produce new information by processing them. Before producing a plot from the traces data, we perform several intermediate steps such as clean-up, filtering, statistics computation, merges, and aggregations. This approach allows building static views in a fully automatic and very efficient way. Although such visualizations could probably be accelerated even further by programming everything in C/C++, the used libraries are already well optimized and benefit from the know-how of data analysts. Furthermore, a combination of small scripts is easier to maintain and adapt to a new necessity or to a particular situation than a rigid monolithic visualization environment.

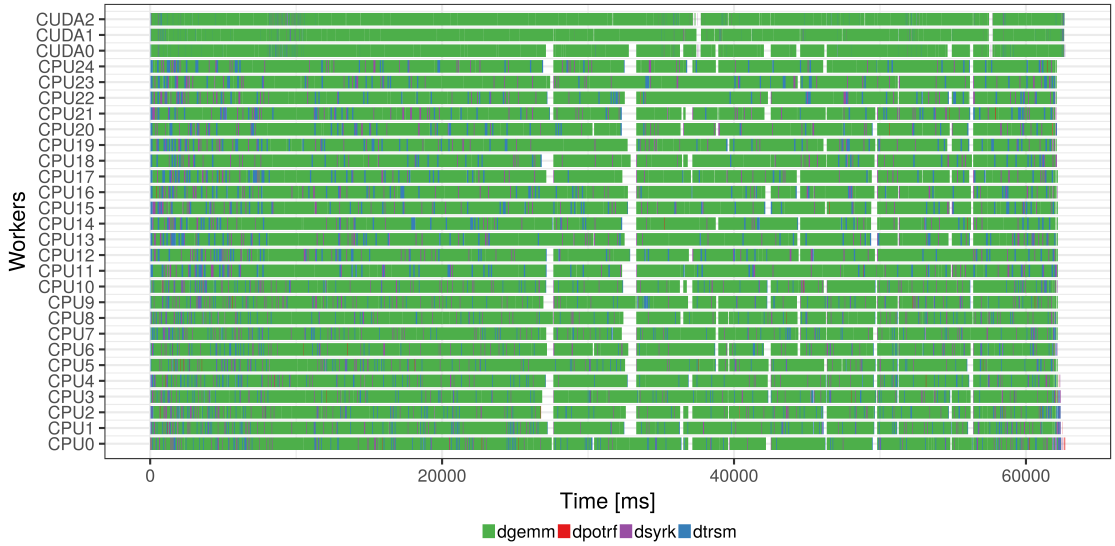
In the remainder of this chapter, in Section 5.1, we discuss how we increment the basic space-time view to highlight application and platform characteristics. In Section 5.2, we depict the additional panels that disclose more details about the application and runtime internal information. Finally, on Section 5.5, we describe the format and content of the input data required by our approach to generating the views presented in Sections 5.1 and 5.2. All the following demonstrations were built using traces from a dense Cholesky decomposition which is further discussed in details in Section 6.1.2. For understanding the rest of this Chapter, it is important to mention that this application is designed with tasks of four types (DPOTRF, DTRSM, DSYRK, and DGEMM). In the figures, each type is represented by a different color. Tasks have implementations for both CPU and GPUs and all tasks of the same type have the same size.

5.1 Enriched Space-time View

In this section, we describe our proposals to improve a basic space-time view as the one of Figure 5.2. This kind of space-time view is provided by several HPC analysis tools but is too simplistic for task-based scenarios on heterogeneous platforms. In the next subsections, we propose enrichments that can be applied to this kind of view to make it more suitable for such scenario. Our enrichments are built on this standard visualization

and are incrementally applied as layers to the basic plot. The addition of these layers does not require a strict order.

Figure 5.2: A basic space-time plot



Source: The Author

5.1.1 Idleness

In the space-time view, the huge amount of tasks might give a false impression that the resources occupation is good. However, in several cases, there are a lot of small idle periods spread during the execution. Since their duration is much smaller than that of computing tasks, they are frequently hidden in the classical graphical representation. To avoid this misleading impression, we compute the overall idleness ratio of each resource, which facilitates the comparison between the occupation of different resources.

To quantify the idleness of a computing resource, we should define what will be considered as idle. One possibility relies on some runtime provided information such as *Idle* and *Sleeping* states traced by StarPU. This option will exclude from idleness all the scheduling related states like *Data transferring* or *Initialization*. Another possibility is to consider all non-computing states as idle periods since the time spent during initialization or fetching data is not directly used to solve the target problem. Figure 5.3 shows two space-time views using the aforementioned approaches to compute resource idleness. For a given computing resource, the idleness ratio can be significantly different from one approach to another, e.g., in Figure 5.3(a), CPU2 spends 0.63% of the time in idle while

in Figure 5.3(b) its idleness ratio is 4.85%.

Figure 5.3: Enriched space-time view with idleness quantification.

Source: The Author

5.1.2 Outliers or Task Duration Anomalies

In regular applications such as dense linear algebra, the task duration should vary only in terms of its type and the type of computing resource where it is executed. This assumption can be visually verified highlighting all tasks whose duration is anomalously larger than others of the same type/resource. This kind of enrichment can help in the recognition of unexpected behaviors such as delayed tasks or platform disturbances. The definition of a threshold value to separate anomalous tasks from normal ones is highly debatable and context specific. Ideally, it should be provided by an analyst with knowledge of the application. In this work, we use the equation 5.1 to exemplify how this view

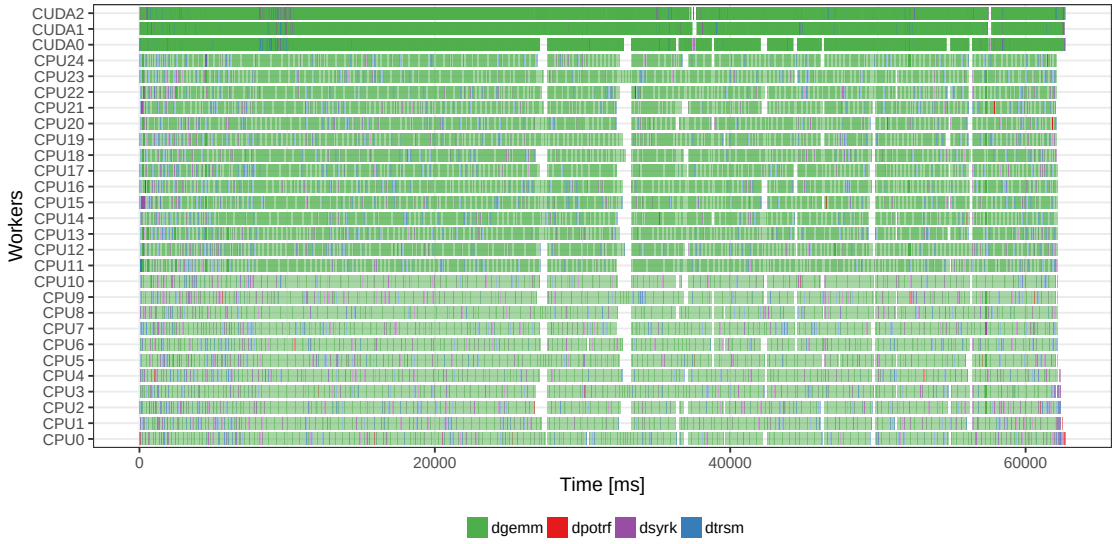
works. The duration of a task is considered anomalous if:

$$task_{duration} \geq Q_3 + 1.5 * IQR \tag{5.1}$$

where Q_3 is the third quartile and IQR is the interquartile range.

Figure 5.4 presents an example of a space-time view enriched with anomalous tasks information. Tasks with normal duration are drawn with lighter colors while the ones with anomalous duration are drawn with darker shades.

Figure 5.4: Enriched space-time view with highlighting of tasks with anomalous duration.



Source: The Author

Since task duration may not be adequated for some application (e.g., sparse linear algebra), this feature can be easily extended to highlight other metrics than task duration, e.g., cache misses, instructions per cycle (IPC), FLOPS.

5.1.3 Bounds for the Makespan

Comparing and understanding the performance of task-based executions on hybrid platforms is challenging. Such executions are inherently stochastic which makes a purely visual analysis unfeasible. The task mapping, for instance, can change from one execution to another even when using the same scheduling policy.

Once a simple visual comparison is impractical, we need some additional information in order to help the performance analysis. The use of makespan bounds is a classical

tool to assess the performance of parallel applications (GRAHAM, 1966, 1969; GAREY; GRAHAM, 1975). In this work, we have implemented two execution time bounds. The Critical Path Estimation (CPE) is obtained from the sum of the duration of each task in the critical path on its fastest implementation (CPU or GPU) while the Area Bound Estimation (ABE) is computed using the linear program defined in (5.2)-(5.5). Where T is the set of task types, R is the set of resource types, $n_{t,r}$ is the number of tasks of type t running on a resource of type r, N_t is the total number of tasks of type t, $w_{t,r}$ is the mean time spent to execute one task of type t on a resource of type r and N_r is the number of resources of type r.

$$\text{minimize:} \qquad makespan \tag{5.2}$$

subject to:

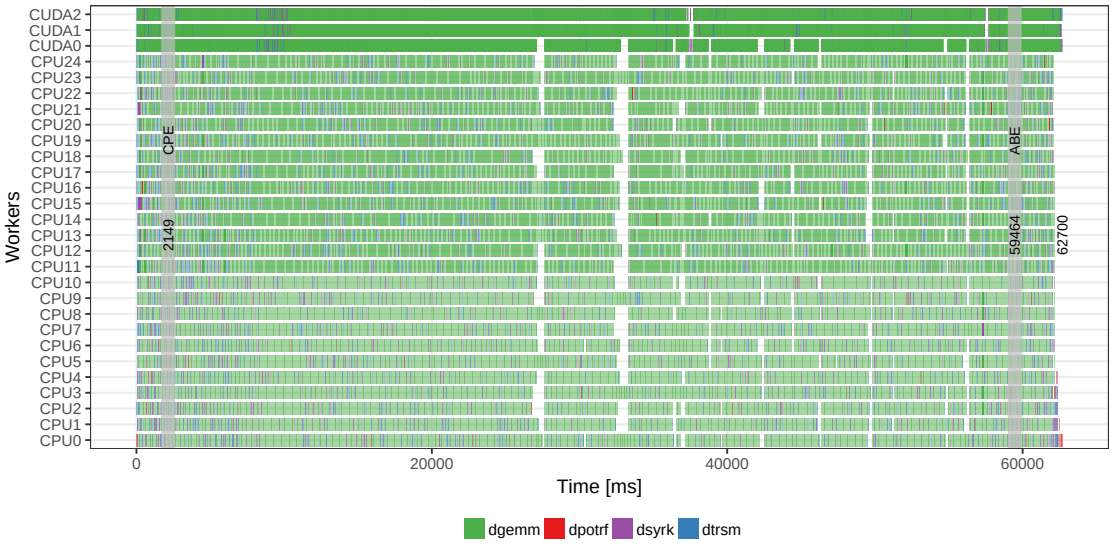
$$\forall t \in T: \qquad \sum_{r} n_{t,r} = N_t \tag{5.3}$$

$$\forall r \in R: \qquad \sum_{t} n_{t,r} * w_{t,r} \leq makespan * N_r \tag{5.4}$$

$$\forall t \in T, r \in R: \qquad n_{t,r} \geq 0 \tag{5.5}$$

The standard space-time view can be enriched to show the CPE and the ABE bounds as presented in Figure 5.5. Note that the CPE bound is too optimistic for an execution of this application with this problem size. This behavior will be discussed in details in Section 6.2.1. However, the ABE bound results in a more realistic value (≈59464 ms) which is much closer to the observed makespan (≈62700). The difference of ≈5.4% between this bound and the real value indicates that there is room for performance improvements. Our ABE bound does not consider the task dependencies, so for smaller problems, it might be less precise. More accurate lower bounds including task dependency constraints (AGULLO, E.; BEAUMONT, et al., 2015) could be used as well. This solution could be better than our ABE bound, in particular, for intermediate size workloads. When executing on distributed-memory platforms, the ABE bound is computed per node. This way, it can be also used to evaluate the load balancing among the nodes.

Figure 5.5: Space-time view with CPE and ABE bounds. The bounds are represented by two vertical gray lines while the real observed makespan is drawn in the position indicated by its value, at the end of the execution (≈62700 ms).



Source: The Author

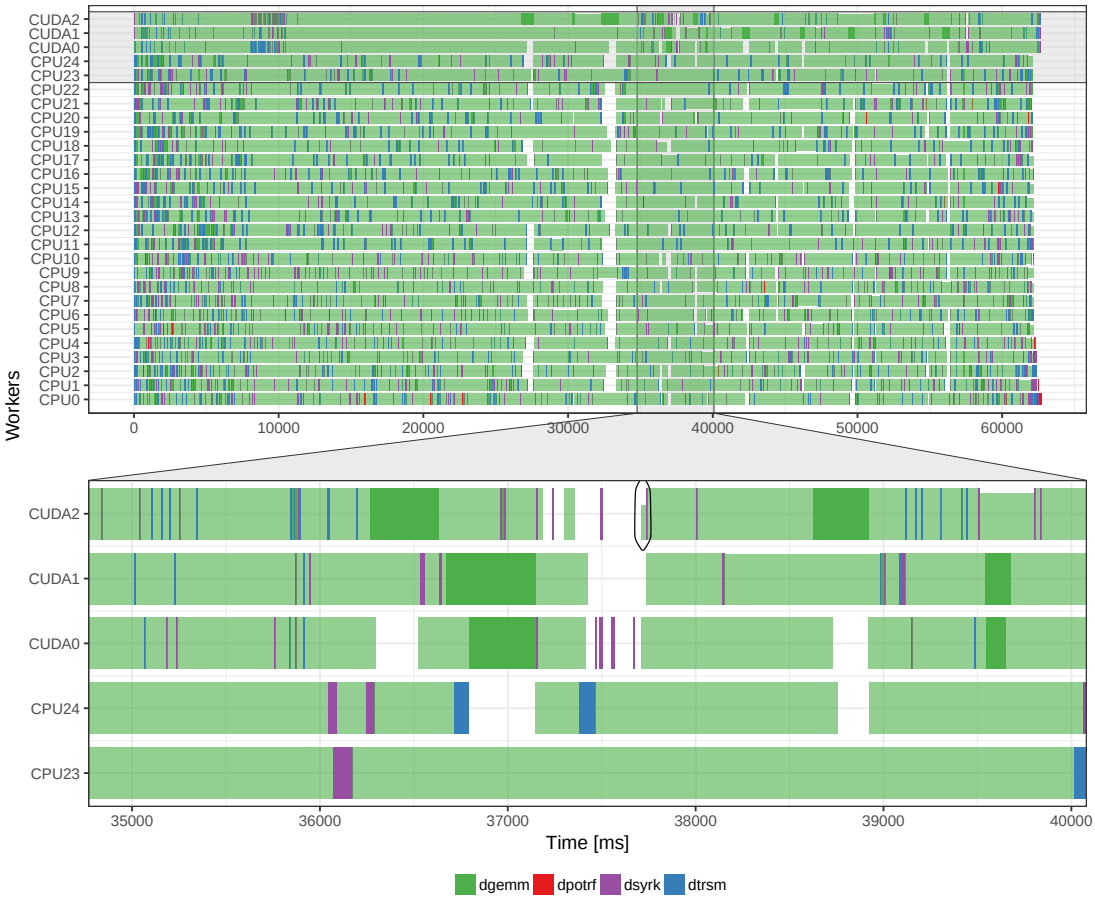
5.1.4 Aggregation

The total number of tasks in task-based applications can be potentially large, e.g., the Cholesky factorization of a 60×60 matrix totalizes 37820 tasks. Plotting all the tasks can significantly increase the complexity of the visualization, which reduces the plot readability and could lead to rendering issues. To reduce the number of objects in the visualization, we have designed and implemented a task-aware temporal aggregation. Our algorithm consists of two main steps: first, we group per-resource consecutive tasks (see Algorithm 1), and after we compute the proportion of time spent by each task type in the group. Ideally, this aggregation procedure is applicated in task-types that are numerous.

The space-time view of Figure 5.6 shows the result of our aggregation technique. In this example, the aggregation procedure is applied to merge consecutive DGEMM tasks. Non-aggregated tasks are represented by full-sized (height and width) rectangles filled with their original task-type related colors. Aggregated tasks are represented by a common rectangle whose width is extracted from the lower *start value* and the higher *end value* among the grouped tasks. This rectangle is filled with a lighter color to differentiate it from those representing non-aggregated tasks. By default, small idle times appearing among aggregated tasks are also aggregated, in that case, the height of the rectangles is used to depicts the percentage of time spent computing tasks of such type (see the black

circle in the zoomed area of the figure). Our technique also enables the exclusion of tasks with longer duration from any kind of aggregation (see darker green tasks in the zoomed area). This exclusion also applies to long idle periods (white areas) as their occurrence indicates potential performance issues.

Figure 5.6: Space-time view with aggregation of DGEMM tasks with duration smaller than 100 ms. Lighter green areas represent aggregated tasks. The circled area illustrates an aggregated time slice where ≈66.03% of the time was spent in the execution of DGEMM tasks.

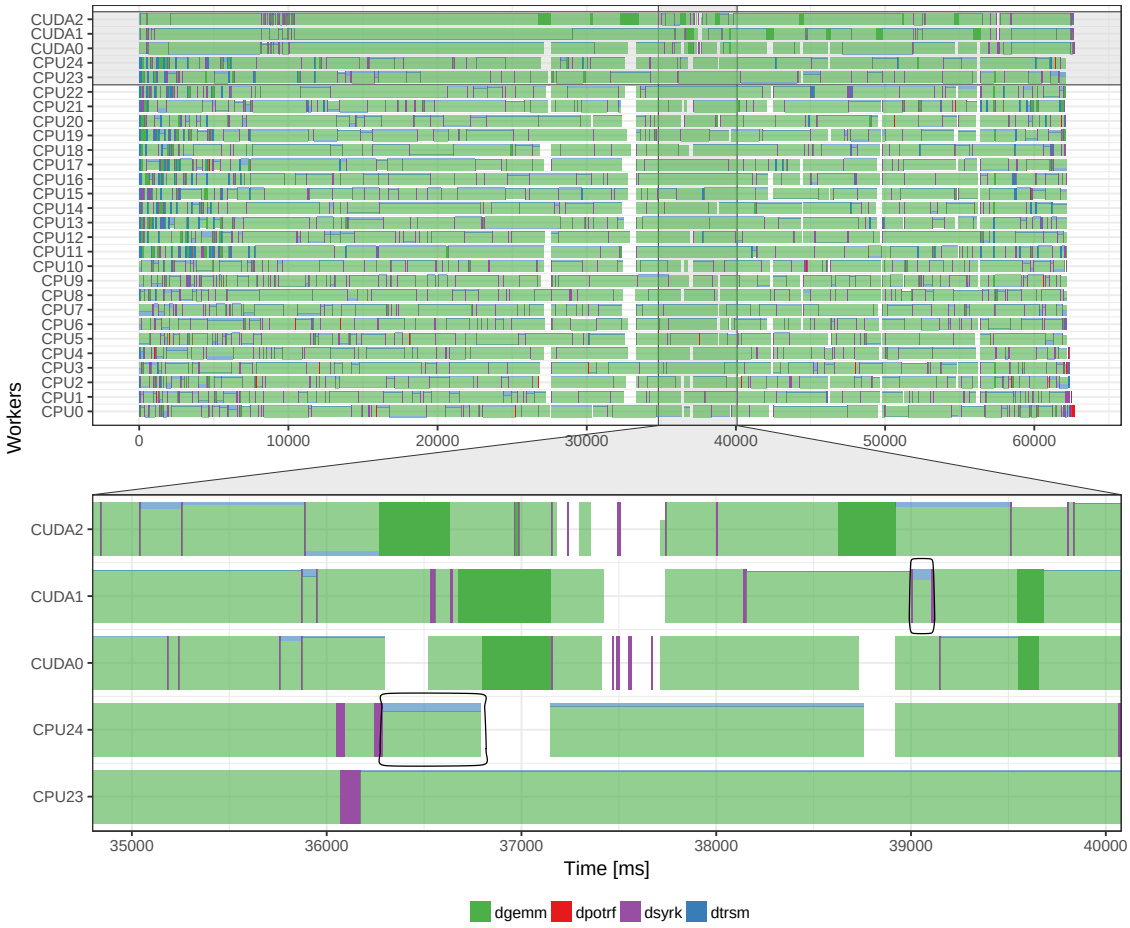


Source: The Author

In some cases, only few task types are relevant to understand the application behavior, so all the other types can be grouped together keeping only an overview about their occurrence in the time. Figure 5.7 shows an example of such cases. DGEMM and DTRSM tasks are grouped together, emphasizing DSYRK tasks that are kept pure. The two circled areas illustrate how we use a kind of stacked bars to represent the aggregation of two or more task types in the same time slice. Our technique also allows the exclusion of specific tasks from any aggregation filter, this is useful when focusing on few tasks of numerous type. Figure 5.8 illustrates how it works when excluding three tasks that were

aggregated in the previous example (Fig. 5.7).

Figure 5.7: Space-time view with aggregation of DGEMM and DTRSM tasks with duration smaller than 100 ms. The circled areas illustrate the stacked representation when two or more task types are aggregated in the same time slice. The height of blue/green rectangles is proportional to the time spent computing the respective task-type during the time slice.



Source: The Author

The main focus of our task-aware temporal aggregation is to reduce the amount of data to be processed or visualized. The aggregation filters presented in the previous examples help us to prevent the hiding of relevant information such as idle gaps, tasks with longer duration or even specific tasks in which the analyst has a special interest. This approach could also be combined with statistical techniques to enrich the visualization highlighting some unusual or unexpected behavior. In Figure 5.9 our technique was associated with a mechanism to detect outliers, so we reduce the information to render by aggregating all task except those whose duration is significantly longer than others of the same type executing in the same computing resource. A non-aggregated view as the one of Figure 5.2 has 37820 objects while the aggregated ones of Figure 5.9 reduce this

Figure 5.8: Space-time view with aggregation of DGEMM and DTRSM tasks with duration smaller than 100 ms (excluding some tasks). The circled tasks were excluded from the aggregation filters.

Source: The Author

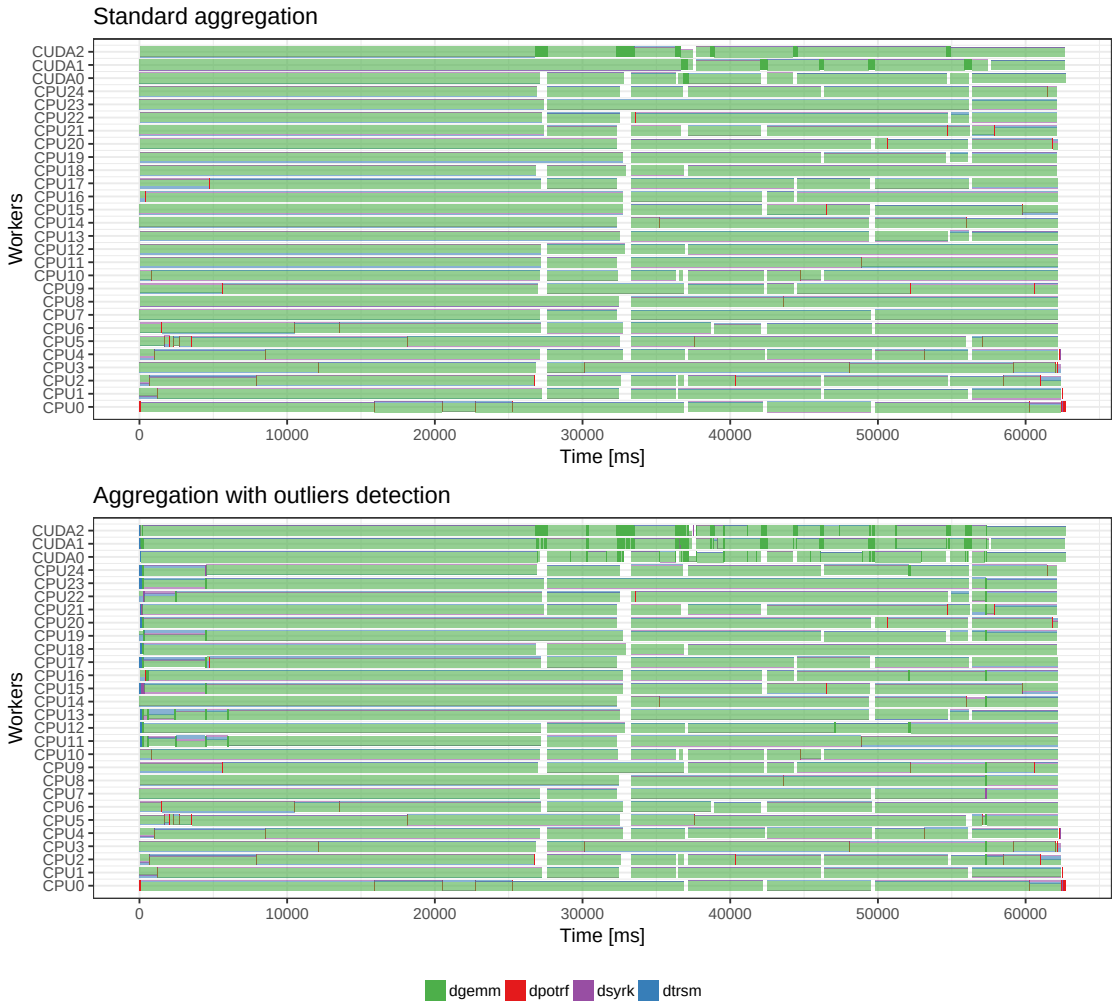
Algorithm 1: Computation of task groups

Input: $tasks$: a table with tasks' execution data (start, end, duration, type, resource)
$threshold$: minimal duration to consider a task as pure
$types$: list of task types to group
$excludeTasks$: list of tasks to be excluded from any kind of grouping
Output: $tasks$: a table with a new column with the task group

```
1  foreach resource r in resources do
2      foreach task t_i in tasks_r do
3          duration_{t_i} ← duration of task t_i;
4          start_{t_i} ← start time of task t_i;
5          end_{t_{i-1}} ← end time of task t_{i-1};
6          if duration_{t_i} > threshold then
7              mark task t_i as a pure task;
8              put task t_i in a new group;
9          else if start_{t_i} > (end_{t_{i-1}} + threshold) then
10             put task t_i in a new group;
11         else if previous task t_{i-1} was a pure task then
12             put task t_i in a new group;
13         else
14             group task t_i in the same group of previous task t_{i-1};
15         end
16     end
17 end
```

amount to 607 objects (standard aggregation) and 1069 objects (aggregation with outliers detection).

Figure 5.9: Space-time view with aggregation of DGEMM and DTRSM tasks excluding outliers.



Source: The Author

5.1.5 Dependencies

The dependency management has a crucial impact on the overall performance of task-based applications. The delay of a critical task can produce idleness in the computing resources due to lack of ready for execution tasks. As a result, some information about task dependencies must be included in the trace visualization panel. Task dependencies, usually, can be statically inferred from the source code (see the cholesky algorithm in Section 6.1.2). Although this information is easily retrievable, plot it in a raw state is

impractical since the number of dependencies is prohibitively large. The DAG of the cholesky factorization with a matrix of 60×60 blocks has 37820 nodes (tasks) with 111630 edges (dependencies). Drawing all the tasks with its respective dependencies would produce a meaningless graphic, once the tasks would be possibly overlapped by the dependency edges. For that reason, some filtering procedure must be applied while keeping only the relevant task dependencies.

Figure 5.10 presents a space-time chart with black lines indicating the dependencies of some delayed tasks. In this example, the number of dependencies varies with the task type. The delayed DGEMM on CPU14 depends on three other tasks: a DTRSM on CUDA2, a DGEMM on CPU14 and a DTRSM on CPU2. Inspecting all the dependencies of a task is relevant, especially when designing the algorithm to keep the sequential consistency and ensures its correctness. However, in terms of performance analysis, only the last dependency is important since it enables the execution of the next task. In the aforementioned DGEMM on CPU14, only the dependency with the DTRSM on CUDA2 is relevant since it enables the execution of the task.

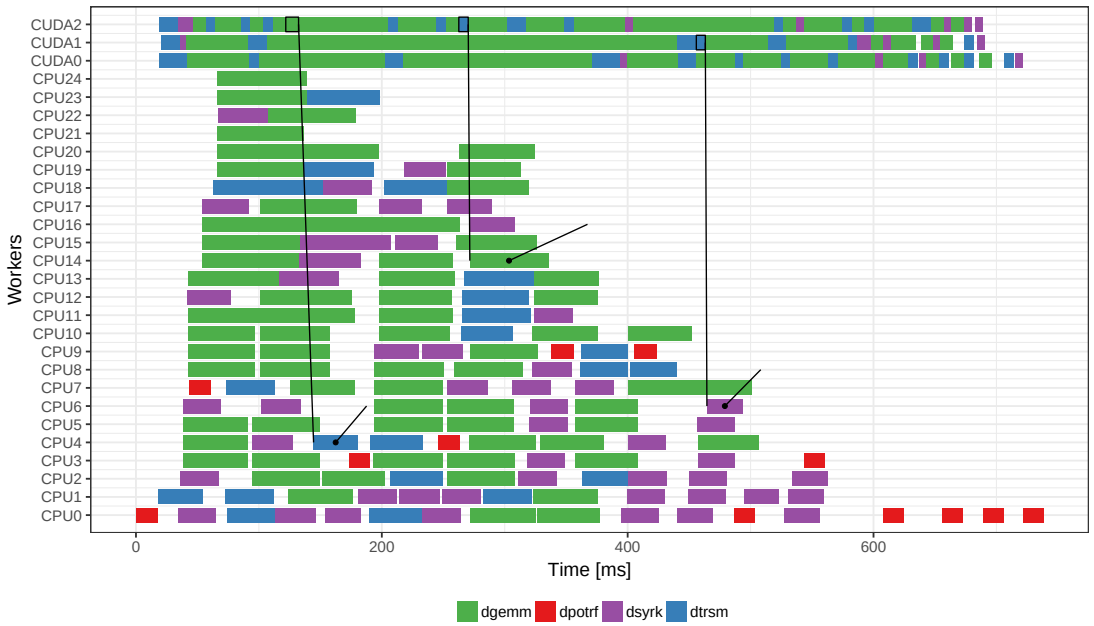
Figure 5.10: Space-time view with all dependencies of some selected tasks.

Source: The Author

To find the last dependency of each task we should combine the static information of the DAG with the execution trace containing the start/end values for each task. The space-time view of Figure 5.11 shows only the last solved dependency of each pinpointed task. The dependency edge inclination indicates the delay between the end of the last dependency and the beginning of the task in question. A horizontal gap could indicate

room for performance improvements once the task has not started as soon as possible. Note that the last dependency of a specific task can be different from one execution to another since the task scheduling is dynamic.

Figure 5.11: Space-time view showing only the last dependency of a some given tasks.

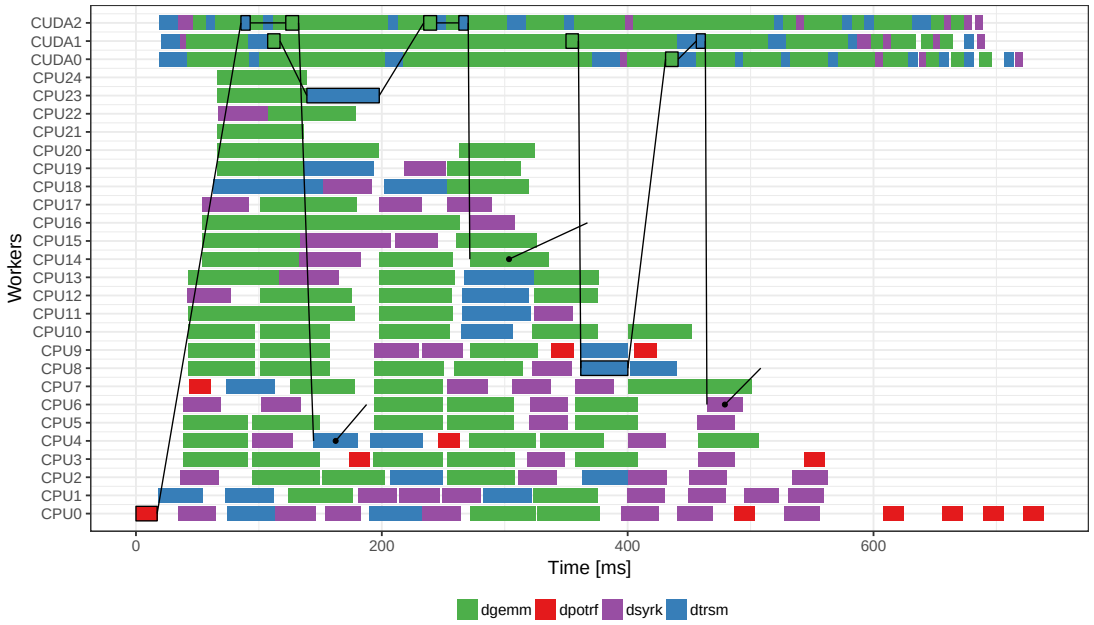


Source: The Author

The space-time view presented in Figure 5.11 depicts the direct dependencies of each task. In several cases, this first-level does not clarify the problem. The delayed DGEMM on CPU14 illustrates one of these cases since this task has started just after its last dependency (DTRSM on CUDA2). To understand why this DGEMM did not start before we should inspect more levels in the DAG. This backward chain is obtained by recursively searching the last task release a given task, i.e., for a given task T_i, we search what was the task T_{i-1} on which it depends on and that finished the latest, similarly, for T_{i-1} its latest predecessor T_{i-2}, etc. In Figure 5.12, the dependency edges are drawn with at least three-levels which discloses the dependency chain, showing another delayed tasks among the antecedents of the aforementioned DGEMM.

In task-based applications, tasks releasing many other tasks are good candidates to filtered dependency track. The view of Figure 5.13(a) shows the dependency tracking for DPOTRF tasks. As expected from the cholesky DAG, a DPOTRF is immediately preceded by a DSYRK, immediately preceded by a DTRSM that would, in turn, be immediately preceded by the DPOTRF of the previous iteration. Since the dependency chain of one DPOTRF will probably meet the dependency chain of the precedent DPOTRF we can try to merge these chains to simplify the view. These merged dependency chains, as shown in

Figure 5.12: Space-time view with the backward dependency chain of some selected tasks.



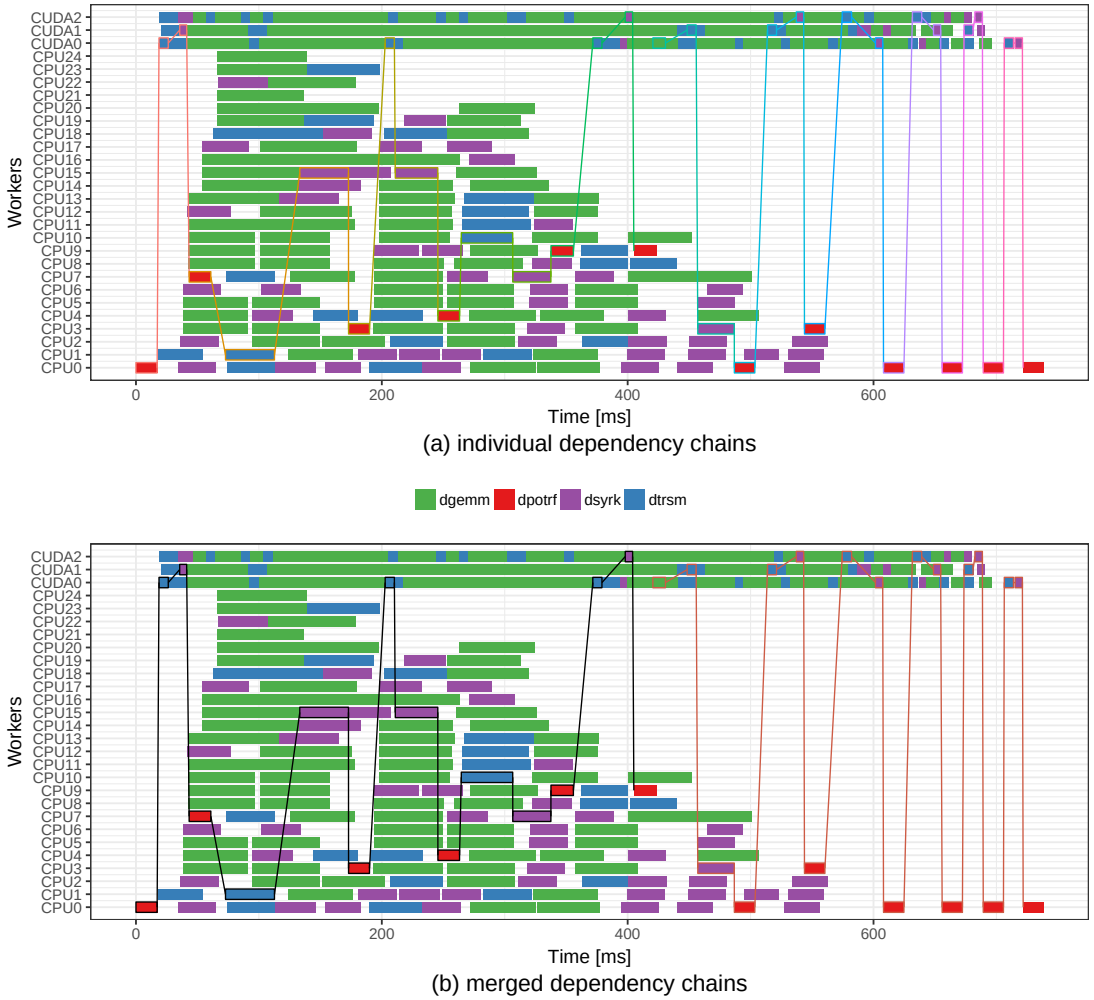
Source: The Author

Figure 5.13(b), are computed using union–find operations for disjoint-set data structures (CORMEN et al., 2009). The original 12 paths were merged into two new ones, that illustrate two moments where the execution has proceeded exactly as expected from the DAG critical path.

5.2 Additional Views

Since the traces of task-based applications can be much richer than classical MPI or OpenMP ones, it is not helpful to include all the information in the space-time view. For this reason, we increment the space-time view with additional panels to describe extra information about the application, the runtime, and the platform. To allow an easier comparison, the temporal axis of all panels are synchronized with the corresponding one in the space-time view. In the remainder of this section, we detail the Application Progression panel in Subsection 5.2.1, the Scheduler Task Metrics in Subsection 5.2.2, and the ABE solution in Subsection 5.2.3.

Figure 5.13: Space-time view with the backward dependency chain of DPOTRF tasks.



Source: The Author

5.2.1 Application Progression

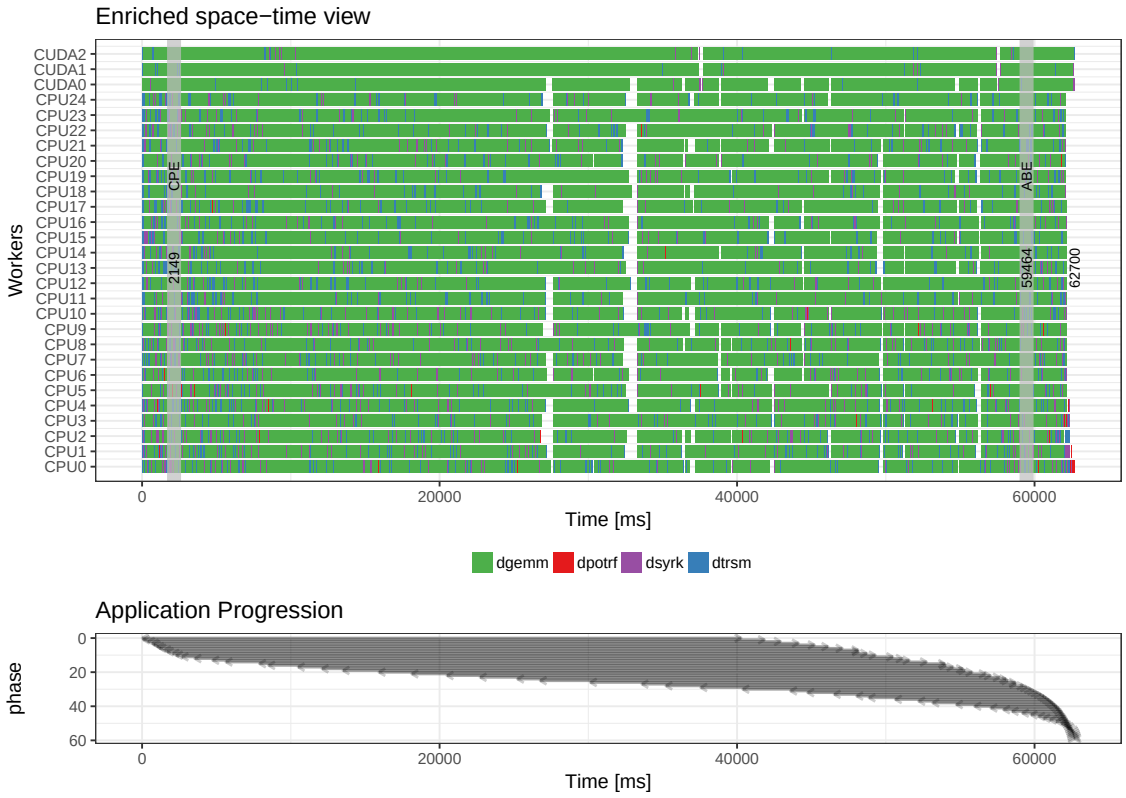
The way the graph of tasks is explored is related to the scheduling policy in task-based applications. In several linear algebra algorithms, the matrices are divided in tiles, and loop iterations are used to apply the algorithm on each tile. Some scheduling policies, like in a sequential execution would traverse the graph in a breadth-first design, executing all tasks of one iteration before starts the next one. However, other scheduling policies can traverse the graph in a depth-first design, favoring task execution on the critical path, and then it can start tasks of the next iteration before the ending of the previous one. This strategy is possible only in models which allows describing the task dependencies in terms of data-accesses (read, write, read-write), such as StarPU or OpenMP 4. In traditional task-based tools that follow the fork-join model to create and synchronize tasks, the DAG is traversed in a breadth-first style. This is the case of programs using the **spawn-sync** (Cilk) or **task-taskwait** (OpenMP) primitives.

Designing a common view that illustrates the progression of any application, whatever its structure, is challenging. However, for iteration-based applications such as tiled linear algebra algorithms, the tracking of the first and the last task in each iteration of the outer-loop provides a useful way to understand how the DAG is handled by the scheduler.

In Figure 5.14, the new panel in the bottom illustrates the progression of a Cholesky execution. To produce this view, all application tasks are tagged according to their membership to a given loop. During the trace processing, we merge this static information with the tasks' begin/end timestamps to identify the first and the last task of each iteration. The temporal axis (x-axis) of this additional panel is aligned with the corresponding one in the space-time view. Each horizontal segment, represents one iteration of the outer loop of the cholesky factorization, indicating an interval where the tasks of the iteration have been executed. Since task dependencies are expressed in a data-flow model, several iterations are processed in parallel. For this specific execution, the maximum number of simultaneous loop-iterations was 30 around 40000ms. In this example, there are 60 outer-iterations since the target matrix was divided into 60×60 tiles. In general, a wider shape indicates more parallelism.

In most cases, the overview with segments showing the duration of each iteration is sufficient to understand the DAG progression and to differentiate executions with multiple schedulers. Despite that, since all tasks were tagged, it is possible to enrich this view to show not only the begin and the end of each iteration but also the distribution of the

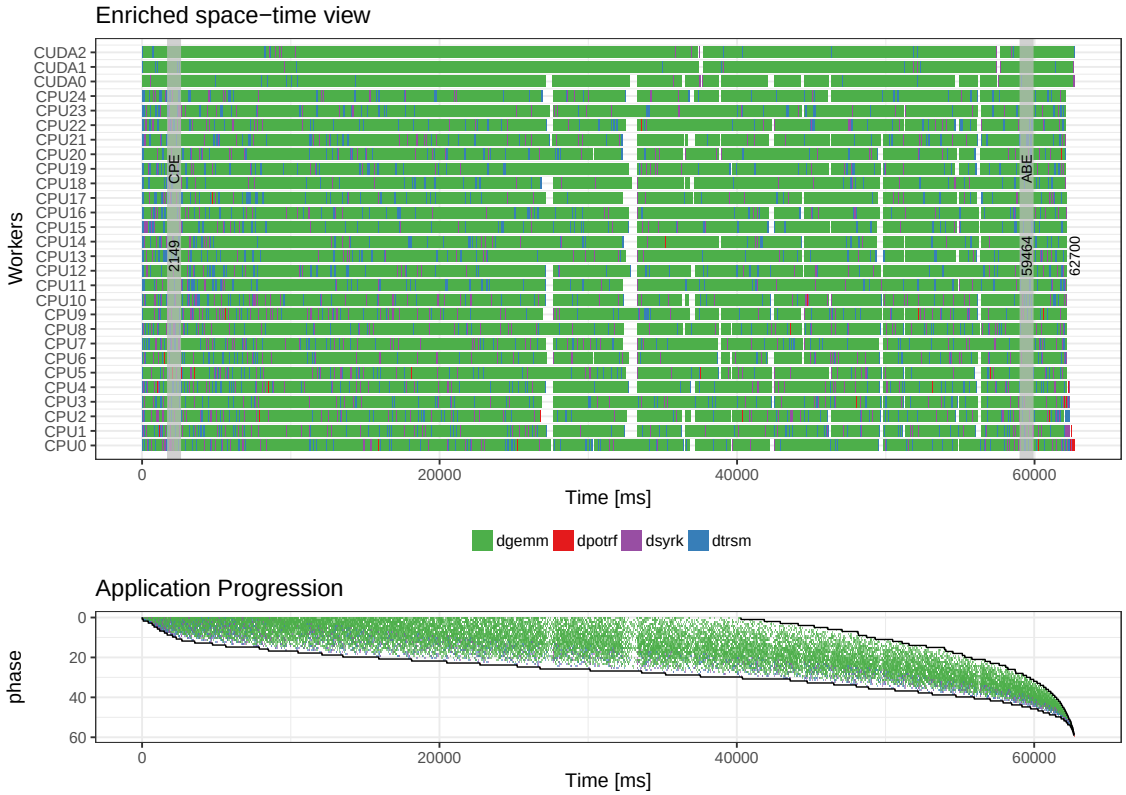
Figure 5.14: Space-time view with application progression.



Source: The Author

tasks inside this interval. In Figure 5.15 the outer curves indicate the begin/end of each iteration while the colorful rectangles between these curves depict the tasks distribution along the iteration duration. Note that the rectangles may be overlapped once several workers might be computing tasks of the same loop-iteration. Similarly, white areas are also possible indicating that there are no workers computing that particular iteration for that specific time interval.

Figure 5.15: Space-time view and the additional progression panel populated with tasks.



Source: The Author

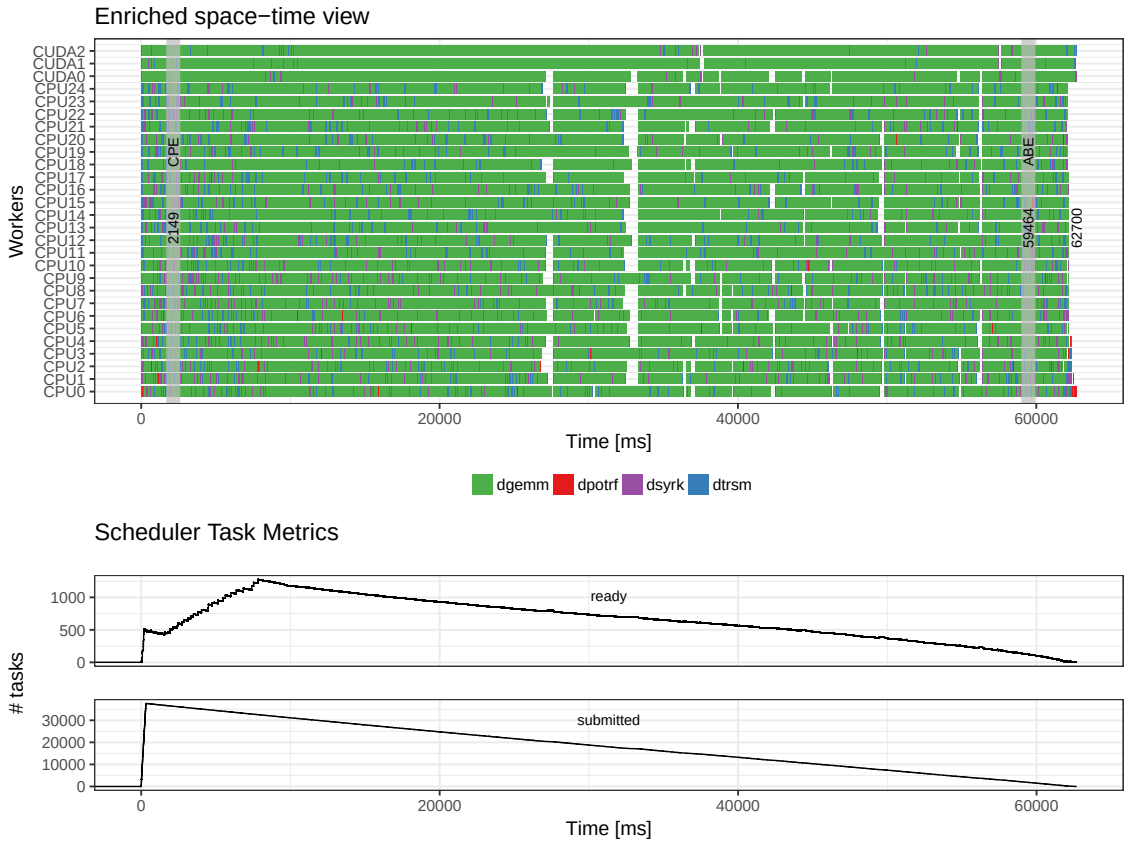
5.2.2 Scheduler Task Metrics

The programmer has no direct control over task availability or synchronizations in task-based applications following the STF model. However, some scheduler task metrics are provided by the runtime system and can be exploited for better understanding the performance. The tasks are created in a sequential way and the runtime system takes care of dependencies and synchronizations. As the execution progresses, the dependencies are solved and the tasks are enabled for execution. For this reason, it is essential to know if

the number of tasks ready for execution at a given moment is sufficient to keep active the computing resources. Some idleness is expected during the beginning and the end of the execution since the DAG is not fully unfolded.

Figure 5.16 shows how this information is attached to the standard space-time view. The two line plots on the bottom of the figure depict the number of submitted and ready tasks along the time. In this example, all the tasks are submitted in the beginning (see the peak in the submitted curve after a few milliseconds of execution). As soon as the initial tasks are executed, the next ones are enabled for execution (growing curve of ready tasks between 1500 and 7800 ms).

Figure 5.16: Space-time view with scheduler metrics. The additional panels (line plots on the bottom) show the number of ready and submitted tasks along the execution.



Source: The Author

5.2.3 ABE Solution

The optimal $makespan$ computed by the Area Bound Estimation (ABE) presented in Section 5.1.3 provides a way to estimate how much further improvement can be ex-

pected. However, from the ABE solution, we can also extract the ideal allocation of each task type on the computing resources. From the equations (5.2)-(5.5) we can verify:

$$\forall r \in R : \qquad \sum_{t} \alpha_{t,r} * n_{t,r} * w_{t,r} \leq makespan \tag{5.6}$$

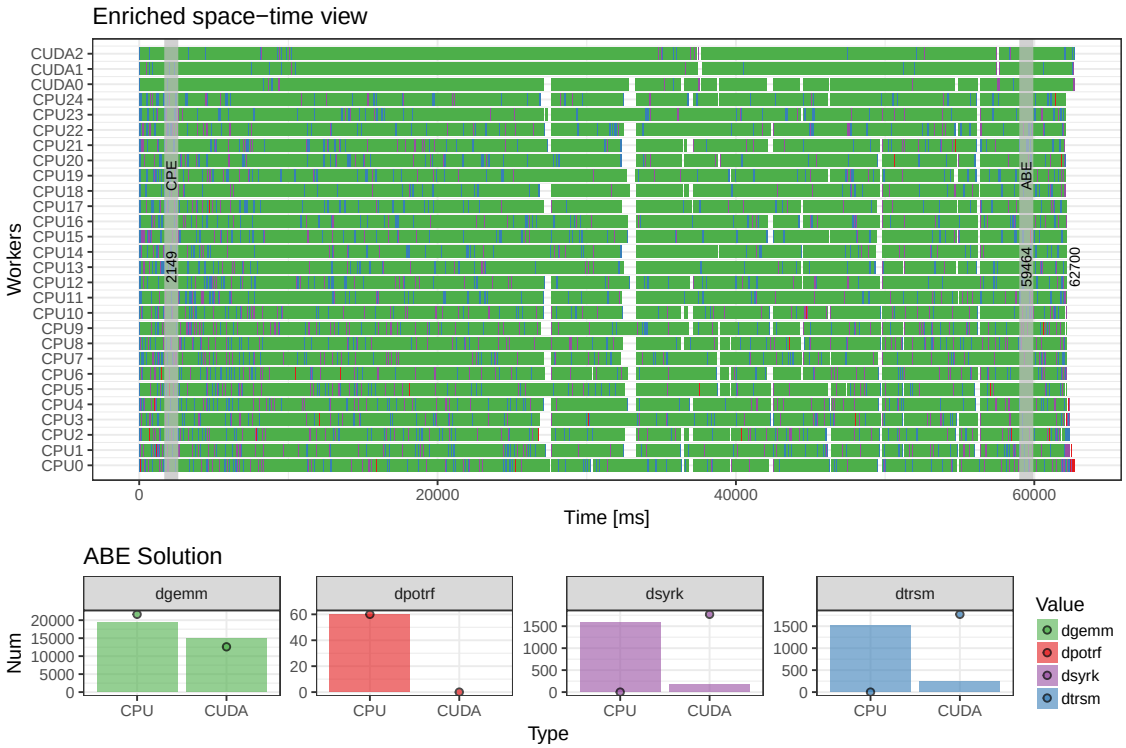
where $\alpha_{t,r}$ is the fraction of tasks of type t that were allocated on a resource of type r. Comparing the $\alpha_{t,r}$ proportion with the actual allocation may help understanding how scheduling could be improved in order to achieve better performance. Figure 5.17 illustrates how the solution of the linear program can be attached to the space-time chart as an additional panel. In this example, the allocation $\alpha_{t,r}$ computed by the linear program indicates that $\approx$63% of the DGEMM tasks should be executed on CPUs (real assignment was 57%) and $\approx$37% on GPUs (real was 43%), for the DTRSM and DSYRK the optimization assigns 100% of the tasks to the GPUs (real was 14% and 10% respectively). This code has no GPU implementation for DPOTRF tasks, so they are always assigned to CPU resources.

5.3 Comparing Views

Comparing the traces from different executions, possibly each one with a different configuration (e.g., scheduling parameter), can be challenging. To draw relevant conclusions, we need to synchronize multiple visualizations panels and filter the unwanted states, which can be difficult when using single instances to visualize each trace. Although some support exists in some tools (PILLET et al., 1995; PAGANO et al., 2013), they do not offer enough customization flexibility for such analysis.

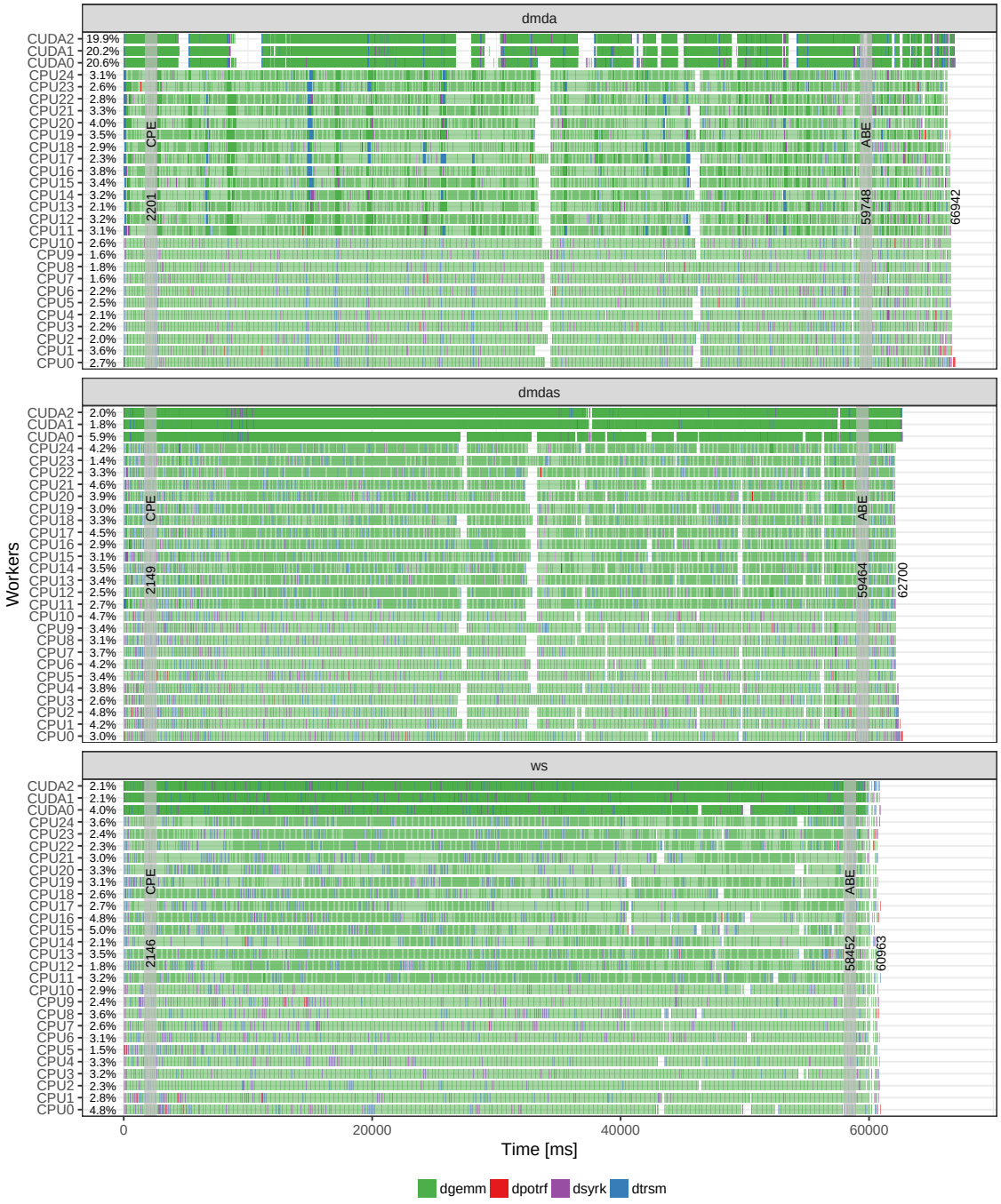
Figure 5.18 presents a comparing view, depicting the space-time visualization for three different executions, each one using a distinct StarPU scheduling policy. Each row on this figure represents one scheduling configuration. The idleness quantification, the outliers detection and the bounds are independently computed for each scenario while the time window is kept synchronized for easy comparison of different settings.

Figure 5.17: Space-time view with CPE and ABE bounds. The additional panel (bar chart on the bottom) shows the optimal task allocation computed by the ABE (number of tasks in the y-axis, task type in the x-axis). This view enables the comparison of the ABE solution (the bullet) with the real execution (the bar).



Source: The Author

Figure 5.18: Comparing views from three executions using different schedulers. Each row represents a different execution. Idle ratio, outliers detection, and bounds for the makespan are computed individually for each execution.



Source: The Author

5.4 Interactive Views

The static views, typically basic X11 window or a PNG/PDF file, of our approach, have disadvantages when compared to some tools described in Chapter 3. Interaction is often useful for the analyst to find what s/he is looking for. This is why we also build on `plotly` (SIEVERT et al., 2017), an online analytics tool, that enables the quick conversion of ggplot2 graphs into HTML/JavaScript interactive, web-embeddable ones usable with a classical web browser[1].

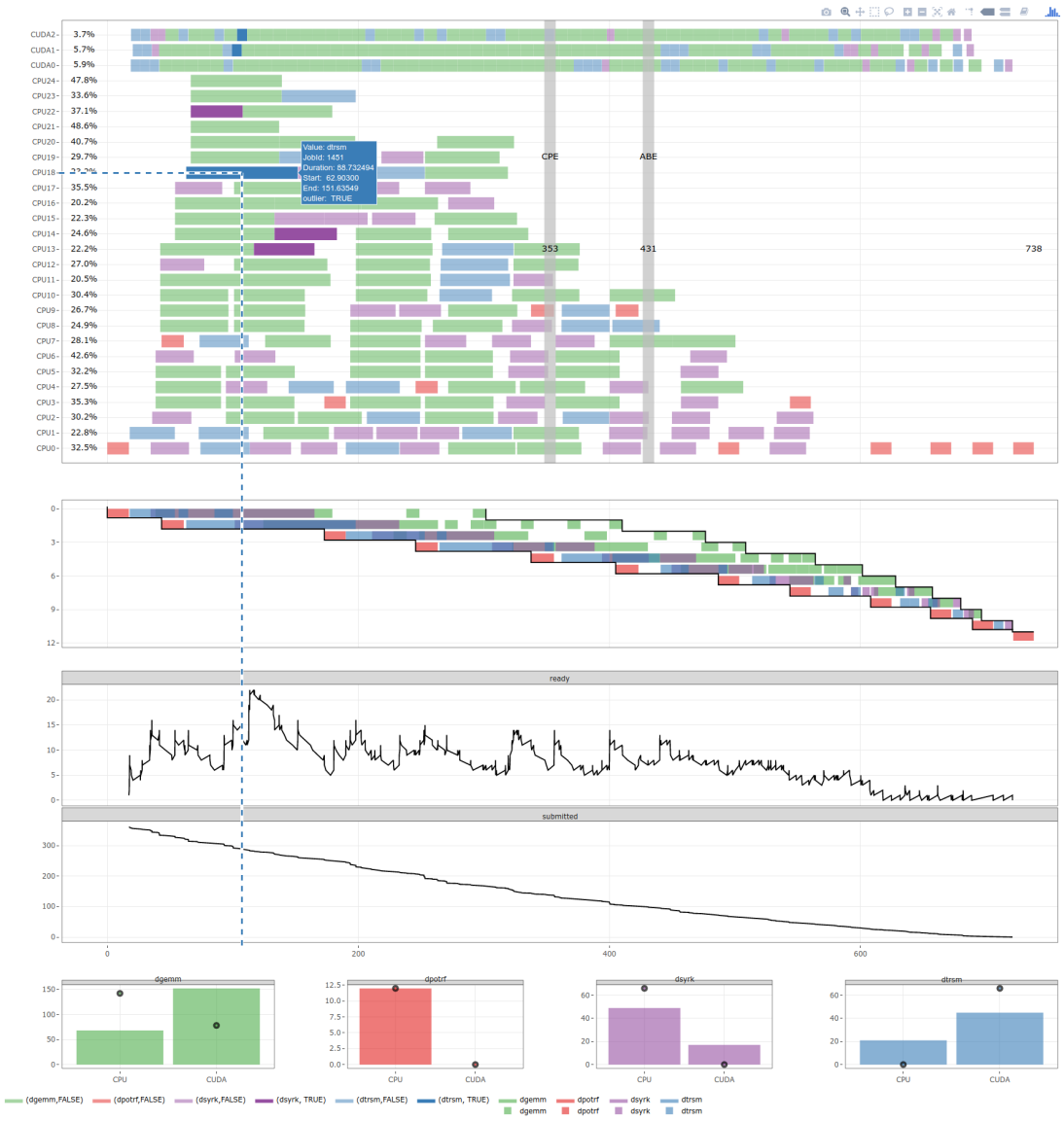
Some illustrations in this report are also available in their interactive version. We believe that putting interaction at the very end together with the scripting capabilities in the core of the analysis process is the key to carrying out the analysis of complex execution traces. Our plotly-based approach is able to show most of the previously discussed views. As an example, Figure 5.19 shows a screenshot of the interactive version of a composite view including the following panels: space-time with outliers highlight and idleness ratio, application progress, scheduler metrics and ABE solution.

Limitations: the `plotly` feature for creating interactive views from static ggplot2 plots is really interesting since we do not need to make big changes in our existing analysis code. At the same time, it enables us to quickly check tasks details such as JobId and the exact duration. However, this solution scales badly in large scenarios, when traces have too much information, with millions of tasks as when there are of many task types or several computing resources. Despite our efforts on data aggregation to reduce the information before creating such graphical objects, the procedure remains usable only for small-scale scenarios. We have also tried a secondary approach using the low-level `plotly` API [2], to produce interactive views without relying on the automatic conversion from ggplot2 objects.

Plotly is incapable of handling too much data, such as when there are many task types or resources. Alternatives to tackle the problem do exist. The `bqplot` or the `googleVis` packages are some of them. The former follows the grammar of graphics philosophy (as ggplot2) for IPython/Jupyter interactive notebooks. The later (googleVis) has functions to generate HTML5 but which are much simpler since they are not layered as we need to create customized views. The `bqplot` library seems more promising because they follow a true Model-View-Controller (MVC) approach, which might scale

[1] See `https://plot.ly/d3-js-for-r-and-shiny-charts/` for more details
[2] `https://plot.ly/r/reference/`

Figure 5.19: Screenshot of an interactive view generated with `plotly`. When the mouse hovers on a task, complementary information is shown, such as its type, ID, Duration, and Start/End values. The visualization of a given task type can be hidden or revealed by clicking on the corresponding legend. The user can also select a region to zoom on it. The corresponding interactive view is available at `http://perf-ev-runtime.gforge.inria.fr/thesis/interactiveView1.html`



Source: The Author

better. We have not yet evaluated any of these alternatives for interactive visualization, adhering with `plotly` for the time being. To minimize its issues, we have reported several bugs to `plotly` developers[3].

5.5 Input Description

The visual performance analysis techniques presented in this work are built from execution traces generated by the StarPU runtime system. The StarPU source code is already instrumented to generate execution traces using the FxT library (DANJEAN; NAMYST; WACRENIER, 2005). This library provides tracing with a limited runtime overhead. After the execution has finished, StarPU generates an FxT binary file with the traces, or per node ones in the case of StarPU-MPI executions. StarPU includes a tool to convert FxT binary traces to Pajé text-based ones (SCHNORR, Lucas Mello et al., 2016).

Assuming a valid Pajé trace, we rely on the `pj_dump` tool from the PajeNG toolkit[4] to convert the trace from the Pajé file format to a basic CSV one. Despite the use of this toolchain based on FxT, StarPU, and Pajé, our approach is not entirely dependent on these tools. Our code expects a basic text-based CSV file describing events and its timestamps as illustrated in Figure 5.21. This input data can be generated from other runtime system or even converted from other trace formats using third-party converters. Each column of Figure 5.21 represents one the required fields: ResourceId indicates the computing unit where an event occurs, Value contains the event name, Start and End indicates when the event has started and when it has finished, JobId is an event unique identifier and Duration depicts the time spent in this event.

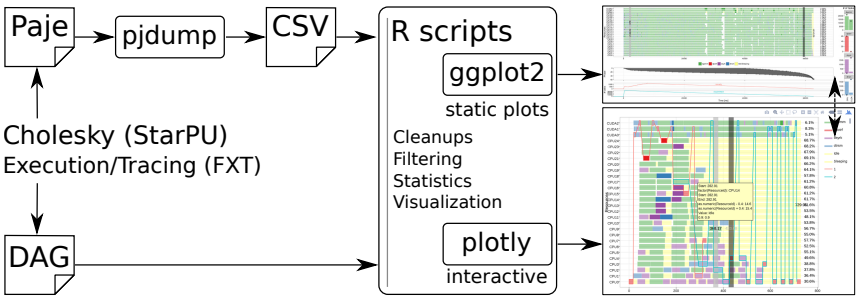
Figure 5.20 shows a simplified representation of our workflow. The left-side represent the steps to be executed in order to obtain the CSV files that will be used by the R code that generates the views. Extending this workflow to support traces from other runtime systems implies to rewrite these initial steps to ensure that the CSV files are properly filled with the data expected by our analysis code.

Several views presented in this Chapter can be built using only a basic trace as the one shown in Figure 5.21. This is the case of *Idleness* (5.1.1), *Outliers* (5.1.2), *Bounds for the makespan* (5.1.3 and 5.2.3), and *Aggregation* (5.1.4). On the other hand, some views

[3] `https://github.com/ropensci/plotly/issues?utf8=%E2%9C%93&q=is%3Aissue+author%3Aviniciusvgp+`

[4] `https://github.com/schnorr/pajeng`

Figure 5.20: A simplified workflow with the steps to generate views from application execution traces.



Source: The Author

Figure 5.21: A fragment of a StarPU execution trace in CSV format after conversion from FxT and Pajé.

	ResourceId	Value	Start	End	JobId	Duration
1:	CPU0	dpotrf	0.00000	17.22442	1347	17.224419
2:	CPU1	dtrsm	17.87772	53.82597	1358	35.948254
3:	CUDA0	dtrsm	18.72615	26.26818	1348	7.542038
4:	CUDA2	dtrsm	18.93579	26.63687	1349	7.701083
5:	CUDA1	dtrsm	20.30117	27.76634	1350	7.465176

360:	CUDA0	dgemm	686.14070	696.44076	1851	10.300062
361:	CPU0	dpotrf	689.27731	705.70472	1857	16.427407
362:	CUDA0	dtrsm	706.39012	713.84567	1858	7.455554
363:	CUDA0	dsyrk	715.15761	721.12105	1861	5.963439
364:	CPU0	dpotrf	721.74629	738.10334	1864	16.357047

Source: The Author

Figure 5.22: Task dependencies information obtained from the application DAG provided by StarPU.

```
      JobId Dependent
   1:  1359      1360
   2:  1360      1347
   3:  1360      1203
   4:  1347      1203
   5:  1372      1373
  ---
2312:  2149      2150
2313:  2152      1858
2314:  2151      2152
2315:  2154      1864
2316:  2153      2154
```

Source: The Author

need some additional data. Tracing the *dependencies* (5.1.5) depends on the application DAG. This information can be provided into the execution trace or in an additional file. In the case of StarPU traces, we retrieve this information from the application DAG (see Figure 5.22) and then, in a second moment, we merge it with the trace dataset. As shown in Figure 5.22, the dependency dataset contains two columns, the event identifier (JobId) and the identifier of an event on which it depends. Events can have more than one dependency, in which case each is described in a different row, e.g., event with JobId 1360 in Figure 5.22.

The basic trace and the additional dependency file can be directly provided by a task-based runtime. However, to create some useful visualizations, we need additional information that can be provided only by the application. This is the case of the *Application progression* view (5.2.1) which relies on the indexes of the loop iteration where the task was created. In the StarPU tracing features, applications can include their specific data in the *Tag* or *iteration* fields. The runtime system can also trace additional data that is not related to specific tasks or the application. This includes internal metrics and statistics such as the ready/submitted values used in the *Scheduler Task Metrics* view (5.2.2).

5.6 Discussion

The visualization strategies presented in this Chapter rely on classical performance analysis instruments such as execution traces and space-time plots. Comparing to the tools discussed in Chapter 3, these strategies present some conceptually similar features but with a strong focus on characteristics of task-based programming. First, we also rely on

space-time (**ST**) views since they are generic enough to be employed in the visualization of task-based applications. In this **ST** plot, we include relevant task dependency edges, which gives some idea about the application structure (**AS**) and at the same time provide some information reduction (**IR**) since we filter only important dependencies. Our approach also provides other two **IR** features: idleness quantification and tasks aggregation. The computation and representation of outliers provide a transparent way to detect anomalies (**AD**) in the task's duration and makes possible to correlate them spacially and temporally. The additional panels, with the temporal-axis aligned with the corresponding one in the **ST** plot, allow us to depict execution metrics (**EMS**) such as the evolution of runtime system variables along the application duration.

Our task-oriented analysis strategies also provide features that are not present in the tools discussed in Chapter 3. The first one is the representation of the application progression which is related to the **AS** but also provides a visual way to check how this structure (i.e., the DAG) is traversed along the time. The second distinct feature is the computation of bounds for the makespan, which provides a way to evaluate executions and also hints about how to improve them. While most of our strategies can work with traces from any application, the application progression view depends on some data from the application. In the case of iterative codes, this can be as simple as tracking the loop-index of each task. In contrast, the bounds are more generic; the only restriction is related to the ABE that works for application with regular tasks, i.e., all tasks of the same type have the same workload.

We believe the proposed visualization strategies can benefit both application and runtime developers since they enable to understand performance issues and then improve the overall performance. Application developers usually need to check how their application behaves on a real platform. In this case, visualization strategies like the space-time plot and the idleness quantification are helpful. Runtime metrics such as the line-plots describing ready and submitted tasks are also useful to confirm that the application parallelism is sufficient to fill the platform. Our strategies can also help application developers when tunning runtime system parameters (e.g., scheduling policy). Using a complete composite view as the one of Figure 5.1 in association with the faceting capabilities as shown in Figure 5.18 helps in understanding and explaining why a certain scheduling policy performs better than others for a given application. On the other hand, runtime developers are looking for issues and mistakes that when fixed, would globally improve the performance of any application running on top of it. An incorrect dependency manage-

ment is most easier to be identified with views as the one of Figure 5.13 than analyzing the raw text log provided by the runtime system. Application progression view and the curve of ready tasks also helps to understand how efficiently the runtime is unrolling the DAG of tasks.

5.7 Chapter Summary

In this Chapter, we present visualization strategies designed to meet the requirements of task-based applications running over hybrid platforms. These strategies get inspiration from classical performance analysis instruments such as execution traces and space-time views.

Our approach is built on top of modern data analytic tools combining `pj_dump`, Org-mode and the *R* programming language, including packages such as *ggplot2*, *lpSolve*, *tidyverse* and *plotly* libraries. The proposed strategies are based on enrichments to the standard space-time view to highlight application and platform characteristics and on synchronized additional panels that disclose more details about the application and runtime internal information.

The next Chapter will present case studies to demonstrate how these strategies can be employed to analyze task-based applications and improve their performance.

6 RESULTS ON VISUALIZATIONS STRATEGIES

In this Chapter, we present how our visualization techniques can be combined together to attain a successful performance analysis. This analysis focus on task-based executions running on the two hybrid platforms described in Section 6.1.1. We present two case studies (Sections 6.2 and 6.3) based on tiled Cholesky decomposition presented in Section 6.1.2.

6.1 Experimental Setup

Each scenario analyzed in this Chapter was executed multiple times to ensure that our observations are reproducible. Although the observed makespans slightly differ, the general behavior and conclusions are the same. General details about the platform and the target application are presented in Sections 6.1.1 and 6.1.2. Low-level hardware and software details, such as cache hierarchy, processor status, linked-libraries or OS configuration can be found in the experiments log files presented in section 4.2 and which are available in the data repository.

6.1.1 Platforms

The case studies presented in this chapter were executed in two heterogeneous platforms. The first one, **idcin2** is composed of a single node with 28 CPU cores and 3 GPU cards. Since StarPU needs one CPU core to handle each GPU, only 25 participate in the computation, totalizing 28 computing units (25 cores + 3 GPUs). The second experimental platform is the cluster **chifflet** from the Grid'5000 site at Lille, which is a distributed platform with nodes composed of 28 CPU cores and 2 GPUs. Similar to **idcin2**, in **chifflet** nodes, only up-to 25 CPU cores are used to compute tasks. As discussed before, two cores are reserved to handle the GPUs, and another one is used to handle the MPI messages exchanges. Table 6.1 shows a summary of the hardware/software configuration of both platforms. Additional information, such as the real used source code, linked libraries, hwloc platform description files, and the StarPU performance and bandwidth models, can be found in the execution log file presented in Section 4.2 of Chapter 4.

Table 6.1: Summary of the hardware/software configuration of the experimental platforms used to collect execution traces.

	idcin2	**chifflet**
Nodes	1	8
Processors per Node	2	2
Processor	Xeon E5-2697v3 2.60GHz	Xeon E5-2680v4 2.40GHz
Cores per Processor	14	14
Core count	28	224
Memory	256GB	756GB
GPUs per Node	3	2
GPU card	GeForce GTX TITAN X	GeForce GTX 1080 Ti
GPU cores (per card)	3072	3584
GPU core count	9216	7168
GPU memory (per card)	12GB	11GB
Interconnection	-	Ethernet 10GbE
Linux kernel	3.2.0-4-amd64	4.9.0-3-amd64
Chameleon	0.9.1 (master)	0.9.1 (master)
StarPU	1.3 (trunk)	1.3 (trunk)
OpenMPI	-	2.0.2
BLAS	Eigen BLAS 3.3	OpenBLAS 0.2.19
CUDA version	7.5	7.0
CUDA Driver	352.39	375.66
GCC	4.7	6.3.0

Source: The Author

6.1.2 Application

In the context of task-based applications, the overall performance is intrinsically related to the efficiency of the runtime system. For that reason, focusing our analysis on the runtime system can help us to identify important issues and mistakes that impact the overall performance of the application. However, to study the runtime system performance, we should rely on a representative and already well-optimized application. Using a non-optimized application can hide runtime system performance issues while a non-representative one could lead us to problems and mistakes that have no significant influence on the performance of commonly-executed applications.

For these reasons, we choose a Cholesky decomposition as our case study application. This factorization is one of the most common linear algebra operations and is used by many scientific applications. The Cholesky method consists on decompose a matrix A into a product of a matrix L and its transpose L^T (DATTA, 2010; LEON, 2014). The factor matrices L and L^T are (respectively) lower and upper triangular with positive diagonal elements:

$$A = LL^t \iff$$

$$\begin{pmatrix} a_{11} & a_{21} & a_{31} & \cdots & a_{i1} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{i2} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{i3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \cdots & a_{ii} \end{pmatrix} = \begin{pmatrix} l_{11} & & & & \\ l_{21} & l_{22} & & & \\ l_{31} & l_{32} & l_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ l_{i1} & l_{i2} & l_{i3} & \cdots & l_{ii} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} & \cdots & l_{i1} \\ & l_{22} & l_{32} & \cdots & l_{i2} \\ & & l_{33} & \cdots & l_{i3} \\ & & & \ddots & \vdots \\ & & & & l_{ii} \end{pmatrix}$$

The Cholesky decomposition is applicable for symmetric positive definite matrices, in which case it is 2 times more efficient than the LU decomposition when solving systems of linear equations (PRESS et al., 2007; DATTA, 2010).

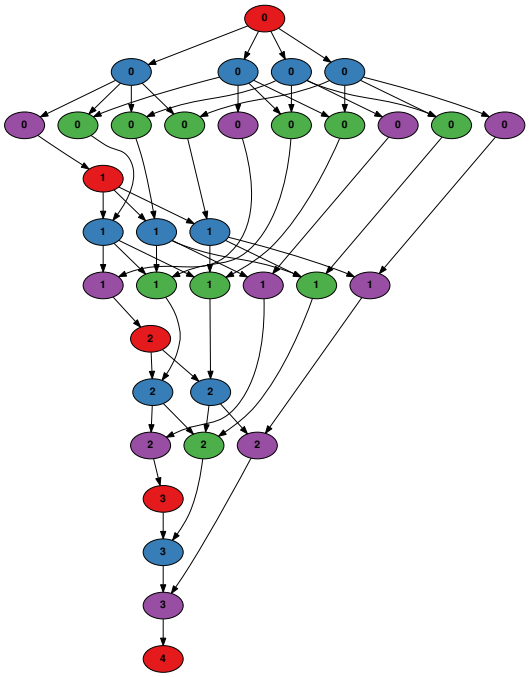
In order to improve the reproducibility and the stability of our tests, we adopt the tiled Cholesky implementation provided by the Chameleon solver. This implementation is built on top of the StarPU runtime system (AUGONNET, Cédric et al., 2011) and compiled with standard BLAS (CPU) and CUBLAS (GPU) libraries. Figure 6.1 shows a simplified version of this application. The lines with calls to DPOTRF, DTRSM, DSYRK, and DGEMM (Figure 6.1(a)) represent the creation of StarPU tasks with double-precision implementations for CPUs and GPUs (except the DPOTRF that has only the CPU version). The underlined labels RW and R indicate the access mode of the subsequent matrix block.

Figure 6.1: The pseudo-code of the tiled Cholesky decomposition and its corresponding DAG for $N = 5$.

(a) Code snippet of the tiled Cholesky decomposition.

```
for (k = 0; k < N; k++) {
  DPOTRF(RW, A[k][k]);
  for (i = k+1; i < N; i++)
    DTRSM(RW, A[i][k], R, A[k][k]);
  for (i = k+1; i < N; i++) {
    DSYRK(RW, A[i][i], R, A[i][k]);
    for (j = k+1; j < i; j++)
      DGEMM(RW, A[i][j], R, A[i][k],
                        R, A[j][k]);
  }
}
```

(b) Cholesky DAG for $N = 5$. The number inside each node represents the k-index of the outer loop.



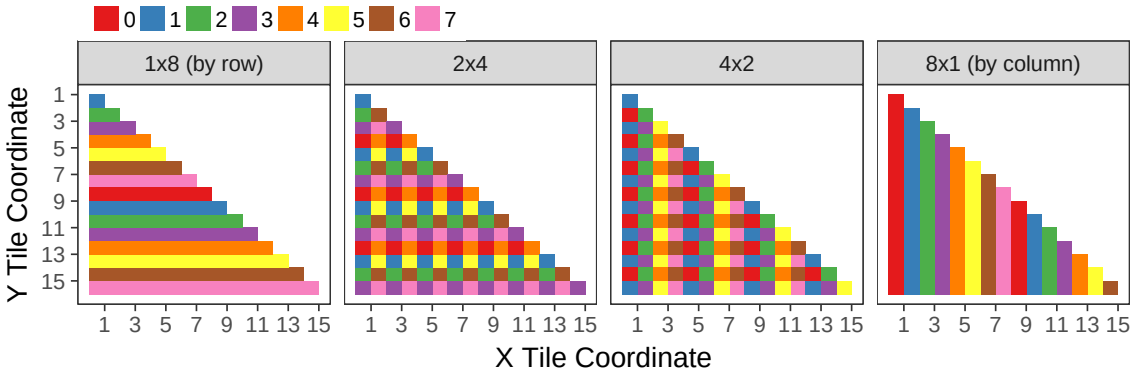
Source: The Author

From these access mode hints, the runtime can infer the dependencies and then build the Directed Acyclic Graph (DAG) of tasks. Figure 6.1(b) shows the corresponding DAG for a 5×5 matrix. In each iteration k of the outer loop, one DPOTRF task enables the execution of $N - k - 1$ DTRSM, then $N - k - 1$ DSYRK tasks, followed by $\approx (N - k)^2/2$ DGEMM tasks. From the dependencies, one can observe that several iterations can be executed simultaneously and that the number of repetitions in the internal loops decreases at the same time as k increases. Finally, the execution time of a task highly depends on its type (DPOTRF, DTRSM, DSYRK, and DGEMM) and on the target resource (CPU or GPU). Note that the color scheme used in this Figure to represent the task types is respected in all the following graphics of this document.

In multi-node executions as the ones of Section 6.3, we use a StarPU-MPI implementation of the Cholesky decomposition. Since in StarPU-MPI executions the domain decomposition is static, the application includes two additional parameters to allow the user to control how the input data will be distributed between the nodes. For this, the classical Two-dimensional Block-Cyclic Distribution (BLACKFORD et al., 1997) has been

previously modified to support multi-node runs with static decomposition under the auspices of StarPU-MPI. The decomposition depends on the P × Q parameter and the number of MPI nodes, governing how the input matrix is partitioned among nodes on a per-tile basis. The value of P can range from one to the number of nodes. Figure 6.2 depicts the four possible situation cases for a Cholesky factorization with eight nodes and a matrix with 16 × 16 tiles: the data decomposition shown in the left facet is obtained when P=1 × Q=8 and leads to a row based distribution of tiles (one color per node); for P=2 × Q=4 and P=4 × Q=2, shown in the center left and right facets, the data distribution is interleaved; finally, when P=8 × Q=1, data distribution is by column as shown by the right facet. In an ideal scenario, for a given number of nodes, the value of P should be defined so as to minimize the communication perimeter of each node as it is related to the total volume of communications.

Figure 6.2: Different static partitioning schemes for DTRSM tasks as dictated by the P parameter when eight nodes are used to run Cholesky: P=1 (left, by row), P=2 (center left), P=4 (center right), and P=8 (right, by column).



Source: (PINTO, Vinícius Garcia; SCHNORR, Lucas Mello; STANISIC; LEGRAND; THIBAULT; DANJEAN, n.d.)

6.2 Case Study: Changing Schedulers on Hybrid Nodes

In task-based programming, the runtime system and its scheduler strategies play a crucial role in the application performance. The makespan may vary significantly depending on the chosen scheduling policy even when using the same tasks implementations and executing on the same hardware. These variations are difficult to explain and understand by regarding only the execution time.

As discussed in Section 2.3.4, StarPU offers several scheduling policies. Some

of them are incrementally built on top of other ones, e.g., DMDA and DMDAS, but, in practice, they present slightly different performance. For that reason, we propose a visual analysis to try to explain the different behavior of three StarPU schedulers: DMDA, DMDAS, and WS.

Another configuration that impacts the execution behavior is the size of the workload, i.e., the DAG size in task-based applications. On one hand, large DAGs are expected to be embarrassingly parallel, almost reaching peak performance. Since such scenario comprises hundreds of thousands of tasks, we need to use macroscopic views and indicators to understand whether and how performance can be improved. On the other hand, small DAGs have little parallelism and idle time will inevitably be incurred by task dependencies. For such executions, microscopic views with fine-grained information on task dependencies should rather be used.

The analysis of this case study was carried out with two representative workloads for the Cholesky factorization. As summarized in Table 6.2, the first one uses large matrices of 60×60 tiles of 960×960, while the second one uses smaller matrices of 12×12 tiles of size 960×960.

Table 6.2: Summary of workloads

Workload	***Large (L)***	***Small (S)***
Matrix Size	57600 × 57600	11520 × 11520
Number of Tiles	60 × 60	12 × 12
Tile Size	960 × 960	960 × 960

Source: The Author

6.2.1 Workload *L* - Cholesky Factorization of 60×60 tiles of size 960×960

Several assumptions can be made when executing large regular DAGs such as the one resulting from the Cholesky algorithm on a 60×60 tiles matrix. We detail below the expectations regarding uniformity, task dependencies issues, application progress, and possible improvements.

- **Uniformity.** Task duration is expected to depend solely on their type (DGEMM, DSYRK, DTRSM or DPOTRF) and on the type of resource (CPU or GPU) on which it is executed. Such assumption should be visually verified, highlighting all tasks whose duration is abnormally large compared to the others of the same type/resource. We treat these tasks as independent outliers, expecting their space/time

location is unrelated to other tasks behavior. If that is not the case, it may mean that the whole platform has been perturbed at particular moments or that the performance of a given resource differs from the others of the same type. To check this uniformity expectation we rely on the enriched space-time view with outliers detection presented in Section 5.1.2.

- **Dependencies management.** Large input matrices generate many tasks, especially when the application starts. We, want to monitor the number of ready and submitted tasks, which can be done using the scheduler task metrics view presented in Section 5.2.2. For this Cholesky implementation, all tasks are expected to be submitted when the application starts. On scale, the number of task dependencies is extremely large. Automatically selecting which ones to display is haphazard. If a detailed view becomes necessary for some task dependencies, we rely on the scripting capability of our framework to select and visualize the offending task dependencies from the performance point of view. These meaningful dependencies can be visualized enabling dependencies backtracking feature discussed in Section 5.1.5. A common way to find problematic task dependencies is to select tasks in front of idle time because there should not be an idle time whenever there is enough parallelism.

- **Application progress.** The task graph resulting from dense linear algebra always share a common structure (see Figure 6.1). In a classical semi-sequential execution, the DAG would be executed much similar to a *breadth*-first search. However, it is also possible to carry out a *depth*-first traversal, favoring task execution on the critical path. This behavior can be visually checked using the application progression view presented in Section 5.2.1. Such view represents the pipelining of the sets of tasks submitted by each outer loop iteration, which can be sufficient to get an overview of how the scheduler is handling the DAG.

- **Idleness quantification.** Displaying information on hundreds of thousands of tasks on a small area in a blunt way generally leads to harmful visualization artifacts (SCHNORR, Lucas M.; LEGRAND, 2013). For example, in a classical space-time chart, visually estimating how much time was spent idle can be quite difficult. This is why it is generally important to aggregate and quantify it in a meaningful and non-ambiguous way as discussed in Section 5.1.1.

- **Potential improvements.** Dependencies are expected to be easily handled with large workloads. In this scenario, the major issue is the load balancing among

CPUs and GPUs. To check whether improvements are still possible for a given execution, we might rely on the classical scheduling bounds, such as the *area bound* and the *critical path bound* discussed in Section 5.1.3. Such bounds, especially the area bound, are expected to be tight when the workload is large and allow to estimate how much further improvement can be expected. Furthermore, an ideal task allocation can sometimes be inferred from such bounds, which may allow helping understanding how scheduling could be improved as showed in Section 5.2.3.

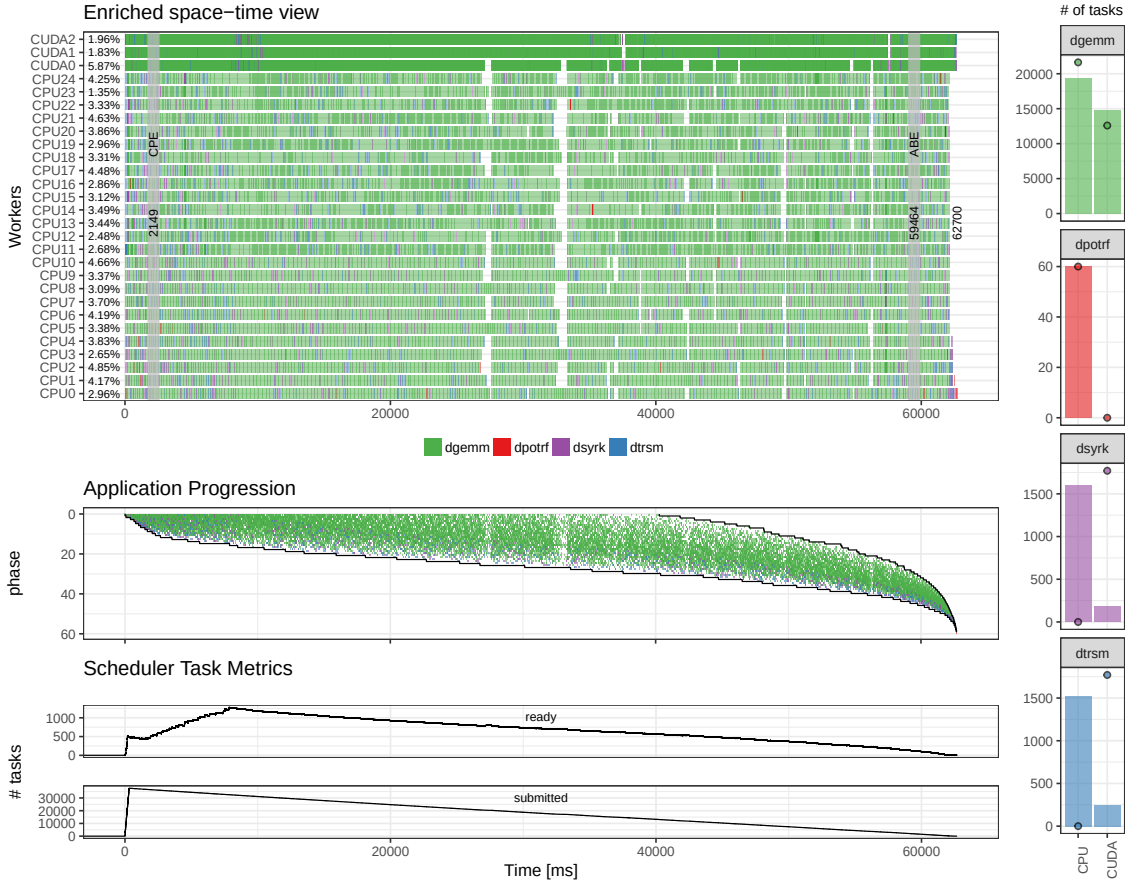
Building on these expectations, we propose an *enriched composite view* that combines a classical space-time view with several techniques presented in Chapter 5. Figure 6.3 shows the behavior of a Cholesky execution with the DMDAS scheduler on idcin2. This view is composed of four panels: the *Enriched space-time view* (Figure 6.3, top) depicts the application states during the time augmented with outliers highlighting, idleness quantification and the bounds for the makespan, the *Application progression* panel (middle) provides insights about how the application DAG is traversed, the *Scheduler Task Metrics* (bottom) shows the number of submitted and ready tasks along the execution, and the ABE solution (right) compares the real execution with the ideal partition found by the ABE.

A first look on the *Enriched space-time view* (top) allows us to deduce that there is room for improvements since the observed makespan is 62700 ms while the ABE is 59464 ms, which means a difference of 5%. The scheduling seems indeed inefficient since there are periods where any useful computation is performed. The total idleness[1] for CPUs varies from 3 to 6%, while for GPUs it ranges from 2 to 6%. This GPU inactivity is likely the main source of potential improvement, even if the idleness rate is similar to that shown on CPUs since a GPU delivers a higher GFlops rate.

By analyzing the values of ready tasks in the *Scheduler Task Metrics* panel (Figure 6.3, bottom), we can verify that this idle time does not come from a sudden lack of ready tasks. The submitted curve clearly indicates that all tasks have been submitted in the beginning and that task execution started immediately after, without waiting for fully unrolling the DAG. As suggested in the *Application Progression* view (Figure 6.3, middle), DAG traversal is rather depth-first. Many outer loop iterations are parallel (the maximum is 30 around 40000 ms of execution), explaining why there is always a sufficient number of ready tasks.

[1] As discussed in Section 5.1.1, the total idleness comprises all non-computing states including not only the "Idle" and "Sleeping" states reported by StarPU but also the scheduling related ones, e.g., Initialization and Data Prefetching.

Figure 6.3: Composite View of a Cholesky factorization with a large matrix (60×60 tiles of 960×960) executed with the DMDAS scheduler. The first panel (top) depicts a classical space-time view with workers on y-axis and tasks duration along x-axis. This basic view is enriched with outliers highlighting, idleness quantification and the makespan bounds (ABE and CPE). The second panel (middle) shows the Application Progression, which in this Cholesky example is given by the timestamp of the iterations of the outer-loop. The third panel (bottom) depicts two metrics provided by the StarPU scheduler to show the amount of submitted and ready tasks along the time. The last panel (vertical bars on the right) shows a comparison between the ideal allocation obtained in the ABE calculation (bullets) and the real allocation (bars).



Source: The Author

Since resource idleness is not a consequence of a lack of ready tasks, we can suppose that such starvation is more likely explained either by data prefetching problem (some tasks are ready but their input data is not yet transferred) or possibly by some priority problem (the priorities, used by the scheduler to choose which task to schedule first when several of them are ready, might be inadequate). The first explanation is likely to be the right one here. Indeed, most large idle periods where some workers are not doing useful computations also coincide with abnormally long DGEMM tasks on GPUs. A further investigation shows that these idle periods correspond to filtered scheduling states (not shown for clarity) where workers try to actively fetch data. For an unknown reason, the GPUs seem to freeze during a task execution inside the proprietary CUBLAS DGEMM kernel, ultimately blocking other tasks eagerly waiting for GPU data. Understanding why GPUs sometimes get stuck would certainly solve the issue[2] but, on the other hand, this fact clearly suggests a weakness of the chosen scheduler which assumes that tasks duration have small variability. Using **other schedulers** may, therefore, alleviate this.

The four-bar plots of the *ABE solution* panel (Figure 6.3, right) show the ideal allocation when calculating the ABE. They show how the GPUs have been overused with DGEMM tasks and under-exploited for DSYRK and DTRSM tasks. It, therefore, suggests **constraining** the DSYRK and DTRSM tasks to run exclusively on GPUs. Performance gains when constraining some tasks to GPUs were already reported in the literature by Lima et al. (2015). However, their results were achieved using scheduler hints provided by programmer annotations. In our case, the suggestion of when and which tasks to constrain to GPUs is inferred from the solution of the ABE without relying on programmer's knowledge about task's architecture affinity.

Our previous analysis based on the *composite view* gave us hints on which changes could be done to improve the performance. First, we have extended our experiments range by including DMDA and WS schedulers which now allow us to analyze executions varying scheduling policies (DMDAS, DMDA, and WS). The second hypothesis has given by *ABE solution* view and suggests constraining some tasks to execute only on GPUs. We have modified the application code to force the allocation of DSYRK and DTRSM tasks to GPUs.

[2]Since the CUDA and CUBLAS are closed-source tools, we cannot check the DGEMM kernel and the transferring implementations. To try to understand this behavior we have based on StarPU-Simgrid executions, using the same StarPU code and the same performance models, to artificially slow-down some DGEMM tasks on GPUs. Despite these efforts, we were not able to reproduce this behavior. We believe that it could be related to a specific configuration of the idcin2 platform (driver, CUDA and GPU cards) since this issue is not reproducible on other platforms and idcin2 is no longer available.

These changes lead us to the six-scenario comparison of Figure 6.4. This figure represents six different executions varying the schedulers (columns) and the code (rows). The previous used DMDAS scheduler (center) is compared with the DMDA (left) and the WS (right). The three executions of the first row (Unconstrained) were performed using the original code, which means the DMDAS one is exactly the same one presented in Figure 6.3. The executions on the second row (Constrained) were done using the modified code where DSYRK/DTRSM tasks can only be executed on GPUs. In each scenario, the space-time view is complemented by two additional panels depicting the *Application Progression* and the ABE solution.

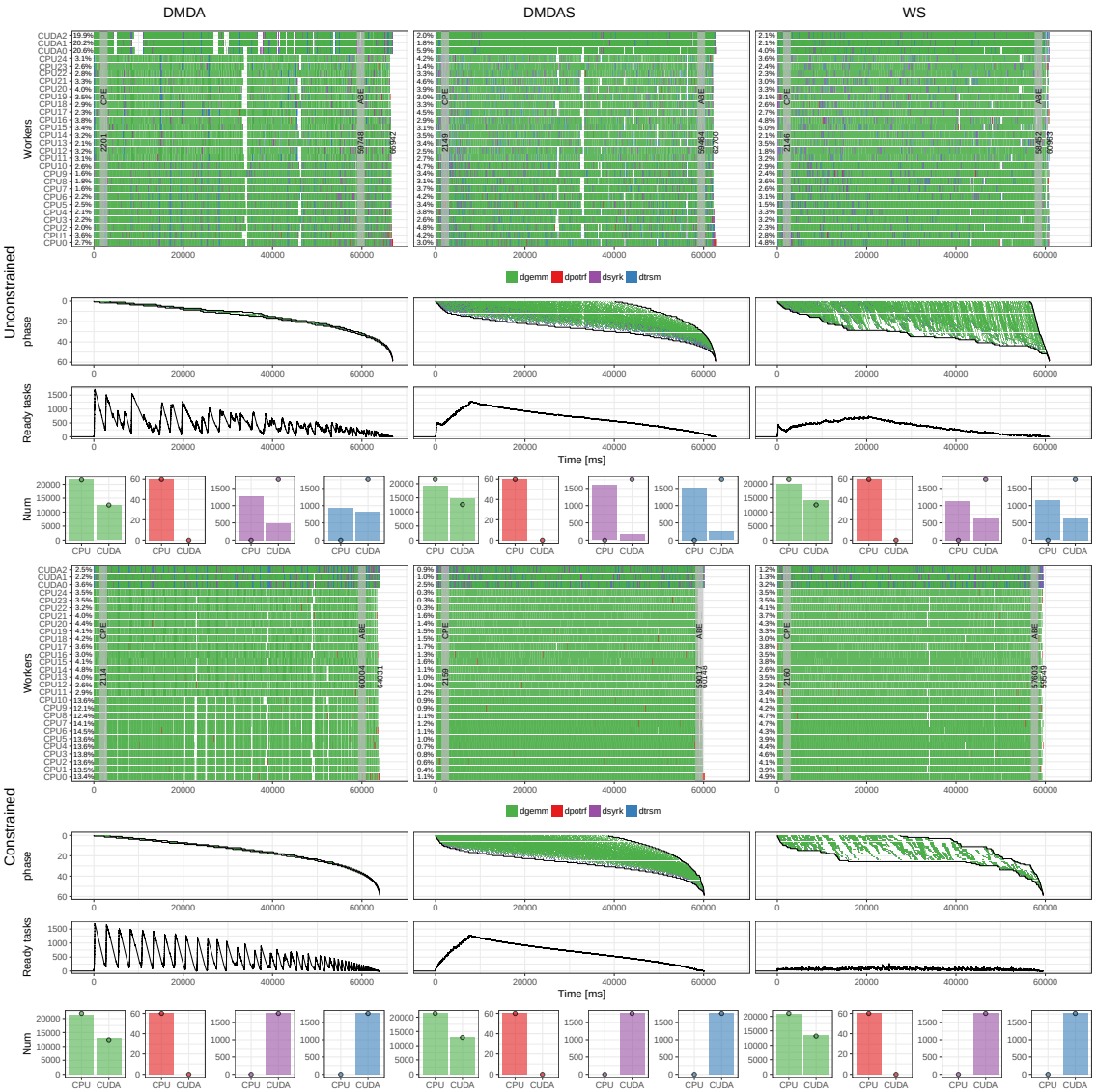
First of all, regarding the *Application progression* panels, it is interesting to see how these three schedulers differ in their traversal of the DAG. While the DMDA algorithm (left) has a breadth-first traversal (very few iterations of the Cholesky outer loop are active at the same time), the DMDAS (center) has a much more depth-first traversal as it takes the priority of the critical path into account. The traversal of the Work Stealing (WS) is even more depth-first as almost all outer loop iterations are still in progress at the end of the execution. Such way of progressing through the DAG is typical of WS and favors local data accesses even though the algorithm does not rely on data transferring modeling used by DMDA and DMDAS.

Second, when constraining the DSYRK and DTRSM tasks to run only on the GPUs (Figure 6.4 bottom row), task allocation of these two task types then corresponds to the ideal one. However, if such constraint allows both DMDAS and Work Stealing to obtain near-optimal executions (within less than 2% of the lower bound as given by the ABE), this helped only moderately the DMDA algorithm. Many synchronized idle periods can be observed and imputed to both dependency issues (not enough parallelism is obtained from such a strict breadth-first traversal) and particularly slow tasks (probably slowed down by simultaneous data transfers). The *Ready tasks* curve of this schedulers is quite unstable and shows some valleys where possibly there is a lack of ready tasks to fill the workers. Interestingly, very few outlier tasks appear in the DMDAS and WS executions although the latter still seems a bit sensitive to this, as inactivity periods on CPUs (white areas) still correlate with the occurrence of DGEMM outliers (darker green) on GPUs.

Finally, we want to stress that such observations are no coincidence. We randomly ran similar scenarios ten times and although the numbers always slightly differ, the general behavior and conclusions are similar.

We also want to highlight the fact that the area bound estimations (ABE) can

Figure 6.4: Comparison view of six executions of the Cholesky factorization. Executions differs in the used schedulers (columns) and the source code (rows). The first row shows executions with the original source code (Unconstrained) using three different schedulers (DMDA, DMDAS and WS), which means the DMDAS one is exactly the same presented on Figure 6.3. The second row shows executions with the modified source code where DTRSM and DSYRK tasks are constrained on GPUs. The additional panels (Application Progression, Scheduler Metrics and ABE solution) are similar to the ones presented on Figure 6.3.



Source: The Author

vary significantly between the two scenarios (e.g., 60004ms for constrained DMDA vs. 57603ms for constrained Work Stealing), which can be initially surprising since these estimates only depend on the total number of tasks and on the average execution time of each type of tasks on each different computing resource. This can, however, be explained by the fact that we use the sample mean of execution time, which may vary a bit. From our investigation, this variation is not really explained by the occurrence of outliers but rather biased toward one or another scheduler. We believe that this is the consequence of a better locality (hence cache usage) but more complex measurements would be needed to fully evaluate this hypothesis.

6.2.2 Workload *S* - Cholesky Factorization of 12×12 tiles of size 960×960

The assumptions that can be made when executing smaller DAGs are much more related to task dependencies. Unfortunately, even in very small instances, the number of dependencies is very large and displaying all them quickly leads to cluttered visualizations. We detail below the expectations regarding potential improvements, uniformity, and idleness.

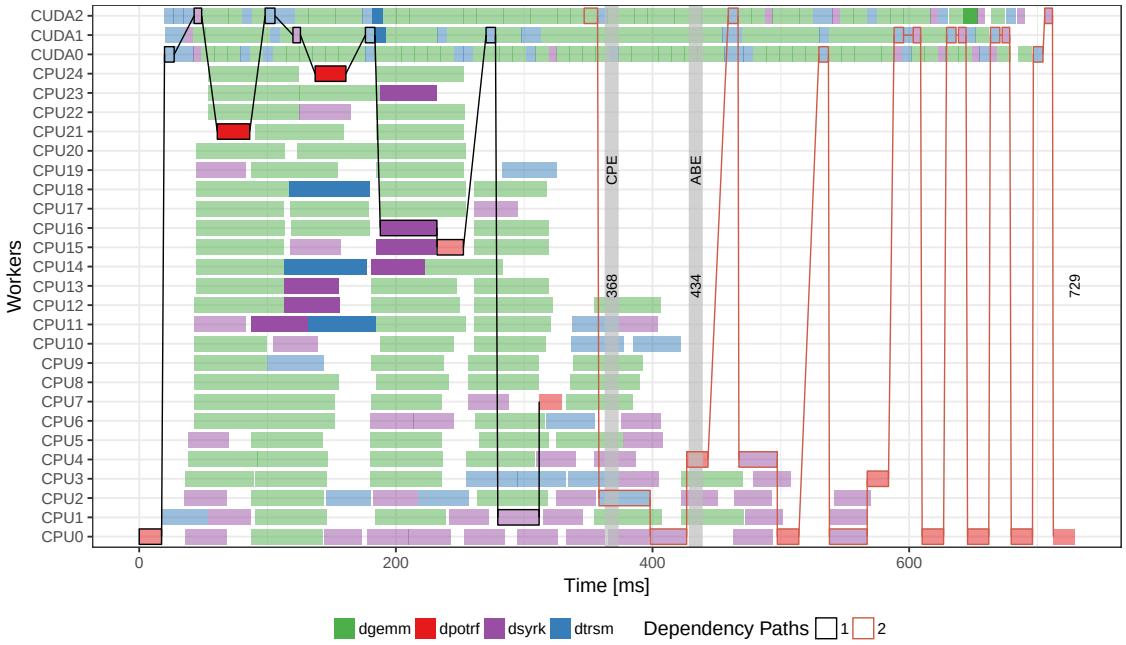
- **Potential improvements.** On small workloads, the area bound (ABE), which ignores task dependencies, is known to be too optimistic. The critical path bound (CPE) is expected to be much more relevant, especially on very small workloads such as the one analyzed here. Still, knowing how tight they are is quite difficult (AGULLO, E.; BEAUMONT, et al., 2015). For this reason, comparing to the ideal allocation (CPU vs. GPU) computed by the ABE is meaningless and we focus therefore mainly on computing and filtering task dependency chains.
- **Uniformity.** Even with few tasks, uniformity on tasks duration is still expected, then highlighting outlier tasks can, therefore, be useful.
- **Idleness**. With a reduced amount of tasks, having a lot of idle time is expected because of dependencies. Regardless this predominant and clearly identifiable idleness, we keep the quantification of idle periods since it helps to understand why some schedulers perform better than others. On the other hand, inspect individual dependencies can give more precise suggestions on performance issues. It is thus important to identify relevant (either because of the DAG structure or because they appear to have been particularly delayed) tasks of the DAG, to backtrack their de-

pendencies and highlight the dynamic critical path, i.e., the last tasks upon which they depended on.

From such expectations, we propose a space-time view enriched with dependencies to pinpoint scheduling mistakes. As previously discussed in Section 5.1.5, task dependencies information can be merged into the execution trace. For a Cholesky execution, the DPOTRF tasks are good candidates to start the investigation of the dependency chain since they release many other tasks. For this reason, in Figure 6.5 we draw the dependency chains starting from each DPOTRF.

The execution illustrated in Figure 6.5 was completed in 729 ms while the ABE is 434 ms and the CPE is 368 ms. The bounds may be loose, but it seems that there is room for improvements. If we start from the end of the schedule and go backward in time, we can see a dependency path (in red) that, until timestamp 400ms, fully respects the alternation DPOTRF–DSYRK–DTRSM. At the very end, all tasks execute right one after the other, which is optimal.

Figure 6.5: Space-time view of a Cholesky factorization with a small matrix (12×12 tiles of 960×960) executed with the DMDAS scheduler. Black and red lines connecting tasks depict the merged dependency critical paths of DPOTRF tasks. Interactive version available at `http://perf-ev-runtime.gforge.inria.fr/thesis/vpacasestudyfig3-interactiveView.html`



Source: The Author

A potential "mistake" appears in time ≈600ms where the DSYRK could have been

executed a little earlier. Slightly before, some DTRSM are not executed right after their DPOTRF maybe because of data transfer or more likely because of a wrong priority. This critical path (red) does not merge with the one obtained for the DPOTRF of the first iterations (black one). Now, when looking at the other (black) dependency path, we can see many times that the tasks are scheduled as soon as possible as if there was some priority problem, which could possibly be solved with **another scheduler**. The scripting feature allows plotting only dependencies whose duration is larger than a given threshold, avoiding graphical clutter.

At the end of the execution, it is possible to identify another problem following the second dependency path (red). Coming back in time from the end to the beginning of the execution, we can verify that tasks are executed on their preferred resource (DPOTRF on CPUs, DSYRK and DTRSM on GPUs). However, slightly before time 600 ms (≈537 ms), critical DSYRK tasks start running on the CPUs, slowing the progression towards the end. Likewise, slightly before time 400 ms (≈358 ms), critical DTRSM tasks are executed on the CPUs whereas they are known to be less efficient on such resources. It seems that this scheduler makes a bad decision and that **constraining** DTRSM and DSYRK to be executed on GPUs may reduce the total makespan. It is important to mention that although the solutions (fix priorities evaluation by changing the scheduler, and constraining DTRSM/DSYRK tasks to GPUs) suggested by our analysis are the same as in the previous use case (Section 6.2.1), the underlying reasons are fundamentally different.

Based on the previous analysis, we have therefore decided to vary again the three schedulers and forcing DSYRK and DTRSM tasks on GPUs. Figure 6.6 compares the six resulting execution using this workload. Comparing the original unconstrained executions (first row on top), we can observe that the behavior demonstrated by the DMDA and the DMDAS are not so different. They both present similar workload partition between CPUs and GPUs, keeping the last ones with occupation ratios about 90% or more. They both have similar runtimes, with unmerged critical paths on which priority and critical task allocation problems can be identified. The small performance gain of DMDAS (≈3.5% faster than DMDA) is probably due to a better usage of the GPU devices. On the right side, WS demonstrates a very bad allocation, which is not surprising because it does not take into account the heterogeneity of the platform. There are several dynamic critical paths in the WS scheduler, with many DTRSM and DSYRK running on CPUs. It is interesting to note that the more equal workload partition of WS does not produce a better performance since it quickly leads to GPUs starvation. This lack of tasks is emphasized by the curve

of *Ready Tasks* of WS.

Analyzing the constrained executions (Figure 6.6, bottom) where DTRSM and DSYRK tasks are forced to execute only on GPUs, we can observe that such restriction does not really help for the DMDA and DMDAS policies. Tasks on the critical path are no longer an issue, but both schedulers still present some priority problems. The behavior demonstrated by DMDA seems easier to understand: we see some typical list scheduling behavior with critical DPOTRF being delayed because CPUs are used for not so critical DGEMM tasks. If one could run these tasks earlier, it appears that the whole makespan would be greatly improved. Surprisingly, Work Stealing strongly benefits from the imposed restriction and now favorably compares against DMDA and DMDAS. It is also interesting to note that WS manages to keep all CPUs busy from the very beginning, unlike the other two schedulers. However, GPUs are not fully exploited, in particular at the end where they should be used to accelerate the DGEMM tasks like the DMDA and DMDAS strategies do. If there were a way to prevent DGEMM task execution on CPU after time 350ms, we would probably get the best of the two scheduling strategies and be much closer to the optimal execution time.

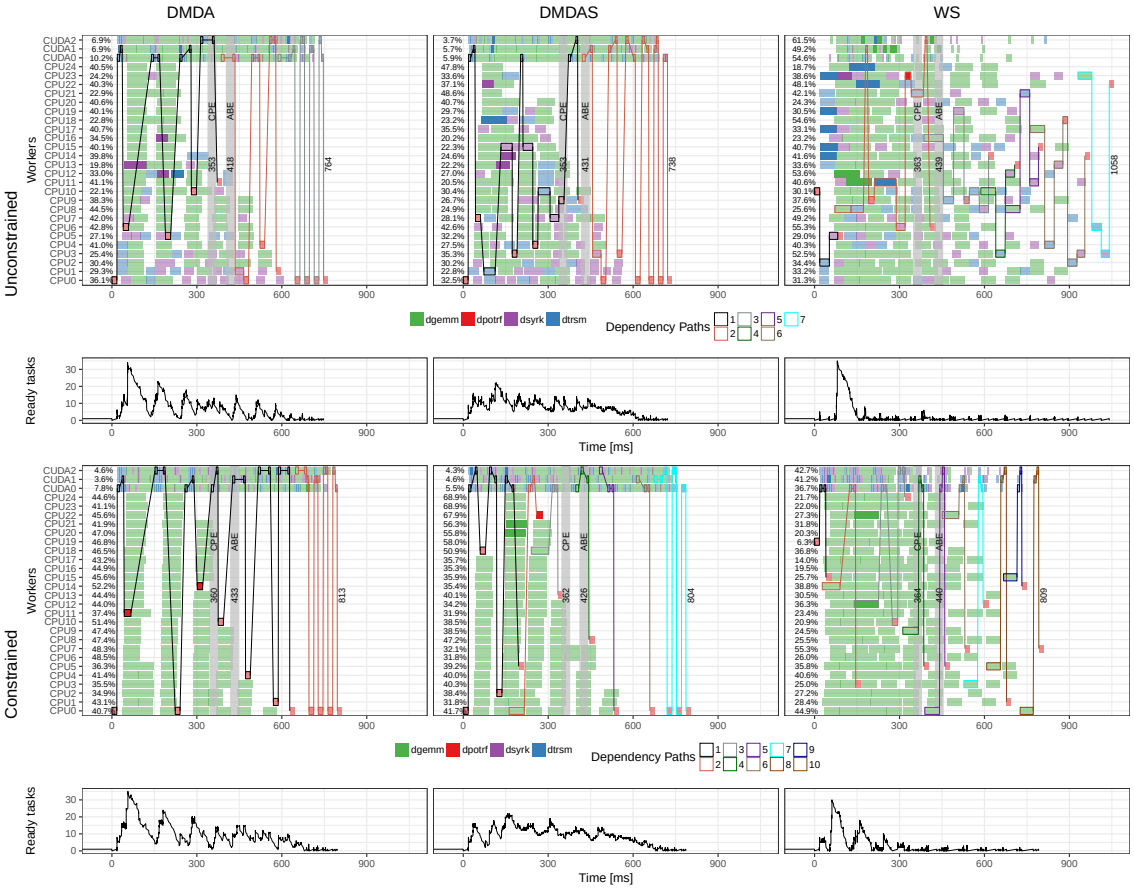
6.3 Case Study: Multi-node Executions with Starpu-MPI

As discussed in Section 2.3.4, the StarPU is able to handle distributed platforms thought its StarPU-MPI extension. The major differences introduced by this extension are the use of several runtime instances (one per node), managing inter-node communications as tasks, and the static domain decomposition. Since these are the main changes from the original single-node version of StarPU, they might be the source of new performance disturbances.

In our previous single-node case study (Section 6.2), we observed that idleness ratio was considerable low for a large enough matrix size, e.g., in average less than 5% for executions with a 60×60 matrix (tiles of 960×960) using the DMDAS scheduler (see Figure 6.4). However, in StarPU-MPI multi-node executions, we can observe very frequent idle periods affecting all the workers (CPU/GPU), especially at the beginning of the application.

Figure 6.7 shows one execution where this problem appears. The idleness ratio is always greater than 10% in both nodes but is significantly greater in the second node ($\approx$16% to $\approx$25%). Despite the spread occurrence of short idle periods during the whole

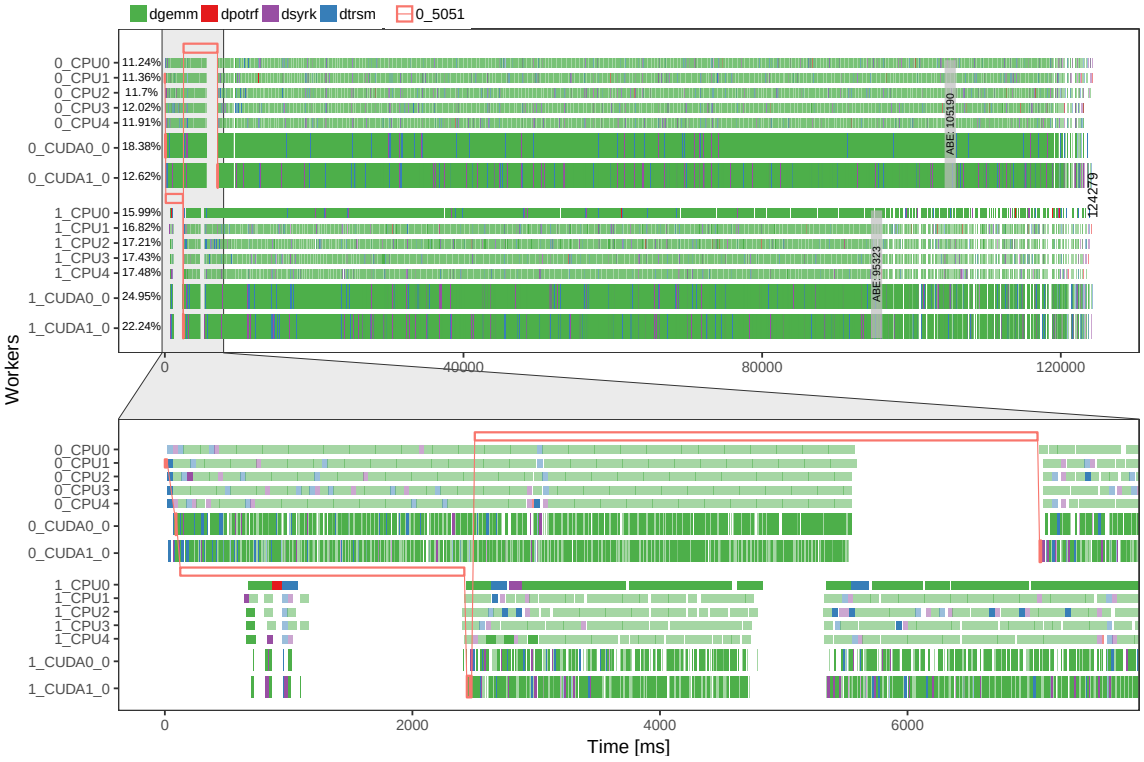
Figure 6.6: Comparison view of six executions of the Cholesky factorization for a matrix of 12×12 tiles of size 960×960. Executions differs in the used schedulers (columns) and the source code (rows). The first row shows executions with the original source code (Unconstrained) using three different schedulers (DMDA, DMDAS and WS). The second row shows executions with the modified source code where DTRSM and DSYRK tasks are constrained on GPUs. All these view are enriched with merged dependency paths starting on DPOTRF tasks. Interactive version available at `http://perf-ev-runtime.gforge.inria.fr/thesis/vpacasestudyfig4-interactiveView.html`



Source: The Author

the execution, the large periods in the beginning seems to be the most impacting ones. Synchronized idle periods, in the beginning, are not totally unexpected since at this moment the parallelism is not yet fully unfolded. However, in the first node, they occur even after the number of ready tasks is already enough to fill the computer resources, e.g., the large idle period at ≈6000ms.

Figure 6.7: StarPU-MPI multi-node execution of the Cholesky factorization with a matrix of 75×75 tiles of 960×960 using PRIO scheduler. On the bottom, a zoom over the first 7500ms of the execution where large synchronized idle periods can be identified. The red curve shows the dependency path of task `0_5051`.



Source: The Author

In Figure 6.7, the space-time view was enriched with dependency edges, which help us to formulate some hypothesis. In this view, the red line shows a backward dependency path starting at a DSYRK task (id `0_5051`) at `0_CUDA1_0` (Node 0). This task was chosen since it is the first task starting after a significantly long idle period. Going back six steps in the dependency path, we can identify that the task `0_5051` has been directly released by a very long MPI operation (red-rectangle at the top from 2504ms to 7050ms). Following-back in the path, this MPI operation was dispatched after a DGEMM task followed by a DTRSM one on `1_CUDA1_0` (Node 1), which in turn were preceded by another long MPI operation (from 125ms to 2420ms). These long MPI operations could be the cause of idle periods appearing during the computation but do not seems to

be related with the slow start of the computation on Node 1, where the first task starts only at 643ms at `1_CPU1`.

From this initial analysis, our investigation has indicated that these idle periods are caused by a combination of two factors: first, the way the MPI thread of StarPU handles asynchronous communications during the beginning of the application which can explain the slow-start on Node 1; second, the MPI threshold configuration between the eager and the rendezvous communication modes which can be related to idle periods during the computation, after the parallelism is sufficiently unfolded.

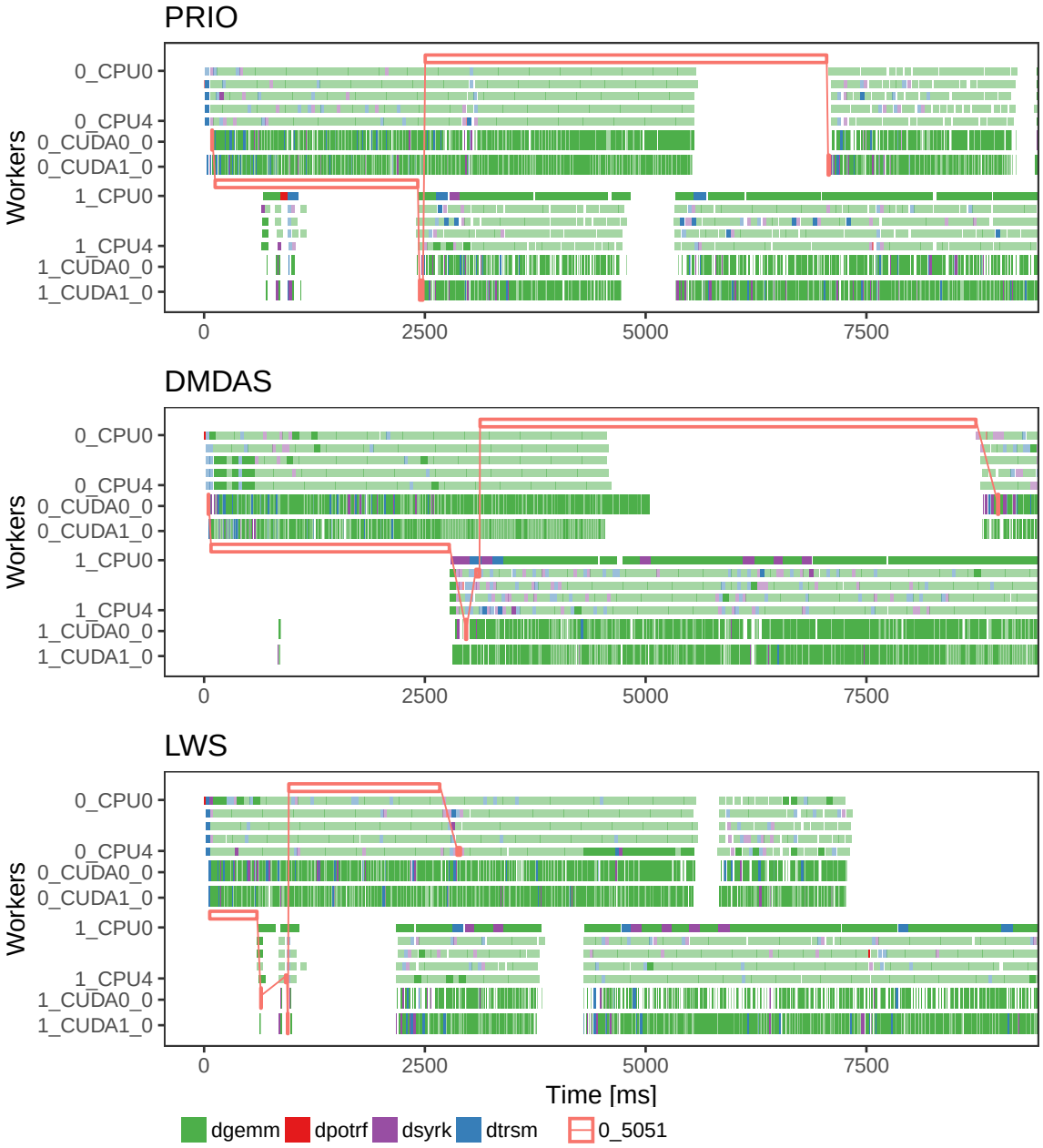
6.3.1 Slow-start in Remote Nodes

The initial analysis enables us to identify an issue during the beginning of the execution where some nodes take a long time to get started. Our hypothesis suggests that this problem is related to how StarPU-MPI handles the MPI asynchronous communications. As showed in Figure 6.8, this behavior was observed in several other executions with different worker combinations, number of nodes, and schedulers.

Since the data partitioning between distributed nodes is static, StarPU identifies all inter-node point-to-point communications to satisfy the data dependencies that cross a node border. Once they are known, the per-node MPI thread of StarPU issues multiple `MPI_Isend` and the corresponding `MPI_Irecv` operations from the start. Depending on the input size and the number of tiles involved in a specific run, the amount of these operations might be very large. The problem is that some of these operations might complete (i.e., the communication finishes) before posting all remaining asynchronous operations. When such scenario occurs, the application is delayed because the MPI thread handling the communication operations has not issued an `MPI_Test` to detect the reception and unlock the corresponding tasks. This negative behavior becomes worse when multiple nodes are used and the borders among tiles have more complex configurations (see $P \times Q$ parameters).

The solution for this first problem was to interleave `MPI_Test` calls between each issue of MPI asynchronous communications. If the test call indicates that a message has been received, the MPI thread of StarPU can satisfy an inter-node data dependency. Since test calls provide a negligible overhead with potentially great benefits at the beginning of the application, this solution is now mainstream in the StarPU code. After the fix, idle periods on remote during the beginning of the computation disappear.

Figure 6.8: The first 8000ms of three StarPU-MPI multi-node executions of the Cholesky factorization with a matrix of 75×75 tiles of 960×960. Slow-start issues on remote nodes are present in the beginning of all executions whatever the scheduler used (PRIO, DMDAS, LWS).



Source: The Author

6.3.2 Idle Periods During the Computation

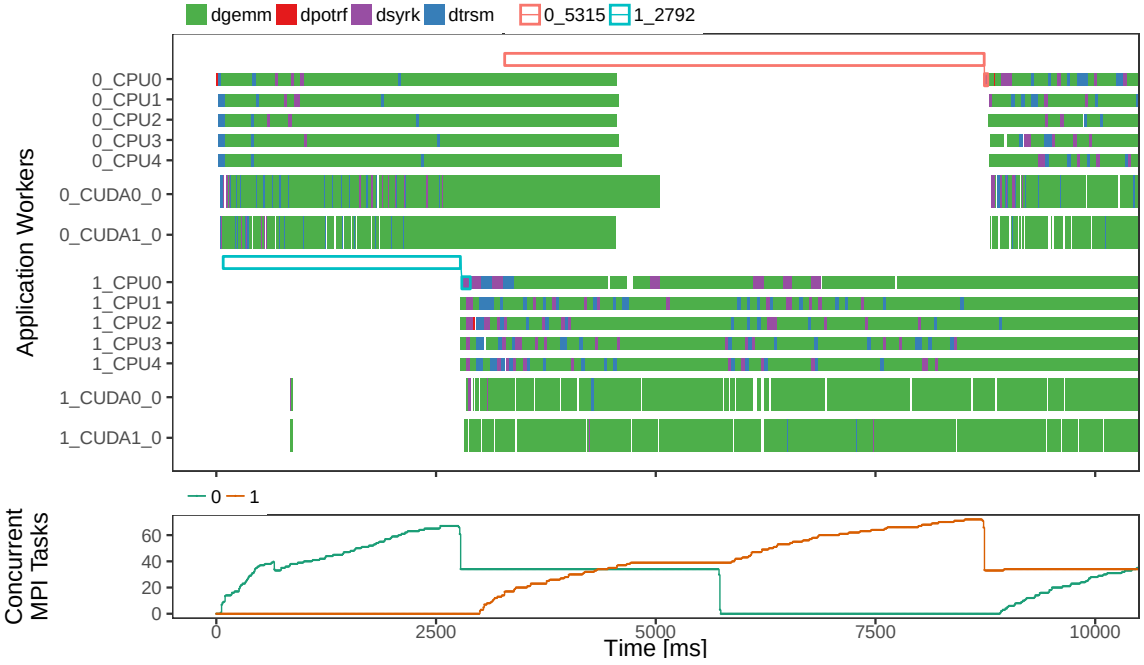
The use of interleaved `MPI_Test` has reduced the idle periods at the beginning of the execution. However, the remaining idleness during the execution is still present after the fix, indicating that the origin is elsewhere.

The root cause of this problem has been identified as the threshold to switch from the eager to the rendezvous communication modes. The eager mode allows a send operation to complete without an acknowledgment from the other side; while the rendezvous mode requires a reception acknowledgment. The change of communication mode is driven by the message size. The default value of the OpenMPI 2.0.2 installation used in all experiments is 64 KBytes. Messages smaller than such size will be sent using the eager mode, favoring asynchronism, while larger messages are sent using the rendezvous, for throughput. In our Cholesky case, the volume of data dependencies depends on the tile size. For example, a commonly used squared tile of 960 8-byte elements occupies $\approx$7.37 MBytes. This implies that only the rendezvous protocol is used throughout the experiments with the default eager limit. Unfortunately, the rendezvous protocol also introduces communication aggregation: MPI requests submitted closely enough will be sent and received together.

This aggregation behavior can be visually checked in Figure 6.9. This view includes an additional panel (bottom) with curves depicting the number (per node) of concurrent MPI operations along the time. In the space-time panel (top), the backtrack of the dependencies of two tasks starting right after long idle periods, highlight they are delayed due to two very long MPI operations (red and blue paths). Correlated with the end of these communications we can observe a large number of MPI operations completing at the same time (around 2700ms and 5700ms on Node 0 and around 8700ms on Node 1). We have typically observed this kind of behavior, with as many as 40 7.37-MByte transfers aggregated together that finish at the same time. This massive network operation may take $\approx$2.5s to complete altogether, instead of completing progressively. Such undesired behavior has been reported to the OpenMPI team, which agreed something needs to be fixed. This behavior is harmless during most of the execution, except in the beginning where little parallelism exists: the execution starts with a single DPOTRF task in the first node, followed by DTRSM tasks, whose results need to be transferred to other nodes as quickly as possible, because all tasks from other nodes depend on these first tasks to start unrolling their part of the DAG. Good schedulers tend to execute the DTRSM tasks as

quickly as possible, but that leads to submitting MPI requests very closely, and thus seeing them all aggregated, and thus received very late. The work-around proposed by the OpenMPI team is to force the eager mode, to avoid aggregation and instead get progressive reception and thus better reactivity, even if it leads to lower network efficiency since clearly, the beginning is very sensitive to pipelined delivery.

Figure 6.9: The first 10000ms of a StarPU-MPI multi-node execution of the Cholesky factorization with a matrix of 75×75 tiles of 960×960 using the DMDAS scheduler. The additional panel depicts Concurrent MPI operations which enables a correlatiion with long MPI transfers delaying tasks `0_5315` and `1_2792`.



Source: The Author

The fix for the second problem is to increase the eager limit to a value that encompasses the tile size in bytes, enabling an asynchronous exchange for all data dependencies. We have increased the tile size to 8 MBytes to confirm that a higher eager limit (using the `btl_tcp_eager_limit` option of OpenMPI) brings performance gains. Figure 6.10 presents a comparison between these two scenarios, the default eager limit (left) and the modified one (right). The fix using a greater eager limit has lead to a reduction in the idleness ratio in all workers of all nodes. Using the default limit (left), it is possible to see some long MPI operations causing significant delays on several tasks (red, green, blue and purple dependency edges) while on the modified version, these large synchronized idle periods affecting all workers on a node have disappeared. The parallelism unfolding can also be verified in the ready tasks panel, where a high number of ready tasks is reached

very quickly and sustained during the execution. The Cholesky Iteration panel shows that the number of tiles being processed at the same time is much higher. The MPI bandwidth performance (not depicted) is higher after the eager limit modification. The curves depicting MPI concurrent operations show that the aggregation behavior has also changed, in the original execution, it possible to see a large number of MPI operations completing at the same time while in the modified one it is possible to see that MPI operations are progressively completed. Such changes enable a 20% execution time reduction, without any application change.

Figure 6.10: Comparison of two StarPU-MPI multi-node executions of the Cholesky factorization with a matrix of 75×75 tiles of 960×960 using LWS scheduler, P=1. On the left, a execution with the default eager limit (64 kB), on the right, a fixed execution with the modified eager limit.

Source: The Author

6.3.3 StarPU-MPI Data Distribution Strategies

Our previous analysis has focused on resource idleness using 2 hybrid nodes. In this section, we present an analysis using all the eight hybrid nodes of the chifflet platform, comprising 48 cores and 16 GPUs. The goal of these experiments is to investigate the influence of data distribution ($P \in \{1, 2, 4, 8\}$) on load balance and resulting perfor-

mance. These executions were performed using the LWS and DMDAS schedulers, using input matrices of 75×75 tiles of size 1440×1440. The MPI was configured with an eager limit higher than the tile size, which in this case is 32 MBytes. In multi-node experiments, as this one, the ABE (Section 5.1.3) is computed per node and can also be used to visualize the load distribution.

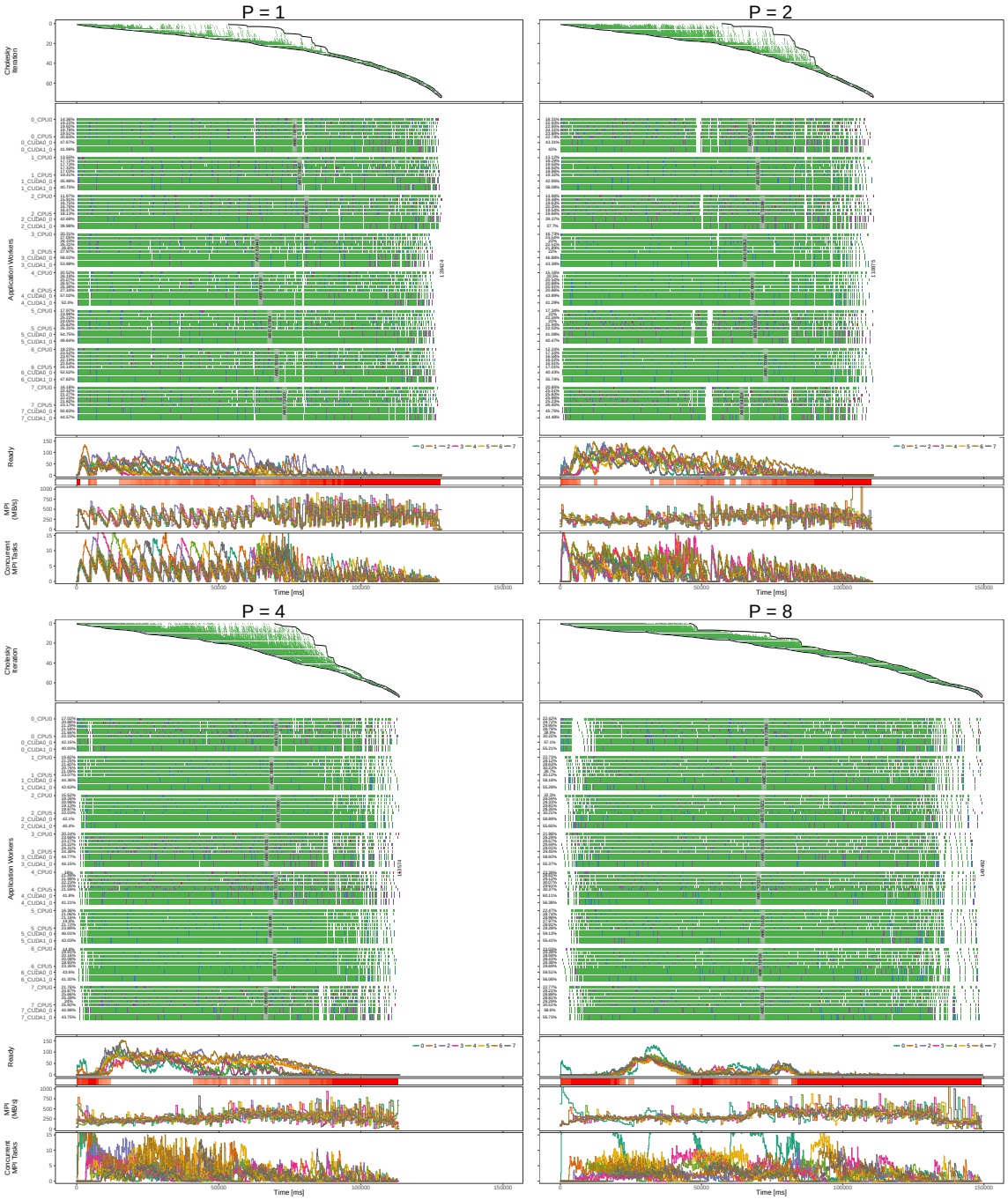
Figure 6.11 shows four executions with P equal to 1, 2, 4, and 8. Using eight nodes, it is expected to have better performance with a P value equal to 2 since it minimizes the nodes communication borders. Using the per-node ABE to analyze the load balancing and comparing it to the value of P, we can observe that load balancing improves when we use a higher value of P, up to the best case with P=8. On the other hand, the overall performance, represented by the makespan, show that the performance is not totally related to a better load balancing. Better results are achieved by the executions using P=2 with a makespan of ≈110875ms, and using P=4 with a makespan of ≈113574ms, despite the fact that the load is unbalanced as shown by the per-node ABE.

Regarding the additional panels, we can verify the reason behind the better performance of executions with P equal to 2 or 4 is that they show a larger number of tiles being computed in parallel, as shown by the Cholesky Iteration panel, and the MPI bandwidth is smoother and flat along the execution time. This led to a better resources usage, as can be confirmed by the idleness ratio, which is smaller, particularly on GPUs. These facts demonstrate that unfolding parallelism on other nodes as early as possible is more important than a better load distribution.

There are many negative observed factors for executions with $P \in \{1, 8\}$. The execution with P=1 is particularly interesting since it demonstrates the largest load imbalance among the four cases. The unequal load distribution force nodes to wait from each other, i.e., the slowest node with the largest load limits the performance of other nodes. This is confirmed through the ups and downs in both MPI bandwidth and number of concurrent MPI operations, like a ping-pong effect. This might indicate a rather sequential execution as shown by the low number of ready tasks along execution. The case with P=8 shows the highest idleness and it is the case where slow-start issues are more representative, even after using the eager protocol in the MPI layer (see the previous section). This is explained by the misplacement of initial DTRSM tasks, responsible for unlocking more parallelism. They are not evenly distributed among nodes, despite the better total load balance.

This case is still very intriguing because of the small 20s window between 25000ms

Figure 6.11: Comparison of four StarPU-MPI multi-node executions of the Cholesky factorization with a matrix of 75×75 tiles of 1440×1440 using LWS scheduler. Each execution uses a different P value (1, 2, 4, 8). Each view is composed of 5 panels: the Cholesky Iteration (first row), the application space-time panel (second row), the curves of ready tasks (third row), the MPI bandwidth (fourth row) and the curves depicting concurrent MPI operations (fifth row).



Source: The Author

and 45000ms where a lot of parallelism is released while the remaining of the execution (before and after) suffers from lack of it. Before that time interval, the runtime is incapable of unlocking enough parallelism to feed all cores because of the column-based distribution, which makes, e.g., node 0 responsible for computing all tasks of the column before letting other nodes to have some work to do. Moreover, it forces node 0 to send a lot of data through the network, as shown in the MPI curve. During and after that 20s-window, the column-based distribution keeps making one after the other responsible for unlocking all parallelism. As long as there are enough iterations computed at the same time, the runtime manages to keep up, but at some point, the runtime is incapable of unlocking parallelism fast enough because of the data distribution, and performance is degraded with a very low number of ready tasks.

A common point of all executions, no matter which P value is used, is the difference between the observed makespans and the values estimated by the ABE. In such multi-node scenarios, matching the ABE is expected to be very hard since it does not take communications costs into account. Despite that, the scheduling and the MPI inter-node communication could be optimized. Regarding the panel depicting the concurrent MPI tasks (fifth row on each one of the four views of Figure 6.11), we can suppose the presumed origin of this performance issue is related to the number of concurrent MPI communications. This in itself is not a problem since these operations might be sometimes carried out by independent pairs of nodes employing independent network links. However, when the communication concurrency is too high, this might delay MPI operations that would unlock parallelism faster. What happens is that since MPI processes requests in the order they were submitted by the runtime, the MPI communications needed by the critical path get delayed by all the previously-submitted requests. This can be observed for the P=8 case, where the first node demonstrates an enormous amount of concurrent MPI operations, delaying all communications by as much corresponding time. Before our investigation, the StarPU-MPI runtime provided no API or configuration to define communication priorities per-task, nor to control the maximum number of concurrent MPI operations.

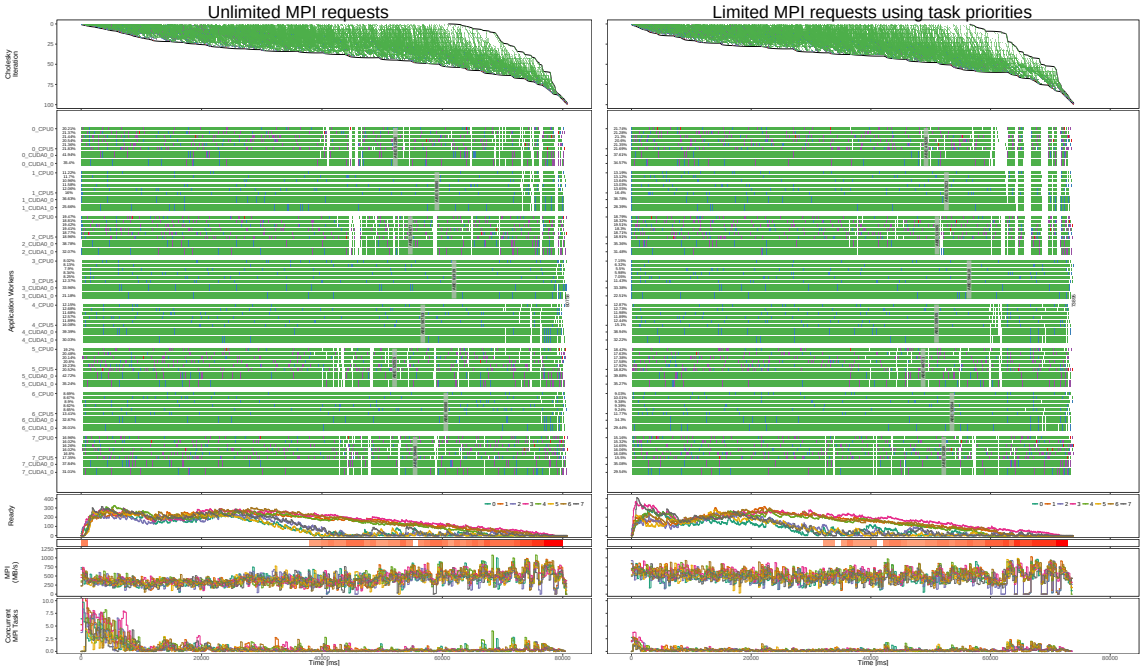
We believe that a better control of MPI operations can be implemented in two ways in StarPU: (a) to give a higher priority to those communications that release parallelism in Cholesky (e.g., the first DTRSM tasks), and (b) to impose a limit on the number of MPI requests issued by StarPU-MPI, so that high-priority requests are delayed at worse by the few requests already issued. These two strategies have been implemented in the

development branch of StarPU after we have identified the problem. Figure 6.12 depicts the scheduling behavior for two representative scenarios with LWS scheduler for two cases: (left) before such modifications and (right) after, using a limit of 10 concurrent MPI requests and giving DTRSM higher communication priority (simply inherited from the task priority, so without application modification). After the changes were introduced, we can see that MPI delivers a higher bandwidth for the application, especially in the crucial starting moments where parallelism is being unfolded. The more controlled MPI requests can be verified in the bottom plot, where the number of concurrent MPI operations is much more contained. The Cholesky Iteration panel shows that the parallelism unfolds much faster, in less than 10000ms, after the implemented changes. This behavior is confirmed by the ready tasks plot, where the number of ready tasks per node (colors) responsible for unfolding parallelism reaches a peak at the beginning of the application, while before (left) such behavior caused a minor yet damaging slow start of the application. This has the direct benefit of exposing in a much faster way the critical path towards application completion and gives much more scheduling opportunities to improve performance. In conclusion, as can be observed in this comparison, these modifications reduce the total makespan by ≈8.5% (from 80758ms to 73895ms). Figure 6.13 shows the results are similar when using the DMDAS scheduler, the gains are of ≈12.7% (from 76851s to 67110s).

6.4 Chapter Summary

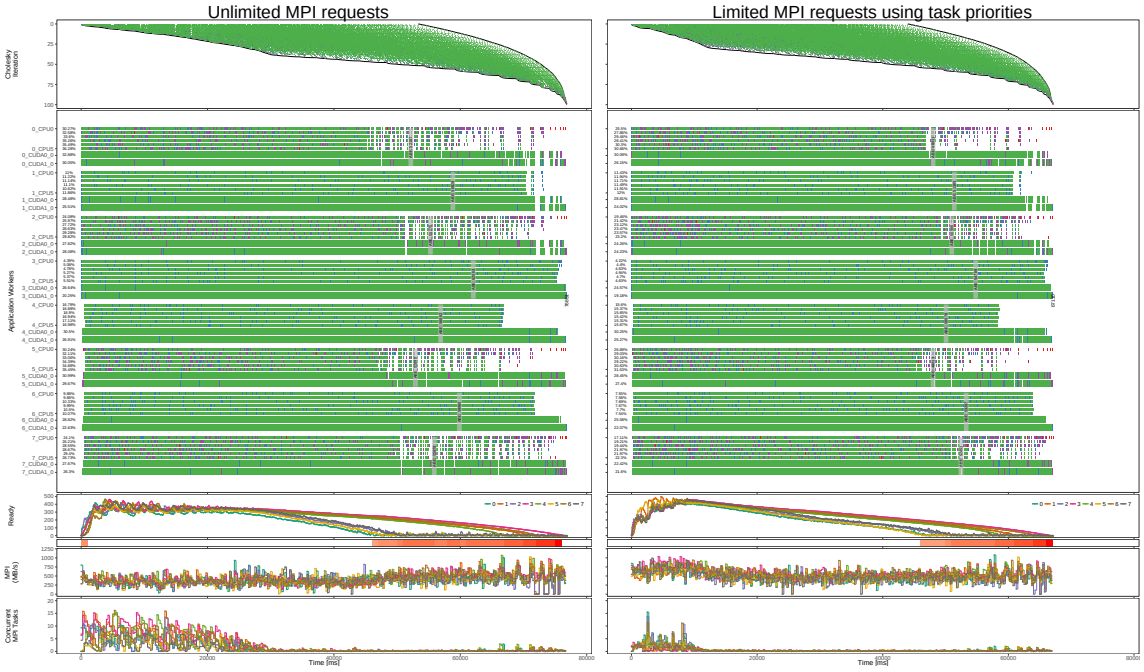
This chapter presents case studies demonstrating how our visualization strategies can be used to identify and fix performance issues. Our first case study shows the impact of different StarPU scheduling policies when executing in a hybrid node. This analysis also provided us hint about how to enhance the task partitioning among the CPU cores and the GPUs to get closer to theoretical lower bounds. The second case study was carried out in a multi-node hybrid platform and focuses on performance analysis of the StarPU-MPI extension. Using our strategies, we identify performance disturbances which enabled us to propose adjustments to mitigate their impact on the overall performance. These improvements comprise a better management of the pipelining strategy of MPI operations to reduce slow-start effects and resource idleness in multi-node executions and changes in the StarPU runtime system to reduce the number of concurrent MPI operations and increase the MPI bandwidth.

Figure 6.12: Comparison of two StarPU-MPI multi-node executions of the Cholesky factorization with a matrix of 100×100 tiles of 960×960 using LWS scheduler and P=2. On the left, a execution using the original StarPU code with unlimited MPI requests. On the right, an execution after the introduction of changes to limit the number (in this case 10) of MPI requests using task priorities of DTRSM tasks.



Source: The Author

Figure 6.13: Comparison of two StarPU-MPI multi-node executions of the Cholesky factorization with a matrix of 100×100 tiles of 960×960 using DMDAS scheduler and P=2. On the left, a execution using the original StarPU code with unlimited MPI requests. On the right, an execution after the introduction of changes to limit the number (in this case 10) of MPI requests using task priorities of DTRSM tasks.



Source: The Author

On considerably large scenarios, the use of our performance analysis strategies has enabled us to propose changes that increase the overall performance. However, in small scenarios, similar changes have amplified unexpected scheduling decisions that seem to be related to task priority issues. Since the information available in the execution traces is not sufficient to clarify these points, we propose, in Chapter 7, a further investigation using Simgrid and GDB. This combination of simulation and debugging enables us to inspect and monitor the internal variables and queues of the scheduler.

7 PROPOSED DEBUG STRATEGIES

Sometimes the insights of the macro visual analysis techniques previously discussed on Chapters 5 and 6 are not enough to understand some scheduler decisions since these techniques rely on the analysis of execution traces after the end of the execution, i.e., post-mortem analysis. This way, our assumptions and its verifications are limited by the amount of observed data collected. However, inspecting in detail some scheduling decisions requires the verification of many other variables, which cannot be traced in a non-intrusive way. Moreover, specific potentially wrong scheduling decisions are very often hard to reproduce because of the stochastic nature of the runtime. In order to better understand why some unexpected decisions are taken by the StarPU scheduler, we propose to use simulated executions in conjunction with a GDB session to collect and inspect the scheduler internal state.

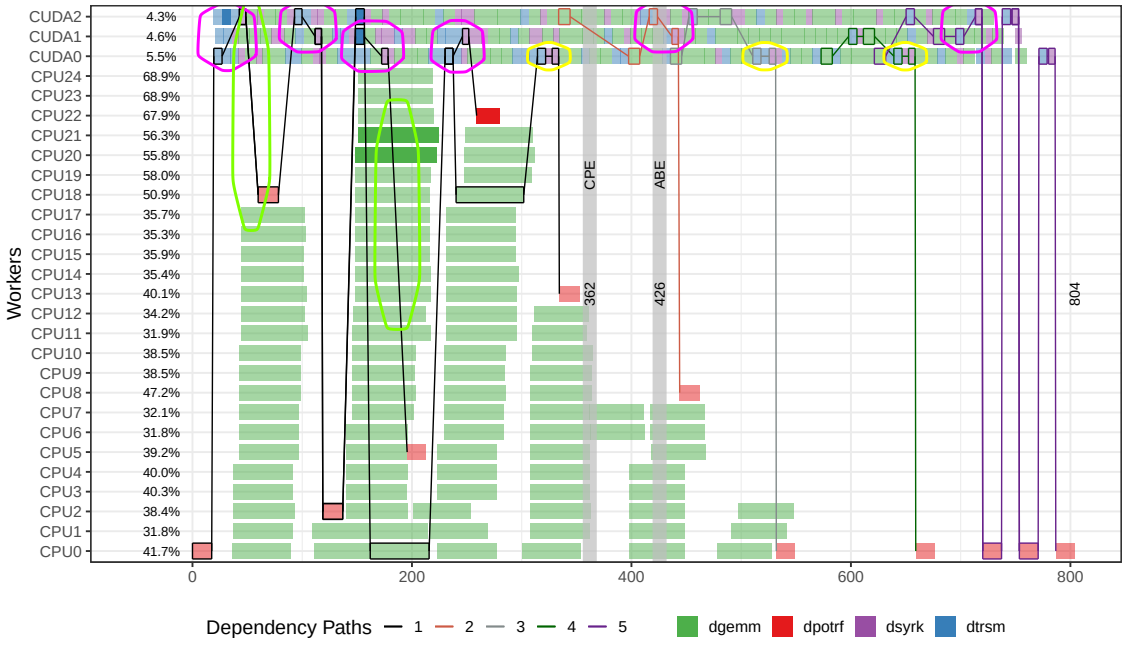
In this Chapter, we detail how we build this strategy and how it can be useful to confirm or refute some assumptions about bad scheduling decisions. In Section 7.1 we present an execution where we highlight some scheduling decisions that seem to be inadequate. Section 7.2 presents the tools and concepts used to build our debug-based analysis strategies. Finally, Section 7.3 presents our approach to visualize the scheduler internal state when scheduling a given task.

7.1 A Representative Example

In Section 6.2.2 we have reported that constraining some tasks to execute only on the GPUs has amplified some potential scheduling mistakes. Figure 7.1 allows us to focus on one of the executions represented in Figure 6.6. The basic view remains the same (DMDAS, constrained), but we have increased the number of backward steps when computing dependencies. As previously discussed, on one hand, constraining DTRSM and DSYRK tasks to GPUs has mitigated the issue of tasks on the critical path, but on the other hand, it has also highlighted dependency issues which seems to be related to the tasks priorities.

The circled edges in Figure 7.1 show some examples of scheduling decisions that seem to be inadequate. Ideally, the execution of a task should start as soon as all its dependencies have been satisfied. In practice, this is not always possible since sometimes a data transfer is required to start the new task. This is probably the scenario of the dependency

Figure 7.1: A space-time view representing the execution of a Cholesky factorization (matrices 12×12 tiles of 960×960). Some dependencies edges are encircled to illustrate some potential mistakes of the StarPU scheduler.



Source: The Author

edges that are circled with green (CPU-GPU) and magenta (GPU-GPU). However, it is not possible to confirm that using only this view.

The dependencies encircled in yellow represent a different case, where the delay when starting the new task cannot be explained by data transfers since both new and previous tasks are allocated on the same device. In these cases, the delays might be related to the task priorities, which are defined by the application programmer to help the scheduler in the decision of which task to schedule first when several of them are ready. However, the priorities are not included in the execution trace. Furthermore, even if they were included, they would be not really useful since they are evaluated considering the scheduler state at the moment when they are tagged as ready and not when their execution actually starts.

7.2 Methods and Materials

As discussed, the information available in the execution traces is not enough to investigate the issues with delayed tasks since we cannot reproduce or understand the estimations computed by the scheduler when scheduling a new task. To proceed a fur-

ther investigation on scheduler decisions we build on the simulated executions offered by StarPU in conjunction with Simgrid which enables us to inspect and collect the values of internal control variables and queues. These values can be used to rebuild the scheduler state at given moment and them confirm or refute assumptions concerning the scheduling decisions. In the following subsections, we present some concepts and tools concerning the simulated executions. To simulate a StarPU execution, Simgrid relies on the Performance Models (7.2.1) collected during a real execution. These models provide the amount of time that should be added to the simulated time to simulate the execution of each task type on each computing resource. Subsection 7.2.2 details how Simgrid is able to simulate StarPU executions. The Subsection 7.2.4 depicts which data is collected during the simulation. Finally, we present the StarPU modifications required to allow our analysis.

7.2.1 Performance Models

As discussed in Section 2.3.4, some StarPU scheduling policies are based on performance models. These models keep a history of the time spent to perform a task on a resource or to execute paired data transfers between different resources. From this information, the scheduler can estimate in advance the duration of a task on a given resource and the time to transfer the data from another resource if needed. By default, these performance models are recorded at the end of each execution in order to improve scheduling decisions of future ones. This way, the performance models can be seem as a part of the output data produced by an execution, and then we gather and include them as a section in the execution log file.

7.2.2 StarPU/Simgrid simulated executions

SimGrid (CASANOVA et al., 2014) is a simulation toolkit that aims to offer versatile simulation of large-scale distributed systems. As reported by Stanisic, Thibault, et al. (2015), the StarPU runtime system has been modified to cooperate with SimGrid in order to simulate task-based executions on hybrid architectures comprising multicores and several GPUs. Their approach is based on both simulation and emulation. In the emulation side, the StarPU control and scheduling code is modified by replacing POSIX synchro-

nization calls to similar ones provided by SimGrid. This modified runtime/scheduler code is actually executed while the real computation of CPU and GPU kernels (the tasks) and the data transfers are simulated by SimGrid that manages the simulated time.

Since this solution has been validated with a similar Cholesky application as well with several other more complex ones, it is the ideal tool to support our investigation. The performance models computed by StarPU can be used to simulate a task-based execution with Simgrid, which allows us to simulate a previous real execution. This environment provides a more controllable, stable and reproducible scenario to investigate the StarPU scheduling decisions. It is also independent of the platform where the experiments were executed, so the investigation can be carried out on the same machine that is used to analyze the traces.

7.2.3 StarPU Scheduler Internals

When using scheduling policies based on performance modeling (e.g., DMDA and DMDAS), StarPU perform a series of estimations to decide on which computing resource the task will be scheduled. These estimations are built considering the current state of the scheduler, the task performance model, the architecture model and some user-defined parameters to prioritize one or other characteristic. In the end, the StarPU scheduler decides to put the task in the worker where it will be completed first.

In order to understand why a given scheduling decision was taken, we should inspect the scheduler state at the moment where a task is being scheduled, just after the computation of all the estimations.

7.2.4 GDB Scripts to Capture the Scheduler State

Since such specific scheduling informations are not available in the execution trace, and it would be expensive to record them for all the application tasks, we decide to use the GDB debugger to pause the execution, collect useful information and then proceed.

We design a set of GDB scripts that are executed when a given task is being scheduled. These scripts collect the following data from the core of the StarPU scheduler framework:

- **estimated duration of the new task**: a list of size $N_{resources}$ (where $N_{resources}$ is the number of computing resources) with the time required to perform the task being scheduled on each resource;
- **estimated data transfers**: a list of size $N_{resources}$ with the expected duration to perform all data transfers required by to be able to execute the new task on the given resource;
- **estimated termination time**: a list of size $N_{resources}$ with the time when the new task will be completed if executed on this given resource;
- **estimated start time**: a list of size $N_{resources}$ with the time when the execution of the new task will start if executed on this given resource;
- **queued tasks**: a table with the ready tasks already scheduled on each computing resource;
- **user fit parameters**: a list of parameters to prioritize some criteria when computing the estimation (e.g., give a higher weight for data transfers);
- **fitness**: a list of size $N_{resources}$ with the fitness value computed to execute the new task on each resource. This value is computed using the state of the worker, the task duration and transfer estimations, and the user fit parameters.

Other less important variables are also collected to help in the investigation.

7.2.5 StarPU Modifications

Some modifications were made in the StarPU code to enable our analysis. The first one is the inclusion of environment variables to raise a signal when a task with a given ID is being scheduled. This enables us to pause the execution of the application and run our GDB scripts to get the scheduler state at the exact moment where the task is scheduled. Other modifications comprise additional fields in the StarPU control variables to help in the reconstruction of the scheduler state.

7.3 A Visual Representation of the Scheduler State

From the data collected by our GDB scripts when a given task is being scheduled, we propose a visual representation of the scheduler state. This view represents the

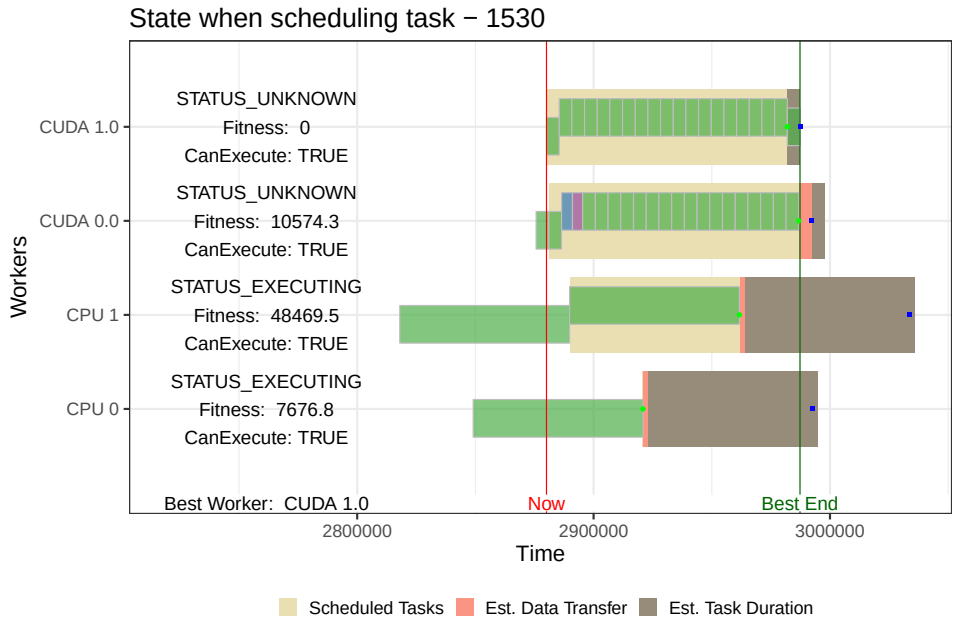
scenario built by the scheduler to estimate which is the best worker to schedule the new task. The view is inspired on the space-time one presented on Section 5.1. It depicts the internal state of scheduler reconstructed by interpreting the data gathered using GDB.

Figure 7.2 shows the scheduler state when scheduling a DGEMM task with ID 1530. In this experiment, DSYRK and DTRSM tasks are constrained to GPUs, so their execution on CPUs is disabled.

This view builds on the space-time view, so it describes the scheduling states of the workers along the time. These states are represented by the colors brown, orange and beige. The brown rectangles are used to represent the estimated task duration. Since these two GPU cards are identical, the estimated durations are almost the same. In the line representing the resource that was chosen to execute the task we also plot a smaller rectangle representing the new task being scheduled. The orange rectangle represents the estimated duration of data transfers required to execute the new task on this resource. In this case, the CUDA1.0 resource is the one which requires the faster transfer to perform the task. The beige rectangles represent an estimation of the time required to finished the tasks that are already scheduled in the worker. This estimation comprises tasks that are scheduled in two queues, for this reason, inside this area, we plot smaller rectangles representing the queued tasks. All these tasks are already tagged as ready for execution. In the higher level we plot the ready tasks that are waiting for execution in the worker queue while in the lower one, we plot tasks that are in the current tasks queue but whose status is not running. This second queue is present only in workers where the pipeline feature is enabled. The vertical red line represents the moment when the execution was paused (now) and the green one indicates the best termination time for the new task without considering the time to perform required data transfers. The fitness computed for each valid resource is also included in the views, as well as, the worker status, the ability to execute the task and the best worker.

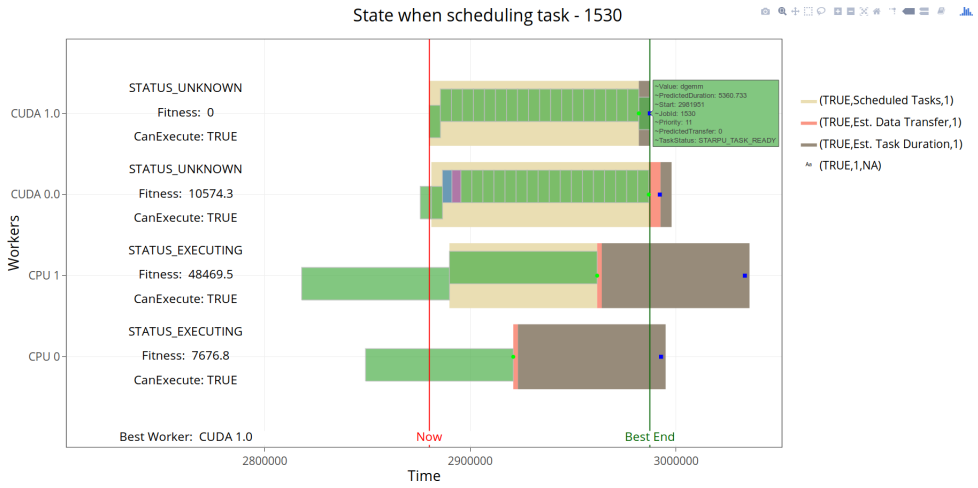
The interactive features provided by `plotly` are particularly useful to check complementary data of tasks in scheduler state views. In this scenario, the `plotly` limitations discussed in Section 5.4 are not a constraint, since the amount of data to plot at a given time is always very small. Figure 7.3 shows a screenshot of the interactive version of the scheduler state view. Hovering over the new task or over the queued ones, it is possible to inspect task details such as task type, prediction duration, predicted start, ID, priority, predicted data transfers, task status, and queue. Comparing such attributes of the new task with the queued ones can help to understand the scheduler decisions.

Figure 7.2: A visual representation of the internal state of the StarPU DMDAS scheduler when scheduling a DGEMM task with ID 1530. For all workers which are able to execute this task, we plot the estimated time to execute the current scheduled tasks (rectangles in beige), to transfer the data required by the new task (rectangles in orange) and to execute the new task (rectangles in brown). Smaller rectangles represent the already queued tasks (green, blue and purple tasks in the beige area) and the new task (green task in the brown area on CUDA 1.0). The color scheme used in DGEMM, DSYRK and DTRSM tasks is the same of previous figures.



Source: The Author

Figure 7.3: Screenshot of a Web interactive view of the StarPU scheduler state generated with `plotly`. When the mouse hovers on a task, complementary information is showed, such as its type, predicted Duration, predicted Start, ID, priority, predicted data transfers and status. The corresponding interactive view is available at `http://perf-ev-runtime.gforge.inria.fr/thesis/interactiveViewSched1.html`



Source: The Author

7.4 Chapter Summary

In this chapter, we present an analysis strategy based on StarPU-Simgrid simulated executions in conjunction with a GDB session to investigate scheduling decisions at the moment they are taken. This combination of debugging and simulation tools gives us additional information that is not available in the traces. From this data, we build a visual representation of the scheduler state which depicts the estimations computed by the scheduler when choosing the best worker to execute a new task.

The next Chapter will present an analysis of the scheduler state for a set of tasks that seem to be delayed due to potential incorrect scheduling decisions. For each task, we compare the estimation computed during the scheduling with the actual execution.

8 RESULTS ON DEBUG STRATEGIES

In this Chapter, we present a detailed investigation of scheduler decisions using the scheduler view presented on Chapter 7. In the analysis, we focus the investigation in three scenarios: dependencies between tasks that execute on different resources of the same type (e.g., GPU0-GPU1), dependencies of tasks executing on different resources of distinct type (e.g., CPU-GPU) and dependencies of tasks that are executed on the same resource (e.g., GPU0-GPU0).

8.1 Experimental Setup

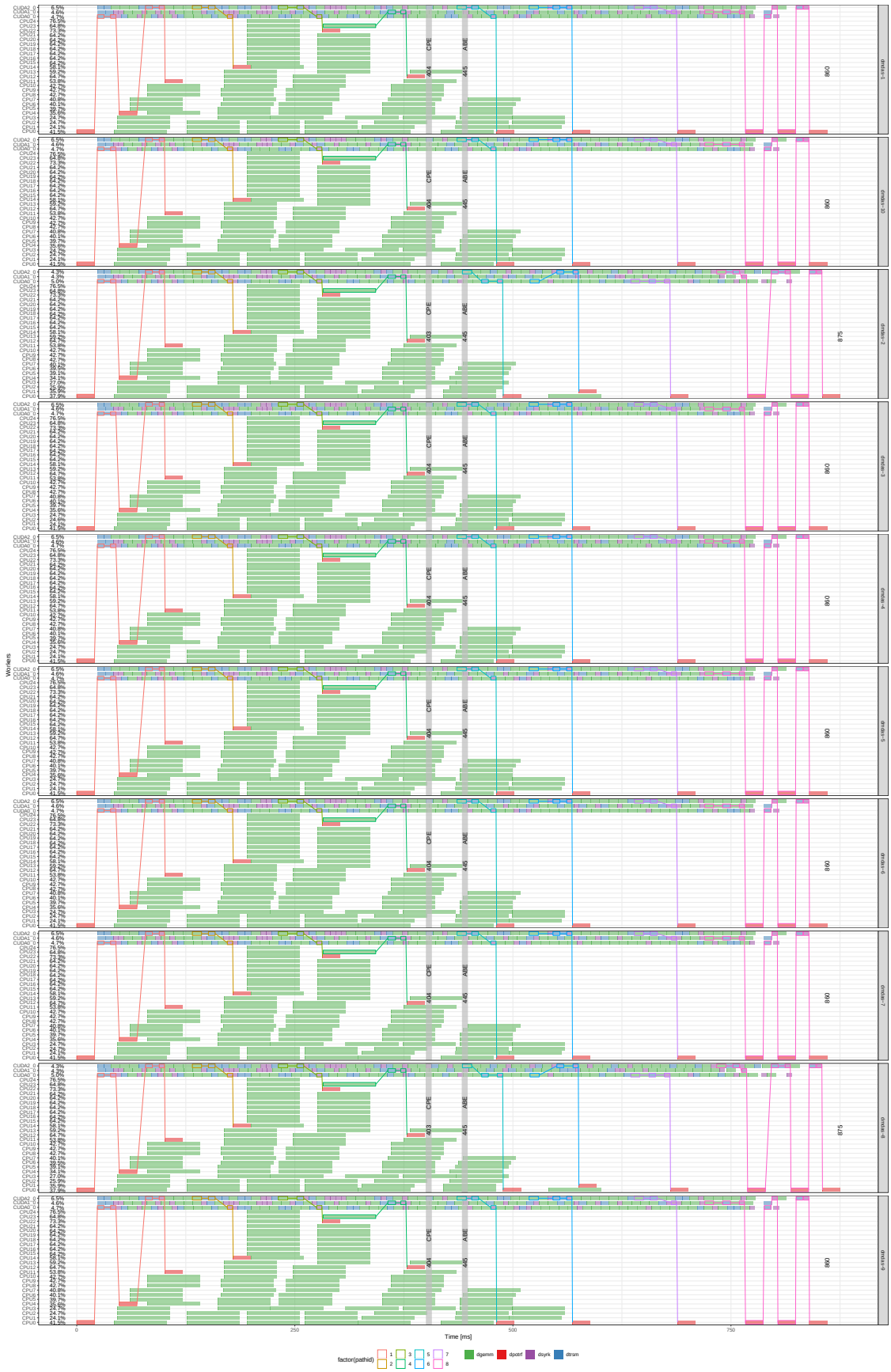
This investigation is performed using a StarPU instance compiled with SimGrid support which provides us a more stable and reproducible environment. To ensure we are not considering a mistake due to the simulation feature or, on the other side, a problem that appears only in real executions, we have selected only issues that are present in both real and simulated executions. In addition, we repeat the simulation ten times, to be sure that the issue in question is omnipresent. As shown in Figure 8.1, all repetitions are stable, the traces are visually very similar and dependency issues are present in all executions. All these executions were executed using the same performance models collected in the experiments discussed in Section 6.2.2.

8.2 Investigating Scheduling Decisions in the First Half of the Execution

At the beginning of the execution, the parallelism is not yet totally unfolded, which means that the number of ready tasks may not be enough to fill all the resources. For this reason, any delay in the execution of tasks in the critical path will further delay this unfolding step and retard the full occupation of computing resources which can significantly impact the overall performance.

To investigate this scenario, first, we select representative tasks (pointed tasks on Figure 8.2), then we rerun the simulation stopping the execution during the scheduling of these tasks to inspect the internal state of the scheduler and try to figure out why they were not executed before. This way, if we are investigating N tasks, we need N simulated runs, each one stopping on the scheduling of a different task. This is required since currently it

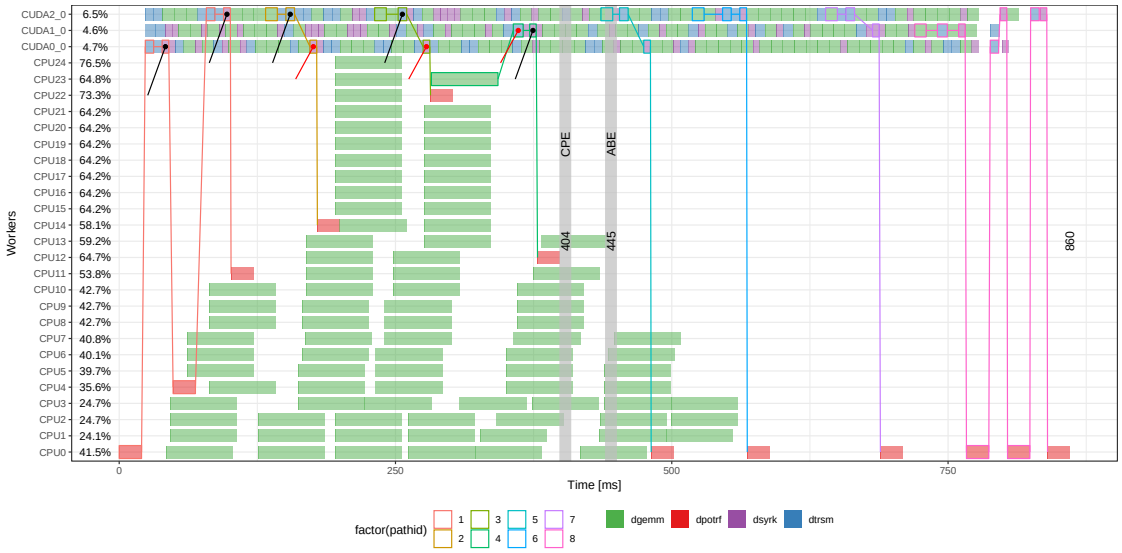
Figure 8.1: The space-time view of ten simulated (StarPU+SimGrid) executions of the Cholesky factorization (matrices with 12×12 tiles of 960×960).



Source: The Author

is not possible to ask to StarPU for stopping the simulation at more than one scheduling point.

Figure 8.2: Space-time view with markers (black pins) to indicate representative delayed tasks. The scheduling state of these tasks will be investigated in the next sections.



Source: The Author

The tasks highlighted in black in Figure 8.2 present a similar behavior, each one executes on the same resource of its last dependency and seems to be delayed by the execution of a third task. This situation seems to be related to a mistake in the task-priorities defined by the application. For this reason, we inspect the scheduler state when they were scheduled.

For the DSYRK task with ID 1333, in the Scheduler state view on the top of Figure 8.3, we can see that best worker to execute this task is CUDA0 which has the lowest fitness value, since it requires the faster data transfer. CUDA1 will be available earlier (green dot) once it has less tasks waiting in the queue, however it requires a longer data transfer. After having chosen a worker, the StarPU scheduler should compute the best position to push the new task to the worker queue of ready tasks. This is done by comparing the priorities of the new task with the queued ones. Since the new task has priority 23, it will be pushed in front of all tasks already waiting in the queue (priorities 19 and 15). The space-time view on the middle of Figure 8.3, that shows the final trace of this execution, allows us to confirm that DSYRK 1333 was executed on CUDA0. The bottom of this Figure shows a comparison between the scenario estimated by the scheduler and the executed one. This zoomed view allows us to confirm that the new task has, in fact, executed before the queued ones (DTRSM tasks with IDs 1324 and 1328). In an ideal scenario, the

task 1333 should also be executed before the task 1322 since this one has lower priority (23 versus 21). However, in practice, this situation is not possible for two reasons. First, task 1322 is already on the current queue (represented by lower rectangles in the figure), which is, in fact, a pipeline queue. This way, even if its status is not yet RUNNING, the scheduler cannot preempt it to prioritize the task 1333. The second reason is the data transfer required by task 1333 to start its execution.

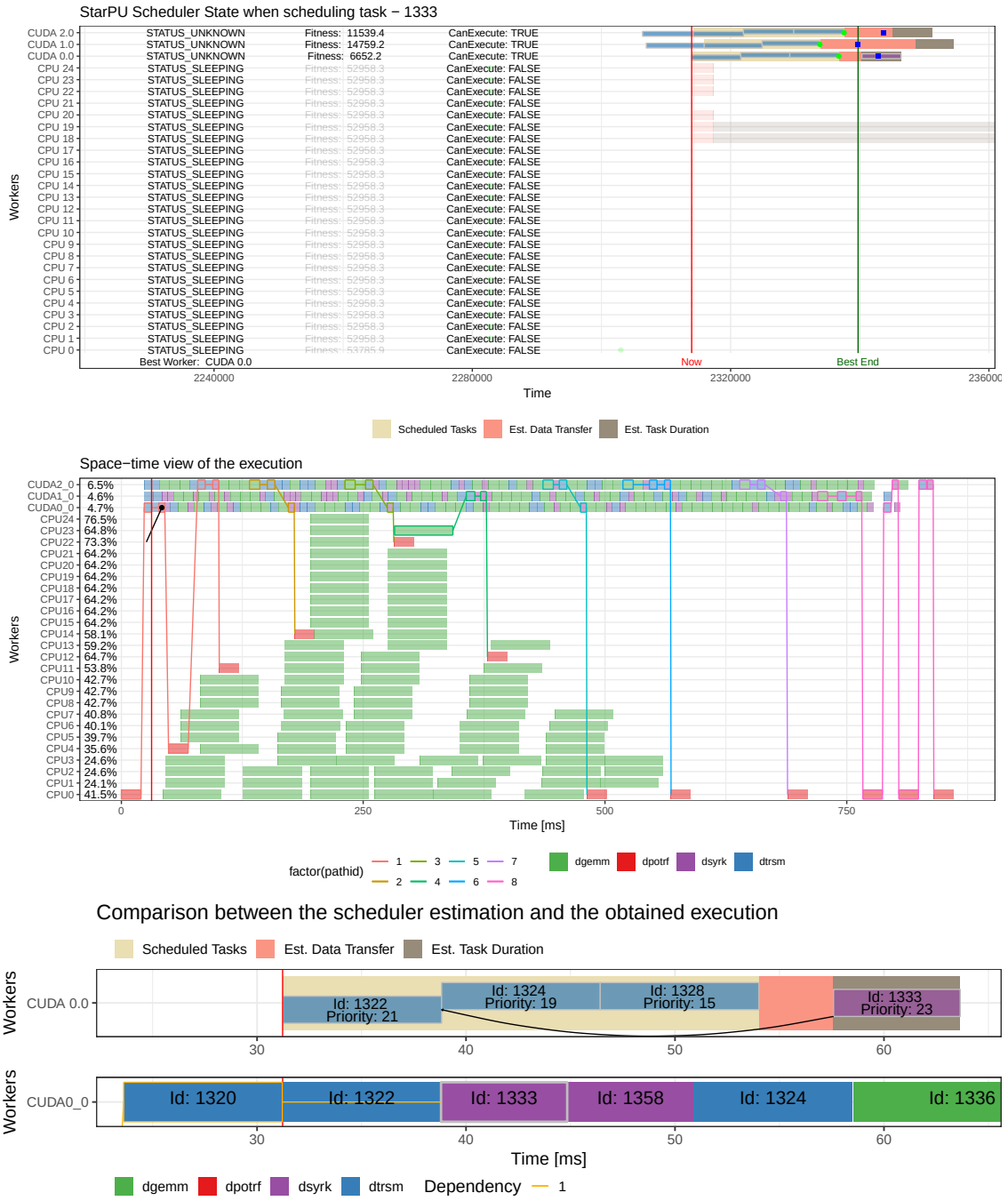
In the end, we can conclude that there is no mistake in this scheduler decision. The best worker was properly selected and the priorities were correctly considered. In fact, the delay of task 1333 is due to the StarPU pipeline mechanism that prevents task 1333 to execute right after the task 1320 (its last dependency).

The analysis of other selected tasks, such as the DSYRK 1434 in CUDA2, the DTRSM 1585 in CUDA2 and the DSYRK 1657 in CUDA1, show they were also delayed because of the pipeline mechanism since the worker choice and the priorities were respected. In Figure 8.4 we present only the comparison between the estimated and the obtained execution to show their similarity with the previously investigated case (DSYRK 1333).

On the other hand, the DTRSM 1510 is a different case. As depicted by the scheduler state view in Figure 8.5, this task does not require a data transfer to be executed on the chosen worker (CUDA2). This way, the only limitation is the pipeline queue which already contains another task (DTRSM 1425) with the same priority. If it was possible to push the new task directly to the front of the pipeline queue, we could estimate a gain of 7.8ms ($\approx$1%) changing only this scheduling decision. This estimation of gain is inferred from the difference between the time the last dependency is completed and the time when the task 1510 starts.

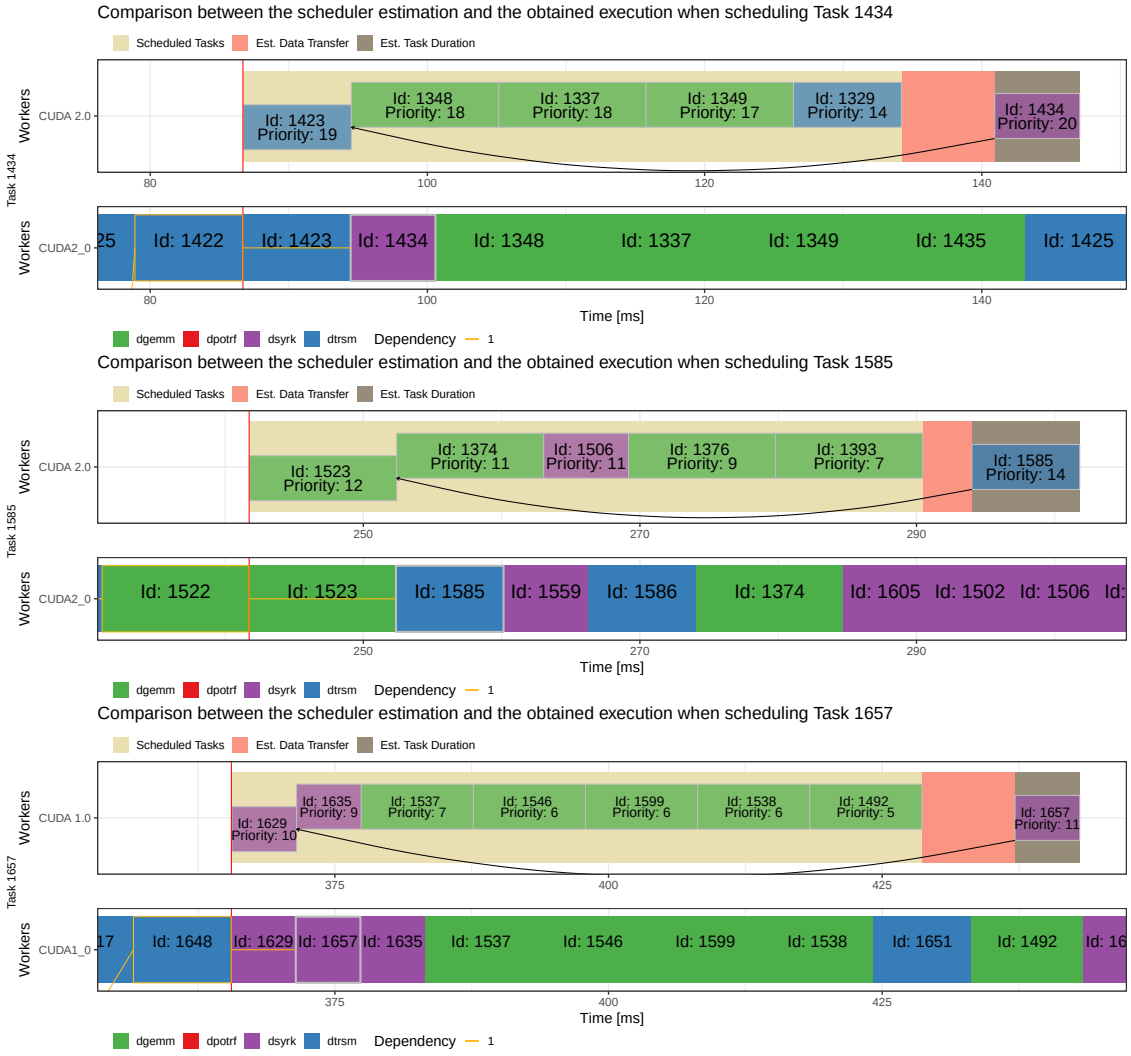
All the previous discussed tasks were scheduled in the same workers of its last dependency. We have also inspected the scheduling of tasks where the last dependency was executed in another worker (tasks marked with red pins in Figure 8.2). In such cases the conclusion is similar, priorities are correctly defined and are properly respected by the scheduler, the new tasks cannot start before due to both required data transfers and the pipeline mechanism, as can be checked in Figure 8.6. Each of the three parts of this figure shows a comparison between the scheduler estimation and the obtained execution of tasks whose last dependency was executed in a different worker. All of them require a data transfer as illustrated by the orange area. We can confirm that their priorities have been correctly respected by comparing the priority value of the new task with the priority

Figure 8.3: A comparison between the estimations computed by StarPU during the scheduling step of the DSYRK task 1333 and the obtained execution. The vertical red line represents the moment when the execution was stooped and when the scheduler state was collected. The top panel shows the estimations computed by the StarPU scheduler when scheduling the task 1333 (interactive version at `http://perf-ev-runtime.gforge.inria.fr/thesis/interactiveSchedView1333.html`). The middle panel shows the trace obtained in the end of the execution (interactive version at `http://perf-ev-runtime.gforge.inria.fr/thesis/interactiveFinalView1333.html`). The bottom one shows a zoom over estimated state of CUDA0, which is the worker chosen to execute the task, and the obtained execution.



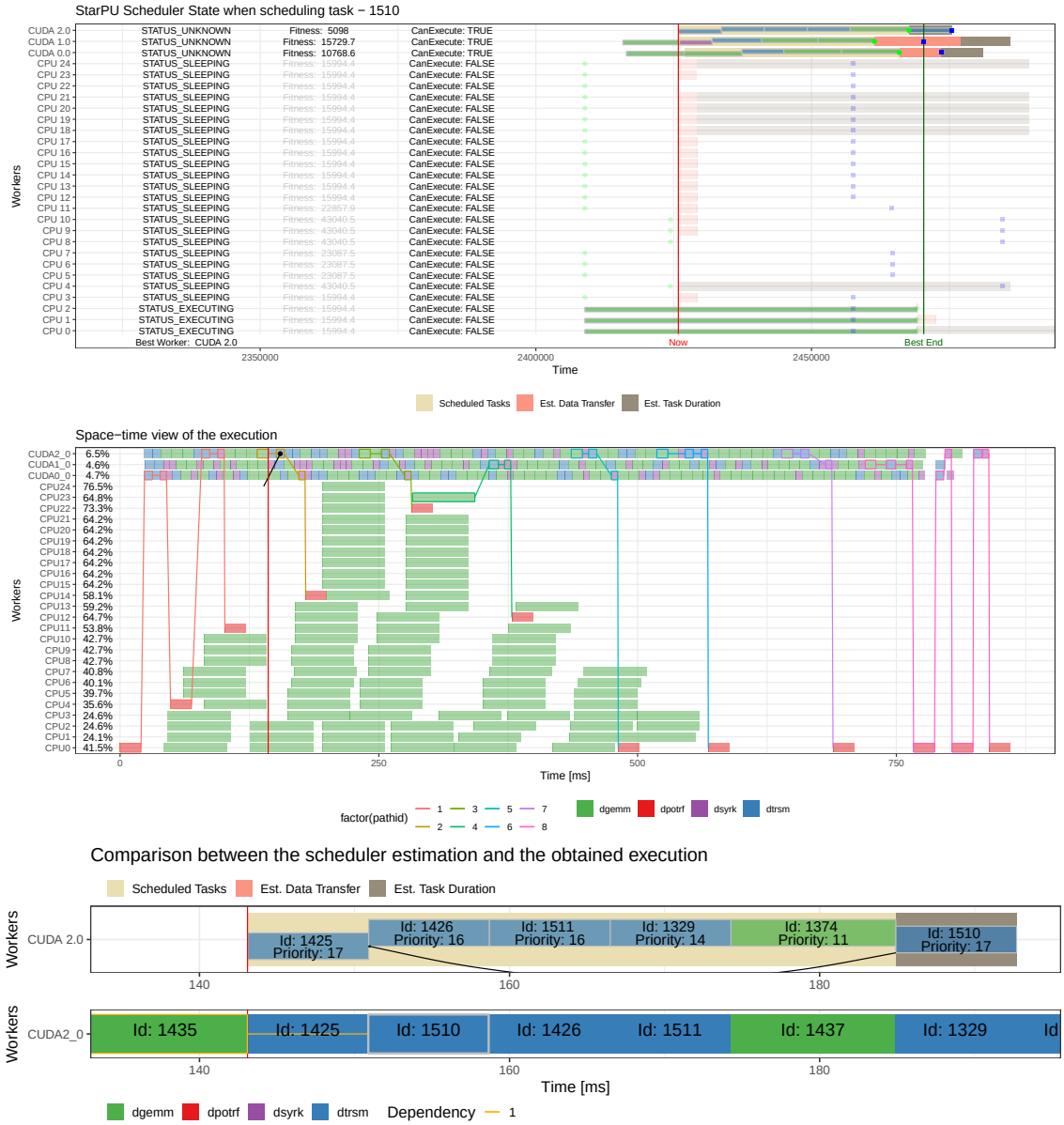
Source: The Author

Figure 8.4: Comparison of the scheduler estimations and the obtained executions for three tasks: DSYRK 1434, the DTRSM 1510 and the DTRSM 1585.



Source: The Author

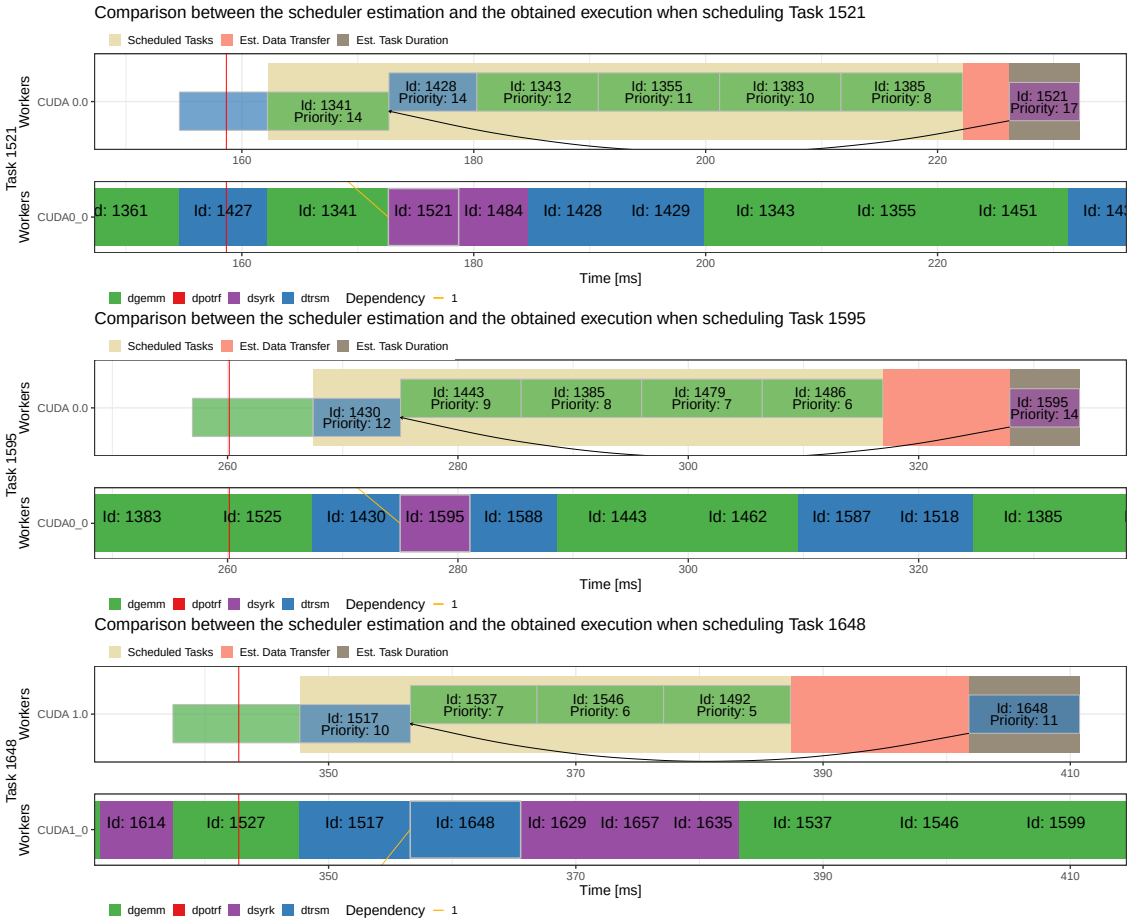
Figure 8.5: A comparison between the estimations computed by StarPU during the scheduling step of the DSYRK task 1510 and the obtained execution.



Source: The Author

of queued ones. By comparing task IDs in the plot depicting the obtained execution we can confirm that new tasks were executed immediately after the pipelined ones.

Figure 8.6: Comparison of the scheduler estimations and the obtained executions for three tasks: DSYRK 1521, the DSYRK 1595 and the DTRSM 1648.



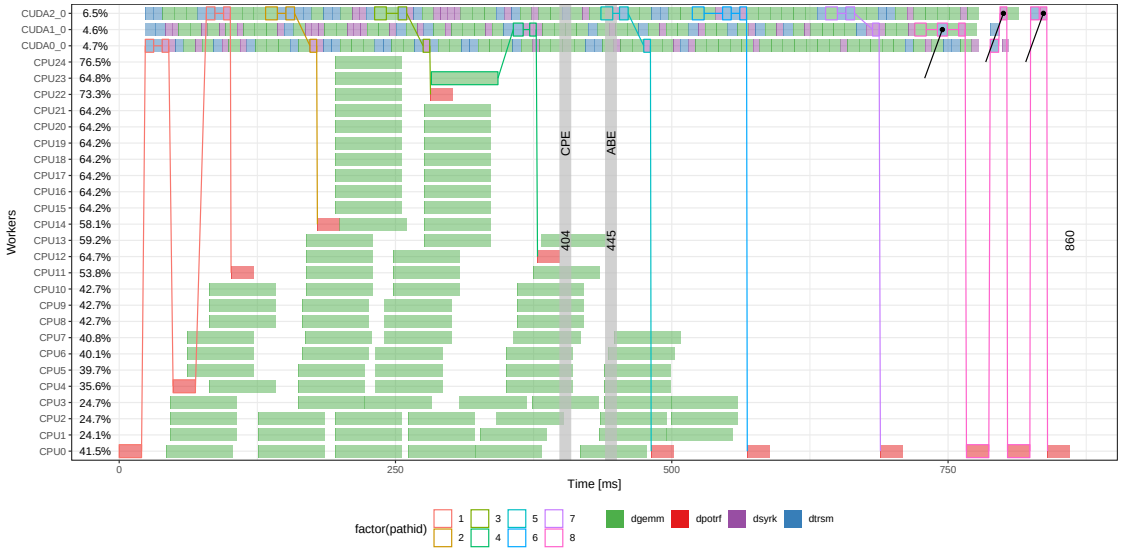
Source: The Author

8.3 Investigating Scheduling Decisions at the End of the Execution

At the end of the execution, there are few tasks to execute and the scheduler does not have much choice. The best decision is to minimize data transfers and execute each task of the critical path on the fastest resource available. In this part of the execution, the repeated simulations are more unstable with variations in the critical path of the last tasks (e.g., third and ninth execution in Figure 8.1) since there are more workers than ready tasks. We have selected three tasks (marked with black pins in Figure 8.7) that are delayed in most part of the simulated executions presented in Figure 8.1.

The first delayed task in this scenario is a DTRSM on CUDA1 with ID 1800. As

Figure 8.7: Space-time view with markers (black pins) to indicate representative delayed tasks in the end of the execution. The scheduling state of these tasks will be investigated in the next sections.



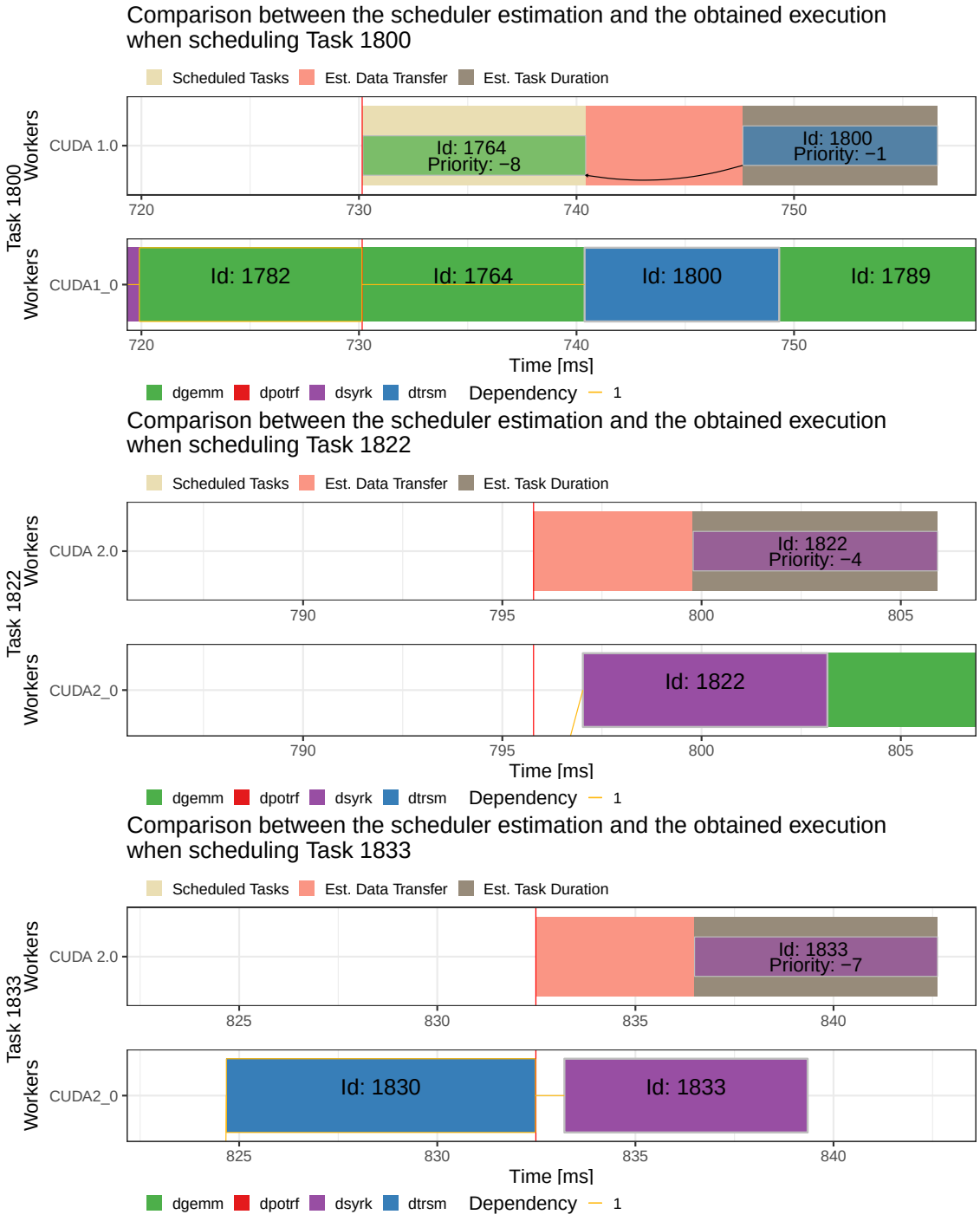
Source: The Author

showed in Figure 8.8, this task is also delayed due to the effects of the pipeline mechanism and to the required data transfers, which are the same reasons as the ones discussed in the previous section. The last two tasks, the DSYRK 1822 and on the DSYRK 1833, both on CUDA2, are delayed by the required data transfers since the worker queues are empty. At this point of the execution, there is almost no room for improvements; the scheduler is working on the critical path, executing sequences of DTRSM-DSYRK-DPOTRF tasks. Since there are only few tasks to be executed, there is not enough time to overlap the data transfer required by the new task with the execution of other ones. This way, sometimes the worker will be in idle waiting for the conclusion of a data transfer before executing the task.

8.4 Experiments with Modified Pipeline Size

As discussed in previous sections, the scheduler decisions taken by StarPU seems to be correct in terms of choice of best worker and task priority. However, some tasks are negatively impacted by the effects of the pipeline mechanism. In GPU resources, this feature allows overlapping task management with the execution of previous tasks, which in general benefits the overall performance. To check the impact of this configuration on the execution of some delayed tasks, we rerun the experiments using a smaller size for

Figure 8.8: Comparison of the scheduler estimations and the obtained executions for three tasks: DSYRK 1800, the DSYRK 1822 and the DTRSM 1833.

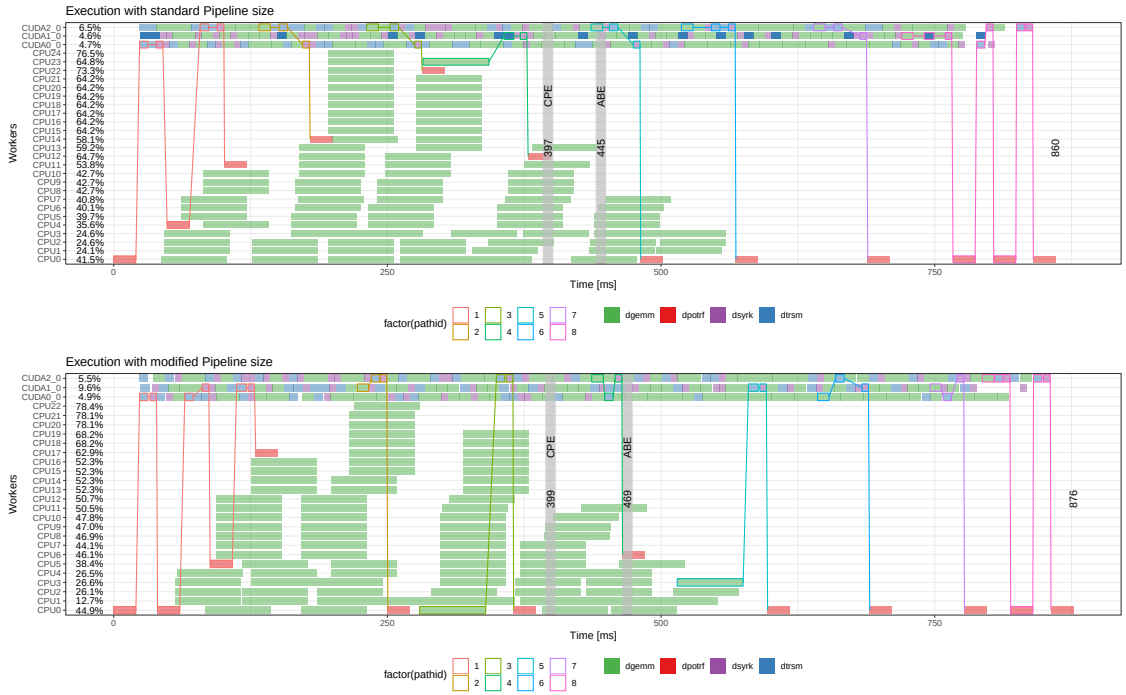


Source: The Author

the pipeline queue.

Figure 8.9 presents the space-time view of two executions, on the top, an execution using the standard pipeline size, which is 2, while on the bottom the modified version with a smaller size (1). As expected, the standard execution is faster since the pipeline mechanism improves the utilization ratio of GPU resources. In contrast, a detailed analysis shows the modified version progress deeper into the critical path, i.e., the first seven DPOTRF tasks are executed sooner in the modified version and their critical paths show fewer delayed tasks thanks to the absence of lower priority tasks in pipeline queue. After this point, StarPU schedules the tasks almost exclusively in the GPU resources, which favors the standard execution that uses these resources more efficiently thanks to the pipeline mechanism.

Figure 8.9: Comparison of a execution with the standard pipeline size (top) and the modified one (bottom).



Source: The Author

8.5 Chapter Summary

In this chapter, we investigate a set of scheduler decisions by using the previously discussed analysis strategy based on simulation and debugging. From the visual representation of the scheduler state, we compare the estimated scenario with the executed one

in order to check where the task was scheduled and how it was allocated in the worker queues.

Our analysis focus on selected tasks whose execution does not starts as soon as their last dependency has finished. We investigate the scheduler behavior regarding respect for priorities and the correct choice of the best worker. We conclude that the scheduler estimations and decisions are correct and that delays in some tasks can be explained by two other reasons: data transfers and the pipeline mechanism. For the transfers issue, we can suppose a solution would be to perform the prefetch sooner. Currently, the prefetch is performed after the moment when a task is marked as ready (all dependencies solved). To avoid this delay, the prefetch should use a speculative strategy to consider not only ready tasks but also those that will be tagged as ready in a near future, i.e., tasks whose last dependency is already running. Regarding the issues caused by the pipeline mechanism, a possible solution would be to preempt tasks with lower priorities from the *current* queue and swap them with the new (higher-priority) one since completely disable pipelining will compromises overall performance.

These suggestions for modifications were reported and discussed with the StarPU developers since the required changes demand extensive knowledge about the runtime system. However, implementing them will deeply impact the scheduling model that is currently used. As a consequence, these changes have not yet been incorporated in the source code.

9 CONCLUSIONS AND PERSPECTIVES

In this chapter, we present the final remarks of this thesis. We start with the conclusions of this work in Section 9.1. The perspectives and future work are discussed in Section 9.2. Finally, in Section 9.3 we list our accepted publications and contextualizes the StarVZ project.

9.1 Conclusions

Current HPC architectures consist of a combination of multicore processors with accelerator devices. This paradigm shift in the HPC landscape has exposed the limitations of traditional parallel programming and analysis tools. In this scenario, task-based programming models are becoming popular as a solution to handle the heterogeneity and the scalability of such platforms. Unfortunately, existing performance analysis tools are unfit to fully understand task-based executions since these ones are supported by dynamic runtime systems. This runtime support provides finer synchronizations, load balancing, automatic data transfers, and dynamic scheduling. For this reason, execution traces of task-based application can be much richer than ones of classical parallel programming applications which motivates the design of specific analysis strategies.

In this thesis, we presented performance analysis strategies for task-based applications executing on hybrid platforms supported by dynamic runtime systems. These strategies were designed to meet the requirements of such complex hardware/software stack and comprise application aspects, platform specification and runtime system state. Our strategies are built on top of modern data analysis tools and of well-established performance analysis concepts. The three main contributions are:

- a set of performance analysis strategies for task-based applications supported by dynamic runtime systems. These strategies are presented in Chapter 5 and comprise the addition of visualization layers to enrich the traditional space-time view and the inclusion of synchronized extra panels to disclose more details about the application, the runtime system, and the platform;
- an incremental and on-demand approach to extend standard visualizations with enrichments and synchronized extra panels. This approach is used to implement the strategies discussed in Chapters 5 and 7 and allows to extend and improve the anal-

ysis strategies building on existing panels without a complete redesign.

- a workflow based on simulation and debugging to inspect and analyze application parameters (e.g., priorities) and runtime scheduler decisions. This debugging approach is presented in Chapter 7 and provides a visualization of the internal state of scheduler showing the estimations computed during task scheduling.
- case-study analyses demonstrating both how to achieve performance improvements and how to understand scheduling decisions. These case studies are presented in Chapters 6 and 8.

We demonstrate the effectiveness of our strategies by analyzing execution traces from a Cholesky decomposition implemented with the StarPU task-based runtime system and running on hybrid (CPU/GPU) platforms. By using our visualization techniques, we are able to identify and fix performance issues comprising the bad task partitioning between computing resources (GPUs or cores), slow-start in distributed executions and mismanagement of MPI operations. By using our debug-based analysis strategies, we are able to inspect the scheduler state which allows us to better understand the StarPU decisions. This kind of analysis helps to confirm or refute assumptions about potential scheduling mistakes.

9.2 Perspectives

This thesis has raised several points that could progress in the future. In this section, we list some opportunities to continue and extend our research on performance analysis strategies.

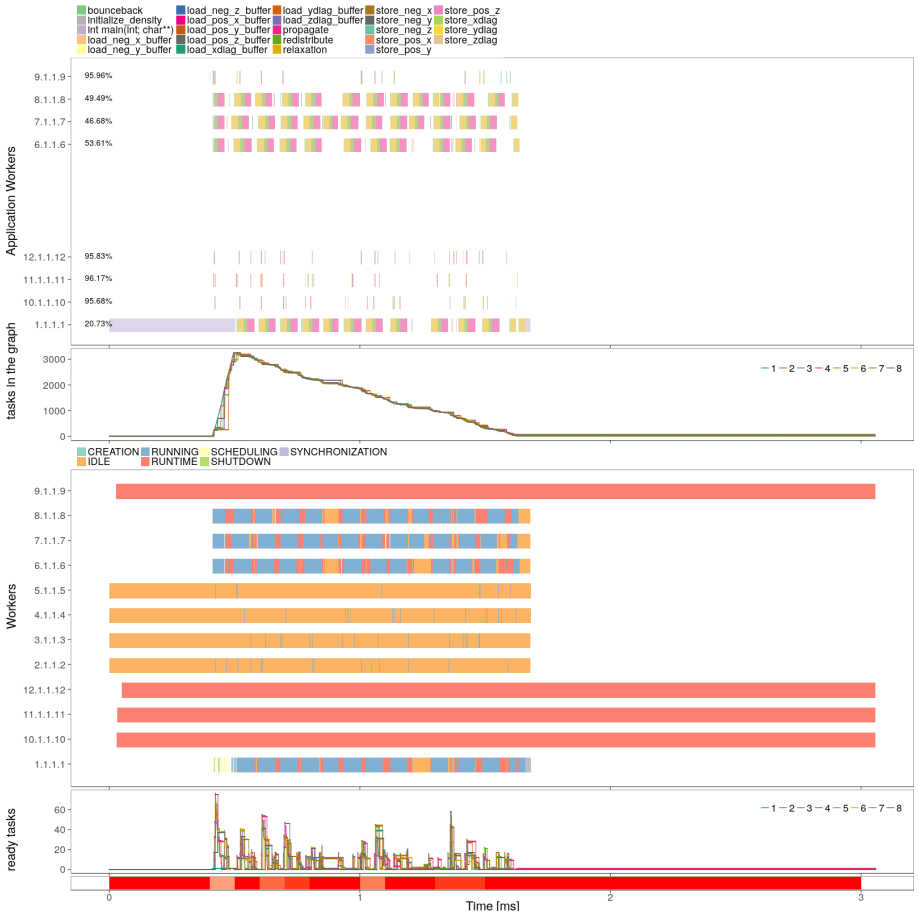
9.2.1 Analysis of other task-based applications and runtime systems

As discussed in 2.3.5, other runtime systems (e.g., XKaapi and OmpSs) are similar to StarPU, and our analysis strategies should be applicable to them as well. Figure 9.1 shows a demonstration of how our analysis strategies can be used to build visualizations from traces of the OmpSs runtime system. OmpSs traces are rich enough to enable us to build visualizations with the same panels as StarPU ones.

It must be highlighted that this visualization is built from a non-optimized and under development application, which explains the bad resources occupation for the time

being. This application provides a CPU-GPU task-based implementation for the D3Q19 Lattice-Boltzmann Method (SCHEPKE; MAILLARD; NAVAUX, 2009) which is a numerical method for simulating viscous fluid flow. As future work, we plan to continue porting our visualization strategies to this runtime system since they can be useful during the application optimization.

Figure 9.1: Visualization of a task-based application executed with the OmpSs runtime system.



Source: The Author

9.2.2 Performance Anomalies on Xeon Phi Knights Landing architecture

GPUs are the dominant accelerator technology for the time being, however other technologies are also used (see Figure 2.2). One of these accelerators is the Intel Xeon Phi Knights Landing (JEFFERS; REINDERS, J.; SODANI, 2016) which is an x86-based many-core processor with up to 72 physical cores and up to 288 hardware threads. In this generation, Xeon Phi processors can be used as the main processor of the system.

To demonstrate the usability of our performance analysis strategies in platforms with such accelerator technology, we execute the tiled Cholesky decomposition provided by Chameleon and implemented with StarPU. We use a platform with a 68-core Intel Xeon Phi 7250 processor with 110 GB of DDR4 memory. Since such platform is very homogeneous, we expect a uniform execution with very few performance disturbances. However, in practice, we observe several curious issues affecting several cores.

Figure 9.2 shows an execution with the DMDAS scheduler on this platform. To reduce potential perturbations, we disabled the hardware threads and dedicated one physical core to the main thread of StarPU. This way, the application is executed with 67 cores. Despite that, we can observe several cores where almost all tasks are tagged as outliers. The idleness ratio varies from 1.54% to 3.91% which suggests some unequal load balancing. As future work, we plan a further investigation to identify the cause and fix these performance issues which can be useful not only to this generation but also give us some insights to tackle future many-core architectures.

9.2.3 StarVZ

Most parts of the visualization strategies proposed in this work are being reimplemented and integrated into an open source project called StarVZ[1]. This project is being conducted in a cooperative effort with other participants. Some figures in Section 6.3 were already built using the new StarVZ code, while others (Sections 5 and 6.2) uses the original prototype code.

For the time being, StarVZ is being extended to support more accurate bounds for the makespan, which probably will be more precise, in particular, for small executions as the ones of Sections 6.2.2 and 7.1.

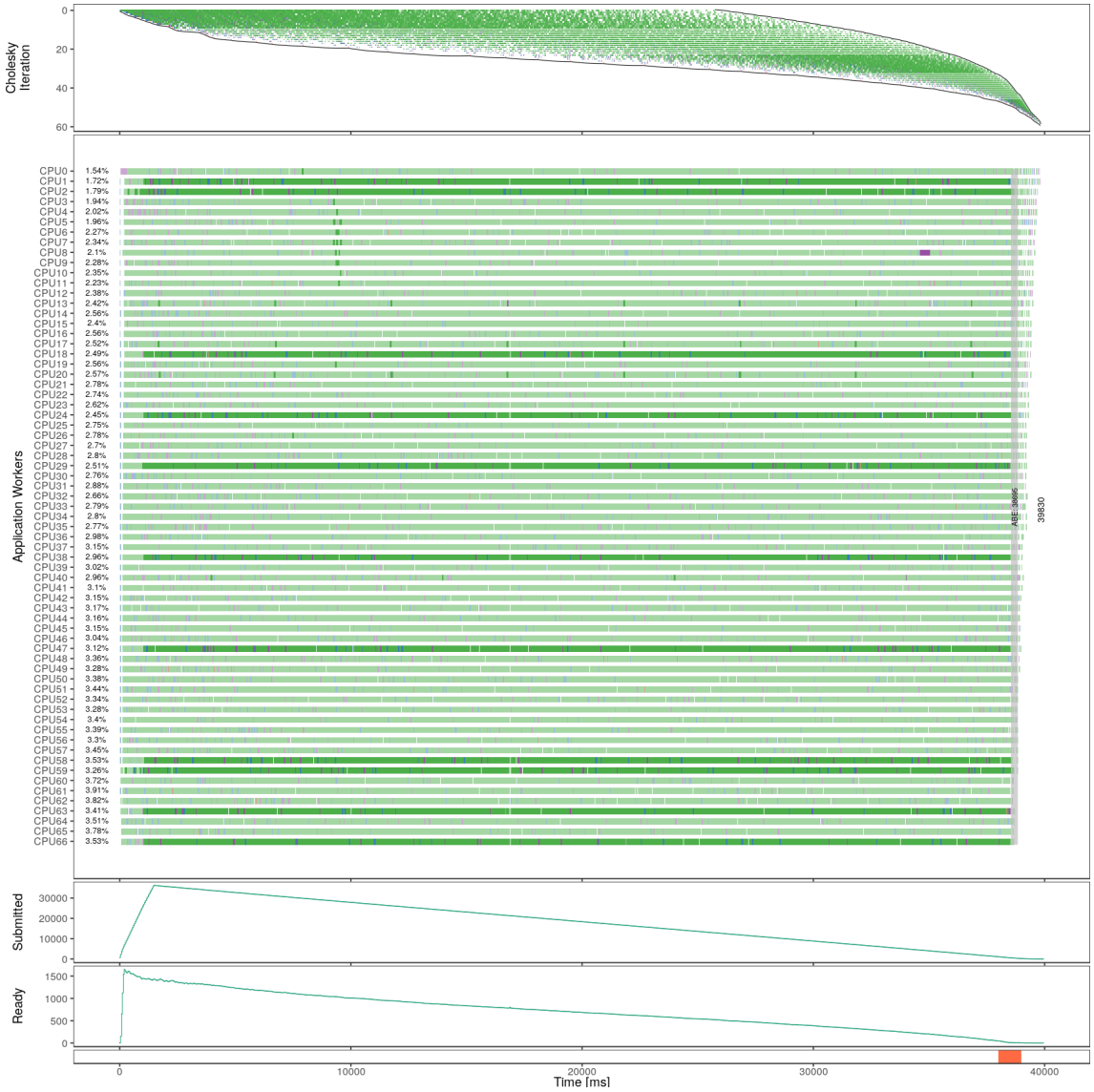
9.3 Publications

9.3.1 Accepted

- PINTO, Vinicius Garcia; STANISIC, Luka; LEGRAND, Arnaud; SCHNORR, Lucas Mello; THIBAULT, Samuel; DANJEAN, Vincent. Analyzing Dynamic Task-

[1] `https://github.com/schnorr/starvz`

Figure 9.2: An overview of a Cholesky execution on the Intel Xeon Phi platform with a matrix of 60×60 tiles of 960×960 using the DMDAS scheduler.



Source: The Author

Based Applications on Hybrid Platforms: An Agile Scripting Approach. In: PROCEEDINGS of the Third Workshop on Visual Performance Analysis, VPA@SC 2016, Salt Lake, UT, USA, November 18, 2016. [S.l.]: IEEE Press, 2016. (VPA '16), p. 17–24. Held in conjunction with SC16. DOI: 10.1109/VPA.2016.008

- PINTO, Vinícius Garcia; SCHNORR, Lucas Mello; STANISIC, Luka; LEGRAND, Arnaud; THIBAULT, Samuel; DANJEAN, Vincent. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. **Concurrency and Computation: Practice and Experience**, v. 0, n. 0, p. 1–27. DOI: 10.1002/cpe.4472
- PINTO, Vinicius Garcia; STANISIC, Luka; LEGRAND, Arnaud; SCHNORR, Lucas Mello; THIBAULT, Samuel; DANJEAN, Vincent. Detecção de Anomalias de Desempenho em Aplicações de Alto Desempenho baseadas em Tarefas em Clusters Híbridos. In: ANAIS do CSBC 2018 - 17º WPERFORMANCE - WORKSHOP EM DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMUNICAÇÃO. [s.l.: s.n.], 2018. p. 97–110

9.3.2 Other accepted publications during the Ph.D.

These articles were also published during the Ph.D. but their content is not directly related to the subject of this thesis.

- PINTO, Vinícius Garcia; LORENZON, Arthur F.; BECK, Antonio Carlos S.; MAILLARD, Nicolas; NAVAUX, Philippe O. A. Energy Efficiency Evaluation of Multilevel Parallelism on Low Power Processors. In: ANAIS do XXXIV Congresso da Sociedade Brasileira de Computação (WPerfomance - XIII Workshop em Desempenho de Sistemas Computacionais e de Comunicação). Porto Alegre: [s.n.], 2014. p. 1825–1836
- PINTO, Vinícius Garcia; HERBSTRITH, Vinicius Alves; SCHNORR, Lucas M. Replicating the Performance Evaluation of an N-Body Application on a Manycore Accelerator. In: 2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW). [s.l.: s.n.], Oct. 2015. p. 19–24. DOI: 10.1109/SBAC-PADW.2015.17

BIBLIOGRAPHY

A.P., Davison et al. Sumatra: A Toolkit for Reproducible Research. In: STODDEN, V.; LEISCH, F.; PENG, R.D. (Eds.). **Implementing Reproducible Research**. Boca Raton, Florida.: Chapman & Hall/CRC, Mar. 2014. p. 57–79.

ABDULAH, Sameh et al. ExaGeoStat: A High Performance Unified Software for Geostatistics on Manycore Systems. **IEEE Transactions on Parallel and Distributed Systems**, IEEE Computer Society, n. 1, p. 1–1, 2018. DOI: 10.1109/TPDS.2018.2850749.

AGULLO, E.; AUMAGE, O., et al. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. **IEEE Transactions on Parallel and Distributed Systems**, PP, n. 99, p. 1–1, 2017. DOI: 10.1109/TPDS.2017.2766064.

AGULLO, E.; BEAUMONT, O., et al. Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop. [S.l.: s.n.], May 2015. p. 34–45. DOI: 10.1109/IPDPSW.2015.35.

AGULLO, E.; BOSILCA, G., et al. Poster: Matrices over Runtime Systems at Exascale. In: _____. **High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:** [s.l.: s.n.], Nov. 2012. p. 1332–1332. DOI: 10.1109/SC.Companion.2012.168.

AGULLO, Emmanuel et al. Implementing Multifrontal Sparse Solvers for Multicore Architectures with Sequential Task Flow Runtime Systems. **ACM Trans. Math. Softw.**, ACM, New York, NY, USA, v. 43, n. 2, 13:1–13:22, Aug. 2016. DOI: 10.1145/2898348.

_____. **qr-mumps**. [S.l.: s.n.]. Available from: <http://buttari.perso.enseeiht.fr/qrmumps/>. Visited on: 2 July 2018.

_____. Task-Based Multifrontal QR Solver for GPU-Accelerated Multicore Architectures. In: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC). [S.l.]: IEEE, Dec. 2015. p. 54–63. DOI: 10.1109/HiPC.2015.27.

ASANOVIC, Krste et al. A View of the Parallel Computing Landscape. **Commun. ACM**, ACM, New York, NY, USA, v. 52, n. 10, p. 56–67, Oct. 2009. DOI: 10.1145/1562764.1562783.

AUGONNET, Cédric et al. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In: **Recent Advances in the Message Passing Interface: 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings**. Ed. by Jesper Larsson Träff, Siegfried Benkner and Jack J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 298–299. DOI: 10.1007/978-3-642-33518-1_40.

AUGONNET, Cédric et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Ltd., v. 23, n. 2, p. 187–198, 2011.

AYGUADÉ, Eduard et al. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: [s.l.]: Springer, Berlin, Heidelberg, 2009. p. 851–862. DOI: 10.1007/978-3-642-03869-3_79.

BADIA, Rosa M.; HERRERO, José R., et al. Parallelizing dense and banded linear algebra libraries using SMPSs. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Ltd., v. 21, n. 18, p. 2438–2456, Dec. 2009. DOI: 10.1002/cpe.1463.

BADIA, Rosa M.; LABARTA, Jesús, et al. Programming Grid Applications with GRID Superscalar. **Journal of Grid Computing**, Kluwer Academic Publishers, v. 1, n. 2, p. 151–170, 2003. DOI: 10.1023/B:GRID.0000024072.93701.f3.

BELLENS, Pieter et al. CellSs: a Programming Model for the Cell BE Architecture. In: ACM/IEEE SC 2006 Conference (SC'06). [S.l.]: IEEE, Nov. 2006. p. 5–5. DOI: 10.1109/SC.2006.17.

BENDER, Michael A.; RABIN, Michael O. Scheduling Cilk Multithreaded Parallel Programs on Processors of Different Speeds. In: PROCEEDINGS of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures. Bar Harbor, Maine, USA: ACM, 2000. (SPAA '00), p. 13–21. DOI: 10.1145/341800.341803.

BERKELAAR, Michel et al. **lpSolve: Interface to 'Lp_solve' v. 5.5 to Solve Linear/Integer Programs**. [S.l.], 2015. R package version 5.6.13. Available from: <https://CRAN.R-project.org/package=lpSolve>. Visited on: 29 Aug. 2018.

BLACKFORD, L. S. et al. **ScaLAPACK user's guide**. [S.l.]: Society for Industrial and Applied Mathematics, 1997.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. In: PROCEEDINGS of the 35th Annual Symposium on Foundations of Computer Science. Washington, DC, USA: IEEE Computer Society, 1994. (SFCS '94), p. 356–368. DOI: 10.1109/SFCS.1994.365680.

BLUMOFE, Robert D.; LEISERSON, Charles E. Scheduling Multithreaded Computations by Work Stealing. **J. ACM**, ACM, New York, NY, USA, v. 46, n. 5, p. 720–748, Sept. 1999. DOI: 10.1145/324133.324234.

BLUMOFE, Robert D et al. Cilk: An efficient multithreaded runtime system. **Journal of parallel and distributed computing**, Elsevier, v. 37, n. 1, p. 55–69, 1996.

BOSILCA, G. et al. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. [S.l.: s.n.], May 2011. p. 1432–1441. DOI: 10.1109/IPDPS.2011.299.

BOSILCA, George et al. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. **Parallel Computing**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 38, n. 1-2, p. 37–51, Jan. 2012. DOI: 10.1016/j.parco.2011.10.003.

BRENDEL, R. et al. Edge Bundling for Visualizing Communication Behavior. In: 2016 Third Workshop on Visual Performance Analysis (VPA). [S.l.: s.n.], Nov. 2016. p. 1–8. DOI: 10.1109/VPA.2016.006.

BRINKMANN, Steen et al. **TEMANEJO 1.3 - Manual**. [S.l.], 2017. p. 27. Available from: <https://fs.hlrs.de/projects/temanejo/Temanejo_manual-1.3.pdf>. Visited on: 31 Jan. 2018.

BRINKMANN, Steffen; GRACIA, José; NIETHAMMER, Christoph. Task Debugging with TEMANEJO. In: CHEPTSOV, Alexey et al. (Eds.). **Tools for High Performance Computing 2012**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 13–21. DOI: 10.1007/978-3-642-37349-7_2.

BSC. **Mercurium Compiler**. [S.l.: s.n.], 2018. Available from: <https://pm.bsc.es/mcxx>. Visited on: 4 Jan. 2018.

_____. **Nanos++ Runtime**. [S.l.: s.n.], 2018. Available from: <https://pm.bsc.es/nanox>. Visited on: 4 Jan. 2018.

BSC. **OmpSs User Guide**. [S.l.: s.n.], June 2017. Available from: <https://pm.bsc.es/ompss-docs/user-guide/OmpSsUserGuide.pdf>. Visited on: 4 Jan. 2018.

_____. **Programming with OmpSs**. [S.l.: s.n.], June 2017. Available from: <https://pm.bsc.es/ompss-docs/book/ProgrammingWithOmpSs.pdf>. Visited on: 4 Jan. 2018.

CALLAHAN, Steven P. et al. VisTrails: Visualization Meets Data Management. In: PROCEEDINGS of the 2006 ACM SIGMOD International Conference on Management of Data. Chicago, IL, USA: ACM, 2006. (SIGMOD '06), p. 745–747. DOI: 10.1145/1142473.1142574.

CASANOVA, Henri 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, June 2014. DOI: 10.1016/j.jpdc.2014.06.008.

CEBALLOS, Germán et al. Analyzing performance variation of task schedulers with TaskInsight. **Parallel Computing**, North-Holland, v. 75, p. 11–27, July 2018. DOI: 10.1016/J.PARCO.2018.02.003.

CEDERMAN, D; TSIGAS, P. Dynamic Load Balancing Using Work-Stealing. In: HWU, Wen-Mei W (Ed.). **GPU Computing Gems: Jade Edition**. [S.l.]: Elsevier, 2011. p. 485–499.

CERN; OPENAIRE; COMMISSION, European. **Zenodo - Research. Shared.** [S.l.: s.n.]. Available from: <https://zenodo.org>. Visited on: 18 Jan. 2018.

CHAPMAN, B. et al. **Using OpenMP: Portable Shared Memory Parallel Programming**. [S.l.]: MIT Press, 2008. (Scientific Computation Series, v. 10).

CHEN, T. et al. Cell Broadband Engine Architecture and its first implementation—A performance view. **IBM Journal of Research and Development**, v. 51, n. 5, p. 559–572, Sept. 2007. DOI: 10.1147/rd.515.0559.

CHIRIGATI, Fernando et al. ReproZip: Computational Reproducibility With Ease. In: PROCEEDINGS of the 2016 International Conference on Management of Data. San Francisco, California, USA: ACM, 2016. (SIGMOD '16), p. 2085–2088. DOI: 10.1145/2882903.2899401.

CORMEN, T.H. et al. **Introduction to Algorithms**. [S.l.]: MIT Press, 2009.

COULOMB, Kevin et al. **Visual trace explorer (ViTE)**. [S.l.: s.n.], 2009.

COUTEYEN CARPAYE, Jean Marie; ROMAN, Jean; BRENNER, Pierre. Design and analysis of a task-based parallelization over a runtime system of an explicit finite-volume CFD code with adaptive time stepping. **Journal of Computational Science**, Elsevier, Mar. 2017. DOI: 10.1016/J.JOCS.2017.03.008.

DANALIS, A. et al. PTG: An Abstraction for Unhindered Parallelism. In: 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing. [S.l.: s.n.], Nov. 2014. p. 21–30. DOI: 10.1109/WOLFHPC.2014.8.

DANJEAN, Vincent; NAMYST, Raymond; WACRENIER, Pierre-André. An Efficient Multi-level Trace Toolkit for Multi-threaded Applications. In: _____. **Euro-Par 2005 Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 166–175.

DATTA, B.N. **Numerical Linear Algebra and Applications, Second Edition**. [S.l.]: Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2010. (Other Titles in Applied Mathematics).

DINAN, James et al. Scalable Work Stealing. In: PROCEEDINGS of the Conference on High Performance Computing Networking, Storage and Analysis. Portland, Oregon: ACM, 2009. (SC '09), 53:1–53:11. DOI: 10.1145/1654059.1654113.

DOMINIK, Carsten. **The Org Mode 7 Reference Manual - Organize Your Life with GNU Emacs**. [S.l.]: Network Theory Ltd., 2010.

DONGARRA, J. et al. With Extreme Computing, the Rules Have Changed. **Computing in Science Engineering**, v. 19, n. 3, p. 52–62, May 2017. DOI: 10.1109/MCSE.2017.48.

DOSIMONT, Damien; CORRE, Youenn, et al. Ocelotl: Large Trace Overviews Based on Multidimensional Data Aggregation. In: NIETHAMMER, Christoph et al. (Eds.). **Tools for High Performance Computing 2014**. [S.l.]: Springer International Publishing, 2015. p. 137–160. DOI: 10.1007/978-3-319-16012-2_7.

DOSIMONT, Damien; LAMARCHE-PERRIN, Robin, et al. **A Spatiotemporal Data Aggregation Technique for Performance Analysis of Large-scale Execution Traces**. en. [S.l.: s.n.], Sept. 2014.

DURAN, Alejandro; AYGUADÉ, Eduard, et al. OmpSs: A Proposal For Programming Heterogeneous Multi-Core Architectures. **Parallel Processing Letters**, v. 21, n. 02, p. 173–193, 2011.

DURAN, Alejandro; KLEMM, Michael. The Intel® Many Integrated Core Architecture. In: 2012 International Conference on High Performance Computing & Simulation (HPCS). [S.l.]: IEEE, July 2012. p. 365–366. DOI: 10.1109/HPCSim.2012.6266938.

ESCHWEILER, Dominic et al. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In: APPLICATIONS, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium. [S.l.: s.n.], 2011. p. 481–490. DOI: 10.3233/978-1-61499-041-3-481.

FAVERGE, Mathieu et al. **PaStiX Parallel Sparse direct Solver**. [S.l.: s.n.]. Available from: <https://gitlab.inria.fr/solverstack/pastix>. Visited on: 2 July 2018.

FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. [S.l.]: Addison-Wesley, 1995. (Literature and Philosophy).

FRIGO, Matteo; LEISERSON, Charles E.; RANDALL, Keith H. The Implementation of the Cilk-5 Multithreaded Language. In: PROCEEDINGS of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. Montreal, Quebec, Canada: ACM, 1998. (PLDI '98), p. 212–223. DOI: 10.1145/277650.277725.

GAMBLIN, Todd. **Spack 0.10 documentation**. [S.l.: s.n.], 2018. Available from: <https://spack.readthedocs.io/>. Visited on: 15 Jan. 2018.

GAMBLIN, Todd et al. The Spack package manager: Bringing order to HPC software chaos. In: IEEE. HIGH Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for. [S.l.: s.n.], 2015. p. 1–12.

GAREY, M. R.; GRAHAM, R. L. Bounds for multiprocessor scheduling with resource constraints. **SIAM Journal on Computing**, SIAM, v. 4, n. 2, p. 187–200, 1975.

GASTER, Benedict. **Heterogeneous computing with OpenCL**. [S.l.]: Morgan Kaufmann, 2012. p. 277.

GAUTIER, T. et al. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. [S.l.: s.n.], May 2013. p. 1299–1308.

GAUTIER, Thierry; BESSERON, Xavier; PIGEON, Laurent. KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-processors. In: PROCEEDINGS of the 2007 International Workshop on Parallel Symbolic Computation. London, Ontario, Canada: ACM, 2007. (PASCO '07), p. 15–23. DOI: 10.1145/1278177.1278182.

GAUTIER, Thierry; FERREIRA LIMA, Joao Vicente, et al. Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures. In: 6TH WORKSHOP ON PROGRAMMABILITY ISSUES FOR HETEROGENEOUS MULTICORES (MULTIPROG). Berlin, Germany: [s.n.], Jan. 2013.

GAVISH, Matan; DONOHO, David. A Universal Identifier for Computational Results. **Procedia Computer Science**, v. 4, p. 637–647, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011. DOI: 10.1016/j.procs.2011.04.067.

GRAHAM, R. L. Bounds for Certain Multiprocessing Anomalies. **Bell System Technical Journal**, Blackwell Publishing Ltd, v. 45, n. 9, p. 1563–1581, Nov. 1966. DOI: 10.1002/j.1538-7305.1966.tb01709.x.

_____. Bounds on Multiprocessing Timing Anomalies. **SIAM Journal on Applied Mathematics**, Society for Industrial and Applied Mathematics, v. 17, n. 2, p. 416–429, 1969.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI: Portable Parallel Programming with the Message-Passing Interface**. [S.l.]: MIT Press, 2014. (Scientific and Engineering Computation).

GROUP, MIT CSAIL Supertech Research. **The Cilk Project**. [S.l.: s.n.]. Available from: <http://supertech.csail.mit.edu/cilk/>. Visited on: 22 Nov. 2017.

GUO, Yi et al. Work-first and help-first scheduling policies for async-finish task parallelism. In: 2009 IEEE International Symposium on Parallel Distributed Processing. [S.l.: s.n.], May 2009. p. 1–12. DOI: 10.1109/IPDPS.2009.5161079.

HAUGEN, Blake et al. Visualizing Execution Traces with Task Dependencies. In: PROCEEDINGS of the 2Nd Workshop on Visual Performance Analysis. Austin, Texas: ACM, 2015. (VPA '15), 2:1–2:8. DOI: 10.1145/2835238.2835240.

HENNESSY, J.L.; PATTERSON, D.A. **Computer Architecture: A Quantitative Approach**. [S.l.]: Elsevier Science, 2017. (The Morgan Kaufmann Series in Computer Architecture and Design).

HENON, P.; RAMET, P.; ROMAN, J. PaStiX: a high-performance parallel direct solver for sparse symmetric positive definite systems. **Parallel Computing**, North-Holland, v. 28, n. 2, p. 301–321, Feb. 2002. DOI: 10.1016/S0167-8191(01)00141-7.

HERMANN, Everton et al. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In: **Euro-Par 2010 - Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II**. Ed. by Pasqua D'Ambra, Mario Guarracino and Domenico Talia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 235–246. DOI: 10.1007/978-3-642-15291-7_23.

HESS, Joey. **git-annex**. [S.l.: s.n.]. Available from: <https://git-annex.branchable.com/>. Visited on: 17 Jan. 2018.

HOQUE, Reazul et al. Dynamic task discovery in PaRSEC. In: PROCEEDINGS of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems - ScalA '17. New York, New York, USA: ACM Press, 2017. p. 1–8. DOI: 10.1145/3148226.3148233.

HOWELL, M. **Homebrew, the missing package manager for OS X**. [S.l.: s.n.]. Available from: <https://brew.sh/>. Visited on: 15 Jan. 2018.

HUYNH, A.; TAURA, K. Delay Spotter: A Tool for Spotting Scheduler-Caused Delays in Task Parallel Runtime Systems. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). [S.l.: s.n.], Sept. 2017. p. 114–125. DOI: 10.1109/CLUSTER.2017.82.

HUYNH, An et al. DAGViz: A DAG Visualization Tool for Analyzing Task-parallel Program Traces. In: PROCEEDINGS of the 2Nd Workshop on Visual Performance Analysis. Austin, Texas: ACM, 2015. (VPA '15), 3:1–3:8. DOI: 10.1145/2835238.2835241.

INRIA. **KSTAR OpenMP compiler**. [S.l.: s.n.], 2018. Available from: <http://kstar.gforge.inria.fr>. Visited on: 5 Jan. 2018.

INRIA; CNRS; UNIVERSITÉ DE BORDEAUX. **StarPU Handbook**. [S.l.: s.n.], 2017. Available from: <http://starpu.gforge.inria.fr/doc/html>. Visited on: 5 Jan. 2018.

INRIA; UTK; UCD; KAUST. **Chameleon: A dense linear algebra software for heterogeneous architectures**. [S.l.: s.n.]. Available from: <https://project.inria.fr/chameleon/>. Visited on: 2 July 2018.

INTEL. **A Guide to Vectorization with Intel® C++ Compilers**. [S.l.: s.n.]. Available from: <https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf>. Visited on: 4 Dec. 2017.

_____. **Intel® Cilk™ Plus**. [S.l.: s.n.]. Available from: <https://www.cilkplus.org/>. Visited on: 22 Nov. 2017.

ISAACS, K. E. et al. Combing the Communication Hairball: Visualizing Parallel Execution Traces using Logical Time. **IEEE Transactions on Visualization and Computer Graphics**, v. 20, n. 12, p. 2349–2358, Dec. 2014. DOI: 10.1109/TVCG.2014.2346456.

ISAACS, Kate. **LLNL/ravel: Ravel MPI trace visualization tool**. [S.l.: s.n.]. Available from: <https://github.com/LLNL/ravel>. Visited on: 7 Feb. 2018.

ISAACS, Katherine E. et al. State of the Art of Performance Visualization. In: _____. **EuroVis - STARs**. [S.l.]: The Eurographics Association, 2014. DOI: 10.2312/eurovisstar.20141177.

JEFFERS, J.; REINDERS, J. **Intel Xeon Phi Coprocessor High Performance Programming**. [S.l.]: Elsevier Science, 2013.

JEFFERS, J.; REINDERS, J.; SODANI, A. **Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition**. [S.l.]: Elsevier Science, 2016. Available from: <https://books.google.com.br/books?id=DDpUCwAAQBAJ>.

KELLER, Rainer et al. Temanejo: Debugging of Thread-Based Task-Parallel Programs in StarSS. In: **Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden**. Ed. by Holger Brunst. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 131–137. DOI: 10.1007/978-3-642-31476-6_11.

KERGOMMEAUX, Jacques Chassin de; OLIVEIRA STEIN, Benhur de. Pajé: An Extensible Environment for Visualizing Multi-threaded Programs Executions. In: _____. **Euro-Par 2000 Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. p. 133–140.

KNÜPFER, Andreas et al. The Vampir Performance Analysis Tool-Set. In: **Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart**. Ed. by Michael Resch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 139–155. DOI: 10.1007/978-3-540-68564-7_9.

KNUTH, D. E. Literate Programming. **The Computer Journal**, Oxford University Press, v. 27, n. 2, p. 97–111, Feb. 1984. DOI: 10.1093/comjnl/27.2.97.

LABARTA, Jesús. **The OmpSs programming model and its runtime support**. [S.l.: s.n.], May 2015. Available from: <https://charm.cs.illinois.edu/workshops/charmWorkshop2015/slides/CharmWorkshop2015_keynote_jesus.pdf>. Visited on: 4 Jan. 2018.

LEON, S.J. **Linear Algebra with Applications**. [S.l.]: Pearson, 2014. (Featured Titles for Linear Algebra).

LIMA, João V.F. et al. Design and Analysis of Scheduling Strategies for Multi-CPU and Multi-GPU Architectures. **Parallel Computing**, North-Holland, v. 44, p. 37–52, Mar. 2015. DOI: 10.1016/j.parco.2015.03.001.

MELLO SCHNORR, Lucas. **Some Visualization Models applied to the Analysis of Parallel Applications**. 2009. PhD thesis. DOI: 10183/37179.

MEUER, Hans Werner et al. **The TOP500: History, Trends, and Future Directions in High Performance Computing**. 1st. [S.l.]: Chapman & Hall/CRC, 2014.

MOR, Stefano; MAILLARD, Nicolas. Dynamic Workload Balancing Deques for Branch and Bound Algorithms in the Message Passing Interface. **Int. J. High Perform. Syst. Archit.**, Inderscience Publishers, Inderscience Publishers, Geneva, SWITZERLAND, v. 3, n. 2/3, p. 77–86, May 2011. DOI: 10.1504/IJHPSA.2011.040461.

MUNAFÒ, Marcus R. et al. A manifesto for reproducible science. **Nature Human Behaviour**, Nature Publishing Group, v. 1, n. 1, p. 0021, Jan. 2017. DOI: 10.1038/s41562-016-0021.

NIELSEN, Lars Holm. **Sharing your data and software on Zenodo**. [S.l.: s.n.], May 2017. DOI: 10.5281/zenodo.802100. Available from: <https://doi.org/10.5281/zenodo.802100>. Visited on: 18 Jan. 2018.

NVIDIA. **NVIDIA CUDA C Programming Guide v9.0**. [S.l.: s.n.], 2017. p. 300. Available from: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Visited on: 11 Dec. 2017.

_____. **PARALLEL THREAD EXECUTION ISA v6.0**. [S.l.: s.n.], 2017. p. 302. Available from: <http://docs.nvidia.com/cuda/pdf/ptx%7B%5C_%7Disa%7B%5C_%7D6.0.pdf>. Visited on: 11 Dec. 2017.

_____. **VampirTrace | NVIDIA Developer**. [S.l.: s.n.], 2018. Available from: <https://developer.nvidia.com/vampirtrace>. Visited on: 6 Feb. 2018.

OPENMP Application Program Interface: Version 3.0, May 2008. [S.l.]: OpenMP Architecture Review Board, 2008. Available from: <http://www.openmp.org/wp-content/uploads/spec30.pdf>. Visited on: 22 Nov. 2017.

OPENMP Application Program Interface: Version 4.0, July 2013. [S.l.]: OpenMP Architecture Review Board, 2013. Available from: <http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>. Visited on: 9 Mar. 2018.

PAGANO, Generoso et al. Trace Management and Analysis for Embedded Systems. In: 2013 IEEE 7th International Symposium on Embedded Multicore Socs. [S.l.]: IEEE, Sept. 2013. p. 119–122. DOI: 10.1109/MCSoC.2013.28.

PILLET, V. et al. PARAVER: A Tool to Visualize and Analyze Parallel Code. In: _____. **Proceedings of WoTUG-18: Transputer and occam Developments**. [S.l.: s.n.], Mar. 1995. p. 17–31.

PINTO, Vinicius Garcia. **Escalonamento por roubo de tarefas em sistemas Multi-CPU e Multi-GPU**. Mar. 2013. MA thesis – Programa de Pós-Graduação em Computação, Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre. DOI: 10183/71270.

PINTO, Vinícius Garcia; HERBSTRITH, Vinicius Alves; SCHNORR, Lucas M. Replicating the Performance Evaluation of an N-Body Application on a Manycore Accelerator. In: 2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW). [S.l.: s.n.], Oct. 2015. p. 19–24. DOI: 10.1109/SBAC-PADW.2015.17.

PINTO, Vinícius Garcia; LORENZON, Arthur F.; BECK, Antonio Carlos S.; MAILLARD, Nicolas; NAVAUX, Philippe O. A. Energy Efficiency Evaluation of Multi-level Parallelism on Low Power Processors. In: ANAIS do XXXIV Congresso da Sociedade

Brasileira de Computação (WPerfomance - XIII Workshop em Desempenho de Sistemas Computacionais e de Comunicação). Porto Alegre: [s.n.], 2014. p. 1825–1836.

PINTO, Vinicius Garcia; MAILLARD, Nicolas. Work Stealing on Hybrid Architectures. In: 13TH Symposium on Computer Systems (WSCAD-SSC 2012). Los Alamitos: IEEE Computer Society, Oct. 2012. p. 17–24. DOI: 10.1109/WSCAD-SSC.2012.28.

PINTO, Vinícius Garcia; SCHNORR, Lucas Mello; STANISIC, Luka; LEGRAND, Arnaud; THIBAULT, Samuel; DANJEAN, Vincent. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. **Concurrency and Computation: Practice and Experience**, v. 0, n. 0, p. 1–27. DOI: 10.1002/cpe.4472.

PINTO, Vinicius Garcia; STANISIC, Luka; LEGRAND, Arnaud; SCHNORR, Lucas Mello; THIBAULT, Samuel; DANJEAN, Vincent. Analyzing Dynamic Task-Based Applications on Hybrid Platforms: An Agile Scripting Approach. In: PROCEEDINGS of the Third Workshop on Visual Performance Analysis, VPA@SC 2016, Salt Lake, UT, USA, November 18, 2016. [S.l.]: IEEE Press, 2016. (VPA '16), p. 17–24. Held in conjunction with SC16. DOI: 10.1109/VPA.2016.008.

_____. Detecção de Anomalias de Desempenho em Aplicações de Alto Desempenho baseadas em Tarefas em Clusters Híbridos. In: ANAIS do CSBC 2018 - 17º WPERFORMANCE - WORKSHOP EM DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMUNICAÇÃO. [S.l.: s.n.], 2018. p. 97–110.

PRESS, W.H. et al. **Numerical Recipes 3rd Edition: The Art of Scientific Computing**. [S.l.]: Cambridge University Press, 2007.

R CORE TEAM. **R: A Language and Environment for Statistical Computing**. Vienna, Austria, 2018. Available from: <https://www.R-project.org/>. Visited on: 29 Aug. 2018.

REINDERS, James. **Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism**. Sebastopol, USA: O'Reilly & Associates, Inc., 2007.

SANDERS, J.; KANDROT, E. **CUDA by Example: An Introduction to General-Purpose GPU Programming**. [S.l.]: Pearson Education, 2010.

SCHEPKE, Claudio; MAILLARD, Nicolas; NAVAUX, Philippe O. A. Parallel Lattice Boltzmann Method with Blocked Partitioning. **International Journal of Parallel Pro-**

gramming, v. 37, n. 6, p. 593–611, Dec. 2009. DOI: 10.1007/s10766-009-0113-x.

SCHNORR, Lucas M.; LEGRAND, Arnaud. Visualizing More Performance Data Than What Fits on Your Screen. In: _____. **Tools for High Performance Computing 2012**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 149–162.

SCHNORR, Lucas Mello et al. **The Paje trace file format**. [S.l.], 2016.

SCHULTE, Eric et al. A multi-language computing environment for literate programming and reproducible research. **Journal of Statistical Software**, Foundation for Open Access Statistics, v. 46, n. 3, p. 1–24, 2012.

SIEVERT, Carson et al. **plotly: Create Interactive Web Graphics via 'plotly.js'**. [S.l.], 2017. R package version 4.7.1. Available from: <https://CRAN.R-project.org/package=plotly>. Visited on: 29 Aug. 2018.

SOTTILE, Matthew; MATTSON, Timothy G.; RASMUSSEN, Craig E. **Introduction to Concurrency in Programming Languages**. 1st. [S.l.]: Chapman & Hall/CRC, 2009.

STALLMAN, Richard et al. **GNU Emacs Manual**. 17. ed. Boston, USA: Free Software Foundation, 2017. p. 635. Available from: <https://www.gnu.org/software/emacs/manual/pdf/emacs.pdf>. Visited on: 4 Dec. 2017.

STANISIC, Luka; LEGRAND, Arnaud; DANJEAN, Vincent. An Effective Git And Org-Mode Based Workflow For Reproducible Research. **ACM SIGOPS Operating Systems Review**, ACM, New York, NY, USA, v. 49, n. 1, p. 61–70, Jan. 2015. DOI: 10.1145/2723872.2723881.

STANISIC, Luka; THIBAULT, Samuel, et al. Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. **Concurrency and Computation: Practice and Experience**, v. 27, n. 16, p. 4075–4090, Nov. 2015. DOI: 10.1002/cpe.3555.

STODDEN, V.; LEISCH, F.; PENG, R.D. **Implementing Reproducible Research**. [S.l.]: Taylor & Francis, 2014. (Chapman & Hall/CRC The R Series).

STONE, John E.; GOHARA, David; SHI, Guochun. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. **Computing in Science & Engineering**, v. 12, n. 3, p. 66–73, May 2010. DOI: 10.1109/MCSE.2010.69.

TOPCUOGLU, H.; HARIRI, S.; WU, Min-You. Performance-effective and low-complexity task scheduling for heterogeneous computing. **IEEE Transactions on Parallel and Distributed Systems**, v. 13, n. 3, p. 260–274, Mar. 2002. DOI: 10.1109/71.993206.

TOSS, Julio; GAUTIER, Thierry. A New Programming Paradigm for GPGPU. In: **Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings**. Ed. by Christos Kaklamanis, Theodore Papatheodorou and Paul G. Spirakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 895–907. DOI: 10.1007/978-3-642-32820-6_88.

VALIANT, Leslie G.; G., Leslie. A bridging model for parallel computation. **Communications of the ACM**, ACM, v. 33, n. 8, p. 103–111, Aug. 1990. DOI: 10.1145/79173.79181.

WANG, Endong et al. Intel Math Kernel Library. In: HIGH-PERFORMANCE Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures. Cham: Springer International Publishing, 2014. p. 167–188. DOI: 10.1007/978-3-319-06486-4_7.

WICKHAM, Hadley. **ggplot2: Elegant Graphics for Data Analysis**. [S.l.]: Springer-Verlag New York, 2016. Available from: <http://ggplot2.org>. Visited on: 29 Aug. 2018.

_____. **tidyverse: Easily Install and Load the 'Tidyverse'**. [S.l.], 2017. R package version 1.2.1. Available from: <https://CRAN.R-project.org/package=tidyverse>. Visited on: 29 Aug. 2018.

WILSON, James M. Gantt charts: A centenary appreciation. **European Journal of Operational Research**, v. 149, n. 2, p. 430–437, Sept. 2003.

YAZDANPANAH, Fahimeh. An approach for analyzing auto-vectorization potential of emerging workloads. **Microprocessors and Microsystems**, v. 49, Supplement C, p. 139–149, 2017. DOI: https://doi.org/10.1016/j.micpro.2016.11.014.