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

LUCAS LEANDRO NESI

Memory Performance Analysis Strategies at Runtime Level for Task-Based Applications over Heterogeneous Platforms

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

Advisor: Prof. Dr. Lucas Mello Schnorr

Porto Alegre
June 2019

CIP — CATALOGING-IN-PUBLICATION

Nesi, Lucas Leandro

Memory Performance Analysis Strategies at Runtime Level for Task-Based Applications over Heterogeneous Platforms / Lucas Leandro Nesi. – Porto Alegre: PPGC da UFRGS, 2019.

111 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: Lucas Mello Schnorr.

1. HPC. 2. Performance analysis. 3. Task-Based programming. 4. Heterogeneous platforms. 5. Memory behavior. I. Mello Schnorr, Lucas. II. Título.

“For even the very wise cannot see all ends.”

— GANDALF THE GREY, J.R.R. TOLKIEN, THE FELLOWSHIP OF THE RING

ACKNOWLEDGEMENTS

To my advisor, Professor Dr. Lucas Mello Schnorr, that received and guided me during this journey. I am very grateful for all the opportunities, time, and motivation that you gave to me. Your advice was essential to the realization of this work.

To CNPq (Conselho de Desenvolvimento Científico e Tecnológico), for grating the scholarship, and permitting that I could conduct this work.

To the members of the jury, for all the time dedicated to the review of this work and for all the comments.

To my parents, – Agradeço aos meus pais, Eliane e Lirio, pelos inúmeros esforços ao longo de toda minha jornada. Vocês são a peça fundamental em todas as minhas realizações e minha principal inspiração.

To my flatmates, friends, Eduardo and Nathália, for welcoming me in Porto Alegre and sharing their time with me. The games, movies, drinks, and runs definitively kept me motivated during this work. For all the good moments that we had in the apartment.

To my friends in the GPPD group, in special, Vinicios, Matheus, Pablo, Marcelo, Váleria, Jéssica, Gabriel, Arthur, Jean. Everyone that I saw almost every day during the realization of this work and helped me. For all the conversations and beer.

To my friends of UDESC, Aurelio, Flavio, Gabriel, and Mateus. Even though most of us are in different cities, we stayed in touch.

To all the Teachers that I had, from elementary school to the university. I am very grateful for all the guidance and instruction.

Memory Performance Analysis Strategies at Runtime Level for Task-Based Applications over Heterogeneous Platforms

ABSTRACT

Programming parallel applications for heterogeneous High Performance Computing platforms is easier when using the task-based programming paradigm, where a Direct Acyclic Graph (DAG) of tasks models the application behavior. The simplicity exists because a runtime, like StarPU, takes care of many activities usually carried out by the application developer, such as task scheduling, load balancing, and memory management. This memory management refers to the runtime responsibility for handling memory operations, like copying the necessary data to the location where a given task is scheduled to execute. Poor scheduling or lack of appropriate information may lead to inadequate memory management by the runtime. Discover if an application presents memory-related performance problems is complex. The task-based applications' and runtimes' programmers would benefit from specialized performance analysis methodologies that check for possible memory management problems. In this way, this work proposes methods and tools to investigate heterogeneous CPU-GPU-Disk memory management of the StarPU runtime, a popular task-based middleware for HPC applications. The base of these methods is the execution traces that are collected by the runtime. These traces provide information about the runtime decisions and the system performance; however, a simple application can have huge amounts of trace data stored that need to be analyzed and converted to useful metrics or visualizations. The use of a methodology specific to task-based applications could lead to a better understanding of memory behavior and possible performance optimizations. The proposed strategies are applied on three different problems, a dense Cholesky solver, a CFD simulation, and a sparse QR factorization. On the dense Cholesky solver, the strategies found a problem on StarPU that a correction leads to 66% performance improvement. On the CFD simulation, the strategies guided the insertion of extra information on the DAG and data, leading to performance gains of 38%. These results indicate the effectiveness of the proposed analysis methodology in problems identification that leads to relevant optimizations.

Keywords: HPC. Performance analysis. Task-Based programming. Heterogeneous platforms. Memory behavior.

Estratégias para análise do desempenho de memória em nível de runtime para aplicações baseadas em tarefas sobre plataformas heterogêneas

RESUMO

A programação de aplicações paralelas para plataformas heterogêneas de Computação de alto desempenho é mais fácil ao usar o paradigma baseado em tarefas, em que um Grafo Acíclico Dirigido (DAG) de tarefas descreve o comportamento da aplicação. A simplicidade existe porque um *runtime*, como o StarPU, fica responsável por diversas atividades normalmente executadas pelo desenvolvedor da aplicação, como escalonamento de tarefas, balanceamento de carga e gerenciamento de memória. Este gerenciamento de memória refere-se as operações de dados, como por exemplo, copiar os dados necessários para o local onde uma determinada tarefa está escalonada para execução. Decisões ruins de escalonamento ou a falta de informações apropriadas podem levar a um gerenciamento de memória inadequado pelo *runtime*. Descobrir se uma aplicação esta apresentando problemas de desempenho por erros de memória é complexo. Os programadores de aplicações e *runtimes* baseadas em tarefas se beneficiariam de metodologias especializadas de análise de desempenho que verificam possíveis problemas no gerenciamento de memória. Desta maneira, este trabalho apresenta métodos para investigar o gerenciamento da memória entre CPU-GPU-disco de recursos heterogêneos do *runtime* StarPU, um popular *middleware* baseado em tarefas para aplicações HPC. A base desses métodos é o rastreamento de execução coletado pelo StarPU. Esses rastros fornecem informações sobre as decisões do escalonamento e do desempenho do sistema que precisam ser analisados e convertidos em métricas ou visualizações úteis. O uso de uma metodologia específica para aplicações baseadas em tarefas pode levar a um melhor entendimento do comportamento da memória e para possíveis otimizações de desempenho. As estratégias propostas foram aplicadas em três diferentes problemas, um solucionador da fatoração de Cholesky denso, uma simulação CFD, e uma fatoração QR esparsa. No caso do Cholesky denso, as estratégias encontraram um problema no StarPU que a correção levou a ganhos de 66% de desempenho. No caso da simulação CFD, as estratégias guiaram a inserção de informação extra no DAG levando a ganhos de 38%. Estes resultados mostram a efetividade dos métodos propostos na identificação de problemas que levam a otimizações.

Palavras-chave: HPC, Programação baseada em Tarefas, Análise de Desempenho, Plataformas Heterogêneas, Comportamento de Memória.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CFD	Computational Fluid Dynamics
CUDA	Compute Unified Device Architecture
CPI	Cycles-Per-Instruction
DAG	Directed Acyclic Graph
DM	Deque Model
DMDA	Deque Model Data Aware
DMDAR	Deque Model Data Aware Ready
DMDAS	Deque Model Data Aware Sorted
FXT	Fast Kernel Tracing
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
HEFT	Heterogeneous Earliest Finish Time
HPC	High performance computing
LWS	Local Work Stealing
LRU	Least-Recently-Used
MPI	Message Passing Interface
MSI	Modified/Owned, Shared, and Invalid
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language
OTF2	Open Trace Format Version 2
SIMD	Single Instruction, Multiple Data
STF	Sequential Task Flow

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

A challenge found in the High Performance Computing (HPC) domain is the complexity of programming parallel applications. The task-based programming paradigm presents numerous benefits, and many researchers believe this is currently the optimal approach to program for modern machines (Dongarra et al., 2017). The tasking related extensions to the OpenMP (in 4.0 and 4.5 version (OpenMP, 2013)), and the upcoming 5.0 standard (OpenMP, 2018) with even more features confirm this trend. In general, a task-based approach is efficient for load-balancing and intelligently using all the resources' computational power in heterogeneous platforms. It transfers to a runtime some activities that are usually carried out by programmers, such as mapping compute kernels (tasks) to resources, data management, and communication. Task-based applications use a Direct Acyclic Graph (DAG) of tasks as the main application structure to schedule them on resources, considering tasks dependencies and data transfers. Among many alternatives like Cilk (BLUMOFE et al., 1996), Xkaapi (GAUTIER et al., 2013), and OmpSs (DURAN et al., 2011); StarPU (AUGONNET et al., 2011) is one example of runtime using this paradigm and is used as the task-based runtime for this work. Its features include the use of distinct tasks' implementations (CPU, GPU), different tasks schedulers, and automatically managing data transfers between resources.

The performance analysis of task-based parallel applications is complicated due to its inherently stochastic nature regarding variable task duration and their dynamic scheduling. Different performance analysis methods and tools can be used to aid on this matter, including analytical modeling of the task-based application theoretical bounds (AGULLO et al., 2015) and the application-runtime simulation which allows reproducible performance studies in a controlled environment (STANISIC et al., 2015a; STANISIC et al., 2015b). StarPU can also collect execution traces that describe the behavior of the application to provide a framework for performance analysis. Possible uses of the information provided by the runtime can be in the form of performance metrics (number of ready and submitted tasks, GFlops rate, etc.), signaling poor behavior (i.e., absence of work in the DAG critical path), or visualization techniques (panels that illustrate the application and the runtime behavior over time). The visualization-based approach can combine all these investigation methods to facilitate the analysis with graphical elements. The StarVZ workflow (PINTO et al., 2018) is an example of a tool that leverages application/runtime traces. It employs consolidated data science tools, most notably specific-tailored `R`

scripts, to create meaningful views that enable the identification of performance problems and testing what-if scenarios.

Interleaving data transfers with computational tasks (data prefetching) is another technique that has a significant impact on performance (AUGONNET et al., 2010). The goal is to efficiently manage data transfers among different memory nodes of a platform: main (RAM), accelerator (GPUs), and out-of-core (hard drive) memories. Factors like the reduction of data transfers between heterogeneous devices and hosts, better use of cache, and smarter block allocation strategies play an essential role for performance. Simultaneously, many applications require an amount of memory greater than the available RAM. These applications require the use of out-of-core methods, generally because disk memory is much larger than main memory (TOLEDO, 1999). Correctly selecting which data blocks to stay on each resource memory is a challenge. The complexity of evaluating these memory-aware methods at runtime level motivates the design of visualization-based performance analysis techniques tailored explicitly for data transfers and general memory optimizations.

1.1 Motivation

Task-based parallel programming is an adopted paradigm for using multiple heterogeneous resources. However, conducting a performance analysis of these applications is a challenge. The creation of specific methods, considering the paradigm particularities, can benefit the programmers of these applications and runtimes. Although some recent tools were proposed on this topic (HUYNH et al., 2015; MUDDUKRISHNA et al., 2016; PINTO et al., 2018), these tools lack techniques to investigate the memory behavior (transfers, presence, allocation) of the runtime and application. Memory is a crucial factor in the performance of parallel applications and can be analyzed to find improvements. The StarPU developers and the StarPU applications' programmers can benefit from specific methods of memory analysis. Features like verify the memory access, flow, and management at runtime level would facilitate the programming, at the same time to enable the discovery of problems that impact performance.

1.2 Contributions

This work focus on the analysis of the StarPU's memory management performance using trace visualization. It enables a general correlation among all factors that can impact the overall performance: application algorithm, runtime decisions, and memory utilization. The main contributions are the following.

- The extension of the StarVZ workflow by adding new memory-aware visual elements that help to detect performance issues in the StarPU runtime and the task-based application code.
- The augmentation of StarPU with extra trace information about the memory management operations, such as new memory requests, additional attributes on memory blocks and actions, and data coherency states.
- Experiments on three different applications with the appliance of the proposed strategies. (I) The first application is the dense linear algebra solver Chameleon (AGULLO et al., 2010). The proposed strategies are used in four cases. In the first case, the methodology identified a problem inside the StarPU software. A fix is proposed, and a comparison of the application performance before and after the correction patch is conducted, leading to ≈66% of performance improvement. The second case analyzes idle times when using out-of-core. The third case explains the performance difference when the application allocate blocks differently using out-of-core memory. The fourth scenario studies the memory/application behavior between the `DMDAS` and `DMDAR` schedulers. The last case shows a proof-of-concept on the use of the strategies on multi-node execution. (II) The second application is a task-based CFD simulation over StarPU (NESI; SCHNORR; NAVAUX, 2019), where the strategies are applied specifically on a method for decomposing the subdomain. The methodology found memory residency problems that lead to possible optimizations thanks to the behavior observed, resulting in 38% performance improvement. (III) The third application is `QR_Mumps`, a sparse QR factorization (BUTTARI, 2012). The strategies found a behavior related to memory transfers, that may be disadvantageous in executions that have high transfer rates.

The combination of these contributions provides multi-level performance analysis strategies of data management operations on a heterogeneous multi-GPU and multi-core platform. The strategies offer a high-level view with the information of application DAG

with the low-level runtime memory decisions, which can guide the application and runtime programmers to identify performance problems. Instead of only using low-level metrics and comparing them with multiple executions, this work focus on the behavior understanding of representative executions. This work presents visualization elements specifically for the performance analysis of memory in a DAG-based runtime, enriching the perception of task-based applications running on heterogeneous platforms.

1.3 Structure

This document is structured as follows. Chapter 2 provides the context and the background about heterogeneous platforms, task-based programming, and the StarPU runtime. Chapter 3 presents related work on the performance analysis and visualization of general HPC applications, task-based applications, and memory management. It also discusses this work's strategies against the state-of-the-art. Chapter 4 presents the strategies and the methodology to investigate the performance of memory operations in the StarPU runtime, employing visualizations and a modern data science framework. Chapter 5 details the experiments conducted in three applications. First, the Chameleon/Morse application using the Cholesky operation. Second, a simple CFD application that the strategies are applied on a partitioning method. Third, a solver for sparse QR factorization, called `QR_Mumps`. Also, it provides a discussion of the limitations of the proposed strategies. Chapter 6 ends this document with the conclusions and future work.

2 TASK-BASED PROGRAMMING PARADIGM FOR HETEROGENEOUS HPC PLATFORMS AND THE STARPU RUNTIME

This chapter provides background concepts on heterogeneous platforms, the task-based programming, and the StarPU runtime. First, Section 2.1 offers some information on the current adoption of heterogeneous platforms on supercomputers and the general problems on programming them. Second, Section 2.2 explains the task-based programming model, some general advantages, some projects that are using it and how it is becoming popular. Third, Section 2.3 gives a brief overview of how StarPU works, a source code example, and its internal data management system related to this work.

2.1 Heterogeneous Platforms

The adoption of diverse accelerators on HPC is vastly increasing. Statistics from the TOP500[1] list of supercomputers on Tables 2.1 and 2.2 offer the accelerations' information of November 2017 and 2018 respectively. The data show an increase of 7.2% in one year. Moreover, the top two supercomputers as of November 2018, Summit and Sierra, are both using GPGPUs. A broad set of applications can benefit from the SIMD architecture present on many accelerators to have their execution time reduced (HENNESSY; PATTERSON, 2011). In addition to the accelerators, the presence of multiple computational nodes with different hardware configurations increases heterogeneity. In these situations where the application performance over heterogeneous nodes is different, robust programming methods must be deployed to use these resources efficiently.

The construction of applications for these heterogeneous platforms is complex because many APIs or programming paradigms are available to deal with specific components of a supercomputer. OpenMP (DAGUM; MENON, 1998) can be applied to use all the cores of a CPU. MPI (GROPP; LUSK; SKJELLUM, 1999) to coordinate the execution of the application and transfer data in different nodes. CUDA (NVIDIA, 2019) and OpenCL (STONE; GOHARA; SHI, 2010) can be used to program GPGPUs. Moreover, not all problems apply to all kinds of architectures. For example, many dense linear algebra operations are desirable on GPUs. When running applications on supercomputers' nodes, it is desirable to take most of it and use all the computational power it has, includ-

[1] https://www.top500.org/

Table 2.1: Accelerators on TOP500 List - November 2017

Accelerator/Co-Processor	Count	System Share (%)	Cores
NVIDIA Tesla P100	42	8.4	1,564,028
NVIDIA Tesla K40	12	2.4	312,456
NVIDIA Tesla K80	7	1.4	332,130
NVIDIA Tesla K20x	6	1.2	758,976
NVIDIA Tesla P100 NVLink	5	1	204,188
PEZY-SC2 500Mhz	3	0.6	2,351,424
NVIDIA Tesla P40	3	0.6	422,000
Intel Xeon Phi 7120P	3	0.6	172,994
NVIDIA 2050	3	0.6	360,256
Intel Xeon Phi 31S1P	2	0.4	3,294,720
NVIDIA Tesla K20m	2	0.4	36,892
Intel Xeon Phi 5120D	2	0.4	122,512
Intel Xeon Phi 5110P	2	0.4	272,136
Others (With Count = 1)	10	2.0	-
Total	102	20.4	-

Source: TOP500 (2018).

Table 2.2: Accelerators on TOP500 List - November 2018

Accelerator/Co-Processor	Count	System Share (%)	Cores
NVIDIA Tesla P100	60	12	2,441,572
NVIDIA Tesla V100	41	8.2	1,645,796
NVIDIA Tesla K20x	4	0.8	731,712
NVIDIA Tesla K80	4	0.8	271,870
NVIDIA Tesla K40	3	0.6	201,328
NVIDIA Tesla P100 NVLink	3	0.6	176,868
NVIDIA Tesla V100 SXM2	3	0.6	609,840
Intel Xeon Phi 5110P	2	0.4	272,136
Intel Xeon Phi 5120D	2	0.4	122,512
NVIDIA 2050	2	0.4	307,008
NVIDIA Volta GV100	2	0.4	3,970,304
Others (With Count = 1)	12	2.4	-
Total	138	27,6	-

Source: TOP500 (2018).

ing both CPUs and accelerators. In the current situation with different APIs, programming languages, and paradigms, programmers are overwhelmed by the number of responsibilities that they should take care of, including the communication of data between different resources' memory, guaranteeing the correctness application behavior, and yet, to achieve the maximum parallel performance that the hardware is capable to deliver. The task-based programming parading is one programming model that can reduce the complexity by pro-

viding a abstraction layer that controls the resources, scheduling tasks, and managing the data. Using this paradigm, the programmer transfers all these responsibilities to a runtime.

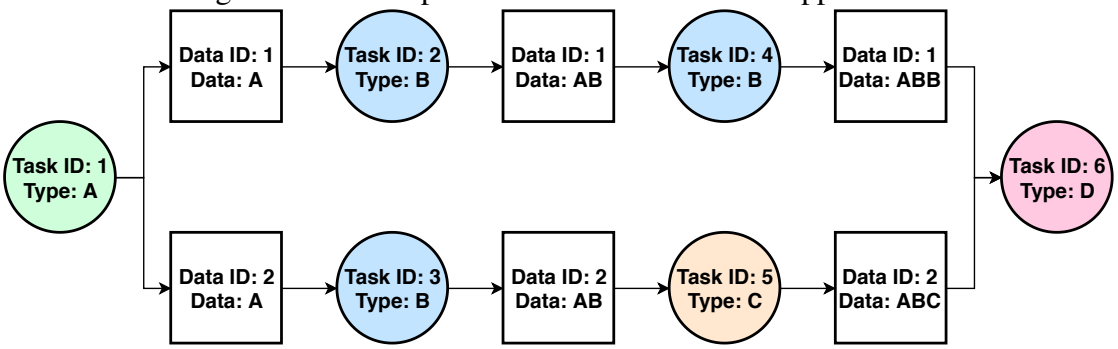
2.2 Task-Based Programming Paradigm

The task-based programming, or data flow scheduling, is a concept (BRIAT et al., 1997) that uses a more declarative strategy to transfer some responsibilities to a runtime system, facilitating the application coding and reducing the programmer control over the execution (Dongarra et al., 2017). This paradigm is becoming more popular since the 2000 decade, where different projects started to use it (THIBAULT, 2018; Dongarra et al., 2017). Furthermore, the runtime system can apply several automatic optimizations with the information inherited from the Task and Data declarations (THIBAULT, 2018).

The structure of a program following the task-based programming paradigm consists of a collection of tasks (usually a sequential code region) that have a specific purpose over a collection of data. Therefore, the interaction between tasks occurs primarily by the use of the same data fragments in different tasks, causing implicit dependencies among them to guarantee computation coherence. Usually, a DAG represents a task-based application, where nodes are tasks, and edges are dependencies. Such dependencies are inherited by data reuse or explicitly inserted by the programmer. Figure 2.1 offers an example of DAG, with four different tasks (each one do a different computation over a data region). The DAG starts with a `Task A`, with `ID 1`, that generates `Data 1` and `Data 2` and will be used by two `B Tasks`. Each `Task B` will process its data that will be used by a `Task B` and a `Task C` respectively. In the end, a `Task D`, with `ID 6`, will use both the results of the `Task B` and `Task C` (`Data 1` and `Data 2`) to end the computation. Each data represents a fragment of the problem data. Although multiple tasks use the same data fragment, the tasks usually modify their content. For example, four tasks used `Data 1`, it started empty, then with "A," after that "AB," and it finished "ABC" when it was read by `Task D` with `ID 6`.

One of the benefits of adopting the task-based model is that the programming of HPC applications is much more abstract and declarative (Dongarra et al., 2017), far from the details of the heterogeneous execution platform. So, instead of an explicit mapping of tasks to resources by the programmer, the parallelism is obtained by letting a middleware take such decisions. Tasks are submitted sequentially to the middleware interface, and the runtime determines where tasks execute. For the example of Figure 2.1,

Figure 2.1: Example of DAG for a task-based application



Source: The Author.

`Task A` would be declared using two fragments of data (`Data 1` and `2`), while `Task C` would be declared only using one (it would read and write over `Data 2`. Because of the uncertain and irregular state of the system environment, the tasks' execution order is stochastic. Many runtimes act like this middleware, for example, PaRSEC (BOSILCA et al., 2012), OmpSs (DURAN et al., 2011), OpenMP (in 4.0 and 4.5 version (OpenMP, 2013)), XKaapi (GAUTIER et al., 2013), and StarPU (AUGONNET et al., 2011).

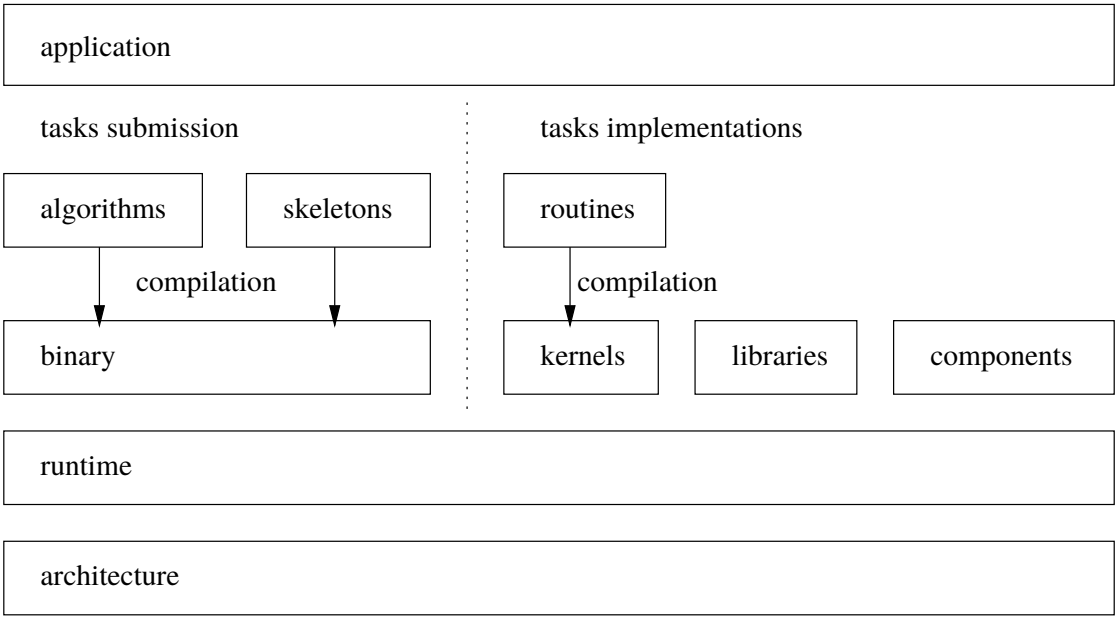
The runtimes can work on heterogeneous platforms, as any resource is just a computation unit. The tasks can have multiple implementations to a variety of accelerators and are scheduled to different devices. One example of runtime that supports the use of CPUs and GPUs is XKaapi. Its support was done by Lima (2014), that also study schedulers for this combination of resources. Although there are similarities among them, this work only details the StarPU runtime, since the strategies use it as the main experimental runtime.

2.3 The StarPU runtime

StarPU is a general-purpose task-based runtime system that provides an interface for task-based applications use to submit jobs over resources. Figure 2.2 presents the software stack and the location of the runtime over an architecture while all the applications and its tasks, kernels, and components are over it. StarPU uses the Sequential Task Flow (STF) model (AGULLO et al., 2016), where the application sequentially submits tasks during the execution and the runtime dynamically schedule them to workers. In such a model, there is no need to unroll the whole DAG of tasks before starting tasks execution, and the application can dynamically create tasks and submit them during the execution. The dependencies among tasks that would structure the DAG are inherited by data that is

shared between tasks. The data used by a task needs to be declared into a data handle, and the reuse of the handles turn possible the creation of dependencies among the tasks and defining a DAG. Therefore, in StarPU, the dependencies among tasks are explicit. This characteristic is simpler and different from other approaches like OpenMP tasks and MPI, where the programmer has to inform the dependencies or communications manually. For example, using OpenMP 4.0, the programmer has the ability to define memory dependencies with the `depend` clause.

Figure 2.2: Software stack when using the StarPU task-based runtime



Source: Thibault (2018).

Moreover, StarPU tasks might have multiple implementations, one for each type of device (such as x86 CPUs, CUDA GPUs, and OpenCL devices). The runtime will decide, base on scheduling heuristics, on which resource each task will execute, selecting the correct implementation. StarPU can manage most of the controlling aspects of these devices, creating modules for each one. For example, for CUDA GPU devices, StarPU is responsible for initializing the GPU, selecting the correct streams, and using the CUDA API for registering and transferring the data handles. However, StarPU still enables the programmer control over these devices. As each CUDA task implementation is essentially one common sequential CPU task that calls a CUDA kernel, it is possible to use any CUDA API that is required. For example, these CUDA tasks usually need to wait for any CUDA asynchronous calls that it make, and do it using the appropriate API calls.

An example of StarPU usage code is present in Figures 2.3 and 2.4. The definition of tasks is present in Figure 2.3 and can be done by defining a `starpu_codelet` struc-

ture and informing the CPU implementation (in this case `task_cpu`, the name of the task, the number of data handles, the access modes of these handles, and the performance model. Figure 2.4 presents the structure of the main function of a StarPU application, responsible for submitting tasks sequentially.

Figure 2.3: Example of code to define tasks using StarPU

```
1  enum starpu_data_access_mode task_modes[1] = {
2          STARPU_RW
3  };
4
5  void task_cpu(void *buffers[], void *cl_arg){
6    struct params par;
7    starpu_codelet_unpack_args(cl_arg, &par);
8
9    int* data = (int *)STARPU_VECTOR_GET_PTR(buffers[0]);
10
11   //DO SOMETHING
12 }
13
14 static struct starpu_perfmodel task_perf_model = {
15      .type = STARPU_HISTORY_BASED,
16      .symbol = "task"
17 };
18
19 struct starpu_codelet cl_task_arrow =
20 {
21   .cpu_funcs = { task_cpu },
22   .cpu_funcs_name = { "task_cpu" },
23   .name = "task",
24   .nbuffers = 1,
25   .dyn_modes = task_modes,
26   .model = &task_perf_model
27 };
```

Source: The Author.

StarPU employs different heuristics to allocate tasks to resources. Depending on resource availability and the heuristic, the scheduler dynamically chooses one of the task versions and places it to execute. Classical heuristics are `LWS` (local work stealing), `EAGER` (centralized deque), `DM` (deque model) based on the `HEFT` (TOPCUOGLU; HARIRI; WU, 2002) algorithm. More sophisticated schedulers consider additional information. The `DMDA` (deque model data aware) scheduler, for example, uses estimated task completion time and data transfer time to take its decisions (AUGONNET et al., 2010). Another example is the `DMDAR` (deque model data-aware ready) scheduler; that

Figure 2.4: Example of main application code using StarPU

```
1  #include <stdio.h>
2  #include <starpu.h>
3
4  #define SIZE 100
5
6  int main(int argc, char** argv){
7          starpu_init(NULL);
8
9          int data[SIZE]
10
11         starpu_data_handle_t h_data;
12         starpu_vector_data_register(&h_data,
13                                     STARPU_MAIN_RAM,
14                                     (uintptr_t)data,
15                                     SIZE, sizeof(int));
16
17         starpu_data_handle_t handles[1];
18
19         handles[0] = h_data;
20
21         starpu_insert_task(cl_task,
22                            STARPU_VALUE, &par,
23                            sizeof(struct params),
24                            STARPU_DATA_ARRAY,
25                            handles,
26                            cl_task.nbuffers,
27                            0);
28
29         starpu_task_wait_for_all();
30         starpu_free(h_data);
31         starpu_shutdown();
32         return 0;
33 }
```

Source: The Author.

additionally considers data handles already present on the workers.

The DM schedulers use performance models of the tasks to predict how much time will take on each resource to calculate the best option. StarPU uses prior executions to creates these models, saving the mean execution time, the standard deviation, the implementation used, and the data sizes. Also, the transfers times of data between resources are calculated at the initialization of StarPU, if not yet done, to estimate the bandwidth transfer ratio for each possible combination of transfer links between resources.

The runtime is also responsible for transferring data between resources, for controlling the presence and the coherence of the data handles. StarPU creates one memory manager for each different type of memory. For example, there is one memory manager for the `RAM` associated with one NUMA node (shared by all CPU cores on that socket), one for each GPU, and so on. StarPU adopts the MSI protocol, with the states `Modified/Owned`, `Shared`, and `Invalid`, to manage the state of each data handle on the different memories. At a given moment, each memory block can assume one of the three states on the memory managers (AUGONNET et al., 2011). When StarPU schedules a task, it will internally create a memory request for one of the tasks memory dependencies to the chosen resource. These requests are handled by the memory managers that are responsible for allocating the block of data and issuing the data transfer. When tasks are scheduled well in advance, StarPU prefetches data, so the transfers get overlapped with computations of the ongoing tasks (SERGENT et al., 2016).

When no memory is available for allocating that data handle, StarPU presents different behaviors depending on the request type (Idle Prefetch/Prefetch/Fetch). In case the handle is a fetch for tasks that can be executed right away, StarPU will cope and remove another data handle. Furthermore, recent versions of StarPU support the use of out-of-core memory (disk i.e., HDD, SSD) when `RAM` occupation becomes too high. The runtime employs a Least-Recently-Used (LRU) algorithm to determine which data blocks should be transferred to disk to make room for new allocations on `RAM`. Interleaving such data transfers with computation and respecting data dependencies on the critical path is fundamental to better performance.

StarPU is able to use multiple computational nodes by using its MPI module (AUGONNET et al., 2012). The application developer, to use this module, follows similar principles of the MPI, where the programmer must do the distribution of data. The main difference from a regular StarPU application is that the data handles have an owner node. Tasks that conduct write operations on handles must be performed on the nodes where the data is owned, restricting the scheduler. Also, each node must unfold the whole DAG to check which tasks it should perform by the ownership of memory blocks that the tasks write. Read-only data handles (not modified) can be automatically transferred by the StarPU-MPI runtime. In that way, this StarPU module presents a distributed scheduling system. Moreover, in StarPU version 1.3, another approach is introduced to handle distributed executions called master-slave (StarPU, 2019). In this approach, a master node will centralize the scheduling, and all other slaves nodes have their resources available for

the master node control. In this master-slave approach, the data distribution is carried by the runtime and not the application developer because the other nodes behave similarly to intra-node resources.

StarPU offers feedback on the application performance in two styles, on-the-fly statistics, and execution traces containing runtime events. Both feedback features need to be specified during the compilation of StarPU because they are optional. The on-the-fly statistics should not degrade the application performance and thus are limited to simple metrics over a set of measurements, like total time running a task, total memory transfers, and scheduling overhead. Differently, the execution traces can contain an accurate, detailed view of the application and the runtime behavior. The trace format used by StarPU is FXT (DANJEAN; NAMYST; WACRENIER, 2005), that aims to trace information during the execution reducing the intrusion. The total overhead caused by the trace system intrusion is low (THIBAULT, 2018). Furthermore, this work concentrates on how to process the trace information while leaving the overhead analysis to StarPU developers. At the end of the application execution, StarPU generates one FXT file per process. In the case of StarPU-MPI usage, it will create one file per computational node. Furthermore, the FXT file can be converted for other more commonly used trace formats in HPC, like Paje (SCHNORR; STEIN; KERGOMMEAUX, 2013), to investigate the application behavior. Such traces contain various internal StarPU events that can be correlated using many approaches to retrieve useful insights.

This work relies on the use of these FXT traces provided by the StarPU runtime, and the use of phase 1 of the StarVZ workflow. The StarPU has an auxiliary tool to convert the FXT traces to Paje, then, StarVZ phase 1 prune, clean and organize these Paje files and save the R structures into an intermediate file. The proposed strategies then rely on reading functions of the StarVZ to load these files and are built around these data structures.

3 TOOLS TO ANALYZE HPC APPLICATIONS

The analysis of HPC applications is an essential step for improving their performance. Many aspects can interfere with the performance, like resources scheduling, platform configuration, and related to this work, memory management. When using the task-based programming model, the runtime will be responsible for many memory operations, like transferring data between resources, deciding when a data fragment is no longer necessary and guarantee coherency between copies. Execution traces can be used to check if there are any problems in these steps. Moreover, tools to help application's and runtime's developers are desirable. Consequently, visualization techniques are employed on the analysis of these applications, and many tools were created to fulfill these needs.

The structure of the chapter is the follow. Section 3.1 describes traditional tools for analyzing the performance of HPC applications. Moreover, Section 3.2 presents tools that are designed for task-based applications. Although memory operations at an architectural level can be a factor in a performance loss of task-based applications (PERICÀS et al., 2014), this work focus at the runtime level and its decisions on the memory management of application data.

3.1 Traditional Tools

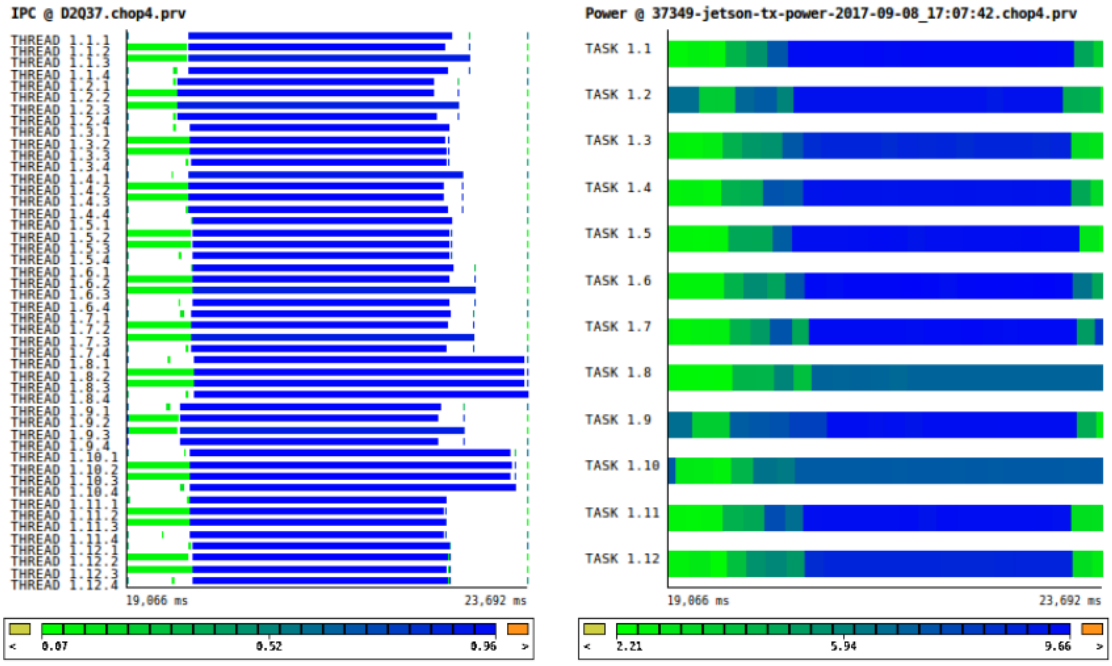
This section presents some traditional performance analysis tools that focus on OpenMP, MPI, MPI/OpenMP, and MPI/CUDA programming paradigms. The tools usually focus on traditional events like communications and resources utilization, information present on any parallel programming paradigm. Their objective is often to present the usage of the resources over time for the analysis of the application on a resource-oriented view. The visualization techniques usually employed are space/time methods where application's states or events are correlated to a specific time represented as an axis, and the resource that they are associated.

3.1.1 Paraver

Paraver was one of the first visualization tools for performance analysis that can be used to different programming models and paradigms (PILLET et al., 1995). It is

currently developed in the Barcelona Supercomputing Center. Paraver trace files can be generated with the runtime measurement system Extrae (GELABERT; SÁNCHEZ, 2011), that supports MPI, OpenMP, pthreads, OmpSs and CUDA. Moreover, the Paraver trace format is open source and can be included in other systems or translated from other trace formats. Paraver focus on being flexible, where the user can create semantics to work on which data should be visualized. Moreover, recent studies with Paraver aims to include energy-aware metrics and visualizations into the tool (MANTOVANI; CALORE, 2018). Figure 3.1 presents an example of visualization of an application running over twelve nodes each one with a single CPU using Paraver. Two panels are displayed. The left panel presents the Instructions per Cycle (IPC) over time for each one of the threads. The right panel presents the power usage by each one of the CPUs.

Figure 3.1: Example of visualization using Paraver



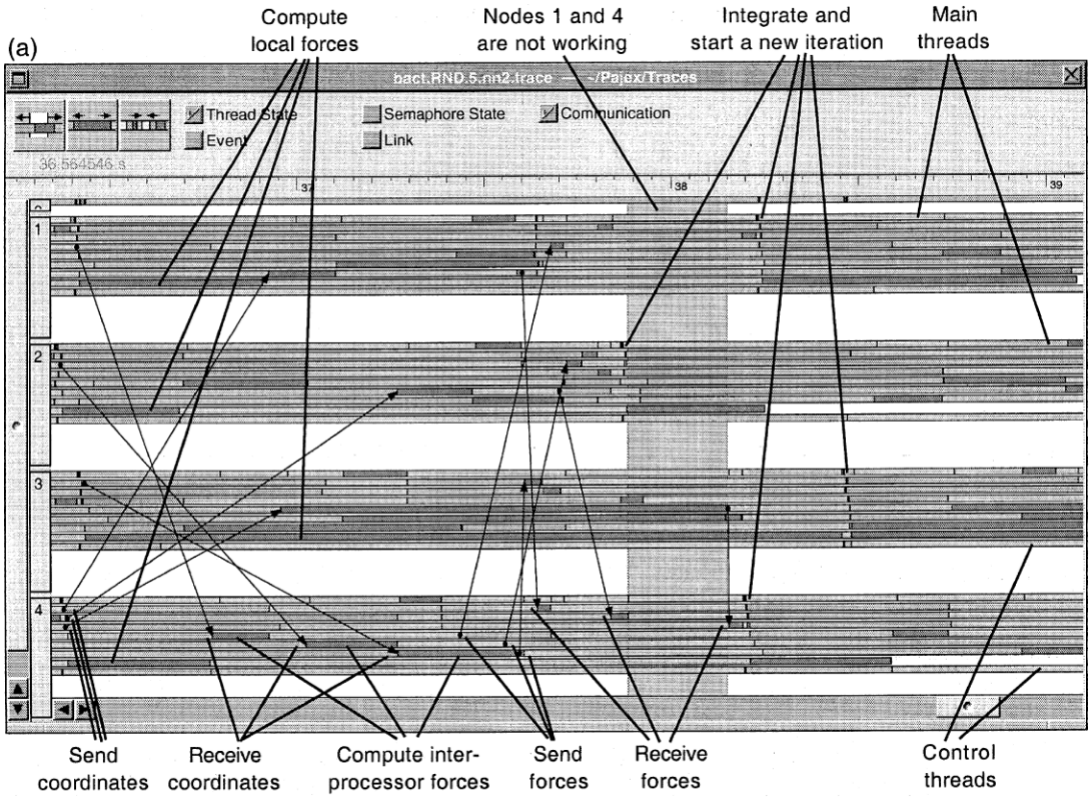
Source: Mantovani e Calore (2018).

3.1.2 Paje

Paje is an open-source visualization tool that enables interactive visualization of execution traces of HPC applications (KERGOMMEAUX; STEIN; BERNARD, 2000). Very similar to Paraver, the tool has a trace file format (SCHNORR; STEIN; KERGOMMEAUX, 2013) to describe the hierarchical structure of the platform; including states,

events, variables, and communications that each resource have. Paje was the first tool capable to hierarchically describe the platform. Two examples of visualizations that Paje provide are the Space/Time View and the Statistical View. Figure 3.2 presents an example of visualization using Paje, presenting four nodes on the Y-axis and their states a long time (X-axis). There are also communications displayed with arrows between nodes. Paje was originally written in Objective-C that is no longer supported, and today it has been replaced by the PajeNG (SCHNORR, 2012) implementation using C++. Although, PajeNG does not have a visualization module, it recommends the use of R+ggplot2 to create plots. Both implementations, Paje and PajeNG, follow a modular philosophy.

Figure 3.2: Example of visualization using Paje



Source: Kergommeaux, Stein e Bernard (2000).

The creation of a Paje trace file can be done manually, by transforming any custom event format into the Paje format or using other tools like Poti (SCHNORR, 2017). Five types of objects can be declared in the Paje file format, Container, States, Events, Variables, and Links (SCHNORR; STEIN; KERGOMMEAUX, 2013), described as follows. (I) The Container is an entity that can be associated with events. This entity can be a hardware resource (Processor, Accelerator, Network), a part of an operational system (Process, thread) or high-level software item (Worker, Memory Node). The containers are

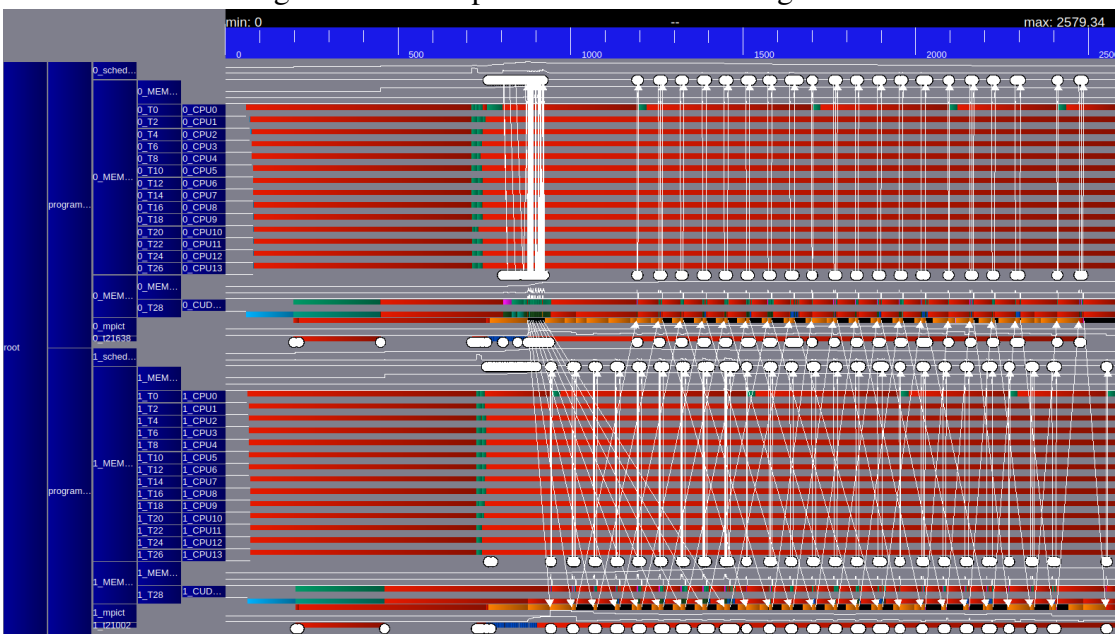
hierarchically organized. Therefore, each one can have multiple inner containers. All the other objects are associated with a container. Moreover, for any hierarchy, there must be a type hierarchy first so semantic checks can be carried out by Paje, identifying possible problems on the structure. (II) The State is an action always associated with only one container that has a start and an end time. (III) The Event is any activity that is affiliated with a container and has only one time informing when it happened. (IV) Variable is an object that represents the evolution of a variable associated with only one container. (V) The Link is an object that represents information associated with two containers, that have a start and an end time. It is primarily used to describe communications.

3.1.3 ViTE

Visual Trace Explorer (ViTE) is an open-source visualization tool for analyzing execution's traces in Pajé or OTF2 format (COULOMB et al., 2009). It aims to provide a fast Gantt visualization of the resources states by directly using OpenGL elements and drawing all data without any aggregation, that is, not grouping events to reduce the number of elements drawn. It provides a simple faithful representation of the traces without computing extra information. The application is a Gantt chart that in the Y-axis it constructs the hierarchical resources and the X-axis is the time. When using StarPU traces, it presents information about the execution of tasks on the resources and internal StarPU actions, like scheduling or idle/sleeping resources. Moreover, communications are arrows between memory nodes, the events appear as dots on a line of the container, and provided metrics are on a line plot. The states can have a color.

Figure 3.3 presents the visualization of the Chameleon Cholesky with StarPU-MPI running over two machines with 14 cores and one GPU each one. The workload is a 9600 × 9600 matrix, with block size 960, `DMDAS` scheduler, and partition 2 × 2. In the visualization, the red states are idle times, and there is communication between nodes and intra-nodes between the memory ram and the GPU represented by the white arrows. There is no information available of specific memory blocks. If multiple events over the same entity occur, it is difficult to distinguish them, because there may be a lot of events for a small space. Moreover, the order of how the elements are in the trace file, or programming decisions of how ViTE renders the elements can change the visualization. This situation can result in different possible interpretations of the different visualizations.

Figure 3.3: Example of visualization using ViTE



Source: The Author.

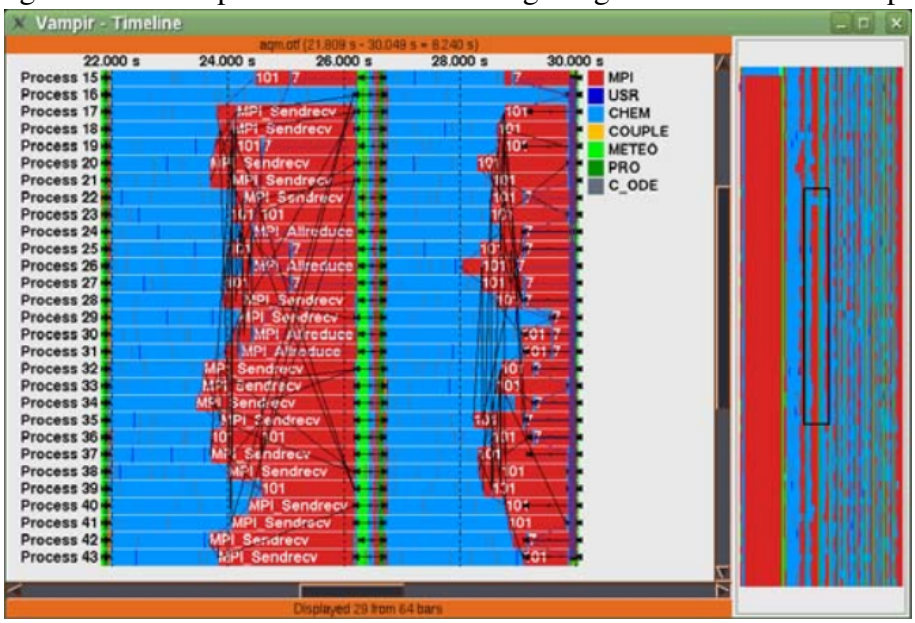
3.1.4 Vampir

Vampir is a closed-source visualization tool for parallel applications traces, available as a commercial product (KNÜPFER et al., 2008). The Vampir visualization tool can read trace in different formats, including the OTF2. Moreover, it has the VampirTrace capture trace system that is available with a BDS license. The trace is generated by instrumentation of the application and internal libraries and can be done automatically for MPI, OpenMP, and MPI/OpenMP programming paradigms. There are four ways that VampirTrace instruments the application, via the compiler, source-to-source, library, and manually. Also, Vampir can use OTF2 traces generated by Score-P (KNÜPFER et al., 2012) trace system.

The visualization of traces can be done using the Vampir tool and opening trace files or using the VampirServer system where it can access files on the computation nodes to generate the visualizations on the analyst client side. This approach provides some advantage like scalability, where the trace data is kept on the computational nodes and only accessed when needed. The visualization of large trace files can be done remotely and interactively. Figure 3.4 presents one of the timeline views, the visualization of the global timeline of Vampir, where it uses a Gantt Chart to present information about the states of the processes. The visualization includes MPI events and uses arrows to show communications between the MPI processes. Figure 3.5 presents one of the Statistics

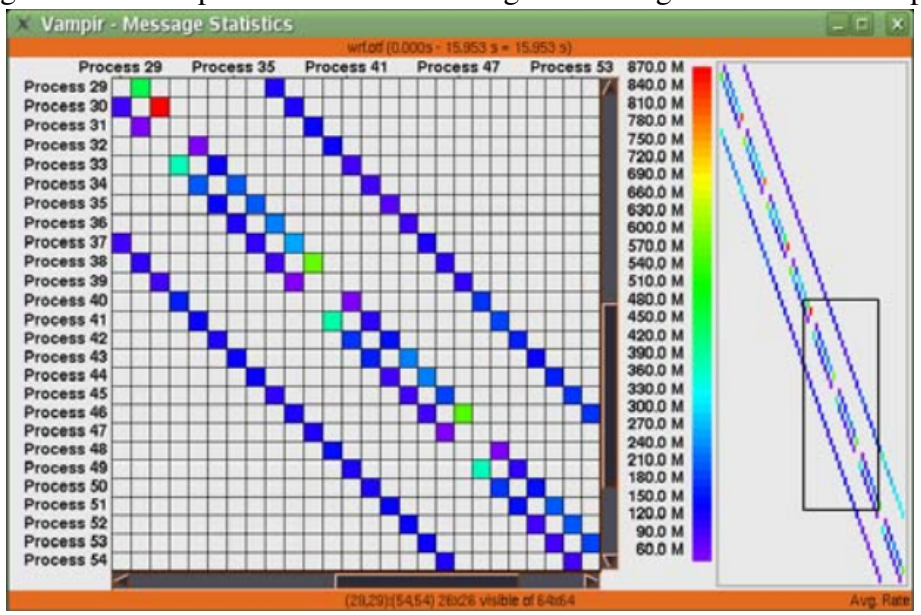
Displays, the Message Statistics, that provides information about how much data was sent and received between pairs of processes. This visualization can also be summarized with a thumbnail to express larger traces visualization.

Figure 3.4: Example of visualization using the global timeline of Vampir



Source: Knüpfer et al. (2008).

Figure 3.5: Example of visualization using the Message Statistics of Vampir



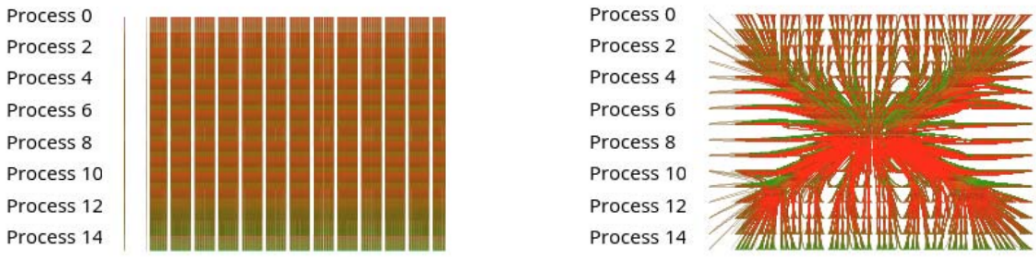
Source: Knüpfer et al. (2008).

Brendel et al. (2016) provides an extension for Vampir visualization focusing on the communication among processes. Using the traditional Vampir visualizations, the

communication is represented by lines among processes, and it can become very saturated if a lot of communication is done in a short period. To overcome this issue, the extension proposes the use of edge bundling that joins lines that have near destinations to form groups, bundles. This strategy is interesting because it reduces lines that in great quantity hides the image, and the notion of communication is lost. Figure 3.6a presents an example of the original master timeline of Vampir while Figure 3.6b presents the same visualization with the edge bundling method.

Figure 3.6: Master timeline of Vampir with and without edge bundling

(a) Original master timeline (b) Master timeline with edge bundling



Source: Brendel et al. (2016).

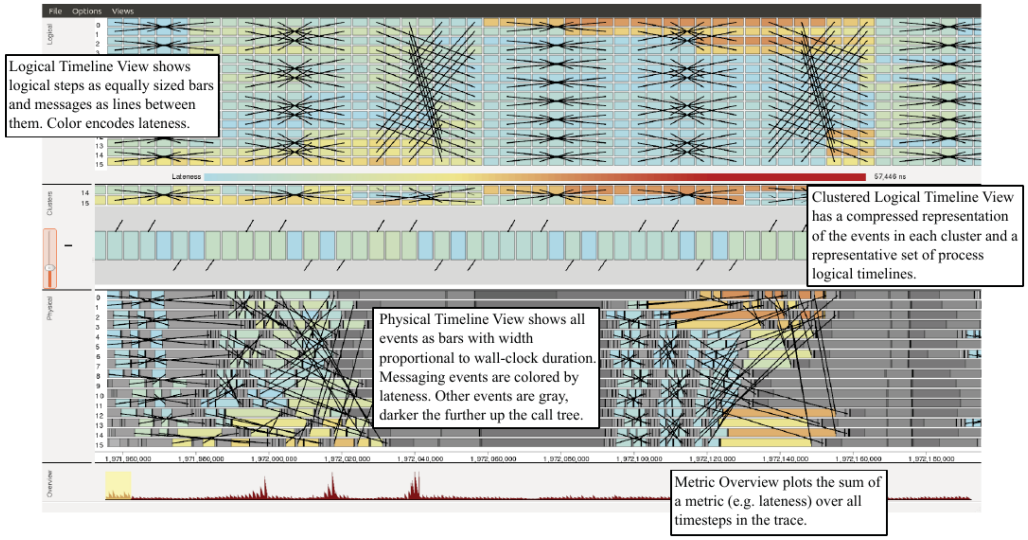
3.1.5 Ravel

Ravel is another example of tool for visualizing parallel applications using execution traces (ISAACS et al., 2014). Instead of using timestamps of the exactly events provided by the trace, the tool uses a logical time, that is calculated considering the sender and receiver communications of the events using the Lamport clock (LAMPORT, 1978). In that way, if process A occurs before B, and A send data to B, the Lamport Clock will attribute a sequence number in A lesser than the number attributed to B. This process is used to improve the visualization of communication between processes and check relationships between them.

Figure 3.7 shows an example of view using Ravel. The upper panel presents the logical timeline of the processes; it is possible to check the behavior of communication among them and the progression of the steps. The structured behavior of communications appears because of the use of the logical time. The second panel presents a selected Cluster of the first visualization and a sequence of tasks and communications. The third panel presents the traditional view using real time. It is possible to notice the difference between the original time and the logical time presented in the upper plot. Moreover, the

last panel presents metric plots, in this case, the lateness of all timesteps.

Figure 3.7: Example of visualization using Ravel



Source: Isaacs et al. (2014).

3.2 Task-Based Tools

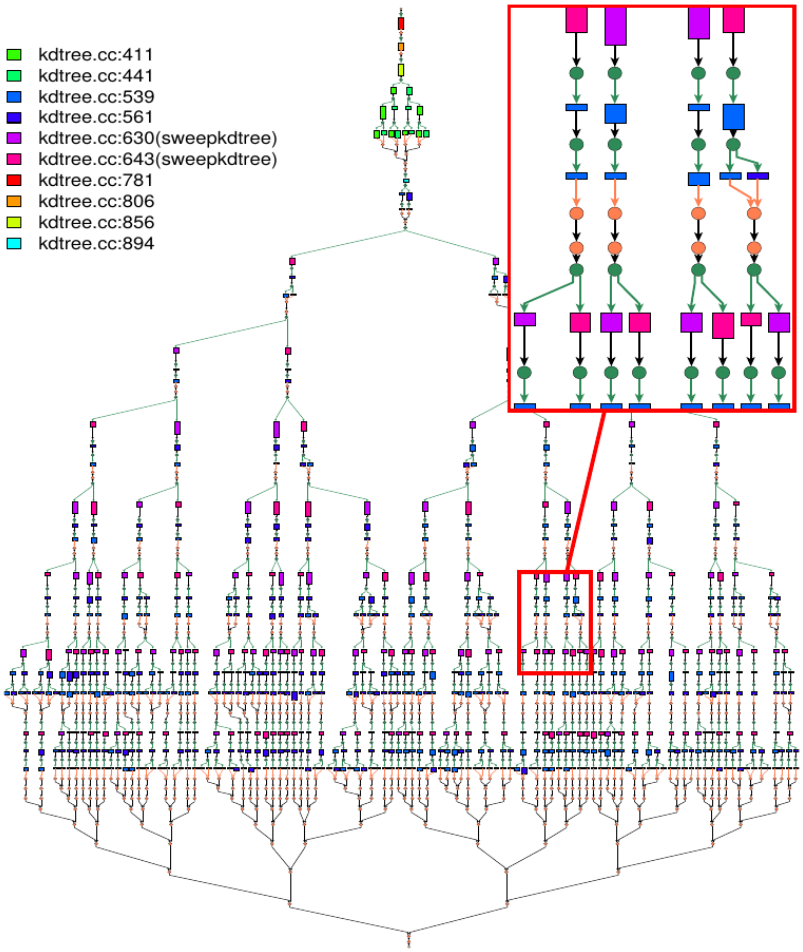
Task-based applications are structured following relationships between the tasks and organized using a DAG, as previously detailed in Chapter 2. Performance analysis can use the DAG structure because it provides extra information about the application, and in most cases, it can impact the performance. Different from traditional parallel applications where the notion of computation steps cannot be very defined, and dependencies are correlated with communications, task-based applications provide a defined structure. The benefits of using these structures to analyze the performance lead to the creation of another category of tools for performance analysis, specially design for these applications. These tools can accommodate a series of programming methods, including OpenMP tasks and task-based runtimes.

3.2.1 Grain Graphs

Grain Graphs (MUDDUKRISHNA et al., 2016) is a visualization tool designed for OpenMP that focus on the visualization of computational chunks, called grains by the authors, that come from tasks or parallel for-loops. The graph of grains is a DAG that

depicts the grains and the fork and join events. Figure 3.8 shows a visualization made with Grain Graphs. Each square task represent a grain, that the legend indicate the source file and line position. Also, the size, color or shapes represent performance metrics in the visualization. For example, the size of the rectangle is related to the execution time of that chunk. Also, the fork and join points are represented by the circle states. The system also derives some metrics related to the Graph.

Figure 3.8: Example of visualization using Grain Graphs



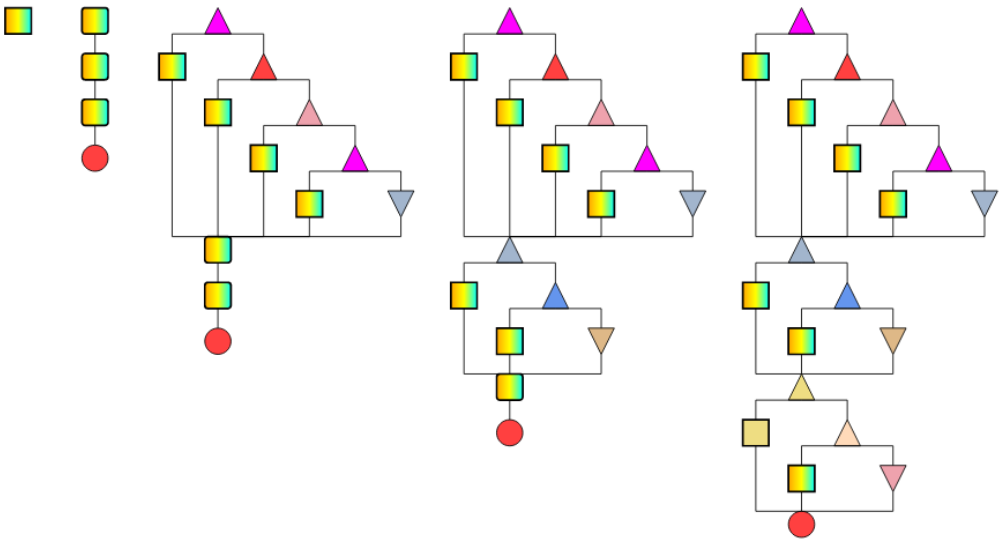
Source: Muddukrishna et al. (2016).

3.2.2 DAGViz

DAGViz is a tool to visualize execution's traces that focus on the task-based application DAG (HUYNH et al., 2015). It currently supports the OpenMP, Cilk Plus, Intel TBB, Qthreads, and MassiveThreads systems. DAGViz provides a model with three primitives to create an application DAG that can be converted to a parallel version using one of

the five supported systems. All applications start as a task, its main function and can use one of the following primitives. (I) CreateTask, the current task creates a child task; (II) WaitTasks, the current task wait for all tasks in the section; (III) MakeSection, create a section from the macro until a WaitTasks. DAGViz extracts the application DAG using a wrapper on the primitives and creates a visualization focusing on the DAG structures and the relationships among tasks without a timeline. Figure 3.9 presents the visualization provided by DAGViz. The squares are a hidden part of the DAG. The upper triangles represents tasks. The down triangles are wait points. Moreover, the red circle marks the end of the application. The color of each component represents the resource that executed it, where multi-color components represent that they were executed by multiple resources.

Figure 3.9: Example of visualization using DAGViz



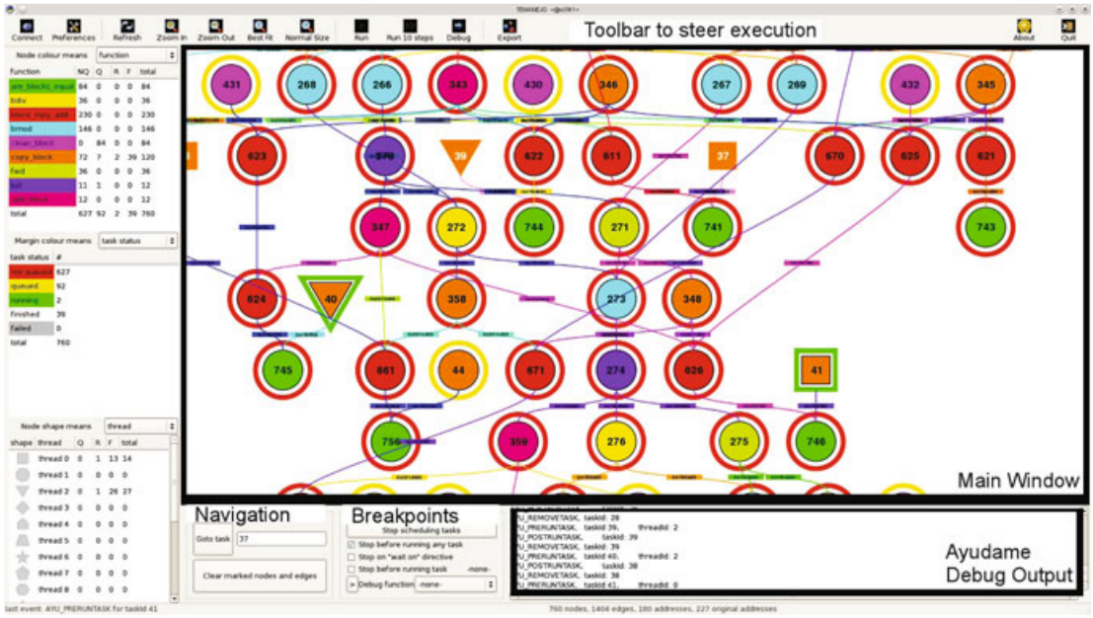
Source: Huynh et al. (2015).

3.2.3 Temanejo

Temanejo is a toolset that contains an online graphical debugger designed for the task-based programming parading (KELLER et al., 2012). Runtimes that support Temanejo include OmpSs (DURAN et al., 2011), StarPU (AUGONNET et al., 2011), OpenMP (OpenMP, 2013), and Parsec (BOSILCA et al., 2012). It relies on the Ayudame library, part of the toolset, to collect and process information while being the connection with the task-based runtime. The main GUI permits to visualize the application DAG during execution as tasks are being submitted to the runtime. Also, it provides infor-

mation about the current states of the tasks and other custom information provided by the runtime. Moreover, it permits interaction with the runtime to give some actions like initiating `gdb` for some task. Figure 3.10 provides an example of visualization using Temanejo, where each component in the main window is a task, and the links represent dependencies among them.

Figure 3.10: Example of visualization using Temanejo



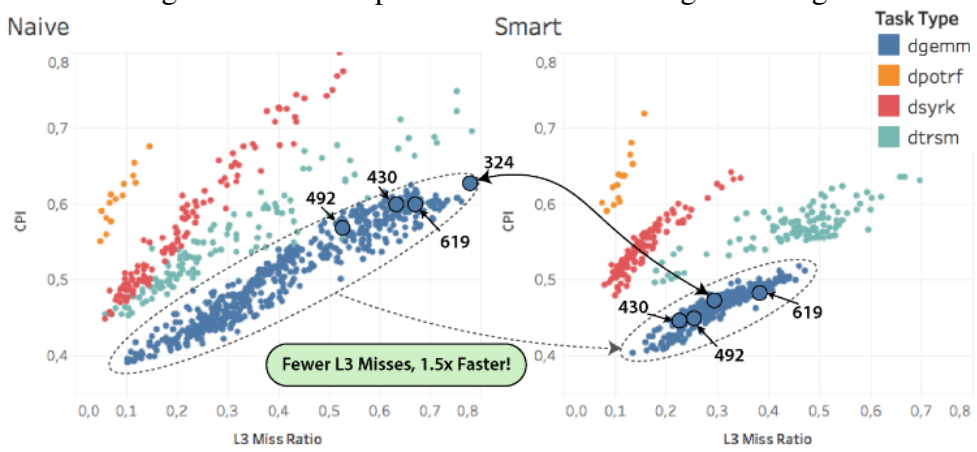
Source: Keller et al. (2012).

3.2.4 TaskInsight

TaskInsight is a tool to evaluate how data reuse among application tasks might affect application performance (CEBALLOS et al., 2018). They quantify the variability of task execution by measuring data-aware hardware counters of some tasks when another task scheduling is being carried out. They focus on low-level architectural metrics that may change based on the order of scheduling of tasks, and this difference would cause performance loss. Some metrics that they analyze are the number of instructions, cycles, cache misses, CPI (cycles-per-instruction), and data reuse by different tasks. Figure 3.11 provides a comparison of a Cholesky factorization execution using two difference schedulers, naive on the left, smart on the right. A naive scheduler that uses BFS policy, sorting the tasks by creation order, while the smart prioritizes tasks children first. The Figure presents the difference in tasks' L3 miss ratio and the impact on the performance (CPI)

when using the two schedulers. The horizontal axis is the L3 miss ratio of the tasks, and the vertical one is the CPI. Each dot represents a different task.

Figure 3.11: Example of visualization using TaskInsight



Source: Ceballos et al. (2018).

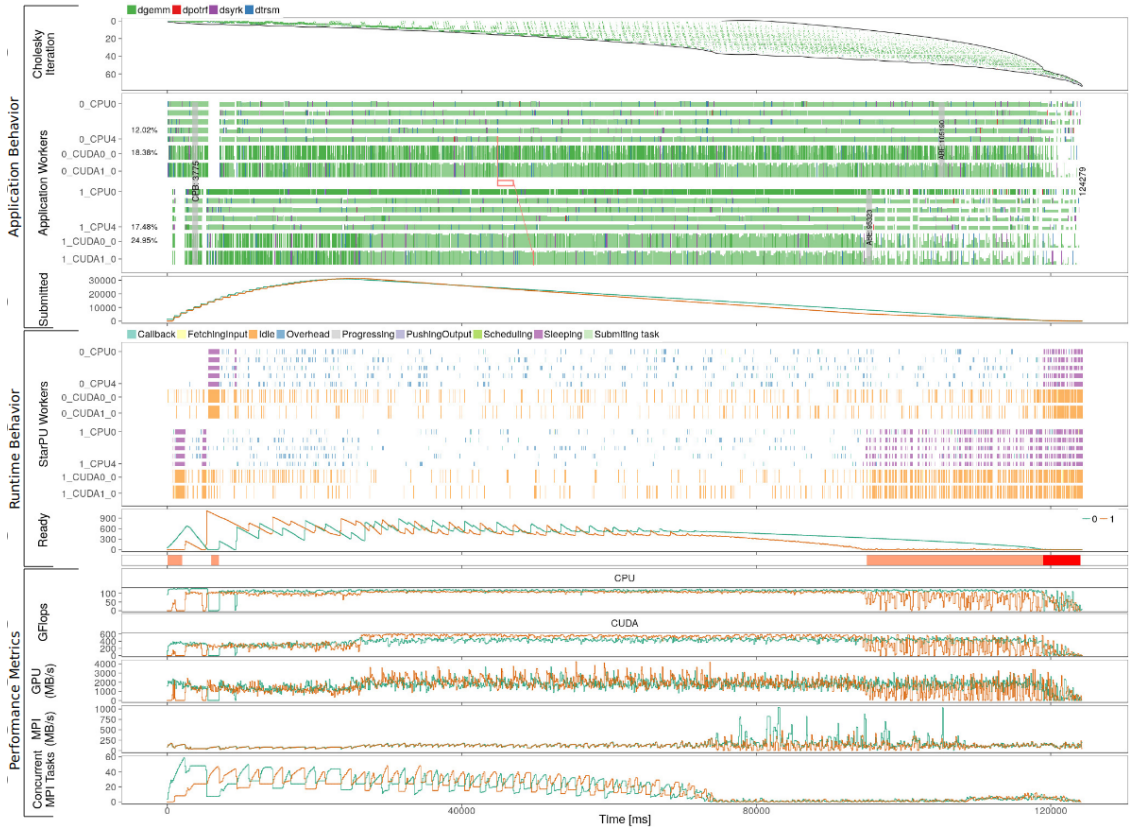
3.2.5 StarVZ

More recently, StarVZ (PINTO et al., 2018) has been proposed as an extensible R-based framework for the performance analysis of task-based HPC applications. It is a open source project and is available as an R package[1]. StarVZ includes several visualization panels enriched with theoretical bounds and task-dependencies delays that correlate observed performance with the DAG and is divided in two phases. In the first one, it reads a Paje trace file and conduct a series of data cleaning and organization saving the results in `feather` files. In the second phase, it reads the `feather` and a configuration file and generates the desirable plots. Furthermore, the data generated after the first phase can be manually worked in an R and reproducible environment to check some specific details of the visualization.

Figure 3.12 presents the visualization of Chameleon Cholesky execution using StarPU over two machines with two GPUs each one. All the panels have in the X-axis the time in ms. The first panel, Cholesky iteration, presents the iteration of the tasks in the Y-axis. The Application workers panel offers an extended Gantt chat of application tasks over the resources in the Y-axis. Extra information is present like resource idleness (the % in the left side), the boundaries (gray rectangles), and the identification of outlier

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

Figure 3.12: Example of visualization using StarVZ



Source: Pinto et al. (2018).

tasks (states with strong coloring). The submitted tasks, ready tasks, GFlops, GPUs, and MPI transfers are panels displaying some variables over time. The StarPU workers panel displays internal StarPU states on the resources.

3.3 Discussion about Memory Features and Summary of Tools

All discussed tools concentrate on the overall application behavior, where the computation or tasks being processing are the focus and are organized on the computational resources. Information about memory is usually expressed by metrics, like the total amount of communication, and events related to data are displayed within other ones (unrelated to memory) based on which resource they occur and not on which memory block. Even if some of these tools provide DAG-oriented support, they generally lack a specific methodology to analyze the impact of different memory block operations on application performance or to correlate the memory behavior of the application with other information. For example, the application memory of linear algebra problems is usually associated with a memory block coordinate, that a performance analysis tool can correlate

with processing aspects and possible optimizations. If an analyst wants to check where a particular memory block is at a specific point in the execution because maybe it is using space on a critical resource, none of the presented tools can provide such information easily. Moreover, all memory blocks are associated with application information. If the runtime copies a memory block to the GPU, most of the tools usually show the communication between the resources, but not specific details on the memory block that is related to the application (the block coordinate for example). An analyst can use this information to correlate performance, memory behavior, and application design. The problem is that the presented tools are unable to provide such actions or deeply investigate the memory at the runtime level.

The proposed strategies of this work are designed to provide a more memory-oriented analysis, based on the information from where the memory blocks are and which tasks they are related to, forming dependencies. The approach provides a multi-level performance analysis of data management operations on a heterogeneous multi-GPU and multi-core platform. This work combines a high-level view of the application DAG with the low-level runtime memory decisions, which guides the analyst in identifying and fixing performance problems. Instead of only using architectural low-level memory metrics and comparing them with multiple executions, this work focus on the behavior understanding of representative executions. This work also designs visualization elements specifically for the performance analysis of memory in a DAG-based runtime, enriching the perception of task-based applications running on heterogeneous platforms. Table 3.1 shows a summary of the presented related works, comparing with this work, informing the type of applications that the tool was designed for and one of the notable contributions.

Table 3.1: Presented tools to analyze HPC applications

Related Work/Tool	**Type**	**Notable Contribution**
Kergommeaux et al. (2000)	Traditional	Generic tool
Pillet et al. (1995)	Traditional	Generic tool
Coulomb et al. (2009)	Traditional	Fast rendering
Knüpfer et al. (2008)	Traditional	Scalable
Isaacs et al. (2014)	Traditional	Logical time
Muddukrishna et al. (2016)	Task-Based	Fork/Dag oriented
Huynh et al. (2015)	Task-Based	Fork/Dag oriented
Keller et al. (2012)	Task-Based	Online DAG GDB
Ceballos et al. (2018)	Task-Based	Tasks memory reuse analysis
Pinto et al. (2018)	Task-Based	Enriched visualizations
This Work	**Task-Based**	**Memory-oriented**

Source: The Author.

Moreover, the tools can be compared by how they present information related to memory. More specificaly about communications and general memory information. Table 3.2 presents such comparison using the following two attributes, communication and memory information.

Table 3.2: Comparison of tools to analyze HPC applications

Related Work/Tool	**Communications**	**Memory Information**
Kergommeaux et al. (2000)	Programmable	Programmable
Pillet et al. (1995)	Programmable	Programmable
Coulomb et al. (2009)	Between resources	Metrics
Knüpfer et al. (2008)	Between resources	Metrics
Isaacs et al. (2014)	Between resources	No
Muddukrishna et al. (2016)	No	No
Huynh et al. (2015)	No	No
Keller et al. (2012)	DAG-oriented	GDB-oriented
Ceballos et al. (2018)	No	Metrics
Pinto et al. (2018)	Metrics	Metrics
This Work	**Memory blocks Oriented**	**Individual memory blocks events/locations**

Source: The Author.

Communications: How the tool displays information about communications, with the following list. (i) Between resources, focusing that a transfer happened between resource A to B without specifying the memory. (ii) Memory blocks oriented, where the information about which block was transferred is used as the main aspect. For example, block A was on resource A on a specific moment and moved to another one. (iii) Metrics, where all the information is summarized. (iv) Programmable, where the tool permits customization of the events. (v) DAG-oriented, where the communications can be inferred by the displayed DAG. (vi) No information about communication is presented or possible.

Memory Information: The aspects of memory information/behavior that the tool displays, a list of values follows. (i) Programmable, where the tool permits customization of the events. (ii) Metrics, where all the information is summarized. (iii) GDB-oriented, where the application can be stopped and opened using GDB so that all information can be checked out. (iv) Individual memory blocks events/locations, where the information is primarily given for each memory block. (v) No information about memory is presented.

4 CONTRIBUTION: STRATEGIES FOR ANALYZING TASK-BASED APPLICATIONS MEMORY AT RUNTIME LEVEL

The management of an application's memory is an essential factor to be analyzed, as it can negatively impact on the overall application's performance. This situation is especially true on task-based applications running on heterogeneous platforms, as the DAG is structured using memory dependencies, and the multiple hybrid devices have distributed memory. Tools that can provide insights about the memory management's behavior and performance are desirable to facilitate the analysis of both task-based applications and runtimes. This chapter presents the main contribution of this work, the methodology and strategies to investigate the memory manager behavior and individual memory blocks' events/locations on the heterogeneous CPU/GPU resources.

The memory-aware visualization panels are designed to leverage the behavioral about memory activities. The presence of memory blocks on each memory manager is used to understand the general behavior of the application. The general objective of these new panels is to analyze the behavior both in a broad vision of the execution, using visualizations that summarize and give a footprint of the memory, as also to deeply verify the details of where each memory blocks are and how they impact in the tasks' execution.

StarPU's data management module is responsible for all actions involving the application's memory. Although the trace system of StarPU already had some events about the data management, detailed information about the memory blocks are still absent from the original trace system. We add or modify in the StarPU's trace system events to track the data allocation on resources, the data transfers between nodes, and extra data blocks identification. All this information now provide individual characteristics for each memory block.

The structure of this chapter follows. All the extensions of the StarPU trace system are present on Section 4.1. Then, five strategies are proposed to improve memory performance analysis. Section 4.2 presents a memory-aware space/time view to understand the activities that the memory managers are conducting and the related memory blocks. Section 4.3 depicts a strategy to have a fast visualization of the presence of a data handle on the resources correlating to the application memory information. Section 4.4 presents a comprehensive temporal behavior of a memory block, presenting several events that affected it during the application execution. Section 4.5 shows a visualization used to track the situation of the memory at a specific point of the application execution. This

visualization can be used with multiple timestamps to generate animations. Section 4.6 presents a heatmap to create a footprint of the presence of the handles on the resources.

4.1 Extra Trace Information added in StarPU to Track Memory Behavior

The capture of traces is a feature already present in the StarPU runtime, and this work only extends it to add new information. Section 2.3 provides some explanation about the StarPU and presents some information about its trace system. This Section describes the modifications done on the trace module of StarPU. Essentially, to include all the necessary information, this work conducts three major modifications. The first modification is to include the events' memory identification on all related events with extra information to allow correlations between runtime activities and to understand the decisions behind it. Second, a modification to trace the memory's coherence update function to keep track of the whereabouts of a memory block during the execution. Third, a modification to track all memory requests (prefetch, fetch, allocation) carried out by the runtime.

Table 4.1 presents the new events added in the StarPU trace system. The first three new events are added to track the modifications on the MSI protocol and added to the StarPU function `_starpu_update_data_state`. These events are the base for all strategies, as they are faithful records of the state of the data handles. With these events, there is no need to infer the location of the handles based on the DAG or transfer events, as the internal MSI protocol of the runtime system provides the correct location.

Moreover, a new event to track the internal memory management transfer request was created and added to the function `_starpu_create_data_request`. The objective with this event is to show exactly when the runtime acknowledges that the data is needed somewhere, the time of this event to the end of the transfer implies on how much time the runtime is losing processing other memory operations. Also, this event can assume a prefetch or a fetch mode, changing the priorities when the runtime process all the requests. The last added event tracks an identification for the MPI communications to the original data handles. This event is important because it makes it possible to correlate the transfers between nodes to an individual memory block.

Table 4.2 shows the already existing FXT events that were modified to include some extra information as follows. The first four events, Driver Copies, were modified to include the thread identification that called the event. The thread identification is recurrent information in other events, however, wasn't included in the Driver copies. The other

Table 4.1: New FXT events created to extend the StarPU trace system

New FXT Event	Description	Function where applied
DATA_STATE_INVALIDATE	Track the invalidate state of the MSI protocol	_update_data_state
DATA_STATE_OWNER	Track the owner state of the MSI protocol	_update_data_state
DATA_STATE_SHARED	Track the shared state of the MSI protocol	_update_data_state
DATA_REQUEST_CREATED	Track the creation of transfers requests	_create_data_request
MPI_DATA_SET_TAG	Track the TAG for MPI communications	_mpi_data_register_comm

Source: The Author.

events, allocation, free, and write back were originally tracing when they occurred and not informing which handle they were working on, so the main modification was to include the handle identification in all these events. The identification is the pointer of the data handle structure that other events also use to associate more information, like application block coordinates. Also, at the start of the allocation events, one modification is to add information if the event is a prefetch or a fetch. In the same way, at the end of these allocation events, the modifications add the result of the action, if it was successfully or not.

Table 4.2: Modified FXT events to extend the StarPU trace system

Modified FXT Event	Modification
START_DRIVER_COPY	Include StarPU thread that called it
END_DRIVER_COPY	Include StarPU thread that called it
START_DRIVER_COPY_ASYNC	Include StarPU thread that called it
END_DRIVER_COPY_ASYNC	Include StarPU thread that called it
START_ALLOC	Include Handle identification and type (prefetch/fetch)
END_ALLOC	Include Handle identification and allocation result
START_ALLOC_REUSE	Include Handle identification and type (prefetch/fetch)
END_ALLOC_REUSE	Include Handle identification and allocation result
START_FREE	Include Handle identification
END_FREE	Include Handle identification
START_WRITEBACK	Include Handle identification
END_WRITEBACK	Include Handle identification

Source: The Author.

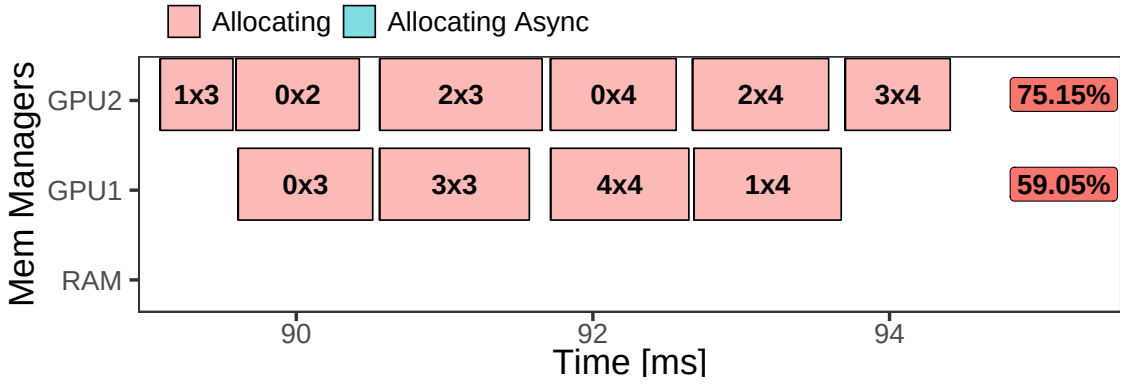
After the addition and modification of the FXT events, the converter of FXT StarPU traces to Paje was modified to reflect the changes. Also, this program could write direct Paje files or use Poti; both approaches were modified to incorporate the new memory events. All these modifications, with the exception of the MPI one, were added

to the StarPU main repository[1].

4.2 Enriched Memory-Aware Space/Time View

Employing Gantt-charts to analyze parallel application behavior is very common, as seen previously in Section 3.1. It is used to show the behavior of observed entities (workers, threads, nodes) along time. This work adapts and enriches such kind of view to inspect the memory manager behavior, as shown in the example of Figure 4.1. On the Y-axis, the figure lists the different memory managers associated to different device memories: `RAM`, different accelerators (memory of GPU and OpenCL devices). This example only has three memory managers: `RAM`, `GPU1`, and `GPU2`. The memory managers can perform different actions like allocating and moving the data over a specific data handle. Each handle can have an extra information provided by the application. In this case, the block coordinates from the Cholesky factorization.

Figure 4.1: The behavior of three memory managers (`RAM`, `GPU1`, `GPU2`) showing allocations states for different memory blocks



Source: The Author.

Figure 4.1 presents the actions of each manager over time with colored rectangles tagged with block coordinates (i.e., for `GPU2`: 1×3, 0×2, and so on) from the application problem. The rectangles in this figure mainly represent different `Allocating` states carried out by those managers, except for the `RAM` manager that had no registered behavior in the depicted 10ms interval. In the right of each manager line, the panel describes the percentage of time of the most recurring state, using the same color. For instance, the `GPU2` manager spent 75.15% of the time of this specific time interval in the `Allocating` state.

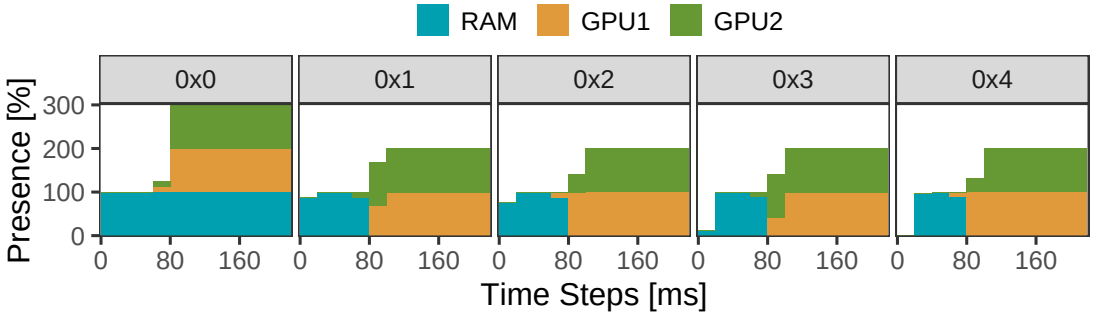
[1] Available on https://scm.gforge.inria.fr/anonscm/git/starpu/starpu.git, commits 784095d, e097aa8, f068631, and 756e365.

This plot provides an overview of the actions of the memory managers, also informing on which block they are applying such actions.

4.3 Block Residency in Memory Nodes

A given block coordinate of an HPC application (i.e., Cholesky factorization) may reside in multiple memory nodes over the execution. For example, there can be many copies of a given block if workers executing on different devices perform read operations only. This situation is due to the adoption of the MSI protocol by StarPU, where multiple memory nodes have copies of the same memory block (see Section 2 for details). Figure 4.2 represents the location of a given memory block over the execution. Each of the five facets of the Figure represents one memory block with the coordinates 0×0, 0×1, 0×2, 0×3, and 0×4 of the input matrix. For each block, the X-axis is the execution time, discretized in time intervals of 20ms. This interval is sufficiently large for the visualization and small enough to show the application behavior evolution. However, other values could be selected for different situations based on the total duration of the application execution. At each time interval, the Y-axis shows the percentage of time that this block is on each memory node (color). For example, if a block is first owned by `RAM` for 18ms and then for 2ms by `GPU2`, the bar will be 90% blue and 10% yellow. Since each block can be shared and hence present on multiple memory nodes, the maximum residency percentage on the Y-axis may exceed 100%. The maximum depends on how many memory nodes there are on the platform. Moreover, if the memory resides for only a portion of the time interval, the percentage would be less than 100%.

Figure 4.2: The location of memory blocks along time



Source: The Author.

With this new visualization, it is possible to check a summarized evolution of data movement and resource's memory utilization. For example, Figure 4.2 details that the

memory block with coordinates 0×0 stayed in `RAM` throughout the execution. StarPU transferred it to both GPUs at time ≈80ms and kept it in all resources until the end of the execution. The other blocks, however, remained in `RAM` only until ≈80ms of the execution and were transferred to both GPUs. An analyst is capable to quickly spot anomalies by correlating the block coordinates residence with the application phases. Very frequently in linear algebra, a lower block coordinate is only used at the beginning of the execution, so it should be absent after some time (which would be demonstrated as 0% occupancy of that block after it is no longer needed).

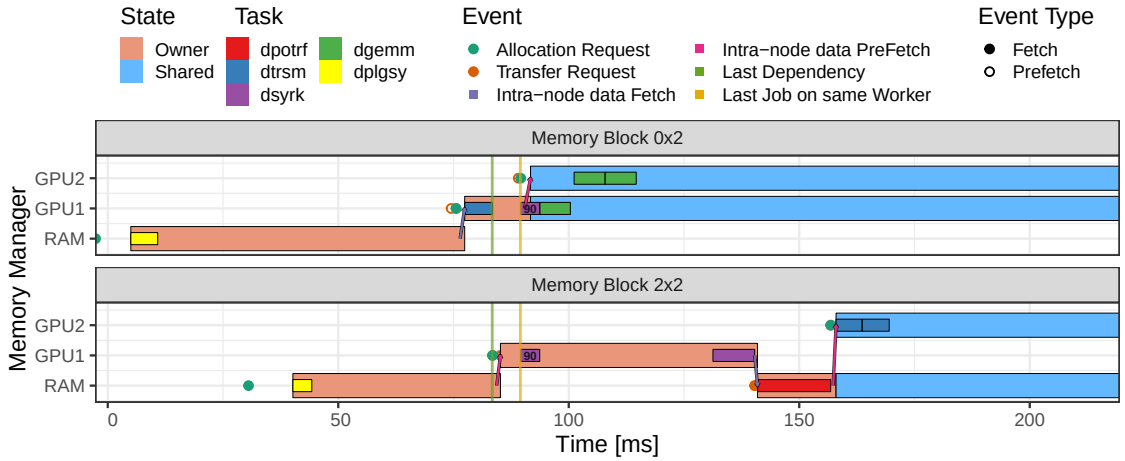
4.4 Detailed Temporal Behavior of Memory Blocks

The previous panel (Section 4.3) shows where a given block is located (on which memory node) throughout the execution, however, it does not provide details of when transfers occurred and what events of StarPU are influencing the data handles. To track these issues, a complete history of events for each data handle would be desirable for understanding specific situations. This is present in Figure 4.3, showing the memory block location with the MSI state, and additionally depicts all the runtime and application tasks activities that affect the block behavior. These activities include runtime transfers between resources, internal runtime memory manager events such as transfer and allocation requests, and task-related information like last dependency and task on the same worker times.

The strategy employs the traditional Gantt-chart as a basis for the visualization, where the X-axis is the time in milliseconds, and the Y-axis represents the different memory managers. There are two types of states, depicted as colored rectangles. The ones shown in the background with a more considerable height represent the residency of the memory block on the managers: the red color expresses when a memory node is an owner, while the blue color indicates the block is shared among different managers. The inner rectangles represent the Cholesky tasks (`dpotrf`, `dtrsm`, `dsyrk`, `dgemm`, and `dplgsy`) that are executing and using that memory block from that memory manager. The strategy presents augmented representation with different events associated with the memory blocks on the respective manager and time. The circles (`Allocation Request`, `Transfer Request`) are either filled or unfilled, for fetch or prefetch operations, respectively. The arrows are used to represent a data transfer between two memory nodes and have a different meaning (encoded with different colors: intra-node prefetch

and fetch). Finally, two vertical lines indicate the correlation (last dependency and last job on the same worker) with a specific application task that one in this example wants to study.

Figure 4.3: The detailed view of all events related to two memory blocks (facets), showing residence (larger height rectangles) and the use by application tasks (inner rectangles)



Source: The Author.

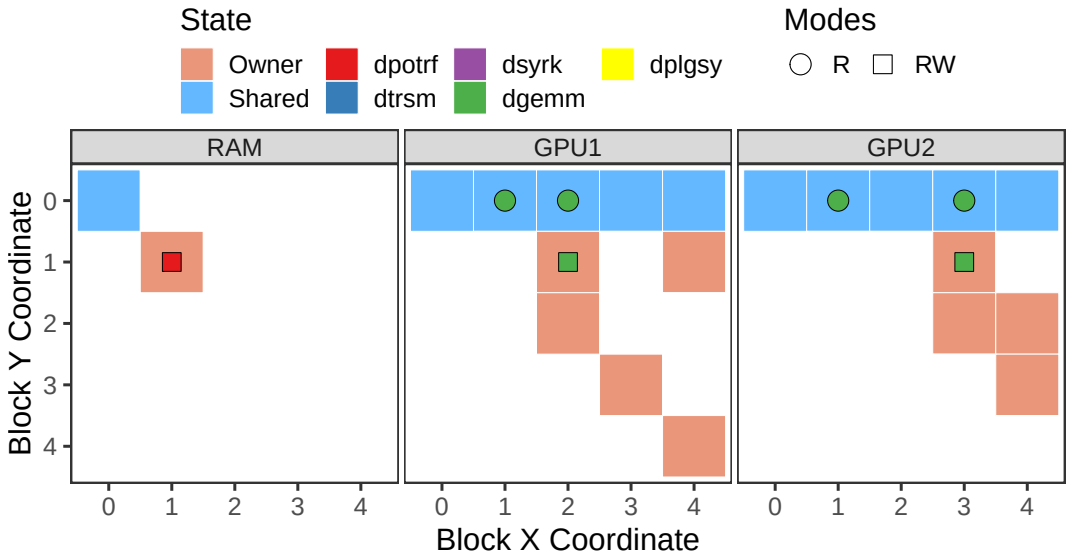
In Figure 4.3 the task ID 90 is highlighted (which is a `dsyrk` task), and only data handles used by task 90 are shown. The green vertical line represents the end of the last dependency that releases task 90, and the yellow represents the end of the last task executed on the same worker. Also, the Figure shows that block 0×2 was owned by memory RAM since the execution of its generation task `dplgsy` and was moved to GPU 1 to be used by a `dtrsm` task. During the execution of task 90, block 0×2 was moved to GPU 2 on a shared state, where its information is duplicated on both GPUs. Before each of the transfers, the events of transfer and allocation requests were present.

4.5 Snapshots to Track Allocation and Presence History

The application running on top of StarPU determines the data and the tasks that will be used by the runtime. Instead of only considering the utilization of resources, it would be useful to correlate the algorithm and the runtime decisions. This strategy creates a view that takes into account the coordinates of the blocks in the original data, illustrating which task is using each block, and their state on the managers (owned, private or shared). Figure 4.4 depicts a snapshot of all memory blocks locations and the running tasks in a specific time. The visualization has three facets, one for each of memory managers (`RAM`, `GPU1`, `GPU2`). Each manager has a matrix with the block coordinates in the X and Y-

axis. On this matrix, each colored square represents one memory block on the coordinates provided by the application, the color of each block informs the actual MSI state of the data handle on the moment of the snapshot. The colored inner squares (write mode access) or circles (read mode access) inside those blocks represent application tasks. With this visualization, it is easy to confirm how the memory data flow correlates with the position of the blocks and understand the progression of the application memory also using the information provided by the application (in this case the block coordinates).

Figure 4.4: A snapshot of memory block residence



Source: The Author.

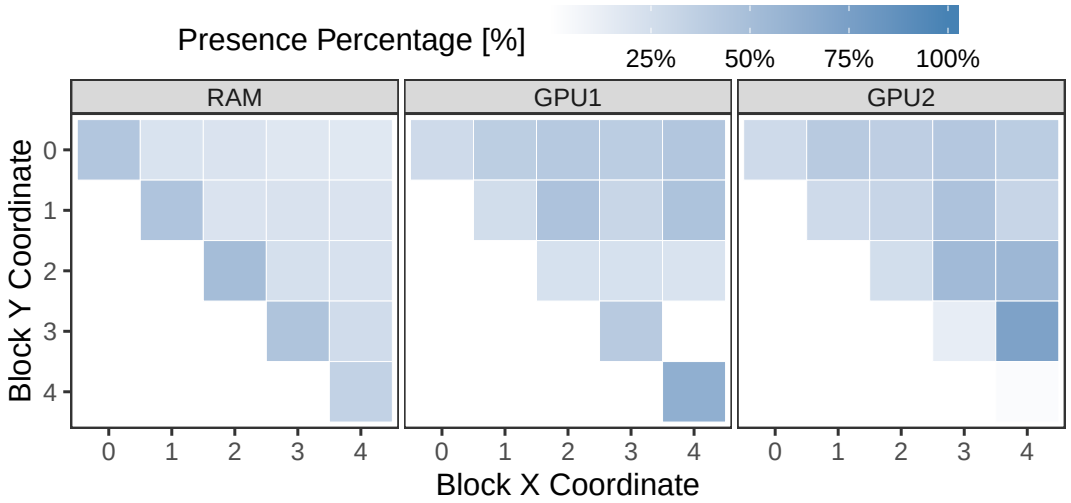
In Figure 4.4, for example, there are only two blocks on `RAM` and that both GPUs share the first row. Moreover, there is a `dpotrf` task executing over block 1×1 in `RAM` and a `dgemm` task on each GPU. `GPU1` has write access on the `dgemm` task on block 1×2, and two read accesses on blocks 0×1 and 0×2. By stacking consecutive snapshots, it is possible to create an animation that shows the residence of memory blocks along time. This feature is particularly useful to understand the algorithm behavior and the data operations.

4.6 Heatmap to verify Memory Block Residence along Time

Apart from the previous memory snapshot visualization, an analyst could be interested in an execution overview of the handles locality among the managers. A footprint of the overall residency of the data handles could provide fast insights about the behavior. The final panel consists of a traditional heat map visualization to provide a summary of

the total presence of the tiles on each manager. Figure 4.5 depicts an example of this strategy. There is one visualization facet for each memory manager; in the example, there is a RAM, GPU 1, and GPU 2 memory managers. Each square represents a memory block positioned on its application matrix coordinate on the X and Y axis. Other structures could be used depending on what extra information the application gives to the memory blocks. The blue color tonality depicts the total amount of time that the block is on the manager during the application execution. While the whiteish color represents low presence on the memory manager, the more blueish color represents high presence.

Figure 4.5: Per-manager heatmaps for memory blocks presence



Source: The Author.

In Figure 4.5, for example, it is possible to observe that all blocks of the diagonal stood more time on `RAM` compared to other blocks. This situation happens because, for the Cholesky, `dpotrf` and `dsyrk` tasks on the diagonal are typically executed on CPUs, to let the GPUs process mostly SYRK and GEMM tasks. Differently, the other blocks were present on both GPUs, as GEMM tasks were preferred executed on GPUs and use all those blocks. Also, all blocks have at least a little presence on the memory ram, as all blocks were generated first on there.

5 EXPERIMENTAL RESULTS ON TASK-BASED APPLICATIONS

This chapter presents the experimental results with three different heterogeneous task-based applications: a dense Cholesky, a CFD simulation, and a sparse QR factorization solver. On each application, the strategies were used for analyzing the memory at the runtime level. Section 5.1 presents the general methodology applied to the experiments, with details about the hardware and software configuration. Section 5.2 investigates the performance of data transfer and out-of-core algorithms with the tile-based dense Cholesky factorization as implemented in the Chameleon solver suite (AGULLO et al., 2010). Section 5.3 explores how the strategies can aid the optimization of a partitioning method of a CFD application (NESI; SCHNORR; NAVAUX, 2019) and what factors can change the decisions of the memory manager. Section 5.4 presents experiments with the software package `QR_MUMPS` (BUTTARI, 2012), a sparse QR factorization software, checking if the memory is causing any performance problems on the application. Section 5.5 ends the chapter discussing the strategies proposed and the known limitations.

5.1 Platform Environment

The experiments present in this document were a result of an exploration of problems in linear algebra and scientific simulation applications. The experiments submit the applications to a series of tests, to stress the memory management system and collect execution traces. With the aid of the strategies for analyzing the memory, we have checked if the memory management causes any problem and impact in the application performance. When problems were found and confirmed, we entered in contact with the developers of the software to inform the findings, discuss solutions, and to provide some insights about memory behavior. Table 5.1 presents the machines used in the experiments, all from the GPPD-HPC group. The `Tupi` machine has an Intel Xeon CPU E5-2620 with eight physical cores, 64 GB of DDR4 RAM, two GeForce GTX 1080Ti 11GB, and a 2TB SSD (WDC WDS200T2B0B) as the auxiliary memory for the out-of-core experiments. The `Draco` machines contain two Intel E5-2640 with eight physical cores, 64GB of DDR3 RAM, one Tesla K20m and are interconnected with a 10Gb Ethernet switch.

The experiments follow the reproducible approach in controlled environments where the data have all the system information alongside with the executions' traces. The companion of the experiments is available at <https://gitlab.com/lnesi/master-companion>.

Table 5.1: Description of machines used in the experiments

Machine	CPU	Memory	GPU	Storage
Tupi	Intel E5-2620	64GB	2× GTX 1080ti	2TB SSD
Dracos[2-3]	2× Intel E5-2640	64GB	1× K20m	2TB HD

Source: The Author.

5.2 Experimental Results with Dense Cholesky

This section presents the experiments conducted with the Cholesky code from the Chameleon solver. Four experiments are conducted analyzing different memory behavior. The machine used in the first four experiments is `Tupi`, and `Dracos` for the last one. The StarPU and the Cholesky code from the Chameleon have been compiled with GCC 7.3.0. The Cholesky runs use a block size of 960x960. The experiment uses Debian 4.16 for the first scenario and Ubuntu 18.04 for the other cases.

Section 5.2.1 described the Chameleon application, with the Cholesky tasks and DAG. Section 5.2.2 presents the identification of a runtime's wrong perception of the total used memory in GPU, causing StarPU to issue numerous allocation requests that impact on performance. Moreover, a correction is proposed and a comparison of the original version of StarPU and the correct one is presented. Section 5.2.3 discuss why the workers were presenting idle times when using the out-of-core feature and possible internal insights on how to solve it. Section 5.2.4 focus only on out-of-core experiments and how the `LWS` and the `DMDAR` schedulers get affected by the matrix input generation with limited `RAM` memory. Section 5.2.5 presents how `DMDAR` and `DMDAS` behave in a CPU-only setup with very limited `RAM` and the differences of scheduler causes in memory management. Section 5.2.6 presents a proof-of-concept using a multi-node MPI execution.

5.2.1 The Chameleon Package

The Chameleon package (AGULLO et al., 2010) contains a series of dense linear algebra solvers implemented using the sequential task-based paradigm. From the set of available solvers, the experiments adopt the task-based solver that implements the dense linear algebra Cholesky factorization on top of the StarPU runtime, because many HPC applications used it as a computing phase. The Cholesky factorization algorithm runs over a triangular matrix divided into blocks, using four different tasks: `dpotrf` (Cholesky

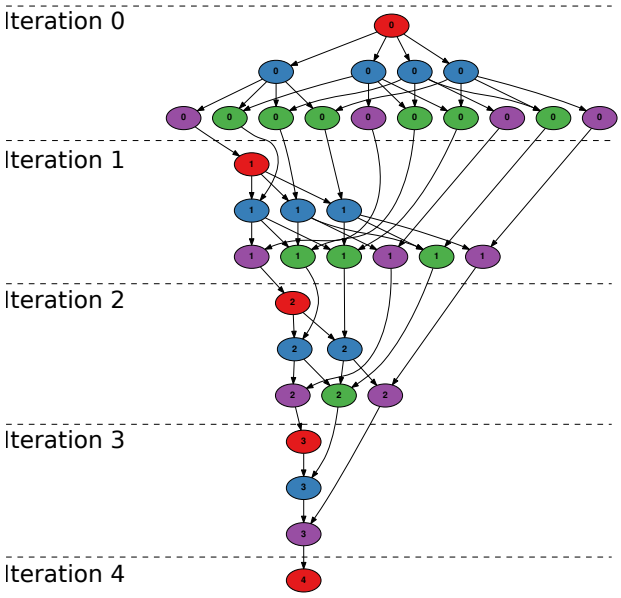
Factorization), `dtrsm` (Triangular Matrix Equation Solver), `dsyrk` (Symmetric Rank-k Update) and `dgemm` (Matrix Multiplication), as shown in Figure 5.0a. The task-based Cholesky factorization divides the input matrix into tiles (blocks), making each task associated with a block. The factorization essentially begins with tasks on lower coordinates blocks and iteratively computes all matrix blocks for all coordinates. The Figure 5.0b demonstrates the resulting DAG for a matrix divided into 25 blocks (N = 5).

Figure 5.1: The tiled Cholesky code and the DAG for $N = 5$

(a) The Cholesky Algorithm

```
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) Corresponding DAG for $N = 5$



Source: The Author.

The Chameleon framework generates the full matrix to conduct numerical checks. Since, in this case, the solver is used independently of real application code without loading a matrix, the Chameleon testing code includes an input generation task called `plgsy` to create floating-point values for the matrix tiles.

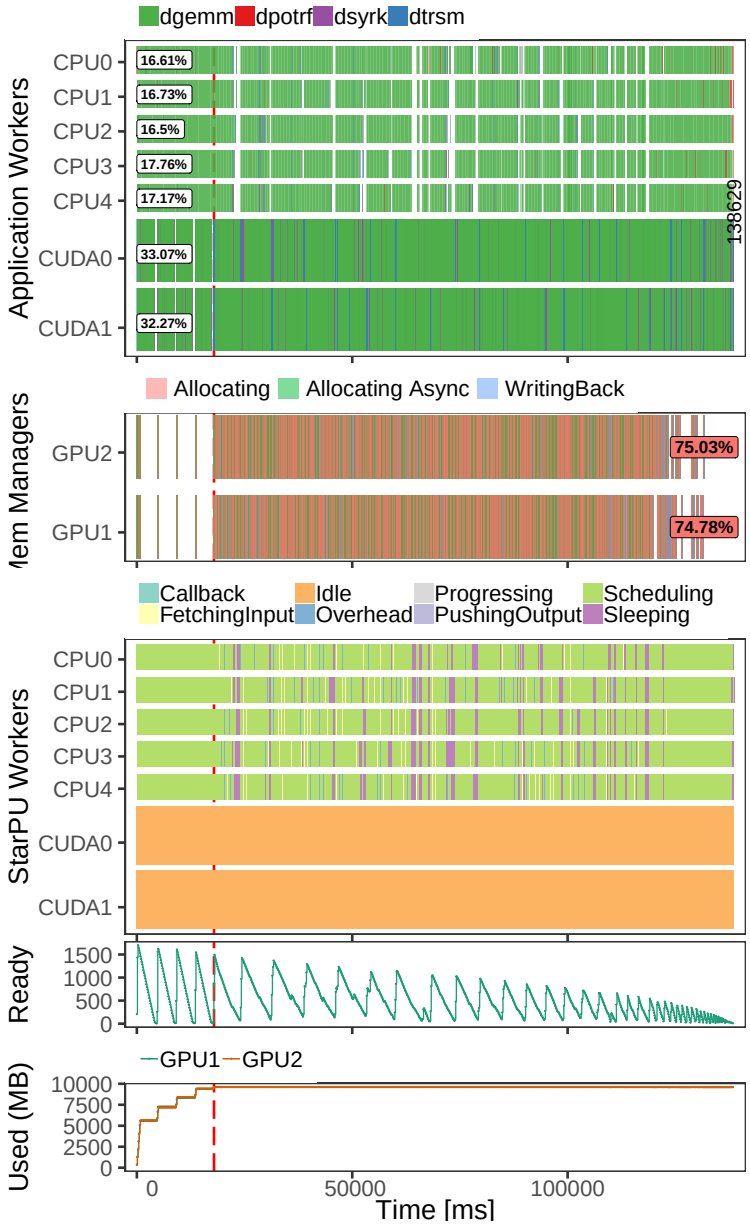
5.2.2 Erroneous control of total used memory of GPUs

Preliminary tests indicated that the dense Cholesky factorization of the Chameleon solver had performance problems when using an input matrix larger than the GPU memory size. The StarVZ overview methods were applied and identified that GPUs are generally idle. This problem motivates further investigation of this issue using other views of the proposed methodology. This section describes the identification of the problem and the complete resolution. In the subsequent executions, the Cholesky uses 60×60 blocks.

With the previously mentioned configuration, five CPU workers, two GPU workers, and the DMDA scheduler, this experiment generate a trace of a problematic execution. First, the StarVZ workflow is applied to check general performance issues in the application. Figure 5.2 presents, from top to bottom, the StarVZ plots: **(a)** Application Workers, where each state is a task, the left percentage in each resource are the total idle time of the execution on that resource, and the right number is the total makespan; **(b)** Memory Manager, the strategy described in Section 4.2, where each state is a memory manager action and the left percentages the total time used on the most present state; **(c)** StarPU Workers, where each state represent an action of StarPU on the resources; **(d)** Ready Tasks, the total number of ready tasks; and **(e)** Used memory, the total amount of used memory registered by StarPU. In **(a)** and **(c)** the Y-axes represent the workers (CPU cores and GPU devices), in **(b)** the memory nodes (in this case, `GPU1`, `GPU2`), in **(d)** the number of ready tasks, and in **(e)** the per-GPU memory utilization in MB. In all plots, the X-axis is the time in milliseconds (ms). Each state has a different color associated with its task or action. The red vertical dashed line has been manually added to emphasize the moment where the total GPU used memory reaches a plateau with its maximum value. The GPUs spend a lot of time idle (`GPU1` with 33%, `GPU2` with 32%), which is impairing the overall application performance. Also, the GPU memory managers started to execute many allocation actions.

The possible correlation between idle times and allocation states led to an investigation of memory manager actions after the memory utilization reached its maximum. An arbitrary time frame is selected since the behavior is similar after achieving the memory utilization peak. Figure 5.3 provides a temporal zoom ($\approx$50ms period) on the Memory Manager panel, depicting multiple `Allocating` actions (red rectangles) for different memory block coordinates of the input matrix. There are many allocating attempts of the same memory blocks occurring many times. This repeated behavior is considered a

Figure 5.2: Multiple Performance Analysis plots for the Cholesky execution

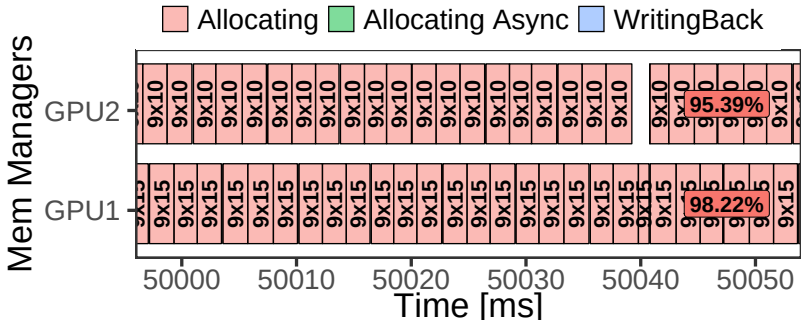


Source: The Author.

problem because a single allocation request should be enough. This analysis leads to an investigation of the StarPU source code responsible for the allocation, which indicated that the runtime keeps trying to allocate blocks indefinitely until it gets a successful allocation. The GPU resources monitor confirmed that the GPUs are using all the memory. One possible explanation is that the devices do not have enough memory, but this should not be a problem, as StarPU could free multiple memory blocks, especially those that would no longer be used by the Cholesky factorization.

Next step was to employ the Block Residency plot to understand the previously described behavior. The selected blocks are in the initial iterations of the outermost loop

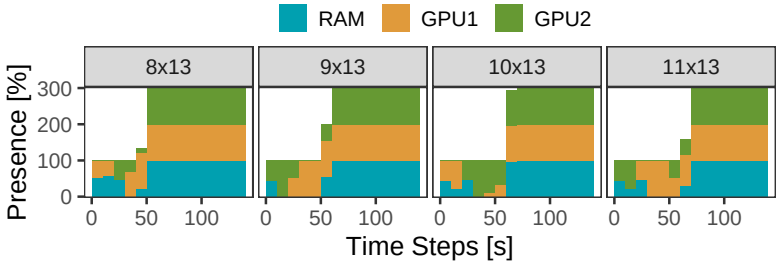
Figure 5.3: Memory Manager states on a time frame of ≈50ms



Source: The Author.

of the tiled Cholesky algorithm, with lower coordinates. They are expected to be more appropriate for being freed rapidly. Figure 5.4 shows that blocks 8×13, 9×13, 10×13, and 11×13 become present in all memory nodes at ≈50s. Shared copies of these blocks remain in each memory manager until the end of execution (at ≈140s). The presence of these blocks in all memory nodes indicates that StarPU is deciding not to free them.

Figure 5.4: Per-node block residency of [8-11]×13 coordinates



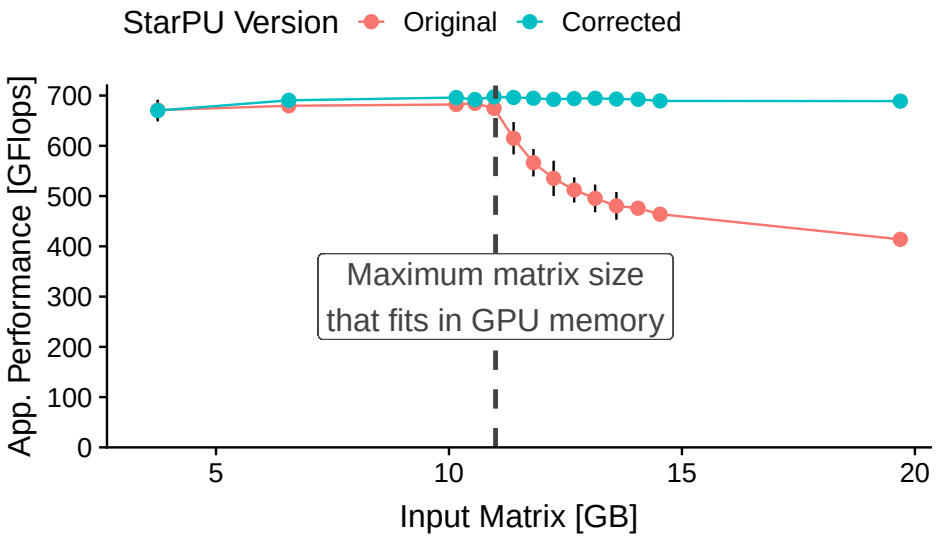
Source: The Author.

All these insights gave enough information to inspect decisions directly. `GDB` is used to check StarPU's functions that free unused memory blocks. StarPU all the time believed that it had free space on the GPUs. Also, a huge difference was detected when comparing the internal StarPU's used memory values to the ones given by the GPU resources monitor. This difference led to the discovery that the CUDA function `cudaMalloc` could allocate more memory than the requested size, and the runtime was not considering it. The function rounds the demanded memory to a device dependent page size, effectively causing internal memory fragmentation. The GTX1080Ti has a fixed page size of 2 MB; however, other GPUs can present different values. Since the used blocks are 960x960 (7200 KB each), every block allocation request in the GPU causes a loss of 992 KB (because it returns four pages to answer that request). That leads to loosing 1800 MB for the 60×60 tiles used in the experiments. StarPU was miss-calculating the GPU memory utilization and kept calling the expensive `cudaMalloc` function even without

GPU memory causing long periods processing the memory management system and the idle times. We then proposed multiple fixes for the StarPU developers, and the chosen one was to verify the memory of the GPU using the CUDA API before the call of the expensive `cudaMalloc` function. Experiments were conducted with the original version of StarPU of commit `be5815e` and the corrected one measuring the Cholesky performance before and after the patch.

Figure 5.5 presents the performance comparison between two StarPU versions: original (red color) and corrected with our patch (blue). For both versions, the experiment employs the same application and runtime configuration with the DMDA scheduler. The figure depicts the application performance in GFlops on the Y-axis as a function of the input size, on the X-axis. The input size distribution has more points around and after the memory limit (marked by a vertical dashed line, calculated based on rounding behavior, number of blocks and the CUDA driver used memory) when the matrix size no longer entirely fits on the GPU memory. The dots represent the mean of ten executions, and error bars are drawn based on a 99% confidence interval. The original version demonstrated a significant performance drop after the memory threshold, falling from the relatively constant ≈700 GFlops rate to poorer values. After the patch, the corrected version keeps the performance stable at the ≈700 GFlops rate. These results demonstrate the effectiveness of the fix, maintaining the program's scalability as the input size increases. The fix was added on the master branch of StarPU on commit `ca3afe9`.

Figure 5.5: Application performance (GFlops) for the original and the fixed version

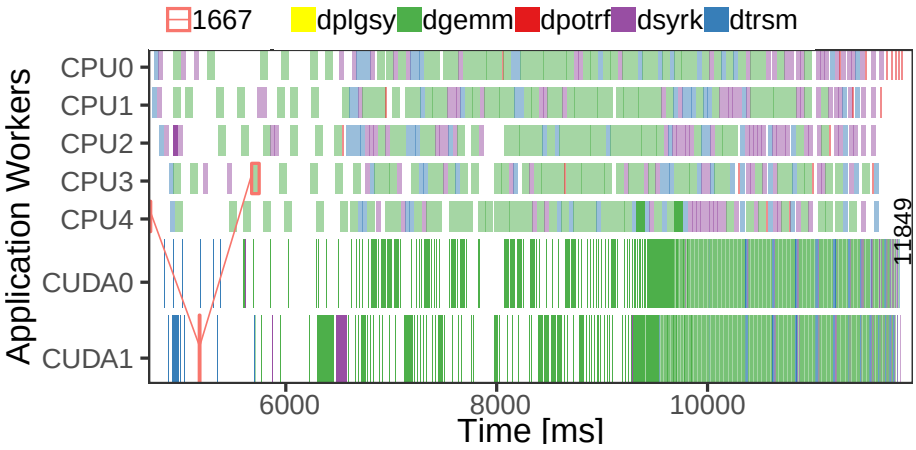


Source: The Author.

5.2.3 Understanding the idle times on out-of-core use

This test case uses the out-of-core version of the dense Cholesky factorization of Chameleon, using the developer branch of StarPU. The first experiment has the following configuration: block size of 960x960, with 20x20 tiles, and the `DMDAR` scheduler. Using StarPU environment variables, the `RAM` was artificially limited to 1.1 GB to stress the out-of-core support. Figure 5.6 depicts the behavior of five CPU and two CUDA workers. The initial phase of the execution presents idle times on all workers. The red highlight task (the right-most selected rectangle in the figure), of identification 1667, is preceded by a significant idle time. This task can aid in the understanding of this poor behavior and why it started was delayed. The red line arriving at the chosen task indicates the two last dependencies to have been satisfied to execute Task 1667. The task dependency that enables task 1667 occur significant time before it, so this is not a task scheduler problem, but maybe a memory one. At least ≈500ms have passed between the execution of the last dependency and the selected task.

Figure 5.6: Application workers using out-of-core, DMDAR scheduler, block size of 960x960 and 20x20 tiles



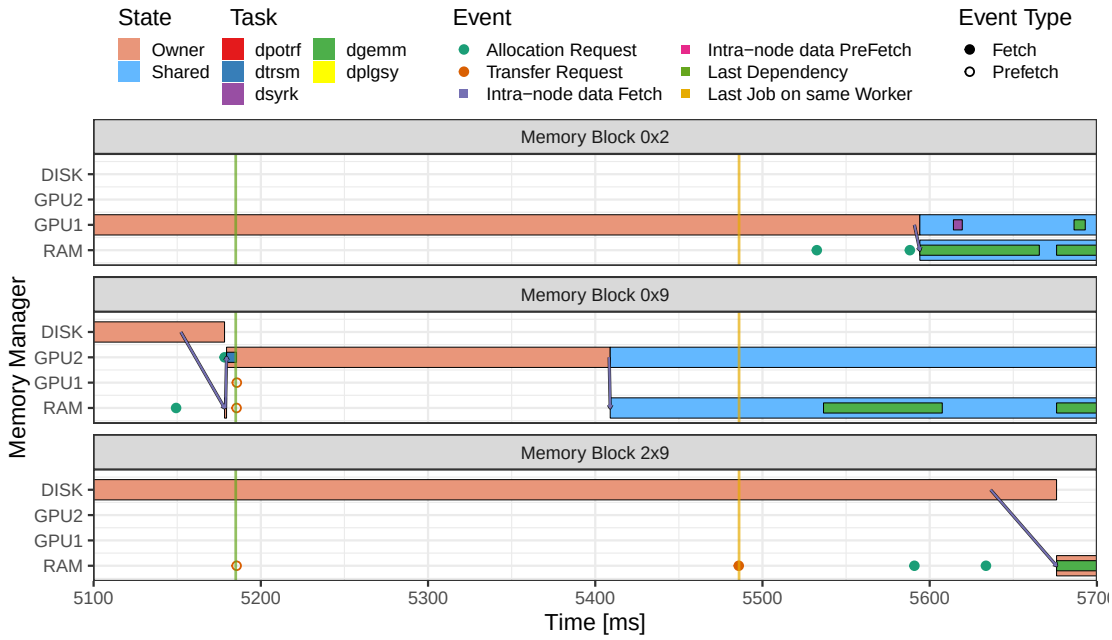
Source: The Author.

Task 1667 uses the memory blocks 0×2, 0×9 and 2×9. The detailed history of these memory blocks is employed to identify if the reason behind the idle times in the initial phase of the execution is due to late data transfers. Figure 5.7 depicts the behavior between 5100ms and 5800ms. Task 1667 is a `dgemm` operation marked as a green inner rectangle in all facets (one per memory block). This task starts exactly at 5676ms, while its last dependency finished at 5185ms: a vertical green line highlights this moment. The last task that executed on the same worker (CPU3, not shown), ended at time 5486ms and

is marked as a vertical yellow line.

Figure 5.7 shows that when the last dependency ends, there are prefetch transfer requests created on blocks 0×9 and 2×9, and this request is satisfied on block 0×9 with a transfer. However, Task 1667 only starts to be executed after the transfer of block 2×9, meaning that it was that memory block that was holding its execution. Also, block 2×9 had a fetch transfer request at time 5486ms when the last job on the same worker ended. The time between this last job and the start of Task 1667 is considered idle time on the worker. Besides, the allocation request of block 2×9 on `RAM` was made almost 100ms after the transfer request. Reducing this gap could lead to less idle time in the worker. Finally, there are two allocation requests on `RAM` of block 2×9 because an allocation request can fail possibly due to the lack memory (RAM, GPU). In this case, StarPU proceeds to free memory blocks by moving a less critical block to the disk. StarPU developers are aware of these issues, and the next release might include a fix to reduce waiting time in an OOC experiment.

Figure 5.7: Identifying delays in data transfer to satisfy the memory blocks necessary to run Task 1667



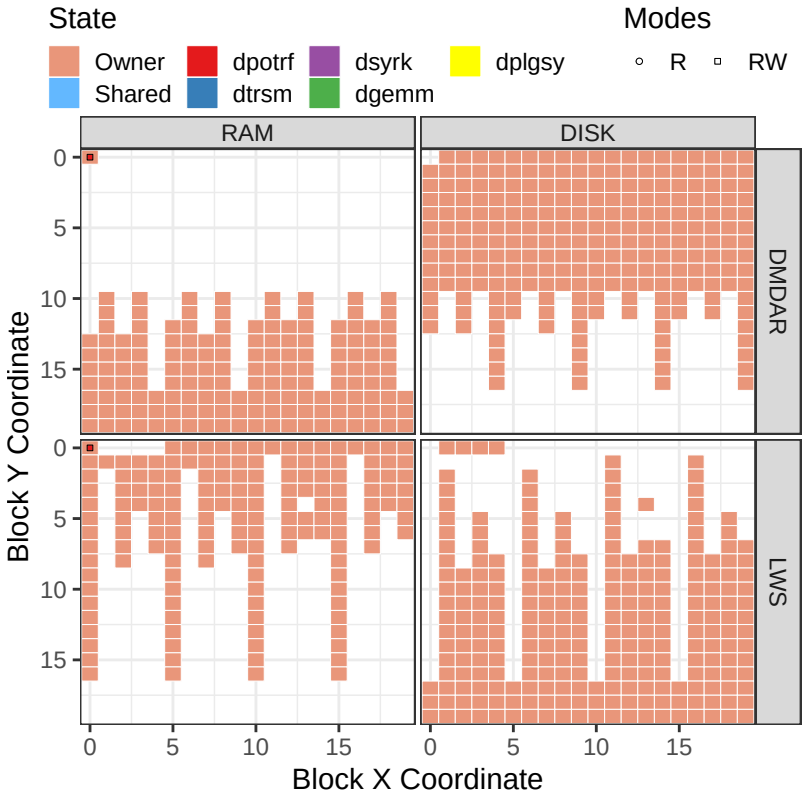
Source: The Author.

5.2.4 Performance impact of matrix generation order using OOC

When comparing the LWS and the DMDAR schedulers, under the same experimental conditions of the previous subsection, the LWS scheduler was consistently faster than DMDAR, as it had a makespan of 10s while the DMDAR had 12s. These results were surprising since DMDAR presented better or same results in other situations. Moreover, DMDAR is supposed to reduce data transfers by prefer running tasks on the location of their input data.

The memory-aware animation snapshots (as detailed in Section 4.5) are employed for each scheduler to understand the good performance of LWS. Figure 5.8 shows the memory snapshots when the first dpotrf task gets executed (on the 0×0 block coordinate – top left) for both LWS (bottom row) and DMDAR (top) schedulers. They are storing the initial input matrix differently. In the LWS, the input matrix is generated from top to bottom, so the required blocks are readily available in RAM. In the DMDAR, the matrix generation is inverted, forcing more data transfers from the disk to RAM at the beginning of the execution, causing performance problems.

Figure 5.8: Snapshots of block presence when application starts

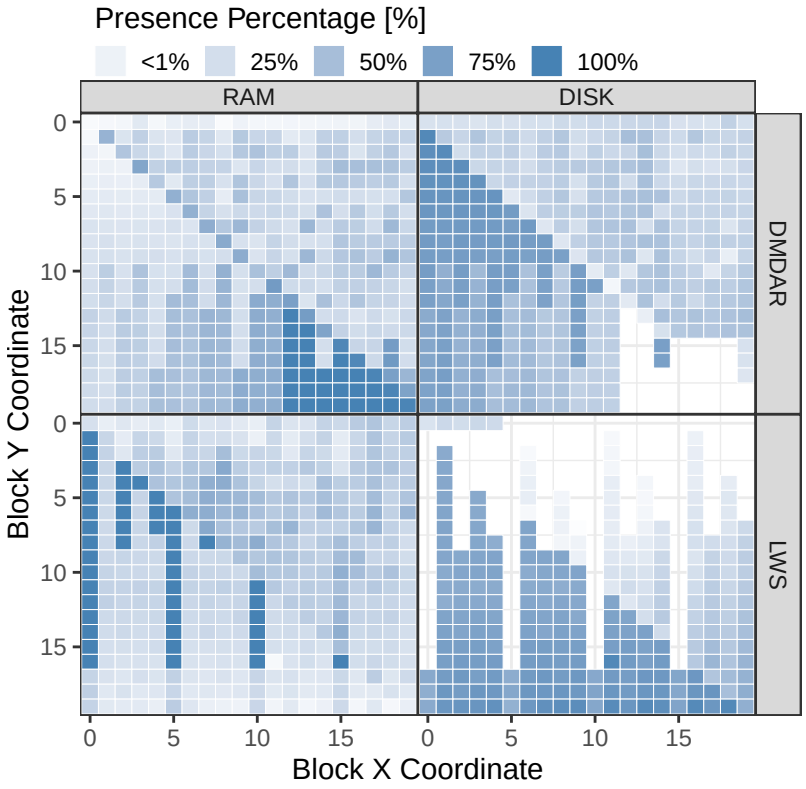


Source: The Author.

Figure 5.9 uses the proposed heatmaps to demonstrate how, in general, the wrong

matrix generation strategy may be harmful to overall performance. The heatmaps show that the upper side of the matrix on the LWS is generally much more present on RAM than the lower side. At the beginning of the execution, the critical low-coordinates blocks are already on RAM leading to fewer transfers from disk during the whole run. Therefore, the unnecessary data transfers of DMDAR justify its poor behavior when compared to the LWS scheduler.

Figure 5.9: Heatmaps showing block presence throughout the run



Source: The Author.

5.2.5 Comparison of DMDAR and DMDAS with CPU-only restricted RAM

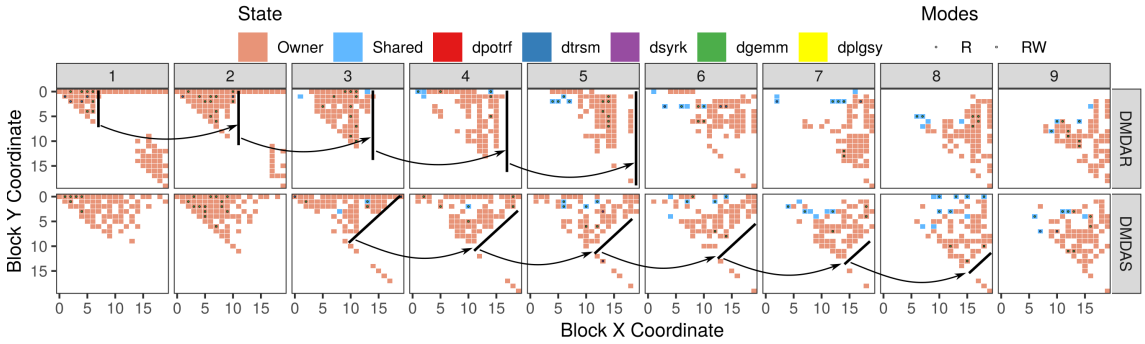
This test case conduct CPU-only experiments, similar to the Sections 5.2.3 and 5.2.4 cases, but with a very restrictive RAM size of 512MB. The goal is to stress the data-awareness of the DMDAS and DMDAR schedulers. An unexpected behavior is that DMDAS is 60% [50s vs 30s] slower than DMDAR. One of the reasons is the different number of transfers done by the schedulers. To take an execution as example, while DMDAR did ≈1981 RAM-to-disk transfers, DMDAS did ≈3097, a increase of ≈50%.

Figure 5.10 shows nine representative block residence snapshots (the horizon-

tal facets) for both schedulers (top and bottom rows) out of two animations with many frames. There are differences in how each block residence evolves for each scheduler. `DMDAR` prioritizes executing tasks whose data are already available in memory (i.e., `ready`). Thus, by linear algebra dependencies, this scheduler tends to work on a linear/columnar way. Therefore, the divisions of the row and column on the visualizations are apparent. For example, the matrix is processed column-wise (the manually-added arrows highlight the behavior) in snapshots one to five. So on each column/row is preferable to be put on `RAM`.

On the other hand, `DMDAS` sorts tasks by priorities, considering the topological distance to the last task. This sorting forces a diagonal advance on the matrix, contradicting row/column locality, a behavior that can be especially seen on snapshots three to eight (arrows highlight the behavior). This diagonal exploration suggests that although task priorities are useful to guide the scheduler towards the critical path, they should not necessarily be enforced too rigorously, to let the scheduler re-use ready data blocks for the better locality. Although no problems are found in this case, the visualization provided the understanding of how these two schedulers affected the behavior of the memory during the execution.

Figure 5.10: Nine block residence snapshots (horizontal facets) for the `DMDAR` (top row) and the `DMDAS` (bottom row) schedulers



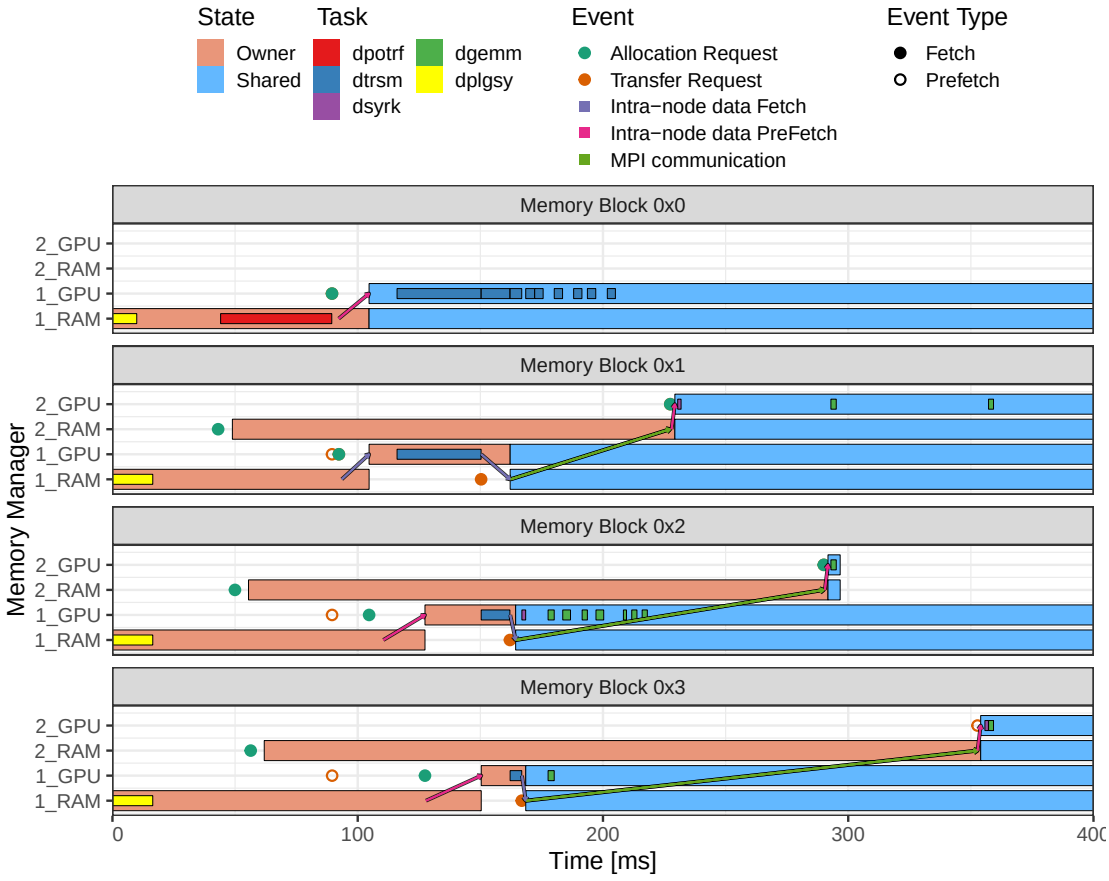
Source: The Author.

5.2.6 Experimenting with MPI executions

The Chameleon solver can use the StarPU-MPI module to explore the execution on multiple nodes. This section applies the detailed behavior of memory blocks strategy in this more complex context to compare general differences in the memory management when another layer (StarPU-MPI) is added. The execution uses a small input matrix with

10×10 blocks and two dracos[2-3], each machine with one GPU. Figure 5.11 presents the behavior of the first four memory blocks. The first difference is that now the memory managers have a prefix number to it (`1_RAM`, `2_RAM`) that indicates the MPI rank. Another difference is the addition of the `MPI communication` event as a green arrow. In these machines, the communication between nodes can only occur from one memory ram to another. This situation is indeed observed on memory blocks 0×1, 0×2 and 0×3 where one data from `1_GPU` is transferred to `2_GPU` but has to be moved to `1_RAM` first as a intra-node communication, then a MPI communication to `2_RAM` and then a intra node communication again. Moreover, the MSI protocol is applied independently per-node. Each memory block will have a different handle on each node, enabling a block coordinate to be owned by more than one node. The StarPU-MPI guarantee the coherence of the blocks by unrolling the DAG on each node and computing when another node needs the data and issuing the transfer.

Figure 5.11: Detailed behaviour of memory blocks 0×0, 0×1, 0×2, and 0×3 of a MPI Cholesky execution



Source: The Author.

5.3 Experimental Results with a CFD Simulation

This section presents the experiments conducted on a CFD simulation code (NESI; SCHNORR; NAVAUX, 2019). The proposed strategies of this thesis were employed on one data partitioning scheme for the CFD to detect problems and improve the performance. Also, to show how the strategies could be used during any developing process. The specific problem found occurs when the memory system of the runtime is stressed, and the memory space required by the simulation is larger than supported on GPUs. The machine used in these experiments is `Tupi` with Ubuntu 18.04. The StarPU and the CFD code have been compiled with GCC 7.3.0 using the default configurations.

Section 5.3.1 presents the CFD application, more specifically the data partition investigated by the strategies. Section 5.3.2 presents the identification of an problem on the first version of the data partitioning using the strategies. Section 5.3.3 provides a comparison of some optimizations proposed to correct these problems.

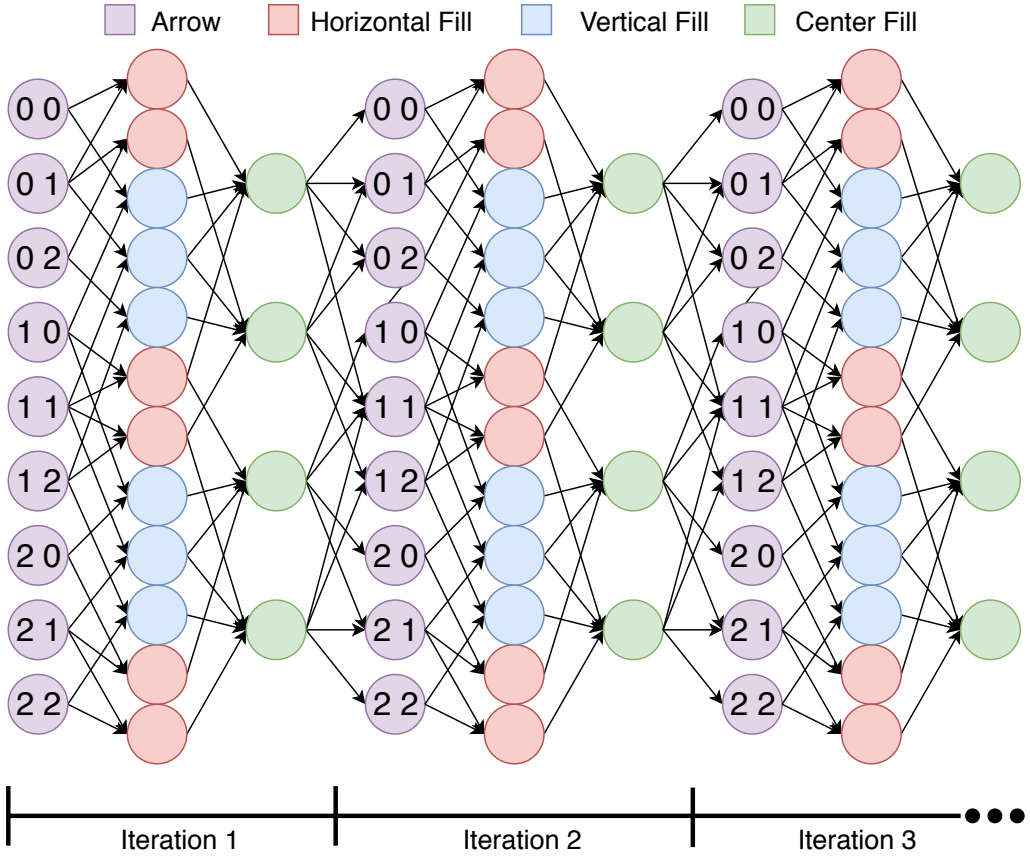
5.3.1 Application description

The application is a task-based and heterogeneous (CPU and GPU) parallel solver of a CFD problem that simulates the 2D flow of an incompressible Newtonian fluid with constant viscosity. The model used to compute the simulation is the Finite Difference Method (FDM), where the simulation space/time is discretized to a regular grid with a horizontal and vertical number of cells given by Nx and Ny. Each cell assumes one value for each variable at one time-step. Iterative methods are executed to compute the fluid's variables over a finite number of time steps where the computation of a new cell time state depends only on its neighbors' variables of the last computed time step. In this 2D simulation, each cell has the same horizontal and vertical size.

A naive implementation would split these 2D cell in blocks and would apply the computation steps in parallel over theses blocks. An alternative strategy of domain decomposition is to increase the computation granularity of the tasks by grouping different model methods in the same application task. This method considers the time as a new dimension on the partitioning. Aggregating tasks (and numerical methods) would prune the DAG structure, reducing any runtime overhead caused by a large number of tasks to be scheduled. The application use a similar version described in (ALHUBAIL; WANG; WILLIAMS, 2016) (identified as swept rule) to decompose the simulation, converting

it to the task-based programming paradigm. The resulting design contains four types of tasks: the Arrow, the horizontal Fill, the vertical Fill, and the center Fill tasks. The resultant task-based DAG for a 3x3 blocks decomposition is presented in Figure 5.12; each color represent a different task, and the coordinates inside the Arrow tasks represent the block position used for that task. On the first iteration, all the Arrow tasks are ready and can be done on any order. After the necessary Arrow tasks, the horizontal (red) and vertical (blue) fills can be executed, and are independent among them, meaning that they can also be done in parallel. In this partition, the simulation time is used as a new computation dimension. Instead of each task only performing its computation on a block at just one time, the tasks advances and compute all cells that it has, in this case the Arrow task. After the Arrow tasks, fill tasks to complete the remaining wholes are applied. Figure 5.13a presents the arrow tasks applied each one a block of a 2×2 block decomposition. Figure 5.13b presents the other fill steps and how the space is completed between four Arrow tasks.

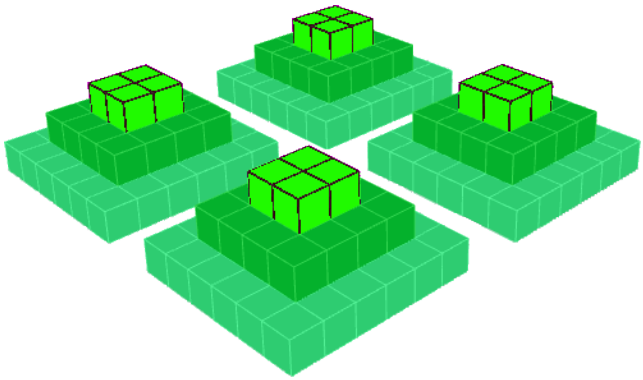
Figure 5.12: The application DAG of the arrow implementation with a 3×3 blocks and the four types of tasks (colors)



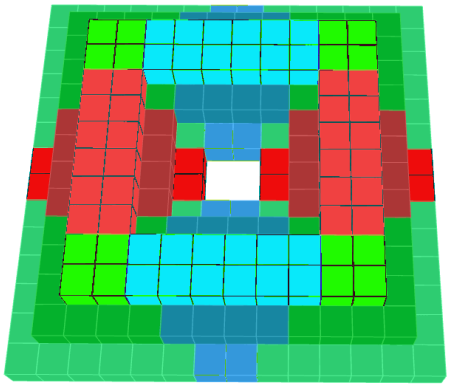
Source: The Author.

Figure 5.13: Data partitioning of the tested CFD application

(a) Four Arrow tasks to compute each one a block of a 2×2 block decomposition. Each block contains 6×6 cells in the base plane and three sub-operations



(b) Two horizontal fill tasks (blue regions) and two vertical fill tasks (red regions) compute the gaps among the four pyramids



Source: The Author.

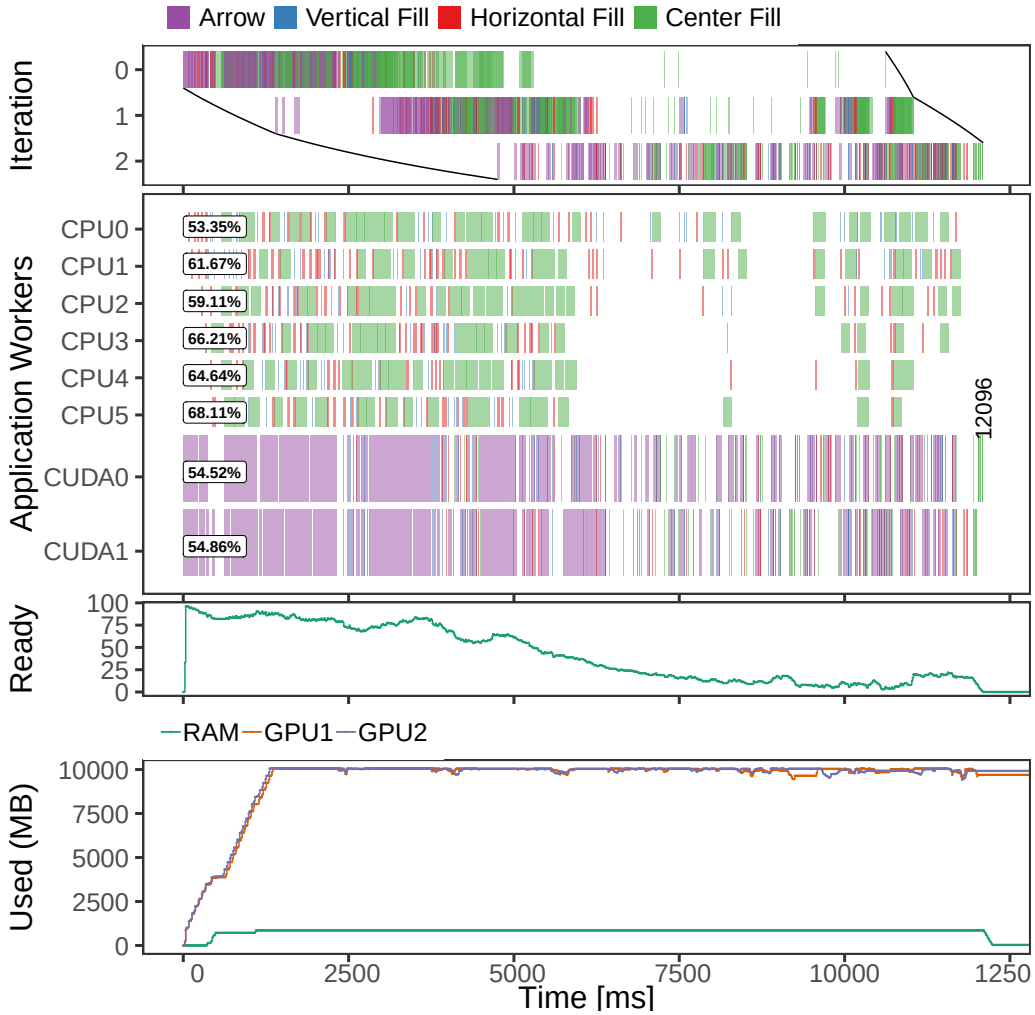
5.3.2 Performance problem identification

Figure 5.14 depicts the original non-optimized version of the Arrow implementation. The non-optimized version contains a high idleness rate for all computing resources (the white areas in the Application Workers panel), ranging from 53-68% in CPUs, and around 54% in GPUs. While innate data dependencies might explain this idleness, other analyses indicate that memory utilization and data transfers between host and GPU devices are the origins of such idle times.

Figure 5.15 indicates the behavior of one base block (2D initial values of the block) data handle (of coordinate 0×0) over time using the proposed strategy of Section 4.4. Each line on the Y-axis represents a different memory node present on this experiment (`GPU2`, `GPU1`, and `RAM`). The specific time frame highlighted by the red rectangle `A` depicts an interesting behavior. After the Arrow task reaches completion to the base block handle, and computed the arrow on another handle, the base handle was kept by StarPU on GPU memory even if the next task to use the handle would access with a write mode, overwriting its content. This StarPU behavior causes extra memory usage on the GPUs. Also, the time between the Arrow task and the Center Fill task is too long, and the data generated by the Arrow task is bigger than the data it reads, using more memory to save temporary data until the execution of the Center Fill task.

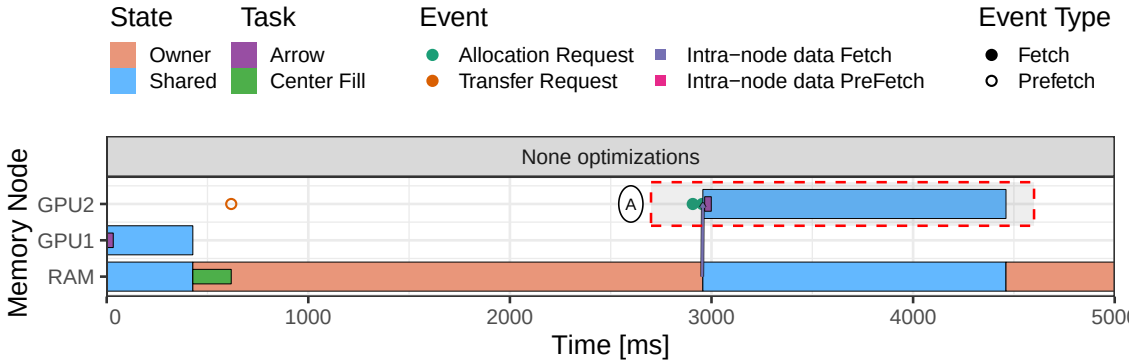
Two approaches are investigated to correct these issues. First, StarPU allows hints about the memory manager: a call to the `starpu_data_invalidate_submit` function informs the runtime that the data handle passed as a parameter is obsolete. This

Figure 5.14: Behavior of the Arrow strategy without optimizations



Source: The Author.

Figure 5.15: Temporal activities regarding the memory block 0×0 in the Arrow implementation without optimizations



Source: The Author.

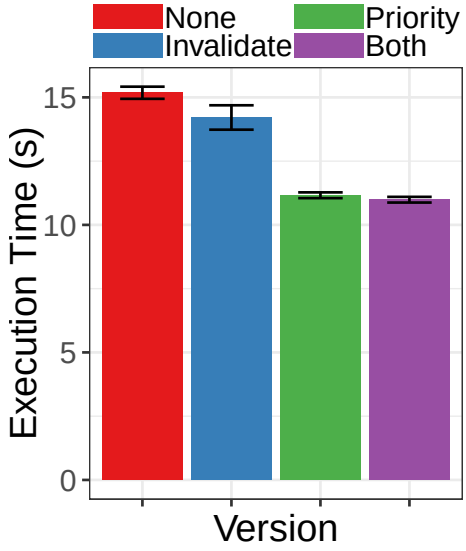
small modification in the code enables memory releases earlier, allowing more prefetches and consequently less idleness. Second, when the application generates many Arrow

tasks, the allocated data is more prominent than the initial one, because of the multiple z levels. Only the completion of Center Fill tasks releases all this data for other uses. For this reason, one solution would be to prioritize the horizontal and vertical fill over the arrow task, and the center tasks over all others.

5.3.3 Performance analysis comparison overview

Figure 5.16 presents the average execution times of four cases: the original non-optimized arrow implementation (red color in the illustration), the manual memory invalidation operations (blue), the adoption of task-priority for Horizontal, Vertical and Center Fill tasks (green), and the combination of the two optimizations (violet). Experimental configuration used is: input size of 30K, 100 blocks (10×10), and the DMDAR scheduler. Also, the experiments consider measurement variability with 30 replications for each configuration on the `Tupi` machine. The results indicate that task priorities improve a lot the performance and also can be combined with explicit memory invalidation. Both optimizations results on gains of 38%.

Figure 5.16: Execution times for the four versions of the Arrow implementation: without optimizations (red), optimization by explicit memory invalidation (blue), optimization with task priorities (green), and both optimizations (violet)

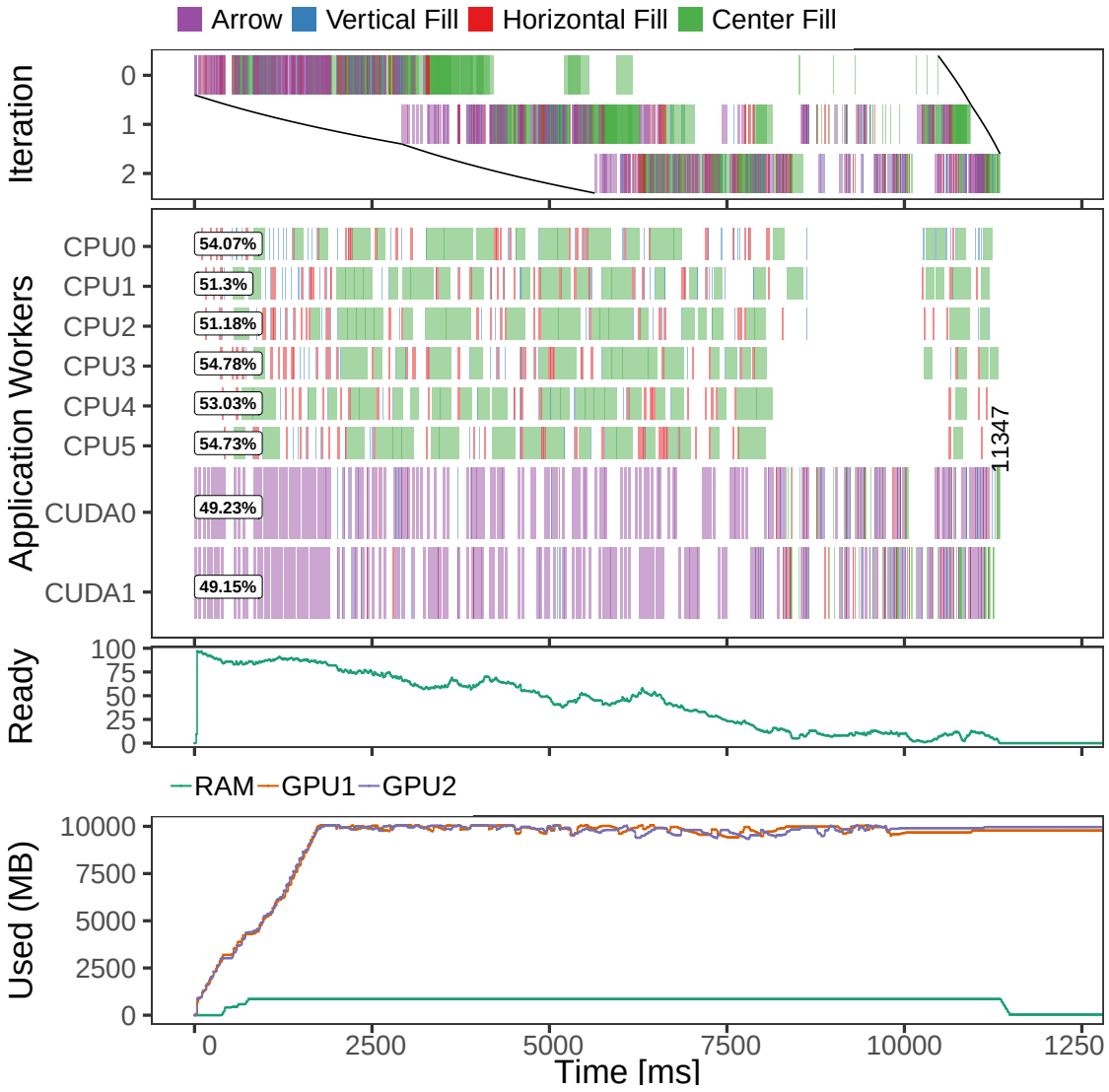


Source: The Author.

StarVZ visualization panels are used to understand the behavior of each version. Four of these plots, from representative executions, presents each one of the four versions: the original non-optimized Arrow version (Figure 5.14), optimization by explicit memory invalidation (Figure 5.17), optimization with task priorities in the Vertical, Horizontal, and

Center Fill tasks (Figure 5.18), and the version with both optimizations (Figure 5.19). A careful analysis of the observed behavior of these different cases indicates that all versions prefer to schedule the Arrow tasks (violet color) on GPUs and almost all the other tasks on CPUs.

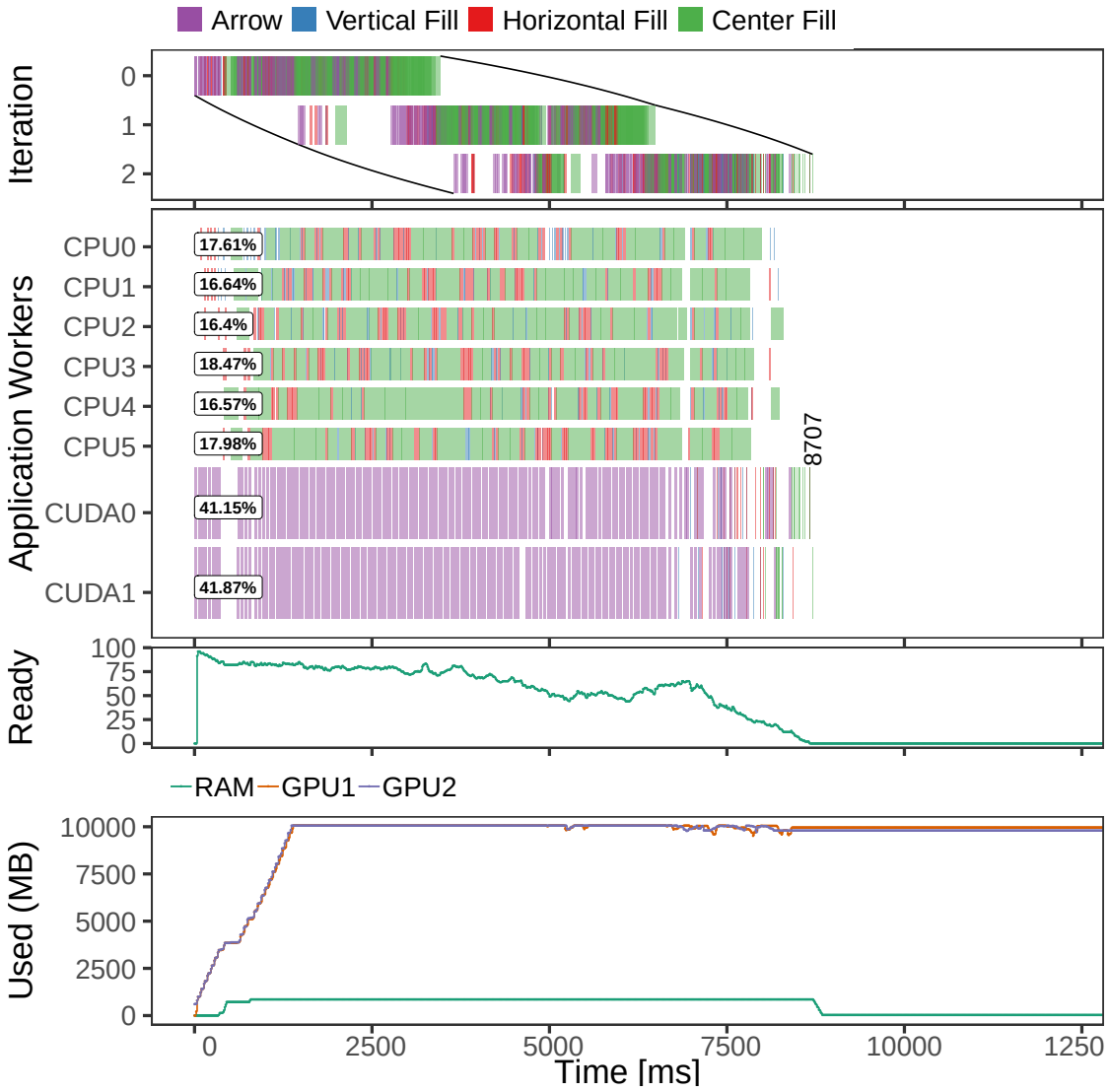
Figure 5.17: Behavior of the arrow strategy with memory invalidation



Source: The Author.

The **non-optimized version** (Figure 5.14) demonstrates too much idleness in all CPU and GPU resources. The lack of memory available on GPUs is a possible cause of such idleness. Moreover, the memory use plot reports maximum memory usage very earlier in the execution. Consequently, prefetches are infrequent, and StarPU only transfers data when a task starts its execution. The optimization with **memory invalidation** (Figure 5.17) demonstrates a small execution time reduction compared against the non-optimized version, but still with high idle times (idleness of 51-54% on CPUs, about 49% on GPUs).

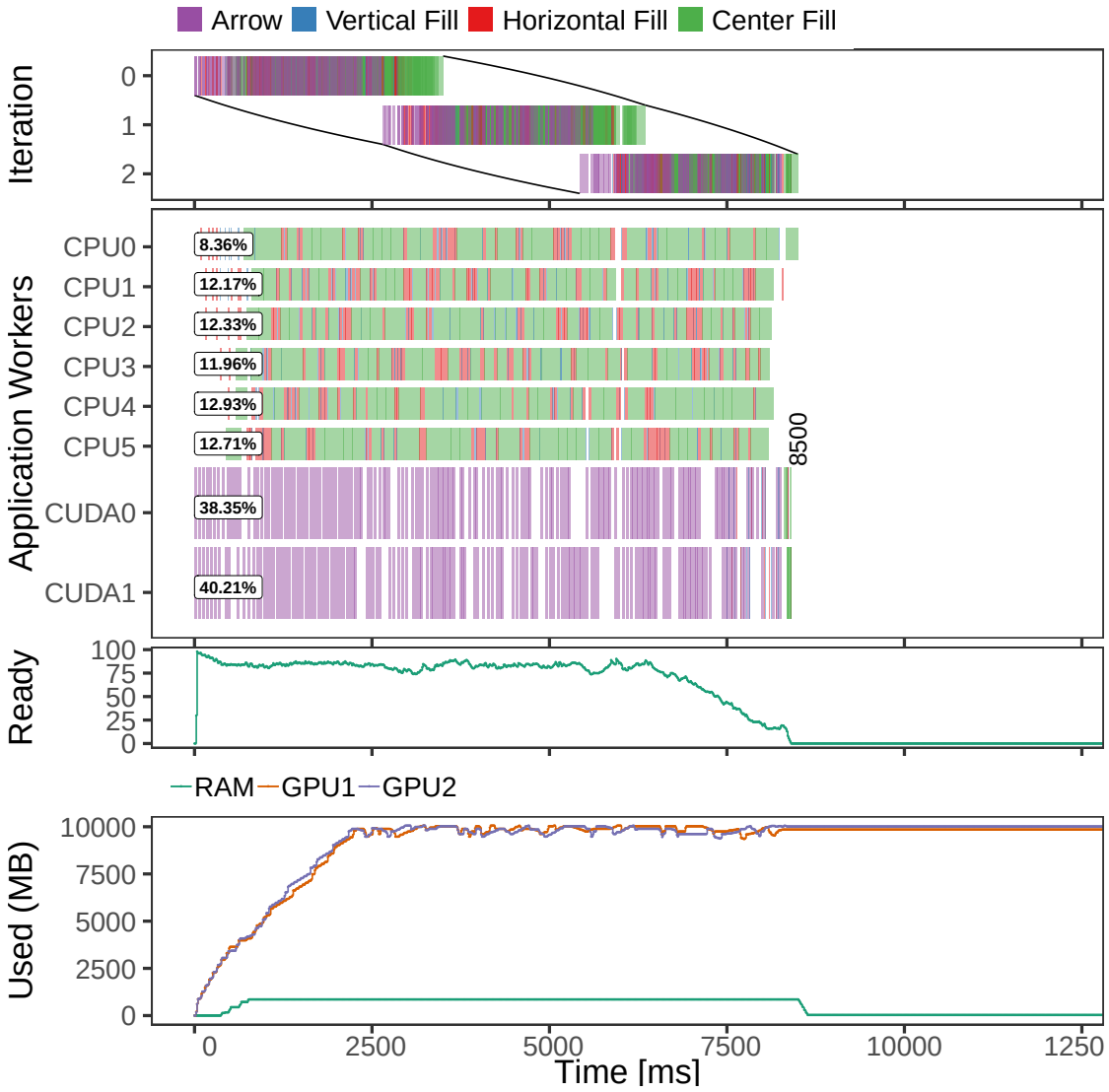
Figure 5.18: Behavior of the arrow strategy with priorities



Source: The Author.

The memory usage is never at its maximum. However, it still suffers from many Arrow tasks generated data, since StarPU adopts the submission order as an implicit priority when there are no task priorities. With **task priorities** (Figure 5.18), there is a significant execution time reduction compared with previous versions. The behavior of ready tasks (see the Ready panel) have a peak following a drop, and the iteration plot shows that some block coordinates reach the last iteration very fast (some states on the third iteration). This behavior is usually a sign of good exploration of parallelism. The **combination of both optimizations** (Figure 5.19) demonstrates a lack of significant difference in the execution time but presents some changes in the detailed behavior. First one, the Iteration panel shows smaller iterations overlap, so no coordinates advance faster than others. Also, the number of ready tasks do not suffer too much fluctuation. Finally, the used memory takes

Figure 5.19: Behavior of the arrow strategy with both optimizations



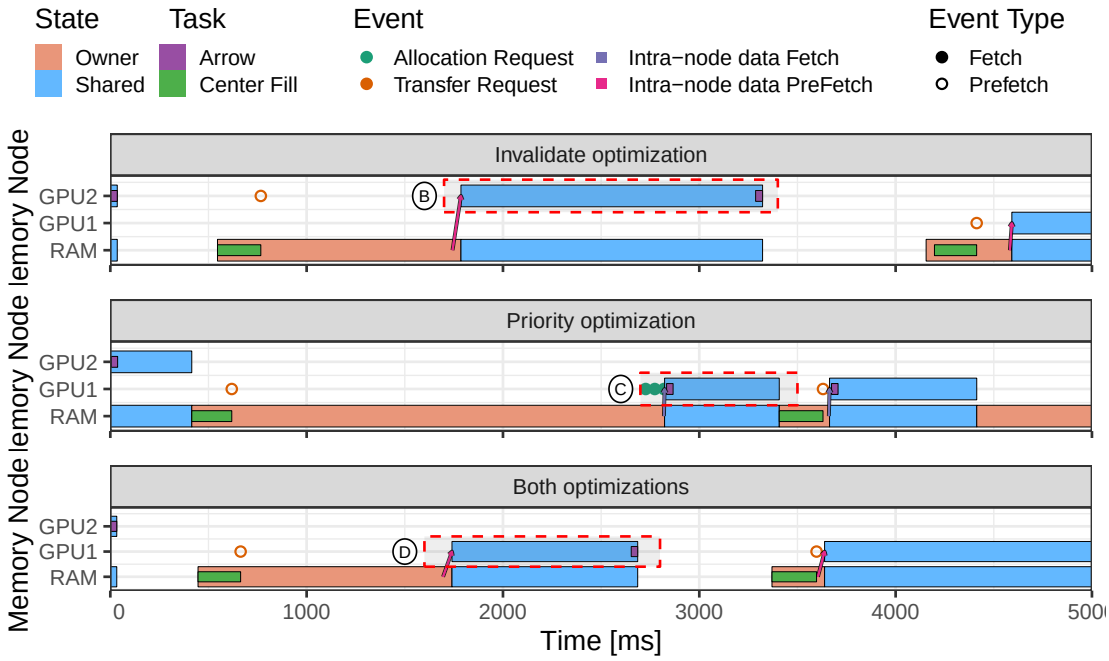
Source: The Author.

more time to reach its peak and rarely is at its limits.

The optimizations interfere directly in the residence of the data handles in the different memory nodes (`GPU2`, `GPU1`, and `RAM`). Figure 5.20 presents the evolution of the base block data handle with coordinate 0x0 in the optimized versions. Figure 5.15 shows the same for the non-optimized version. The main difference is the time that a data handle remains on one of the GPU memory nodes. In the non-optimized version, the handle resides much more time when compared to the optimized versions. Even if the next task that would use the handle access it with write permissions, StarPU continues to keep it on the GPUs memory node after the Arrow task. When the memory invalidation optimization is applied, the data handle is deleted from all memory nodes, causing an free extra space for prefetching operations, which improves even more the performance. The

rectangle B of Figure 5.20 shows that the data is now available before the Arrow task, and it disappears right after the task completion. When the priority version is applied (see rectangle C), the Center Fill task happens sooner, and because the runtime prefers to execute it in CPUs rather than GPUs, the data handle becomes invalidated on GPUs. The rectangle C highlights another scenario with a similar problem of the original version. After the completion of the Arrow task, the data remains in the memory node. Also, the transfers that appear in this case are fetches and not prefetches. Finally, the rectangle D highlights the scenario with both optimizations. This case demonstrates that data is now present before the Arrow task, and released immediately after. Moreover, while the transfers on the non-optimized version are fetches, now they are only prefetches.

Figure 5.20: Memory block 0×0 behaviour with three cases, invalidate optimization, priority optimization, and both optimizations



Source: The Author.

In this stage, the analysis suggests that the tasks executing on CPUs are the dependency of Arrow tasks scheduled on GPU, causing the idleness. The application may have global improvements if the code of the four types of tasks is further optimized. It is important to notice that GPU kernels are simple ones, without the adoption of shared memory optimizations, for instance. Moreover, the Arrow strategy does more integer calculations than the other strategies. These additional operations are necessary to cope with the more complex coordinate system. A comparison is hard to obtain because all these computations are included in the task code, being hard to isolate.

5.4 Experimental Results with Sparse QR Factorization

This section presents the experiments conducted with the sparse QR factorization available in the `QR_Mumps` application (BUTTARI, 2012). The machine used in this experiments is `Tupi` with Ubuntu 18.04. The StarPU and `QR_Mumps` have been compiled with GCC 7.3.0. Moreover, in this executions two workers per GPU were used. The matrix used for experiments is the TF18 sparse matrix available at SuiteSparse Matrix Collection[1]. Section 5.4.1 describes the `QR_Mumps` application, and Section 5.4.2 presents some earlier analysis of the `QR_Mumps` application.

5.4.1 QR Mumps Application

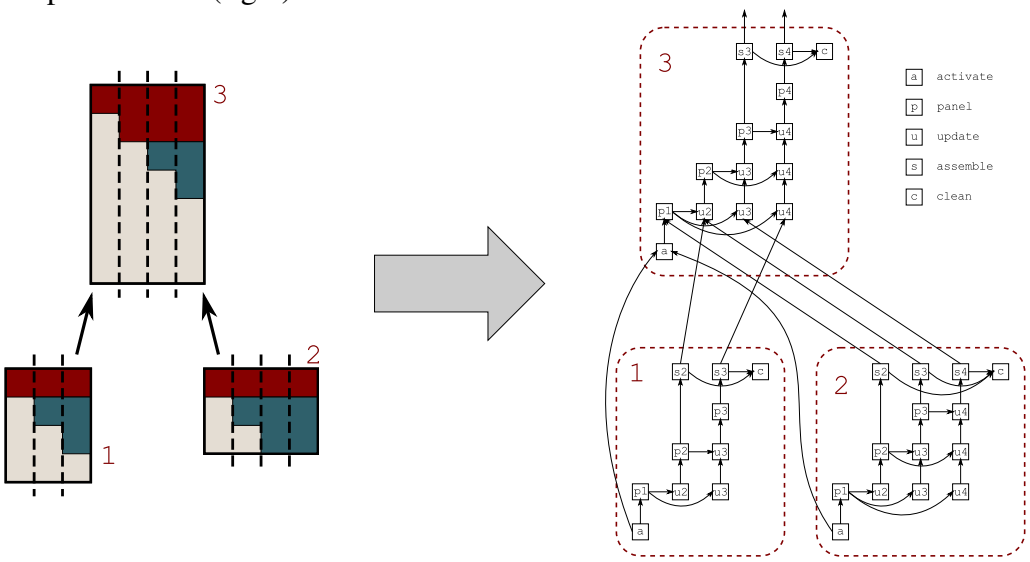
The `QR_Mumps` is a software package design to solve sparse linear systems on multicore computers. Its implementation is based on the QR factorization of an input matrix and can be executed on heterogeneous platforms thanks to its version using `StarPU`. The package can be used to solve least-squares problems and to compute the minimum-norm solution of sparse, underdetermined problems. The `QR_Mumps` uses the task-based paradigm to achieve parallelism, where its tasks can be, for example, BLAS or LAPACK routines. The development of `QR_Mumps` over StarPU was conducted in (AGULLO et al., 2013) and evaluates the original implementation of `QR_Mumps` with the StarPU version. Figure 5.4.1 presents an example of the `QR_Mumps` elimination tree with three nodes (left) and the resultant DAG representation (right).

5.4.2 Checking if memory is a problem

The experiments of this section have the objective to check the overall performance of the `QR_Mumps` application using a large matrix as input. Figure 5.22 presents the applications workers, submitted and ready tasks for the execution with the TF18 matrix. The first noted problem is the idle times in the middle of the execution while other tasks are waiting for a `do_subtree` task. This problem is caused by how `QR_Mumps` process the sparse matrix, trying to reduce computation on fragments of the matrix that contains zeros. The ordering and the elimination tree algorithms used are responsible for

[1] SuiteSparse Matrix Collection Website: <https://sparse.tamu.edu/>

Figure 5.21: The QR_Mumps elimination tree example with three nodes (left) and its DAG representation (right)



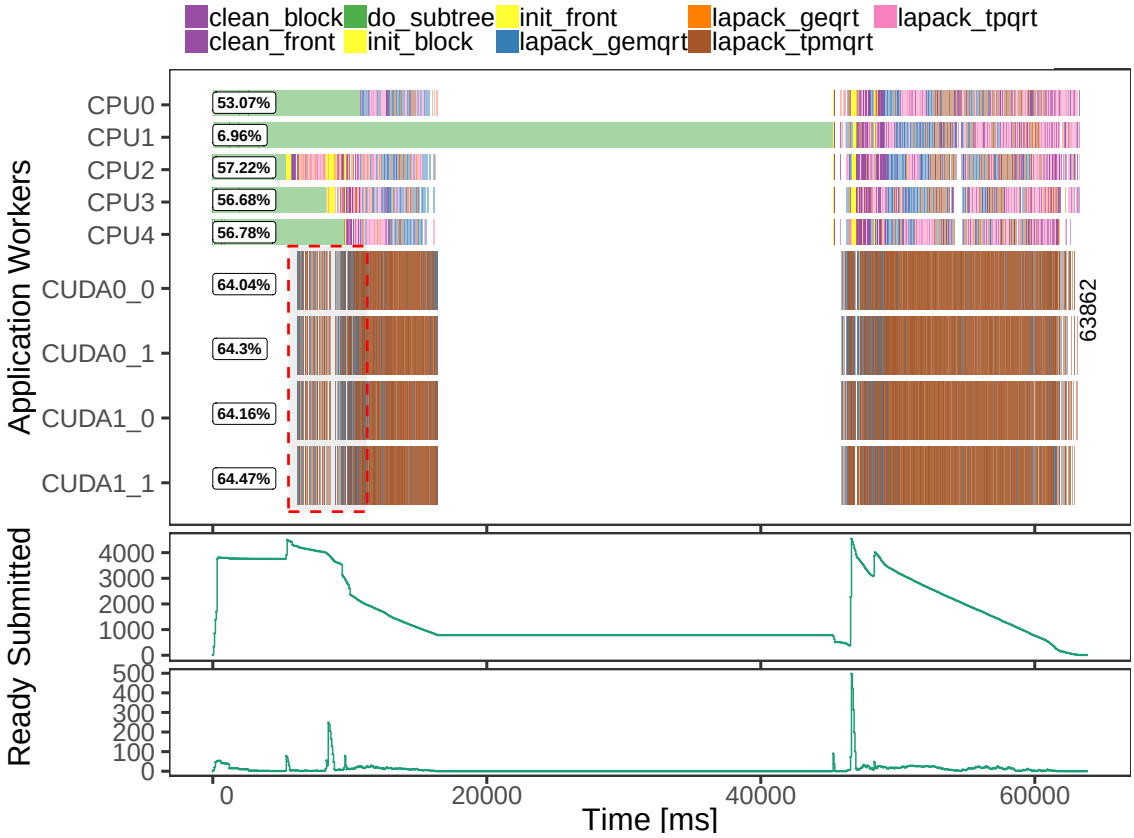
Source: Agullo et al. (2013).

this idle time. Although this idle time is the cause of a huge loss in performance, it is not caused by memory problems. However, other parts of the execution have idle times, and their study could lead to performance improvements. For example, consider the idle times after the first task on the GPUs, highlighted in Figure 5.22 by a dotted red rectangle. As it can be observed, the amount of idle times is very high until the GPU becomes fully utilized.

The idle times of the tasks on the GPUs could be caused by problems related to transfers from the RAM to the device memory. The analysis of a task that has idle time just before it would help in the understanding of this problem. Figure 5.23 shows the selection of task 600, a `lapack_tpmqrt` task executed on `CUDA0_1`, and its last dependencies highlighted in the red path. The last task before it, task 598, was a `lapack_tpqrt` task executed on CPU3. The time between both tasks indicates that just after the end of task 598, StarPU transfers the necessary data of task 600 from RAM to GPU, and execute the task. However, the last dependency of task 598, another `lapack_tpqrt` task, executed long before it. The memory strategies analysis are used to check the memory flow and understand if task 598 could be executed earlier.

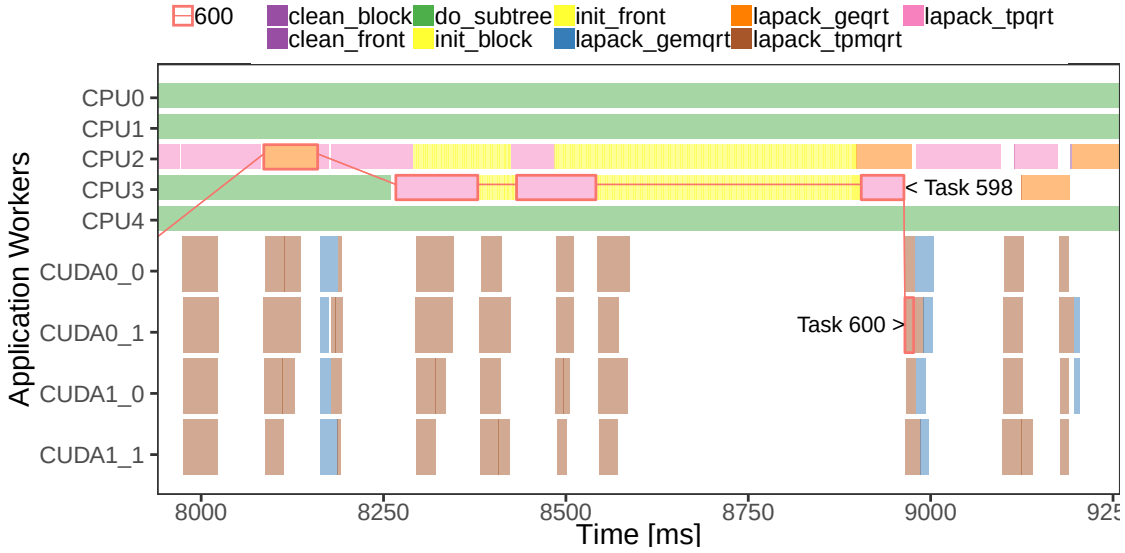
The detailed temporal behavior of memory block can be used to investigate the behavior of the data used by both tasks 598 and 600. Figure 5.24 presents the behavior of the memory blocks used by task 598. Different from the strategies proposed, there is a green vertical line manually added to show when the StarPU worker pop task 598. According

Figure 5.22: Multiple Performance Analysis plots for the QR_Mumps execution



Source: The Author.

Figure 5.23: Fragment of the application workers during some GPU idle times

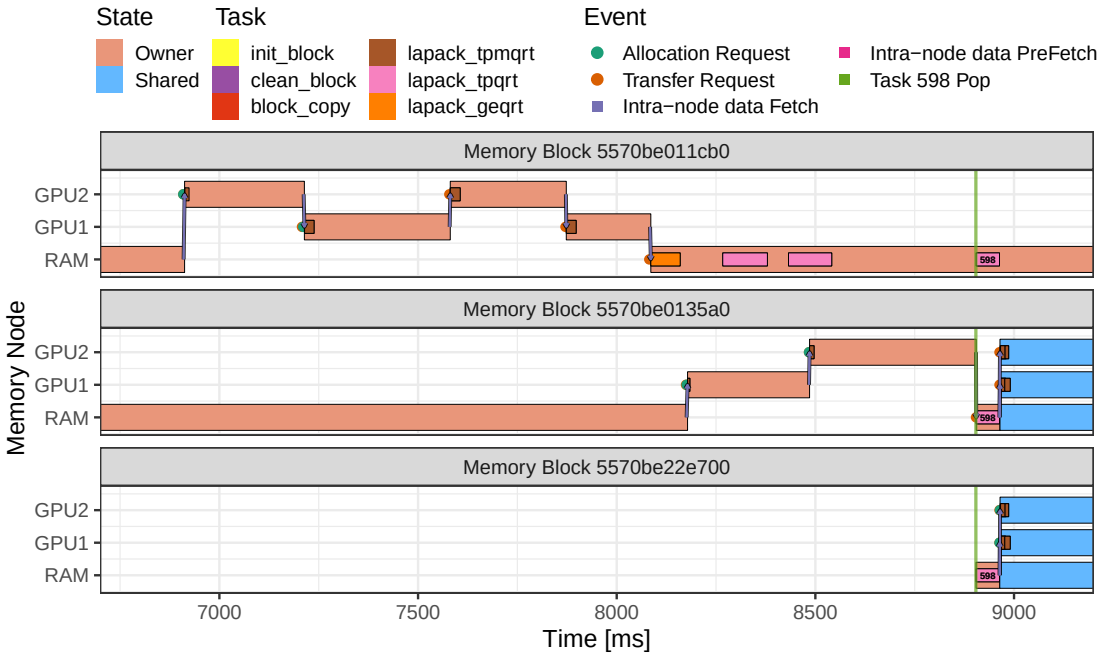


Source: The Author.

to the visualization, from the three memory blocks used, the first one `5570be011cb0` and the last one `5570be22e700` did not need to be transferred to the scheduled worker, they were already on RAM. However, the second block, `5570be0135a0`, was on GPU 2 and had to be transferred to the RAM. When the worker needs the task, it pops the task

from the scheduler queue, and the transfers occur immediately, without causing much idle time. From this perspective, the idle times are mostly caused by problems on how the task 598 was scheduled, and not by any problem on how the StarPU manages the memory.

Figure 5.24: Detailed behavior of the memory blocks used by task 598



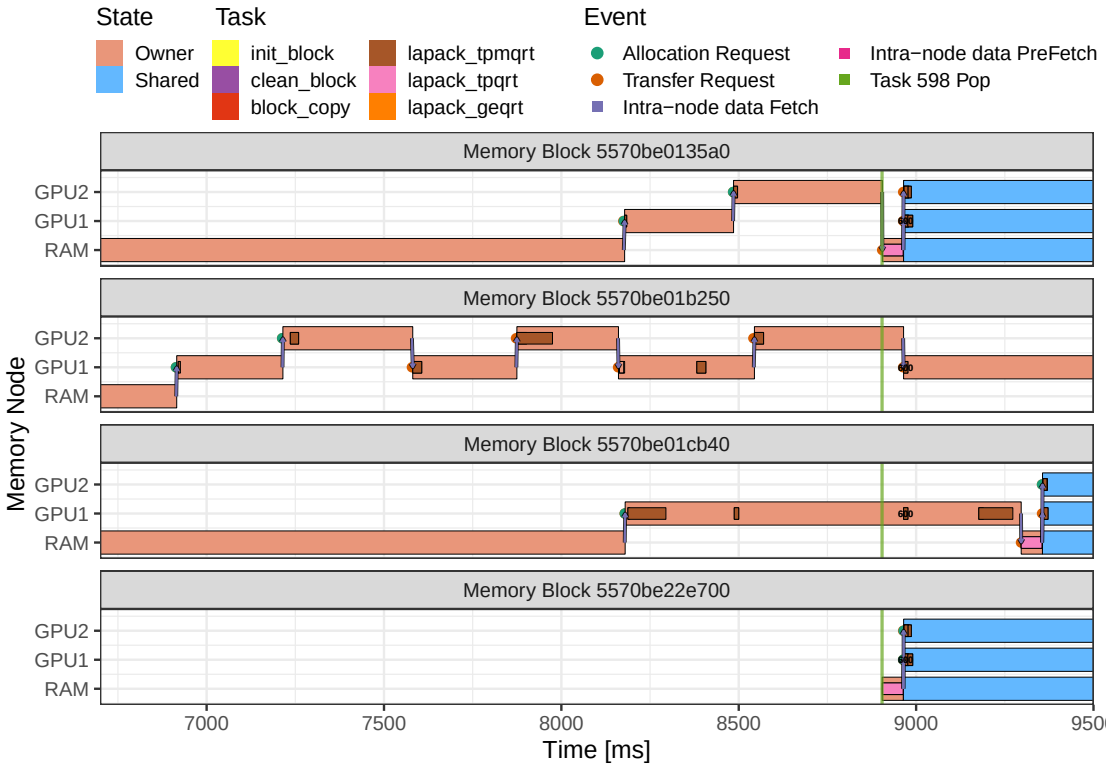
Source: The Author.

Although the memory management was unrelated to the idle times, another situation appears in the analysis. Similar to the last visualization, Figure 5.25 shows the blocks used by task 600. The interesting aspect here is the behavior of the second block, `5570be01b250`. The StarPU is transferring the block from one GPU to the other, causing a ping-pong situation. In this case, it is not causing any performance problem, as StarPU is executing the transfer immediately, and the transfer time of the block is low. However, in some other critical situations, this ping-pong behavior could be a disadvantage for the performance, as the transfers are being repeated periodically. Moving the tasks to the same GPUs would cause fewer transfers. However, maybe this is an unavoidable situation because if StarPU schedules these tasks on the same GPUs, others would need to be moved to a different device, causing the same behavior.

5.5 Discussion and Known Limitations

This section details known limitations of the proposed strategies and the current implementation. Four main points are described as follows and are based on the applica-

Figure 5.25: Detailed behavior of the memory blocks used by task 600



Source: The Author.

tions and the runtime used.

- The methodology is general but specifically tailored for the StarPU runtime and its applications. The views depend on traces enriched with events about memory operations that are only available, as far as we know, in the StarPU runtime. Other runtimes (such as for OmpSs (DURAN et al., 2011) and OpenMP) must be instrumented to make them work with StarVZ. The methodology is easier to extend for runtimes that divide the memory into blocks and use the MSI memory coherence protocol.
- Although this work apply the strategies for three specific applications, the views would work for any application that uses the StarPU runtime. For example, results of Section 5.2.2 benefit all StarPU applications with a similar memory block-based layout and problems larger than the GPU memory. The out-of-core case studies bring positive results that are exclusive to the Cholesky application. However, this understanding can lead to potential improvements to other applications that use the out-of-core capabilities of StarPU. Moreover, the visualizations are specific for data in this 2D format. Specific strategies, like the snapshot and heatmaps, would require adaptations for the application data structure, for example, if they are cubical, 4D

visualizations and so on.

- This work presents most of the experiments on single node executions because even if the number of nodes is increased, the identified problems would still be there, just with more information to be processed. Problems surface in multi-GPU memory interactions, reinforcing the obstacles in memory management for heterogeneous platforms. The case using the StarPU-MPI module is a proof-of-concept that the strategies would work on multi node platforms as well.
- The views require knowledge about both the runtime and the application to make conclusions. However, the methodology is intuitive and easy to use for researchers with some HPC experience. The strategies in the hands of advanced analysts or HPC researchers could provide a broad understanding of the final application performance and how to improve it, as show in many examples.

6 CONCLUSION

The task-based programming paradigm is a great candidate for programming heterogeneous parallel applications because a runtime assists in some responsibilities. However, as other HPC approaches, the performance analysis of these applications is challenging. Many aspects can impact on the performance. One studied in this work is the memory management of the application at the runtime level. The runtime is responsible for taking many decisions regarding the memory system, including guaranteeing the coherence of the memory, transferring it to the needed devices and removing unused memory. However, the runtime is not alone responsible for the management. For the runtime conduct better decisions, the application programmer has to structure the DAG of tasks, and the application correctly. Tools that help on the performance analysis of the memory of task-based applications are then desirable.

This work presents a visual performance analysis methodology based on known visualization techniques (space/time, heatmaps, etc.) that are modified in a novel way to support the identification of data transfer and management inefficiencies both in the StarPU task-based runtime and application code. The proposed strategies, to investigate the performance of the memory of task-based application at runtime level, are applied on three different applications, the Chameleon suite to solve dense algebra problems, a CFD application, and the `QR_Mumps` solver for sparse QR factorization.

The strategies are applied on the computation of a tile-based dense Cholesky factorization of the Chameleon application using heterogeneous platforms with GPUs, out-of-core algorithms that move extra memory to DISK, and distributed StarPU-MPI executions on multiple nodes. Five scenarios are investigated and are described as follows. (i) The wrong perception by StarPU of GPU memory utilization, issuing too many memory allocations requests that ultimately hurt performance. The problem was corrected and lead to 66% of performance gains. (ii) Identifying significant idle times in applications with the out-of-core feature and providing rich details on the memory operations that caused it. (iii) The identification of unexpected performance issues caused by the input generation on StarPU schedulers under limited memory and out-of-core feature use. (iv) The explanation of the performance differences between two StarPU schedulers on a CPU-only execution with severe memory constraints and out-of-core. (v) A proof-of-concept on the use of the strategies employing a multi-node execution with StarPU-MPI.

The strategies are also applied on a CFD simulation code, during the analysis of

one of the data partition schemes. The experiments present application execution problems with high memory use on GPUs, resulting in idle times. The proposed strategies found a prejudicial behavior of having unnecessary memory blocks on limited memory GPUs. The discovery of this problem leads to the creation of two optimizations informing task priorities and when data is not anymore needed. Essentially, the optimizations expose how the runtime can take better decisions on memory management when the application developer gives extra hints in the DAG. Additional information, in this case, are task priorities and informing when some data blocks are not needed anymore. These optimizations lead to an increase of 38% on application performance. Moreover, the strategies in the case of the `QR_Mumps` application perceived a ping-pong behavior of the memory caused by the task scheduling decisions and not related to memory problems.

The case studies indicate that the proposed strategies are fundamental to identify the reasons behind performance problems related to memory operations. These panels lead to understanding how to fix memory operations of the StarPU runtime and how to individual optimize the applications.

Future work includes the analysis of data transfers on other distributed StarPU-MPI executions. Moreover, another possibility is the study of the strategies on other applications, especially ones that did not have the data organized in a 2D format. Also, export the methodology and the tool to other task-based runtimes, including the OpenMP tasks.

6.1 Publications

The main publication of this work, containing the explanation of strategies and the first four Chameleon Cholesky cases is the following.

- NESI, L. et al. Visual performance analysis of memory behavior in a task-based runtime on hybrid platforms. In: IEEE. **2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)**. Larnaka, Cyprus, 2019. **(Qualis A1)**

Other publications made during the masters are:

- NESI, L. L.; SCHNORR, L. M.; NAVAUX, P. O. A. Design, Implementation and Performance Analysis of a CFD Task-Based Application for Heterogeneous CPU /

GPU Resources. In: SENGER, H. et al. (Ed.). **High Performance Computing for Computational Science – VECPAR 2018**. Cham: Springer International Publishing, 2019. p. 31–44. ISBN 978-3-030-15996-2. **(Qualis B2)(Best Paper)**

- NESI, L. L.; SERPA, M. S.; SCHNORR, L. M. Impacto dos parâmetros do starpu no desempenho do qr_mumps. **ERAD/RS, Escola Regional de Alto Desempenho/Região Sul**, 2019.
- NESI, L. et al. Task-Based Parallel Strategies for CFD Application in Heterogeneous CPU/GPU Resources. **Concurrency and Computation: Practice and Experience**, Wiley, 2019. **(Submitted)**

REFERENCES

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

AGULLO, E. et al. Bridging the gap between performance and bounds of cholesky factorization on heterogeneous platforms. In: **IEEE Intl. Parallel and Distributed Processing Symp. Workshop**. [S.l.: s.n.], 2015.

AGULLO, E. et al. Multifrontal qr factorization for multicore architectures over runtime systems. In: WOLF, F.; MOHR, B.; MEY, D. an (Ed.). **Euro-Par 2013 Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 521–532. ISBN 978-3-642-40047-6.

AGULLO, E. et al. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. **ACM Tr. Math. Softw.**, ACM, New York, NY, USA, v. 43, n. 2, 2016. ISSN 0098-3500.

ALHUBAIL, M. M.; WANG, Q.; WILLIAMS, J. The swept rule for breaking the latency barrier in time advancing two-dimensional pdes. **CoRR**, abs/1602.07558, 2016. Disponível em: <http://arxiv.org/abs/1602.07558>.

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

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

AUGONNET, C. et al. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. **Conc. Comp.: Pract. Exp., SI:EuroPar 2009**, John Wiley and Sons, Ltd., v. 23, 2011.

BLUMOFE, R. D. et al. Cilk: An efficient multithreaded runtime system. **Journal of parallel and distributed computing**, Elsevier, v. 37, 1996.

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

Brendel, R. et al. Edge bundling for visualizing communication behavior. In: **2016 Third Workshop on Visual Performance Analysis (VPA)**. [S.l.: s.n.], 2016. p. 1–8.

BRIAT, J. et al. Athapascan runtime: Efficiency for irregular problems. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 1997. p. 591–600.

BUTTARI, A. Fine granularity sparse qr factorization for multicore based systems. In: JONASSON, K. (Ed.). **Applied Parallel and Scientific Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 226–236. ISBN 978-3-642-28145-7.

CEBALLOS, G. et al. Analyzing performance variation of task schedulers with taskinsight. **Parallel Computing**, v. 75, p. 11 – 27, 2018. ISSN 0167-8191.

COULOMB, K. et al. **Visual trace explorer (ViTE)**. [S.l.], 2009.

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

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

Dongarra, J. et al. With extreme computing, the rules have changed. **Computing in Science Engineering**, v. 19, n. 3, p. 52–62, May 2017. ISSN 1521-9615.

DURAN, A. et al. Ompss: a proposal for programming heterogeneous multi-core architectures. **Paral. Proces. Letters**, World Scientific, v. 21, n. 02, 2011.

GAUTIER, T. et al. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: **IEEE Intl. Symposium on Parallel and Distributed Processing**. [S.l.: s.n.], 2013. ISSN 1530-2075.

GELABERT, H.; SÁNCHEZ, G. Extrae user guide manual for version 2.2. 0. **Barcelona Supercomputing Center (B. Sc.)**, 2011.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI: portable parallel programming with the message-passing interface**. [S.l.]: MIT press, 1999. v. 1.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Fifth Edition: A Quantitative Approach**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X, 9780123838728.

HUYNH, A. et al. DAGViz: A DAG Visualization Tool for Analyzing Task-parallel Program Traces. In: **Proceedings of the 2nd Workshop on Visual Performance Analysis**. New York, NY, USA: ACM, 2015. (VPA '15), p. 3:1–3:8. ISBN 978-1-4503-4013-7.

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

KELLER, R. et al. Temanejo: Debugging of thread-based task-parallel programs in StarSs. In: **Proc. of the 5th Intl. Workshop on Paral. Tools for High Performance Computing, September 2011, ZIH, Dresden**. [S.l.]: Springer, 2012. p. 131–137.

KERGOMMEAUX, J. C. de; STEIN, B.; BERNARD, P. PajÉ, an interactive visualization tool for tuning multi-threaded parallel applications. **Parallel Comput.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 26, n. 10, p. 1253–1274, set. 2000. ISSN 0167-8191. Disponível em: <http://dx.doi.org/10.1016/S0167-8191(00)00010-7>.

KNÜPFER, A. et al. The Vampir performance analysis tool-set. In: **Proc. of the 2nd Intl. Workshop on Parallel Tools for High Performance Computing**. [S.l.]: Springer, 2008. p. 139–155.

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

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Commun. ACM**, ACM, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359545.359563>.

LIMA, J. V. F. **A Runtime System for Data-Flow Task Programming on Multicore Architectures with Accelerators**. Tese (Doutorado) — Université de Grenoble, 2014.

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

MUDDUKRISHNA, A. et al. Grain graphs: Openmp performance analysis made easy. In: **24th ACM SIGPLAN Symp. on Principles and Practice of Paral. Prog.** [S.l.: s.n.], 2016.

NESI, L. et al. Task-Based Parallel Strategies for CFD Application in Heterogeneous CPU/GPU Resources. **Concurrency and Computation: Practice and Experience**, Wiley, 2019.

NESI, L. et al. Visual performance analysis of memory behavior in a task-based runtime on hybrid platforms. In: IEEE. **2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)**. Larnaka, Cyprus, 2019.

NESI, L. L.; SCHNORR, L. M.; NAVAUX, P. O. A. Design, Implementation and Performance Analysis of a CFD Task-Based Application for Heterogeneous CPU / GPU Resources. In: SENGER, H. et al. (Ed.). **High Performance Computing for Computational Science – VECPAR 2018**. Cham: Springer International Publishing, 2019. p. 31–44. ISBN 978-3-030-15996-2.

NESI, L. L.; SERPA, M. S.; SCHNORR, L. M. Impacto dos parâmetros do starpu no desempenho do qr_mumps. **ERAD/RS, Escola Regional de Alto Desempenho/Região Sul**, 2019.

NVIDIA. **CUDA Toolkit Documentation v10.1.168**. Santa Clara, CA, USA: NVIDIA Corporation, 2019. Disponível em: <https://docs.nvidia.com/cuda/>. Acesso em: 20 de Maio de 2019.

OpenMP. **OpenMP Application Program Interface Version 4**. OpenMP Architecture Review Board, 2013. Disponível em: <https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>.

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

PERICÀS, M. et al. Analysis of data reuse in task-parallel runtimes. In: **High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation**. Cham: Springer International Publishing, 2014. p. 73–87. ISBN 978-3-319-10214-6.

PILLET, V. et al. Paraver: A Tool to Visualize and Analyze Parallel Code. In: NIXON, P. (Ed.). **Proceedings of WoTUG-18: Transputer and occam Developments**. [S.l.: s.n.], 1995. p. 17–31. ISBN 90 5199 222 X.

PINTO, V. G. et al. A visual performance analysis framework for task based parallel applications running on hybrid clusters. **Concurrency and Computation: Practice and Experience**, 2018.

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

SCHNORR, L. Poti repository. **GitHub Repository. https://github.com/schnorr/poti/**, 2017.

SCHNORR, L.; STEIN, B. de O.; KERGOMMEAUX, J. C. de. Paje trace file format, version 1.2.5. **Laboratoire d'Informatique de Grenoble, France, Technical Report**, 2013.

SERGENT, M. et al. Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System. In: **21st Intl. Workshop on High-Level Paral. Prog. Models and Supportive Environments**. Chicago, USA: [s.n.], 2016.

STANISIC, L. et al. Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers. In: **The 21st IEEE International Conference on Parallel and Distributed Systems**. Melbourne, Australia: [s.n.], 2015.

STANISIC, L. et al. Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. **Concurrency and Computation: Practice and Experience**, Wiley, p. 16, maio 2015.

StarPU. **StarPU Handbook version 1.3.1**. StarPU Developers Team, 2019. Disponível em: <http://starpu.gforge.inria.fr/doc/html/index.html>.

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

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

TOLEDO, S. A survey of out-of-core algorithms in numerical linear algebra. **Ext. Mem. Alg. and Vis.**, v. 50, p. 161–179, 1999.

TOP500. **TOP500 statistics list**. 2018. <https://www.top500.org/statistics/list/>. Accessed: 2019-05-24.

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

APÊNDICE A — RESUMO EXPANDIDO EM PORTUGUÊS

This appendix presents a extended summary of the work in the portuguese language. The structure of this appendix is the same of the english document.

Neste apêndice é apresentado um resumo expandido do trabalho na língua portuguesa. A estrutura deste apêndice é a mesma do documento na língua inglesa.

A.1 Introdução

Um desafio encontrado na área de Computação de Alto Desempenho (HPC) é a complexidade na programação de aplicações paralelas. O paradigma de programação baseado em tarefas apresenta inúmeros benefícios, e muitos pesquisadores acreditam que esta é atualmente a melhor abordagem para programar máquinas modernas (Dongarra et al., 2017). A abordagem baseada em tarefas transfere para um *runtime* algumas atividades que normalmente são executadas por programadores incluindo o gerenciamento de memória. Aplicações baseadas em tarefas usam um grafo acíclico dirigido de tarefas como a estrutura principal para o escalonamento em recursos, considerando dependências de tarefas e transferências de dados. Entre muitas alternativas, como Cilk (BLUMOFE et al., 1996), Xkaapi (GAUTIER et al., 2013) e OmpSs (DURAN et al., 2011); StarPU (AUGONNET et al., 2011) é um exemplo de *runtime* e é utilizado neste trabalho.

A análise de desempenho de aplicações paralelas baseadas em tarefas é complexa devido à sua natureza estocástica em relação à duração das tarefas e seu escalonamento dinâmico. O StarPU também pode coletar rastros de execução que descrevem o comportamento da aplicação para fornecer dados para análise de desempenho. A abordagem baseada em visualização pode facilitar a análise de desempenho com elementos gráficos.

Fatores como a redução de transferências de dados entre recursos heterogêneos, melhor uso de cache e estratégias mais inteligentes de alocação de blocos desempenham um papel essencial para o desempenho. Escolher corretamente quais blocos de dados ficar em determinada memória de um recurso é um desafio. A complexidade de avaliar a memória no nível de *runtime* motiva a construção de técnicas de análise de desempenho baseadas em visualização adaptadas explicitamente para transferências de dados e otimizações gerais de memória em aplicações baseadas em tarefas.

A criação de métodos específicos, considerando as particularidades do paradigma, pode beneficiar os programadores de aplicações e *runtimes* do paradigma baseado em ta-

refas. Recentemente, algumas ferramentas foram propostas neste tópico (HUYNH et al., 2015; MUDDUKRISHNA et al., 2016; PINTO et al., 2018) , entretanto, essas ferramentas não apresentam técnicas relacionadas ao comportamento da memória (transferências, presença, alocação) do *runtime* e aplicação. Funcionalidades como, verificar o acesso à memória, fluxo e gerenciamento no nível de *runtime* facilitariam a programação; ao mesmo tempo, permitir a descoberta de problemas que afetam o desempenho.

A.1.1 Contribuição

Este trabalho se concentra na análise do desempenho do gerenciamento de memória do StarPU usando a visualização do rastreamento das execuções. Esta estratégia permite uma correlação geral entre todos os fatores que podem afetar o desempenho geral: o algoritmo da aplicação, as decisões do *runtime* e a utilização da memória. As principais contribuições são as seguintes.

- A extensão da ferramenta StarVZ adicionando novos elementos visuais especializados na análise de memória. Ajudando a detectar problemas de desempenho no StarPU e no código das aplicações.
- A adição no StarPU de informações extras de rastreio sobre as operações do gerenciamento de memória, como novas solicitações de memória, atributos adicionais em ações e blocos de memória e os estados da coerência dos dados.
- Experimentos em três aplicações diferentes com o uso das estratégias propostas. A primeira aplicação é o solucionador de álgebra linear densa Chameleon (AGULLO et al., 2010) onde as estratégias são usadas em quatro cenários. No primeiro caso, a metodologia identificou um problema dentro do software StarPU. Uma correção é proposta, e uma comparação do desempenho antes e depois do patch de correção é conduzida, resultando em $\approx$66% de melhora no desempenho. A segunda aplicação é uma simulação CFD (NESI; SCHNORR; NAVAUX, 2019), onde as estratégias são aplicadas especificamente em um método para decompor o subdomínio. A metodologia encontrou algumas otimizações possíveis graças ao comportamento observado, levando a uma melhora no desempenho de 38%. A terceira aplicação é o `QR_Mumps`, um solucionador de fatoração QR esparsa (BUTTARI, 2012). As estratégias encontraram um comportamento relacionado às transferências de memória que pode ser desvantajoso.

A combinação destas contribuições fornecem estratégias de análise de desempenho em vários níveis das operações do gerenciamento de memória em uma plataforma multi-GPU e multi-core heterogênea. As estratégias oferecem uma visualização de alto nível com as informações do DAG da aplicação com as decisões de memória de tempo do *runtime*, que podem orientar os programadores de aplicações e *runtimes* para identificar problemas de desempenho. Em vez de usar apenas métricas de baixo nível e comparálas com várias execuções, este trabalho se concentra na compreensão do comportamento de execuções representativas. Este trabalho projeta visualizações especificamente para a análise de desempenho de memória, enriquecendo a percepção de aplicações baseados em tarefas executados em plataformas heterogêneas.

A.2 Programação Baseada em Tarefas para Plataformas Heterogêneas e o Runtime StarPU

A adoção de diversos aceleradores em HPC está aumentando. As estatísticas da lista dos supercomputadores do TOP500 [1] mostram um aumento de 7,2% entre 2017 e 2018. Nessas situações em que o desempenho da aplicação em relação a nós heterogêneos é diferente, é necessário implantar métodos de programação robustos para usar esses recursos de maneira eficiente. A construção de aplicações para estas plataformas heterogêneas é complexa porque muitas APIs ou paradigmas de programação estão disponíveis para lidar com componentes específicos de um supercomputador. Na situação atual, com diferentes APIs, linguagens de programação e paradigmas, os programadores ficam sobrecarregados pelo número de responsabilidades que eles têm, incluindo a comunicação de dados entre a memória de diferentes recursos, garantir o comportamento correto da aplicação e, ainda assim, alcançar o desempenho máximo que o sistema permite. Um modelo de programação que pode reduzir a complexidade, controlando os recursos, escalonando tarefas e gerenciando os dados é a programação baseado em tarefas.

A programação baseada em tarefas, ou programação de fluxo de dados, é um conceito que usa uma estratégia mais declarativa para transferir algumas responsabilidades para um *runtime*, facilitando a codificação da aplicação e reduzindo o controle do programador sobre a execução (BRIAT et al., 1997; Dongarra et al., 2017). A estrutura desses programas consiste em uma coleção de tarefas que têm um propósito específico sobre uma coleção de dados. Portanto, a interação entre tarefas ocorre principalmente pelo uso

[1] https://www.top500.org/

dos mesmos fragmentos de dados em diferentes tarefas, causando dependências implícitas entre elas para garantir a coerência computacional. Normalmente, um Grafo Acíclico Dirigido (DAG) representa uma aplicação baseado em tarefas, em que os nós são tarefas e as ligações são dependências. Tais dependências são herdadas pela reutilização de dados ou inseridas explicitamente pelo programador. A execução é guiada por um *runtime*, por exemplo, PaRSEC (BOSILCA et al., 2012), XKaapi (GAUTIER et al., 2013) e StarPU (AUGONNET et al., 2011), usado neste trabalho.

O StarPU é um *runtime* baseado em tarefas para fins gerais que fornece uma interface para aplicações submeterem tarefas para recursos. O StarPU usa o modelo STF (Sequential Task Flow) (AGULLO et al., 2016), em que a aplicação submete sequencialmente as tarefas durante a execução e o *runtime* as escalona dinamicamente. Os dados usados por uma tarefa precisam ser declarados em um bloco de memória e a reutilização dos blocos permite que dependências sejam criadas entre as tarefas e estruturem o DAG. Portanto, no StarPU, as dependências entre as tarefas são explícitas. Além disso, as tarefas do StarPU podem ter várias implementações, uma para cada tipo de dispositivo (como CPUs x86, GPUs CUDA e dispositivos OpenCL). O *runtime* pode utilizar diferentes heurísticas para escalonar tarefas para recursos. Dependendo da disponibilidade de recursos e da heurística, o *runtime* escolhe dinamicamente uma das versões da tarefa e a executa.

O *runtime* também é responsável pela transferência de dados entre recursos, para controlar a presença e a coerência das blocos de memória. O StarPU cria um gerenciador de memória para cada tipo diferente de memória e adota o protocolo MSI, com os estados `Modificado/Dono`, `Compartilhado` e `Inválido`, para gerenciar o estado de cada identificador de memória nas diferentes memórias. Em um dado momento, cada bloco de memória pode assumir um dos três estados nos gerenciadores de memória (AUGONNET et al., 2011). Quando não há memória disponível para alocar esse bloco de memória, o StarPU apresenta diferentes comportamentos, dependendo do tipo de solicitação (*Idle Prefetch* / *Prefetch* / *Fetch*, incluindo o uso de memória fora do núcleo (out-of-core). O StarPU é capaz de usar vários nós computacionais usando seu módulo MPI (AUGONNET et al., 2012). E oferece feedback sobre o desempenho da aplicação em duas formas, estatísticas instantâneas e rastros de execução contendo eventos do *runtime*. No final da execução da aplicação, quando o rastreamento está ativo, o StarPU gera um arquivo FXT por processo. Além disso, o arquivo FXT pode ser convertido para outros formatos de rastros mais usados em HPC, como o Paje (SCHNORR; STEIN; KERGOMMEAUX, 2013) e ser usado para investigar o comportamento da aplicação.

A.3 Ferramentas para Analisar Aplicações de HPC

A análise de aplicações em HPC é uma etapa essencial para melhorar o desempenho. Muitos aspectos podem interferir no desempenho das aplicações, como escalonamento de recursos, configuração da plataforma, e para este trabalho, o gerenciamento de memória. Os rastros de execução podem ser usados para verificar se há algum problema nessas situações e ferramentas para ajudar desenvolvedores de aplicações e *runtimes* são desejáveis. Consequentemente, técnicas de visualização são empregadas na análise dessas aplicações e muitas ferramentas foram criadas para preencher essas necessidades. Existem duas categorias de ferramentas, as tradicionais e para as aplicações baseadas em tarefas.

As ferramentas tradicionais de análise de desempenho enfocam os paradigmas de programação OpenMP, MPI, MPI/OpenMP e MPI/CUDA. As ferramentas geralmente se concentram em eventos tradicionais como comunicação e utilização de recursos, informações gerais de qualquer paradigma. Seu objetivo é muitas vezes apresentar o uso dos recursos ao longo do tempo para a detecção de problemas, como por exemplo, recursos ociosos. As técnicas de visualização geralmente empregadas são métodos de espaço/-tempo, onde estados ou eventos são correlacionados a um tempo específico representado em um eixo.

Os analistas de desempenho podem usar a estrutura do DAG de aplicações baseadas em tarefas, pois fornecem informações extras sobre a aplicação e, na maioria dos casos, podem afetar o desempenho. Diferente de aplicações paralelas tradicionais, em que a noção de etapas de computação não pode ser muito definida e as dependências são correlacionadas com as comunicações, as aplicações baseadas em tarefas fornecem uma estrutura definida. Os benefícios de usar informações do DAG para analisar o desempenho levaram à criação de outra categoria de ferramentas para análise de desempenho, especialmente projetadas para aplicações baseados em tarefas.

Todas as ferramentas discutidas se concentram no comportamento geral da aplicação, em que a computação ou as tarefas que estão sendo processadas são o foco e são organizadas nos recursos computacionais. As informações sobre memória geralmente são expressas por métricas, como a quantidade total de comunicação, e os eventos relacionados a dados são exibidos dentro de outros (não relacionados à memória) com base em qual recurso eles ocorrem e não em qual bloco de memória. Mesmo que algumas dessas ferramentas forneçam suporte orientado pelo DAG, elas geralmente não possuem

uma metodologia específica para analisar o impacto de diferentes operações de blocos no desempenho da aplicação ou para correlacionar o comportamento da memória com outras informações. Por exemplo, a memória da aplicação de problemas de álgebra linear é geralmente associada a uma coordenada de um bloco de memória, que uma ferramenta de análise de desempenho pode correlacionar com aspectos de processamento e possíveis otimizações. Por exemplo, queremos verificar onde um determinado bloco de memória está em um ponto específico na execução, porque talvez ele esteja usando espaço em um recurso crítico e nenhuma outra etapa de computação o utilizará novamente. Nenhuma das ferramentas apresentadas pode fornecer essas informações. Além disso, todos os blocos de memória estão associados às informações da aplicação. Se o *runtime* copiar um bloco de memória para a GPU, a maioria das ferramentas geralmente mostrará a comunicação entre os recursos, mas não da detalhes específicos sobre o bloco de memória relacionado (a coordenada do bloco na matriz, por exemplo). Um analista pode usar essas informações para correlacionar o desempenho, o comportamento da memória e o projeto da aplicação. O problema é que as ferramentas apresentadas são incapazes de fornecer tais ações ou investigar profundamente a memória no nível de *runtime*.

As estratégias deste trabalho são projetadas para fornecer uma análise mais orientada à memória, baseada nas informações de onde estão os blocos de memória e com quais tarefas eles estão relacionadas, formando dependências. A abordagem fornece uma análise de desempenho em vários níveis das operações do gerenciamento de dados em uma plataforma multi-GPU e multi-core heterogênea. Este trabalho combina uma visão de alto nível do DAG da aplicação com as decisões baixo nível de memória do *runtime*, que orienta o analista na identificação e correção de problemas de desempenho. Em vez de usar apenas métricas arquiteturais de baixo nível de memória e compará-las com várias execuções, esse trabalho se concentra na compreensão do comportamento de execuções representativas. Este trabalho também projeta elementos de visualização especificamente para a análise de desempenho de memória de um *runtime*, enriquecendo a percepção de aplicativos baseados em tarefas executados em plataformas heterogêneas.

A.4 Contribuição: Estratégias para Analisar a Memória de Aplicações Baseadas em Tarefas no nível de Runtime

Este capítulo apresenta a principal contribuição deste trabalho, a metodologia e as estratégias para investigar o comportamento do gerenciador de memória e os even-

tos/locais dos blocos de memória individuais nos recursos heterogêneos de CPU/GPU. O módulo de gerenciamento de dados do StarPU é responsável por todas as ações que envolvem a memória da aplicação.

A captura de rastros de execução é um recurso já presente no *runtime* StarPU, e este trabalho somente o estende para adicionar novas informações. Essencialmente, para incluir todas as informações necessárias, este trabalho conduz três grandes modificações. A primeira modificação é incluir a identificação de memória em todos os eventos relacionados com informações extras para permitir correlações entre as atividades do *runtime* e para entender as decisões por trás dele. Segundo, uma modificação para rastrear a função de atualização da coerência da memória para acompanhar o paradeiro de um bloco de memória durante a execução. Terceiro, uma modificação para rastrear todas as solicitações de memória (pré-busca, busca, alocação, sincronização) realizadas pelo *runtime*. Todas essas modificações, com exceção de uma MPI, foram adicionadas ao repositório principal do StarPU[2].

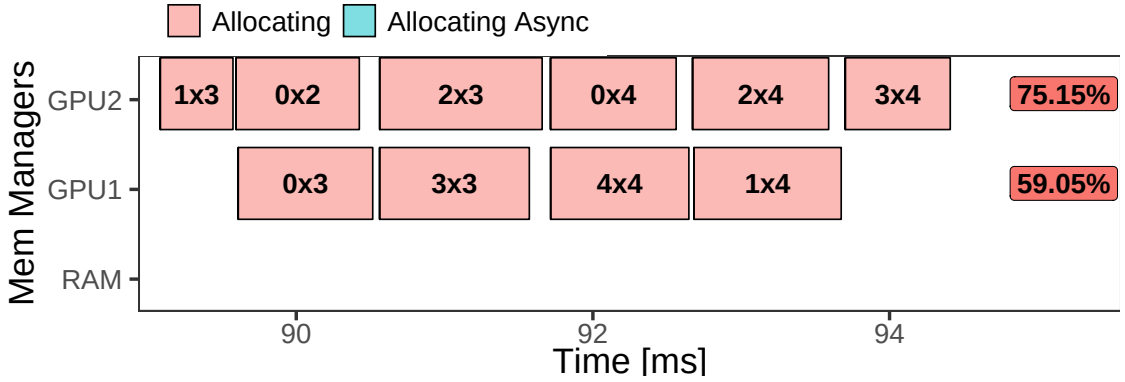
A.4.1 Visualização Tempo/Espaço dos Gerenciadores de Memória

Empregar gráficos de Gantt para analisar o comportamento de aplicações paralelas é comum. É usado para mostrar o comportamento de entidades observadas (trabalhadores, threads, nós) ao longo do tempo. Este trabalho adapta e enriquece esse tipo de visualização para inspecionar o comportamento do gerenciador de memória, como mostrado no exemplo da Figura A.1. No eixo Y, a figura lista os diferentes gerenciadores de memória associados a diferentes memórias de dispositivos: `RAM`, diferentes aceleradores (memória de dispositivos GPU e OpenCL). Este exemplo tem apenas três gerenciadores de memória: `RAM`, `GPU1` e `GPU2`. Os gerenciadores de memória podem executar ações diferentes, como alocar e mover dados. Cada bloco de memória pode ter uma informação extra fornecida pela aplicação. Neste caso, as coordenadas de bloco da fatoração de Cholesky.

A Figura A.1 apresenta as ações de cada gerenciador ao longo do tempo com retângulos coloridos marcados com as coordenadas dos blocos (por exemplo, `GPU2`: 1×3, 0×2 e assim por diante). Os retângulos nesta figura representam diferentes alocações sendo executados pelos gerenciadores, exceto pelo gerenciador da memória `RAM`. que não teve nenhum comportamento registrado no intervalo de 10ms representado. À direita de

[2]Disponível em https://scm.gforge.inria.fr/anonscm/git/starpu/starpu.git, commits 784095d, e097aa8, f068631, e 756e365.

Figura A.1: Comportamento dos três gerenciadores de memória mostrando os estados de alocação para diferentes blocos de memória



Fonte: O Autor.

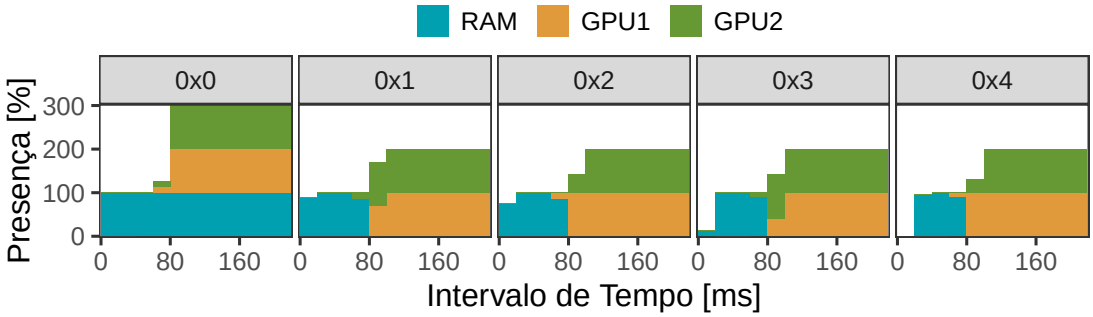
cada gerenciador, o painel descreve a porcentagem de tempo do estado mais recorrente usando a mesma cor. Por exemplo, o gerenciador `GPU2` gastou 75,15% do tempo desse intervalo de tempo específico no estado de alocação. Este gráfico fornece uma visão geral das ações dos gerenciadores de memória, informando também em qual bloco eles estão aplicando tais ações.

A.4.2 Residência dos Blocos de Memória nos Recursos

Um determinado bloco de uma aplicação HPC (fatoração de Cholesky por exemplo) pode residir em vários nós de memória durante a execução. Por exemplo, pode haver muitas cópias de um determinado bloco se os trabalhadores que executam em dispositivos diferentes realizarem somente operações de leitura. Esta situação é devida à adoção do protocolo MSI pelo StarPU, onde vários nós de memória têm cópias do mesmo bloco de memória. A Figura A.2 representa a localização de determinados blocos de memória durante a execução. Cada uma das cinco faces da figura representa um bloco de memória com as coordenadas 0×0, 0×1, 0×2, 0×3 e 0×4 da matriz de entrada. Para cada bloco, o eixo X é o tempo de execução, discretizado em intervalos de tempo de 20ms. Este intervalo é suficientemente grande para a visualização e pequeno o suficiente para mostrar a evolução do comportamento da aplicação. No entanto, outros valores podem ser selecionados para diferentes situações com base na duração total da execução da aplicação. Em cada intervalo de tempo, o eixo Y mostra a porcentagem de tempo que esse bloco está em cada nó de memória (cor). Por exemplo, se um bloco pertencer primeiro a `RAM` por 18 ms e depois a 2 ms pela `GPU2`, a barra será 90% azul e 10% amarela. Como cada bloco

pode ser compartilhado e, portanto, presente em vários nós de memória, a porcentagem máxima de residência no eixo Y pode exceder 100%. O máximo depende de quantos nós de memória existem na plataforma. Além disso, se a memória residir apenas em uma parte do intervalo de tempo, a porcentagem seria menor que 100%.

Figura A.2: O local dos blocos de memória ao longo do tempo



Fonte: O Autor.

Com essa nova visualização, é possível verificar uma evolução resumida do movimento de dados e da utilização da memória do recurso. Por exemplo, a Figura A.2 detalha que o bloco de memória com coordenadas 0×0 permaneceu na `RAM` durante tota a execução. O StarPU transferiu o bloco para ambas as GPUs no tempo ≈80ms e manteve-a em todos os recursos até o final da execução. Os outros blocos, no entanto, permaneceram na `RAM` somente até ≈80ms da execução e foram transferidos para ambas as GPUs. Um analista é capaz de identificar rapidamente anomalias correlacionando a residência das coordenadas do bloco com as fases da aplicação. Muito frequentemente na álgebra linear, uma coordenada de bloco inferior é usada somente no início da execução, o que seria demonstrado como 0% de ocupação daquele bloco depois que ele não é mais necessário.

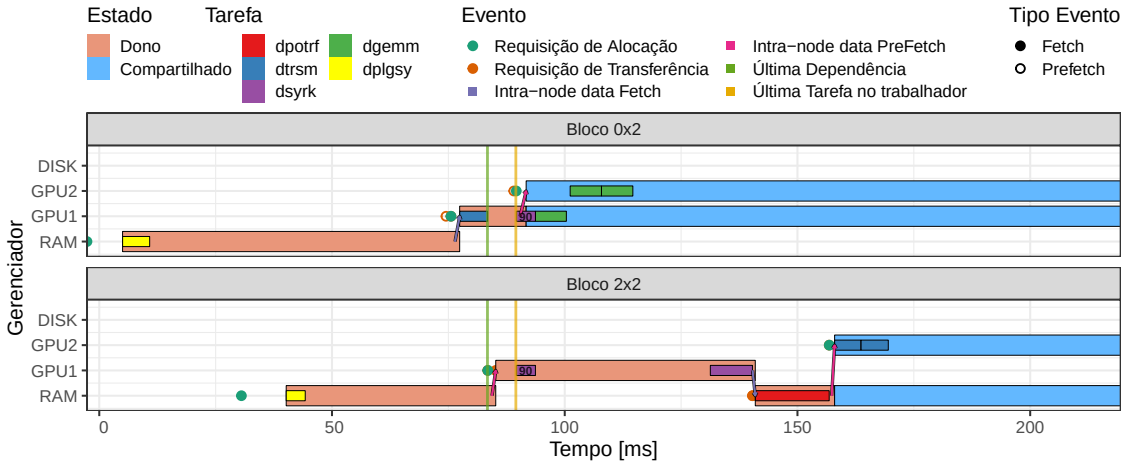
A.4.3 Comportamento Temporal Detalhado dos Blocos de Memória

A estratégia anterior mostra onde um determinado bloco está localizado (em qual nó de memória) durante a execução, no entanto, ele não fornece detalhes de quando as transferências ocorreram e quais eventos do StarPU estão influenciando os gerenciadores de memória. Para solucionar esses problemas, um histórico completo de eventos para cada identificador de memória seria desejável para o entendimento de situações específicas. Isso está presente na Figura A.3, mostrando o local do bloco de memória com o estado MSI e, além disso, descreve todas as atividades de tarefas do tempo de execução e aplicação que afetam o comportamento do bloco. Essas atividades incluem transferências

do *runtime* entre recursos, eventos internos do gerenciador de memória do *runtime*, como solicitações de transferência e alocação, e informações relacionadas à tarefa, como última dependência e o tempo da última tarefa no mesmo trabalhador.

A estratégia emprega o tradicional gráfico de Gantt como base para a visualização, onde o eixo X é o tempo em milissegundos e o eixo Y representa os diferentes gerenciadores de memória. Existem dois tipos de estados, representados por retângulos coloridos. Os mostrados mais ao fundo com uma altura maior representam a residência do bloco de memória nos gerenciadores: a cor vermelha expressa quando um nó de memória é proprietário do bloco, enquanto a cor azul indica que o bloco é compartilhado entre diferentes gerenciadores. Os retângulos internos representam as tarefas Cholesky (`dpotrf`, `dtrsm`, `dsyrk`, `dgemm` e `dplgsy`) que estão executando e utilizando esse bloco de memória desse gerenciador de memória. A estratégia apresenta uma representação aumentada com diferentes eventos associados aos blocos de memória no respectivo gerenciador e tempo. Os círculos (`Requisição de alocação`, `Requisição de transferência`) estão preenchidos ou não preenchidos, para operações de *fetch* ou *prefetch*, respectivamente. As setas são usadas para representar uma transferência de dados entre dois nós de memória e possuem um significado diferente (codificado com cores diferentes: *fetch* ou *prefetch* intra-nó). Finalmente, duas linhas verticais indicam a correlação (última dependência e a última tarefa no mesmo trabalhador) com uma tarefa selecionada pelo analista.

Figura A.3: Eventos detalhados relacionados a dois blocos de memória (faces), mostrando a residência (estados maiores) e seu uso pelas tarefas (estados menores)



Fonte: O Autor.

Na Figura A.3, a tarefa 90 é destacada (que é uma tarefa `dsyrk`), e apenas blocos de memória usados pela tarefa 90 são mostrados. A linha vertical verde representa o final

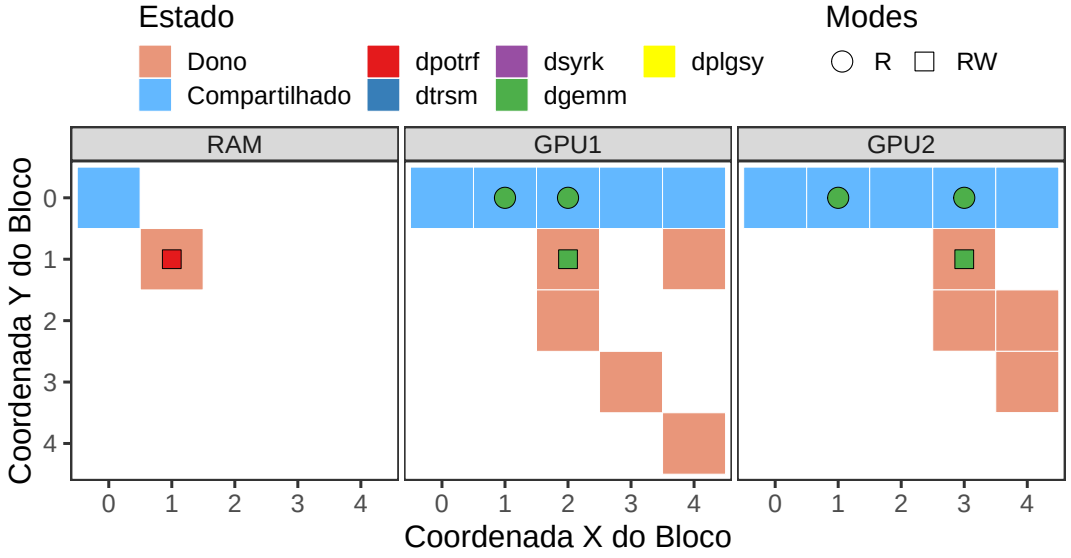
da última dependência que libera a tarefa 90 e o amarelo representa o final da última tarefa executada no mesmo trabalhador. Além disso, a figura mostra que o bloco 0×2 era de propriedade da memória RAM desde a execução de sua tarefa de geração `dplgsy` e foi movido para GPU 1 para ser usado por uma tarefa `dtrsm`. Durante a execução da tarefa 90, ela foi movida para GPU 2 em um estado compartilhado, onde suas informações são duplicadas em ambas as GPUs. Antes de cada uma das transferências, os eventos de requisição de transferência e alocação foram executados.

A.4.4 Momentos Específicos e Instantâneos para Avaliar o Bloco de Memória

Uma aplicação sobre o StarPU determina os dados e as tarefas que serão usadas pelo *runtime*. Em vez de considerar apenas a utilização de recursos, seria útil correlacionar o algoritmo e as decisões de *runtime*. Essa estratégia cria uma visão que leva em consideração as coordenadas dos blocos nos dados originais, ilustrando qual tarefa está usando cada bloco e seu estado nos gerenciadores (dono, privados ou compartilhados). A Figura A.4 mostra um momento instantâneo de todos os locais dos blocos de memória e as tarefas em execução em um momento específico. A visualização tem três faces, uma para cada gerenciador de memória (`RAM`, `GPU1`, `GPU2`). Cada gerenciador possui uma matriz com as coordenadas do bloco nos eixos X e Y. Nesta matriz, cada quadrado colorido representa um bloco de memória nas coordenadas fornecidas pela aplicação, a cor de cada bloco informa o estado MSI real do identificador de memória no momento utilizado. Os quadrados internos coloridos (acesso ao modo de gravação) ou círculos (acesso ao modo de leitura) dentro desses blocos representam tarefas da aplicação. Com esta visualização, é fácil confirmar como o fluxo de dados da memória se correlaciona com a posição dos blocos e entender a progressão da memória usando as informações fornecidas pela aplicação (neste caso as coordenadas do bloco).

Na Figura A.4, por exemplo, existem apenas dois blocos em `RAM` e que ambos as GPUs compartilham a primeira linha. Além disso, há uma tarefa `dpotrf` executando sobre o bloco 1×1 em `RAM` e uma tarefa `dgemm` em cada GPU. `GPU1` tem acesso de gravação na tarefa `dgemm` no bloco 1×2 e dois acessos de leitura nos blocos 0×1 e 0×2. Ao utilizar momentos instantâneos consecutivos, é possível criar uma animação que mostre a residência dos blocos de memória ao longo do tempo. Esse recurso é particularmente útil para entender o comportamento do algoritmo e a as operações sobre os dados.

Figura A.4: Momento específico da residência dos blocos de memória



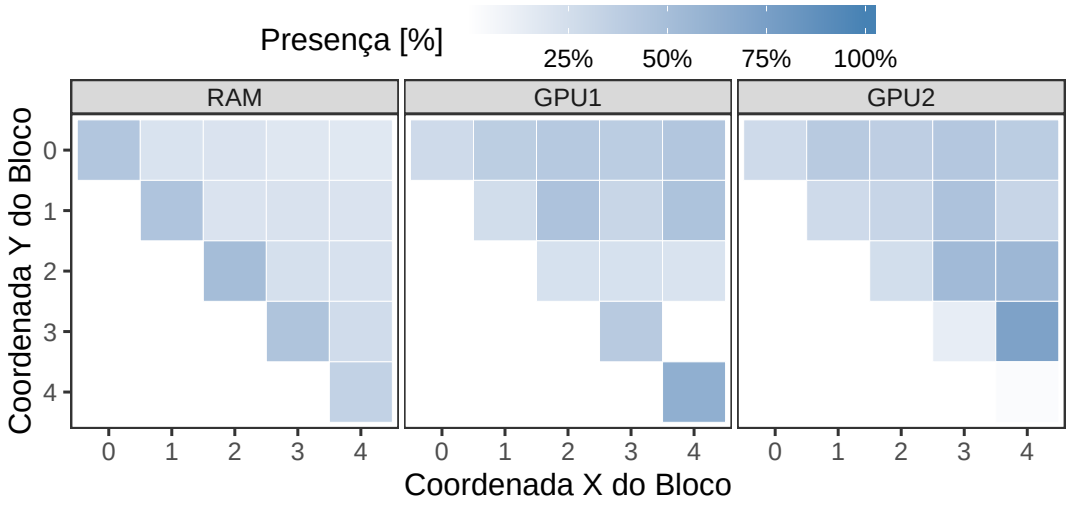
Fonte: O Autor.

A.4.5 Mapa de Calor para Verificar a Residência dos Blocos de Memória

Um analista pode estar interessado em uma visão geral da execução da localidade dos blocos entre os gerenciadores. Um *footprint* da residência geral dos blocos de memória poderia fornecer *insights* rápidos sobre o comportamento. O painel final consiste em uma visualização tradicional de mapa de calor para fornecer um resumo da presença total das blocos. A Figura A.5 descreve um exemplo dessa estratégia. Há uma face de visualização para cada gerenciador de memória. No exemplo, há um gerenciador de memória RAM, GPU 1 e GPU 2. Cada quadrado representa um bloco de memória posicionado em sua coordenada matricial da aplicação nos eixos X e Y. Outras estruturas podem ser usadas dependendo da informação extra que a aplicação fornece aos blocos de memória. A tonalidade da cor azul representa a quantidade total de tempo que o bloco está no gerenciador durante a execução da aplicação. Enquanto a cor branca representa baixa presença no gerenciador de memória, a cor mais azulada representa alta presença.

Na Figura A.5, por exemplo, é possível observar que todos os blocos da diagonal ficaram mais tempo na memória `RAM` em comparação com outros blocos. Essa situação acontece porque, para as tarefas Cholesky, `dpotrf` e `dsyrk` na diagonal, são executadas normalmente em CPUs, para permitir que as GPUs processem principalmente tarefas SYRK e GEMM. Diferentemente, os outros blocos estavam presentes em ambas as GPUs, pois as tarefas GEMM eram preferenciais executadas nas GPUs e usavam todos esses blocos. Além disso, todos os blocos têm pelo menos uma pequena presença na memória

Figura A.5: Mapas de calor por gerenciador da residência total dos blocos



Fonte: O Autor.

RAM, pois todos os blocos foram gerados primeiro nela.

A.5 Resultados Experimentais em Aplicações Baseadas em Tarefas

Este capítulo apresenta os resultados experimentais com três diferentes aplicações heterogêneas baseadas em tarefas. Um solucionador de Cholesky denso, usando a suíte Chameleon (AGULLO et al., 2010). Uma simulação de CFD (NESI; SCHNORR; NAVAUX, 2019), focando em um método de particionamento específico dos dados. Um solucionador da fatoração QR usando matrizes esparsas, do pacote de software `QR_MUMPS` (BUTTARI, 2012). Os experimentos foram realizados usando a abordagem reprodutível, e as máquinas utilizadas fazem parte do GPPD (Grupo de Processamento Paralelo e Distribuído) UFRGS.

As estratégias são aplicadas no cálculo de uma fatoração de Cholesky densa baseada em blocos da aplicação Chameleon usando plataformas heterogêneas com GPUs, algoritmos *out-of-core* que movem memória extra para o disco, e execuções distribuídas com StarPU-MPI em múltiplos nós. Cinco cenários são investigados e são descritos a seguir. (i) A percepção errada pelo StarPU da utilização da memória da GPU, emitindo muitas solicitações de alocação de memória que prejudicam o desempenho. O problema foi corrigido e levou a 66% de ganho de desempenho. (ii) Identificação de tempos ociosos significativos em execuções com o recurso *out-of-core* e fornecendo detalhes sobre as operações de memória que causaram isso. (iii) A identificação de problemas de desempenho inesperados causados pela ordem de geração dos dados de entrada usando dois

escalonadores (DMDAR e LWS) do StarPU sob memória limitada e uso de *out-of-core*. (iv) A explicação das diferenças de desempenho entre dois escalonadores (DMDAR e DMDAS) do StarPU em uma execução somente na CPU com restrições de memórias e *out-of-core*. (v) Uma prova de conceito sobre o uso das estratégias empregando uma execução com vários nós usando StarPU-MPI.

As estratégias também foram aplicadas em um código de simulação CFD, durante a análise de um dos esquemas de partição dos dados. Os experimentos apresentam problemas com alto uso de memória nas GPUs, resultando em tempos ociosos nos recursos. As estratégias propostas encontraram um comportamento prejudicial de ter blocos de memória desnecessários nas GPUs. A descoberta desse problema levou à criação de duas otimizações, a primeira de informar as prioridades das tarefa e informar quando os dados não são mais necessários. Essas otimizações levam a um aumento de 38% no desempenho da aplicação. Além disso, no caso da aplicação QR_Mumps, as estratégias perceberam um comportamento de *ping-pong* da memória causado pelas decisões de escalonamento de tarefas e não relacionadas a problemas de memória.

A.6 Conclusão

O paradigma de programação baseado em tarefas é um ótimo candidato para programar aplicações paralelas heterogêneas porque um *runtime* ajuda em algumas responsabilidades. No entanto, como outras abordagens de HPC, a análise de desempenho dessas aplicações é desafiadora. Muitos aspectos podem afetar o desempenho. Um aspecto estudado neste trabalho é o gerenciamento de memória da aplicação no nível de *runtime*. O *runtime* é responsável por tomar muitas decisões em relação ao sistema de memória. Ferramentas que ajudam na análise de desempenho da memória de aplicações baseados em tarefas são então desejáveis.

Este trabalho apresenta uma metodologia de análise de desempenho baseada em técnicas de visualização conhecidas que são modificadas de forma inovadora. Estas metodologias tem como objetivo ajudar na identificação de ineficiências de transferência e gerenciamento de dados tanto no *runtime* StarPU quanto no código da aplicação. As estratégias propostas, para investigar o desempenho da memória de aplicações baseadas em tarefas, são aplicadas em três problemas diferentes, a suíte Chameleon para resolver problemas de álgebra densa, uma aplicação CFD, e o QR_Mumps um solucionador para fatoração QR esparsa.

Os estudos de caso indicam que as estratégias propostas são fundamentais para identificar as razões por trás dos problemas de desempenho relacionados às operações de memória. As estratégias levaram a entender como consertar as operações de memória do *runtime* StarPU e como otimizar as aplicações individualmente. Correções e otimizações resultantes do uso das estratégias propostas levaram a 66% de ganho de desempenho no caso da fatoração Cholesky e 38% no caso da aplicação CFD.

Trabalhos futuros incluem a análise de transferências de dados em outras execuções distribuídas do StarPU-MPI. Além disso, outra possibilidade é o estudo das estratégias em outras aplicações, especialmente aquelas que não possuem os dados organizados em um formato 2D. Além disso, adaptar a metodologia e a ferramenta para outros *runtimes* baseados em tarefas, incluindo o OpenMP tasks.