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

JONAS MULLER KORNDORFER

High Performance Trace Replay for Event Simulation of Parallel Programs Behavior

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

Advisor: Prof. Dr. Lucas Mello Schnorr

Porto Alegre
October 2016

CIP — CATALOGING-IN-PUBLICATION

Muller Korndorfer, Jonas

High Performance Trace Replay for Event Simulation of Parallel Programs Behavior / Jonas Muller Korndorfer. – Porto Alegre: PPGC da UFRGS, 2016.

75 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2016. Advisor: Lucas Mello Schnorr.

1. Parallel Application, Performance Analysis, High Performance, Big Data, Trace Replay. I. Mello Schnorr, Lucas. II. Título.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

ACKNOWLEDGMENTS

This work is the result of two and a half years of personal development supported by several people.

In the first place, I would like to thank my advisor Dr. Lucas Mello Schnorr for its close participation and motivation during all the work. Lucas, you were essential for the completion of this work, and I really want to thank you for all the “tricks and tips” about organization, text writing, English, “procrastination hehe” and many other knowledge that you gave me. I have really evolved with your orientation and I think you are an example of advisor, I hope that we can work together again in the future.

I am also grateful to the professor Dr. Claudio R. Geyer that have initiated me in the PPGC, believing in me and helping me several times.

Second and no less important, or even most important, I have to greatly thank my girlfriend, Marjorie Moraes! Marjorie, how you were able to handle with me all this time? I was so boring! Thank you for being so patient with me, thanks for motivate me all this time, thanks for all the “earful” that made me back to work, thanks for being my girlfriend, I love you.

I would like to thanks my family, mother (Elena Muller), father (Sergio A. Korndorfer) and sister (Rita C. M. Korndorfer), by staying on my side all this time, motivating and supporting me in the "moments of despair" and all other moments, of course hehe.

Thank you my classmates, Vinícius G. Pinto, Luís F. Millani, Flavio Alles, Alef Farah Herbstrith Vinicius, for helping me several times with many knowledge during this work. I also would like to make an extra and special thanks for the colleagues Vinícius G. Pinto and Luís F. Millani that, besides all, have watched my presentation and continue to help me even being outside of the country.

Lastly, I would like to thank my friends, Arthur M. Daudt, Cesar P. Purper, Gabriel Maciel, Guilherme M. Daudt, Letícia A. Kruse, Marcus L. Rohden, Marjorie Moraes (yes, she is also one of my best friends), Ricardo M. Oliveira and Suliane Cardoso. Thank you! You have made my life, together with this work, happier and lighter, making me laugh and relax. An special thank for my friend Guilherme that gave me some tips about writing in English which have really improved this text.

Thanks!

ABSTRACT

Modern high performance systems comprise thousands to millions of processing units. The development of a scalable parallel application for such systems depends on an accurate mapping of application processes on top of available resources. The identification of unused resources and potential processing bottlenecks requires good performance analysis. The trace-based observation of a parallel program execution is one of the most helpful techniques for such purpose. Unfortunately, tracing often produces large trace files, easily reaching the order of gigabytes of raw data. Therefore trace-based performance analysis tools have to process such data to a human readable way and also should be efficient to allow an useful analysis. Most of the existing tools such as Vampir, Scalasca, TAU have focus on the processing of trace formats with a fixed and well-defined semantic. The corresponding file format are usually proposed to handle applications developed using popular libraries like OpenMP, MPI, and CUDA. However, not all parallel applications use such libraries and so, sometimes, these tools cannot be useful. Fortunately, there are other tools that present a more dynamic approach by using an open trace file format without specific semantic. Some of these tools are the Paraver, Pajé and PajeNG. However the fact of being generic comes with a cost. These tools very frequently present low performance for the processing of large traces.

The objective of this work is to present performance optimizations made in the PajeNG tool-set. This comprises the development of a parallelization strategy and a performance analysis to set our gains. The original PajeNG works sequentially by processing a single trace file with all data from the observed application. This way, the scalability of the tool is very limited by the reading of the trace file. Our strategy splits such file to process several pieces in parallel. The created method to split the traces allows the processing of each piece in each thread. The experiments were executed in non-uniform memory access (NUMA) machines. The performance analysis considers several aspects like threads locality, number of flows, disk type and also comparisons between the NUMA nodes. The obtained results are very promising, scaling up the PajeNG about eight to eleven times depending on the machine.

Keywords: Parallel Application, Performance Analysis, High Performance, Big Data, Trace Replay.

Ferramenta de Alto Desempenho para Análise do Comportamento de Programas Paralelos Baseada em Rastos de Execução

RESUMO

Sistemas modernos de alto desempenho compreendem milhares a milhões de unidades de processamento. O desenvolvimento de uma aplicação paralela escalável para tais sistemas depende de um mapeamento preciso da utilização recursos disponíveis. A identificação de recursos não utilizados e os gargalos de processamento requere uma boa análise desempenho. A observação de rastros de execução é uma das técnicas mais úteis para esse fim. Infelizmente, o rastreamento muitas vezes produz grandes arquivos de rastro, atingindo facilmente gigabytes de dados brutos. Portanto ferramentas para análise de desempenho baseadas em rastros precisam processar esses dados para uma forma legível e serem eficientes a fim de permitirem uma análise rápida e útil. A maioria das ferramentas existentes, tais como Vampir, Scalasca e TAU, focam no processamento de formatos de rastro com semântica associada, geralmente definidos para lidar com programas desenvolvidos com bibliotecas populares como OpenMP, MPI e CUDA. No entanto, nem todas aplicações paralelas utilizam essas bibliotecas e assim, algumas vezes, essas ferramentas podem não ser úteis. Felizmente existem outras ferramentas que apresentam uma abordagem mais dinâmica, utilizando um formato de arquivo de rastro aberto e sem semântica específica. Algumas dessas ferramentas são Paraver, Pajé e PajeNG. Por outro lado, ser genérico tem custo e assim tais ferramentas frequentemente apresentam baixo desempenho para o processamento de grandes rastros.

O objetivo deste trabalho é apresentar otimizações feitas para o conjunto de ferramentas PajeNG. São apresentados o desenvolvimento de um estratégia de paralelização para o PajeNG e uma análise de desempenho para demonstrar nossos ganhos. O PajeNG original funciona sequencialmente, processando um único arquivo de rastro que contém todos os dados do programa rastreado. Desta forma, a escalabilidade da ferramenta fica muito limitada pela leitura dos dados. Nossa estratégia divide o arquivo em pedaços permitindo seu processamento em paralelo. O método desenvolvido para separar os rastros permite que cada pedaço execute em um fluxo de execução separado. Nossos experimentos foram executados em máquinas com acesso não uniforme à memória (NUMA). A análise de desempenho desenvolvida considera vários aspectos como localidade das *threads*, o número de fluxos, tipo de disco e também comparações entre os nós NUMA. Os resultados obtidos são muito promissores, escalando o PajeNG cerca de oito a onze vezes, dependendo da máquina.

Palavras-chave: Aplicação Paralela, Análise de Desempenho ,Alto Desempenho, Big Data, Trace Replay.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The continuous increasing demand of computing power tends to the development of efficient parallel applications that better use the available resources. Modern high performance systems comprise thousands to millions of processing units connected by complex hierarchical networks. The development of parallel applications for such systems is a complicated task, affected by at least two factors: the first is the high scalability provided by them which is hard to achieve; the second is the lack of a deterministic execution. Since processing nodes are independent, the understanding of such applications can be very painful. Some libraries like OpenMP (DAGUM; MENON, 1998) and MPI (GROPP; LUSK; THAKUR, 1999) are used to facilitate the implementation of these applications. However, even with supporting libraries, the successful development of a high performance parallel application depends on an accurate mapping of the application's processes on top of available resources.

The identification of unused resources and potential processing bottlenecks requires good performance analysis tools that are able to observe many entities over long periods of time. This observation is often initiated by collecting significant performance measurements through event tracing. The collected data is usually from the application level. Tracing tools commonly register local and global states of the program, the amount of data transferred by messages and hardware counters for specific functions. This data enables the detection of several complex patterns, such as late communications, costly synchronization or train effects (SCHNORR; LEGRAND; VINCENT, 2012). Unfortunately, tracing often produces large trace files, easily reaching the order of gigabytes of raw data. Therefore the processing of such data to a human readable way should be efficient to allow a quick and useful performance analysis.

Over the years, several tracing tools have been developed for performance analysis of parallel applications. The major challenges of these tools are the processing and interpretation of the large amount of data produced by parallel applications. The data is frequently used to replay/simulate the behavior of the program execution. This results in visual representations such as space/time views, communication graphs among others (SCHNORR, 2009). Most of the existing tools such as Vampir (MÜLLER et al., 2007), Scalasca (GEIMER et al., 2010), TAU (SHENDE; MALONY, 2006) have focus on the processing of trace formats with a fixed and well-defined semantic. The corresponding file format are usually proposed to handle applications developed using popular libraries like OpenMP, MPI, and CUDA. However, not all parallel applications use such libraries and so, sometimes, these tools cannot be used. Parallel runtimes provided by StarPU (AUGONNET et al., 2011) and XKaapi (GAUTIER et al., 2013) are examples where usual HPC tracing tools are incapable to obtain correct performance data. Fortunately, there are other tools that present a more dynamic approach. They have an open trace file format without being associated with any specific semantic. This way they can handle with a wide range of parallel applications including parallel run-times such as those mentioned before. Some of these tools are: Paraver (PILLET et al., 1995), Paje (KERGOMMEAUX; STEIN; BERNARD, 2000) and PajeNG (SCHNORR, 2014b). The fact of being more generic comes with a cost. These tools very frequently present low performance for the processing of large traces. In some occasions, they handle the increasing data size by applying aggregation techniques which can remove important performance information from the original trace data.

This work focuses in the optimization of the PajeNG's toolset, more specifically the Paje simulator responsible for replaying traces and re-creating the original behavior of the observed parallel application. This toolset works with the Pajé trace format (SCHNORR, 2014a) which has an open semantic. PajeNG relies on its Paje simulator and built on top of that tools for performance analysis. Examples of such tools are the space/time view provided by `pajeng` and `pj_dump`. The latter is capable to export performance data in a CSV-like textual format well tailored to conduct R analysis (IHAKA; GENTLEMAN, 1996). The current implementation of the Paje simulator is sequential and therefore not scalable when dealing with large trace files. This dissertation presents a parallelization strategy for PajeNG, intending to improve its performance and scalability. This document is organized as follows.

Chapter 2: Basic Concepts. The main purpose of this chapter is to present basic concepts that are important for a good understanding of this work. It explains two ways of how to measure application performance, and it also provides an overview about the Non-Uniform Memory Access (NUMA) architecture.

Chapter 3: Related Work. This chapter makes an overview about the current scenario of performance analysis tools based on trace replay. The most common approaches used to build such tools with high performance are presented. The chapter also presents common parallelization patterns/techniques.

Chapter 4: Parallel Trace Replay in PajeNG. This chapter presents the implemented parallelization technique for the PajeNG simulation library. It details the needed modifications and the optimizations made by our work.

Chapter 5: Performance Analysis and Results. This chapter presents a performance analysis of our solution. We detail and explain the impressive results, scaling up to eleven times the original PajeNG with the developed strategy. The chapter initiates describing the experimental platform and the performance methodology. Then it continues to an overview about the achieved speedup and efficiency, and terminates by a more detailed analysis including disk measurements and comparisons between different NUMA nodes.

Chapter 6: Conclusion. This chapter lists the main contributions of this work. It also includes some ideas for future work.

2 BASIC CONCEPTS

There are many ways to evaluate the performance of parallel applications. Common examples include the energy consumption, instructions per second, memory usage, speedup, efficiency and so on. This chapter presents two of these metrics: how to determine the speedup and the corresponding efficiency achieved by a parallel program. The chapter also brings an overview about how the NUMA (Non-Uniform Memory Access) architecture works. This knowledge is essential to understand our performance analysis Chapter 5, since the experiments were performed on NUMA machines. For this, the Chapter is divided in three sections: Section 2.1 presents the Amdahl's Law (AMDAHL, 1967) for strong scalability and the Section 2.2 with the Gustafson's Law (GUSTAFSON, 1988) for weak scalability. Section 2.3 offers an overview about the NUMA architecture.

2.1 Amdahl's Law and Strong Scalability

The ideal speedup/efficiency of a parallel program would be linear. Doubling the number of processing elements should halve the runtime maintaining the efficiency at 100%. The ideal speedup is hard to obtain. Gene Amdahl have shown the limit on a parallel program's speedup/efficiency as it executes with more cores while solving the same problem size (strong scalability). The Figure 2.1 illustrates such idea: the program should ideally take $1/K$ amount of time to perform the result for the same problem, with K processes. Amdahl proposes some equations as follows. Equation 2.1 shows the sequential program's running time divided by the parallel program's running time with K processes, ultimately defining the speedup metric. Amdahl's efficiency is shown in Equation 2.2: it calculates how far the program is from the ideal speedup.

$$Speedup(N, K) = \frac{Tseq(N, 1)}{Tpar(N, K)} \tag{2.1}$$

$$Efficiency(N, K) = \frac{Speedup(N, K)}{K} \tag{2.2}$$

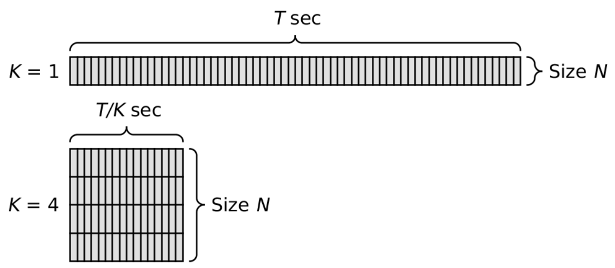


Figure 2.1 – Visual example for strong scalability. Figure adopted from (KAMINSKY, 2013).

Gene Amdahl asserts that every parallel program has a sequential fraction that limits the performance gains. So even increasing the number of cores, the runtime will never be faster than such fraction's time. Equation 2.3 describes this interaction providing a program's total running time using K cores. The sequential fraction F is the total running time that must be performed in a single core. When executing with one core, the program's total execution time is T, the sequential portion takes time FT, and the parallelizable portion takes $(1 - F)T$. When using K cores, the parallelizable portion may experience an ideal speedup taking $(1 - F)T/K$ time, but the sequential portion still takes FT time. The sequential portion F therefore limits the possible parallelization gains. This is famous equation called Amdahl's Law (AMDAHL, 1967).

$$T(N,K) = FT(N,1) + \frac{1-F}{K}T(N,1) \tag{2.3}$$

2.2 Gustafson's Law and Weak Scalability

John Gustafson has explained another way to check the scalability of a parallel program when more cores are available. Differently from Amdahl's law where the problem size is kept fixed, Gustafson propose an alternative manner by letting the problem size change according to the number of resources available. This is known as weak scaling as exemplified by the Figure 2.2: as the number of cores increases, the problem size is also increased in the same proportion. So ideally it should take the same amount of time to perform a K times larger problem. Therefore, the main idea is how much we can increase the problem size/number of cores maintaining the runtime. With that Gustafson asserts that as the problem size increases, the program's parallelizable portion's running time also increases. So how the program's sequential portion typically remains the same, it will becoming less and less significative when considering the program's total running time. Thus the speedup still increases without hitting the limit imposed by Amdahl's law.

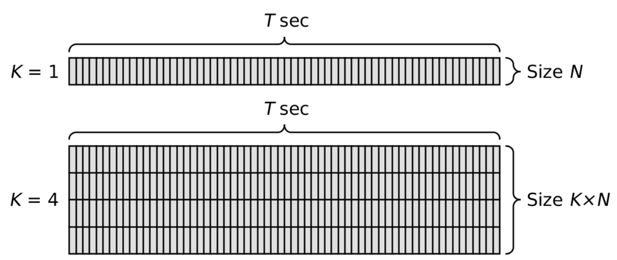


Figure 2.2 – Visual example for weak scalability. Adopted from (KAMINSKY, 2013).

The main equations for weak scaling are the Sizeup and Efficiency. The Sizeup equation 2.4 is the main metric for measuring weak scaling. Considering the T time of the sequential program with the 1 problem divided by the T time of the parallel program with K cores and K problems. The Efficiency equation 2.5 tells how close a program is to the ideal Sizeup. This is much like the strong scaling efficiency, that is dividing the $Sizeup(N,K)$ by K number of cores.

$$Sizeup(N,K) = \frac{N(K)}{N(1)} * \frac{Tseq(N(1),1)}{Tpar(N(K),K)} \tag{2.4}$$

$$Efficiency(N,K) = \frac{Sizeup(N,K)}{K} \tag{2.5}$$

2.3 The Non-Uniform Memory Access (NUMA) Achitecture

The Non-Uniform Memory Access (NUMA) architecture was designed to improve the scalability limits of the Symmetric Multi-Processing (SMP) architecture in which all memory access are posted to the same shared memory bus. The SMP works relatively well for a small number of CPUs, however with a larger number of CPUs there are much concurrency to access the shared memory bus. Therewith NUMA intends to reduce these bottlenecks by limiting the number of CPUs on a memory bus. The NUMA architecture classifies memory into NUMA nodes in which a certain number of cores reside. All cores of the same node have equivalent memory access characteristics, and all nodes are inter-connected by means of a high speed interconnect.

The Figure 2.3 shows a visual representation of what is a NUMA architecture. Each rectangle presents a NUMA node with its respective cores and local memory range. The black lines mean the high speed interconnect mentioned above. The NUMA architecture has some particular features that are very important for a program running on it. All cores of the system can access the entire memory of any node, however not at all with the same speed. Two concepts were created to differ the memory access pattern: *local node* that is when a core access the memory range of its own node; and *remote node* that occurs when a core allocates memory that resides in another node local memory. The *local node* access is much faster than a *remote node*. Therefore a parallel program has to be prepared to execute in a NUMA machine, otherwise it will lose performance. The development of a program for NUMA commonly uses the *NUMA aware* approach which means maximize the local memory allocation.

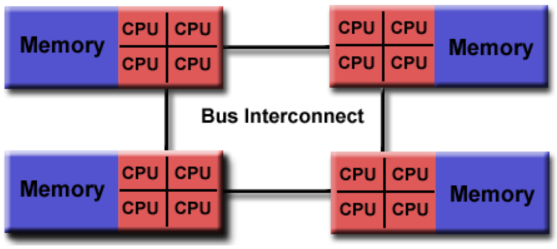


Figure 2.3 – Simple visual representation of the NUMA architecture. Adopted from (BARNEY et al., 2010).

3 RELATED WORK

The goal of this chapter is to present what are the most common parallelization techniques to build high performance tools. This includes parallelization techniques/-patterns and the internal structure of several existing tools. The chapter is structured in two sections as follows. Section 3.1 presents parallelization techniques while Section 3.2 makes an overview about performance analysis tools. The objective of latter is to detail the main features of each tool and its processing model. The last Section 3.3 makes a comparison between the surveyed tools and presents some conclusions.

3.1 Parallelization Techniques

Parallelization techniques are divided here in implicit techniques for automatic compiler-based parallelization and explicit parallelization techniques/patterns.

3.1.1 Automatic Compiler-based Parallelization Techniques

This section was inspired by E. Raman's thesis (RAMAN, 2009) and presents some of the most general and important techniques for automatic compiler-based parallelization. We first explaing DOALL, one of the first automatic parallelization techniques. Then, we describe DOACROSS and DSWP, both designed to handle of inter-iteration dependencies during automatic parallelization.

Independent Multi-Threading (IMT) and the DOALL Technique

Independent Multi-Threading (IMT) techniques are characterized by the nonexistence of communication between the threads that execute loops in parallel. The main parallelization technique for this approach is DOALL (LUNDSTROM; BARNES, 1980; ALLEN; KENNEDY, 2002). It extracts iteration level parallelism by partitioning the loop iterations into threads and executing them concurrently with no or very few restrictions. This means that IMT techniques requires loops without inter iteration dependencies. The great point of the DOALL technique is that its performance is independent of the communication latency between cores, since the threads do not communicate within the loop body giving a near linear speedup. On the other hand, this technique has a limited applicability since most loops have several dependencies between the iterations.

Cyclic Multi-Threading (CMT) and the DOACROSS Technique

Cyclic Multi-Threading (CMT) techniques intend to use synchronization methods to extract parallelism from loops even when there are iteration dependencies. DOACROSS (CYTRON, 1986) is one of the firsts and much important techniques for this approach. Like IMT techniques, DOACROSS obtains parallelism by executing iterations concurrently across threads. However the iterations mapping to the threads is guarded by points of synchronization and communication. The code 3.1 shows an example of a loop with inter-iteration dependencies due to the pointer chasing load LD. By using the DOACROSS technique, all threads are synchronized sending and receiving tokens from each others resulting in a cyclic communication among them. The great advantage of the DOACROSS technique is its applicability that covers more possibilities than the DOALL. On the other hand its performance depends on the number of stall cycles: the speedup decreases from the linear according to how many stall cycles have occurred during the execution.

Listing 3.1 – Loop incrementing a linked list with inter-iteration dependence due to the pointer chasing load. Adopted from (RAMAN, 2009).

```
ptr = first;
While(ptr = ptr->next) //LD
{
    ptr->val += 1; //A
}
```

Pipelined Multi-Threading (PMT) and the DSWP Technique

Pipelined Multi-Threading (PMT) also intends to extract parallelism even with the existence of inter-iteration dependencies. Decoupled software pipeline (DSWP) (RANGAN et al., 2004; OTTONI et al., 2005) is one of the most general techniques for this approach. DSWP works by splitting the loop body in a sequence of pipeline stages and executes each one in a separated flow. The first operation of the DSWP technique is construct the program dependence graph (PDG) (FERRANTE; OTTENSTEIN; WARREN, 1987) of the loop and identify which are the Strongly Connected Components

(SCC) (TARJAN, 1972). Thus a new PDG can be formed based on the SCCs in which each SCC composes a single node of the graph. Finally, it forms the directed acyclic graph (DAG) that can have each node mapped to a separated thread. Therefore each SCC must be processed by a single thread: it prevents that two or more threads share operations of the same SCC, thus avoiding cyclic communication between them. The speedup obtained with the DSWP technique is limited by the execution latency of the thread with the longest execution time. So differently from IMT and CMT techniques, this approach is not scalable according to the number of iterations or input size.

3.1.2 Explicit Parallelization Techniques and Patterns

The pipeline (MATTSON; SANDERS; MASSINGILL, 2004) pattern describes a way to handle data in parallel. It proposes the connection of tasks in a producer–consumer relationship, basically creating a linear sequence of stages. Conceptually all stages of the parallel pipeline are active at once and can be updated as the data flows through them. There are two basic approaches to implement the pipeline (MCCOOL; REINDERS; ROBISON, 2012). The first sets a worker to process some stage and the data bounded for that. The second puts a worker to process a piece of data through all pipeline stages. The differences between these approaches are their applicability: the first approach is better for large stages and small data, while the second is preferred for smaller stages and larger data. The Figure 3.1 shows an abstraction of the pipeline pattern implemented by the Intel TBB (REINDERS, 2007) library. This is a modified version of the second approach commented above. In this case a worker receives one of the five input data items and carries it through as many stages as possible. When a worker finishes its task, it checks if there are more data for processing. The pipelines are commonly useful for serially dependent tasks which can be decoupled forming a fixed number of stages. Therefore the pipeline speedup scalability is limited by the slowest stage, for example: a pipeline with three balanced stages achieves a maximum speedup of three, since it is directly attached to the number of stages.

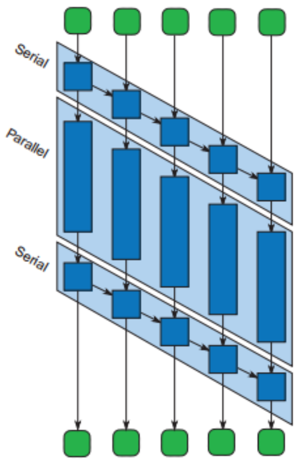


Figure 3.1 – A DAG model for the pipeline processing with five input items. Figure adopted from (MCCOOL; REINDERS; ROBISON, 2012).

The stencil (MCCOOL; REINDERS; ROBISON, 2012) pattern is an alternative approach to map data which proposes that an elemental function can access a set of data in an input collection, and not just a single element. This pattern intends to provide parallel access to an input collection where each worker pick up a set of fixed offsets for processing. Stencil is commonly used for image filtering, median filtering, motion estimation, simulations like fluid flow and lots of others applications. The recurrence technique addresses a complex case where loop iterations can depend one each other. This works like a map but can use the outputs from a iteration of adjacent elements as inputs for the next. Recurrence can be implemented with a sequence of stencils. The outputs of stencils (that is offsets of data processed) are iterated and can be reinterpreted as recurrences over space-time. Recurrences can be used by several applications like: image processing, sequence alignment, and also by performance analysis tools to infer behavioral patterns of a program execution (COOK; WOLF, 1998; CORNELISSEN; MOONEN, 2007).

3.2 Performance Analysis Tools

This section presents some of the most important trace-based performance analysis tools. The text have focus on the internal structure of such tools intending to explain how they achieve high performance on the processing of large amounts of data. This Section is structured as follows. The Subsection 3.2.1 presents Vampir, one of the most powerful tools existing today. The following Subsection 3.2.2 exposes Paraver, which is the most similar to this work. The Subsection 3.2.3 introduces Jumpshot that is a portable tool implemented in Java. The Subsection 3.2.4 explains the multi-format trace visualizer ViTE. The Subsection 3.2.5 presents PajeNG tool-set for trace replay built in C++ which is the main focus of this work.

3.2.1 Vampir

The Vampir toolset (NAGEL et al., 1996) is a commercial infrastructure for performance analysis of parallel programs built for MPI, OpenMP, Pthreads, CUDA, and OpenCL. Currently at version 8.3[1], the tool is available for Unix, Windows, and Apple platforms. It is composed by the Vampir GUI for interactive post-mortem visualization, the VampirServer for parallel analysis, the VampirTrace instrumentation and runtime tracing system, and the Open Trace Format(*OTF/OTF2*) as file format and access library.

Vampir's Trace Input Data

The Vampir performance visualization tool works based on execution traces. Thus it requires a working monitoring system to be attached in the parallel program. There are some ways to generate performance data compatible with Vampir: VampirTrace, Score-P and Event Tracing for Windows (ETW).

VampirTrace (JURENZ, 2006) is part of the Vampir toolset and supports event tracing for the programing languages C, C++, Fortran, and Java. It also allows the instrumentation of parallel programs that use MPI, OpenMP, POSIX Threads, CUDA or combinations of these libraries. The VampirTrace generates OTF trace files which are stored in the current working directory from the application execution. The tool provides several ways that can be combined for the instrumentation of an application, some of these are:

1. Fully-automatic compiler instrumentation, which have support in the most relevant compilers existing today, the GNU compiler collection, OpenUH compilers, and the commercial compilers from Intel, Pathscale, PGI, SUN, IBM, and NEC;
2. Source-code instrumentation by using VampirTrace API;
3. By binary rewriting of a ready executable either in a file or a memory image.

[1]Vampir's website: https://www.vampir.eu

The VampirTrace has great features and used to be the recommended monitoring facility for Vampir. However, according to the site of Vampir, the tool still available as open source software but no longer under active development. They recommend the Score-P[2] as code instrumentation and run-time measurement framework for the newest versions of the Vampir toolset. The OTF file format is now considered deprecated and replaced by OTF2, which is generated by the Score-P infrastructure.

Vampir's Performance analysis framework

The Vampir performance analysis (BRUNST et al., 2010) tool provides a framework which can quickly display the programs behavior at many levels of detail. The traces can be used to generate several graphic views. The tool allows profiling and event-based tracing. Profiling is the summarized run-time behavior of programs with accumulated performance measurements. Event-based analysis allows a more detailed comprehension of the program's execution behavior. This approach records timed events like function calls and message communication, in a combination of timestamps, event type, and event specific data. Tracing allows a more precise analysis and several different representative ways of seeing the date like time-line views and communication matrices.

The Figure 3.2 shows one of the most useful views supported by the Vampir. This is called master time-line view which contains detailed information about functions, communication, and synchronization events over the time. In the left side of the Figure 3.2 each row presents a single process and, in the right side, the events that occurred on it. The horizontal axis shows the selected execution time interval. The different colors represent a specific group of functions or synchronization events, for example `MPI_Send` belonging to the function group MPI.

The Figure 3.3 shows another view provided by Vampir. It is a derivation of the Figure 3.2 showing a single process time-line. This is divided in different levels of function call stacks. These levels show the initial function at the first level and the sub-functions in a level beneath and so forth. The functions that have returned to its caller are just represented by going back to the level above.

[2]Score-P available at: http://www.vi-hps.org/projects/score-p/

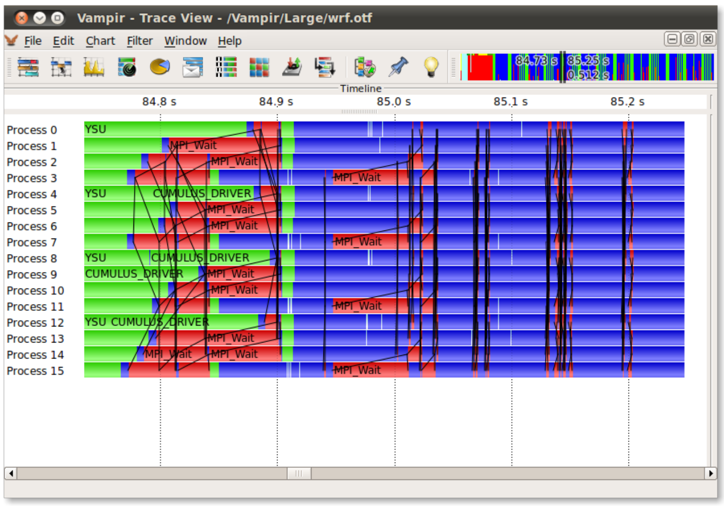


Figure 3.2 – Master time-line view taken from Vampir's site.

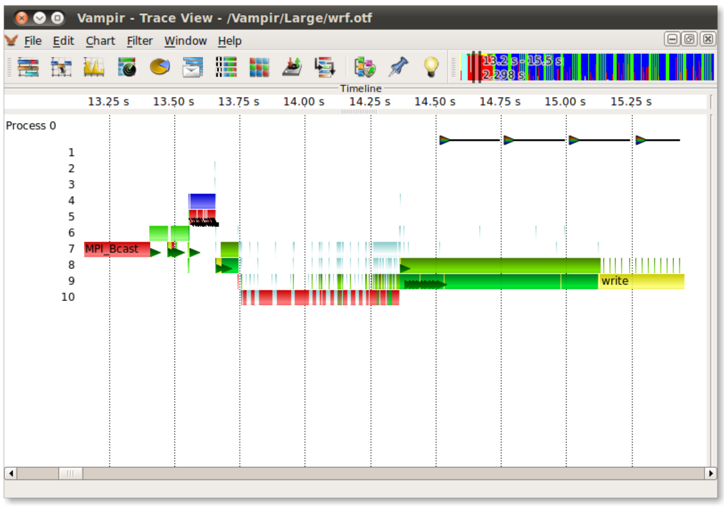


Figure 3.3 – Process time-line view adopted from Vampir's site.

Vampir's Performance Architecture

The Vampir toolset provides two ways for trace data processing, Figure 3.4. The fist is using the Vampir GUI in the client machine, as shown on the top of Figure 3.4. This way is commonly used to analyze small programs which generate light trace files. The second is the VampirServer (KNÜPFER et al., 2008), as shown on the bottom of Figure 3.4. This is a parallel and distributed high performance client-server framework that provide these features:

- The performance data are kept where it has been created, avoiding data transfer for processing.
- The parallel/distributed process in each node increase the scalability of the analysis processing.

- It works efficiently from every end-user platforms.
- Large performance data can be remotely browsed and visualized interactively.

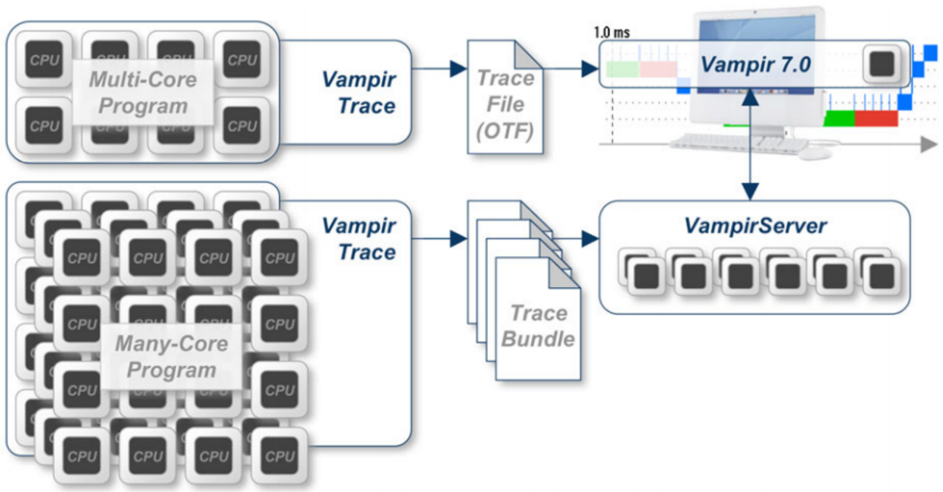


Figure 3.4 – Simple representation of the workflow executed by the Vampir. In the top, Vampir GUI for the client machine. In the bottom, VampirServer for remote distributed processing. Figure adopted from (BRUNST et al., 2010).

The VampirServer architecture is based on a master-slave relationship as shown in Figure 3.5. The MPI master process does the communication with the clients and handle the workers. The workers are a set of identical MPI processes, each one with a main thread that does the communication with the master process. The session threads are present in every MPI process of the architecture and are dynamically created according to the number of clients. The communication between local threads is made through shared buffers that are synchronized by mutexes and conditional variables.

The worker's session threads are subdivided in three modules: trace format, event data base and analysis. The trace format module is a parser for some trace formats. The event data base module stores objects divided in categories like functions, messages etc. Finally, the analysis module performs the event data provided by the data base module.

The master's session threads have different classification. These are subdivided as: analysis merger, endian conversion and client comm, like shown 3.5. The analysis merger combines the results received from several workers. The endian conversion converts the results to a platform independent format. Finally, the client comm layer does the communication with the clients.

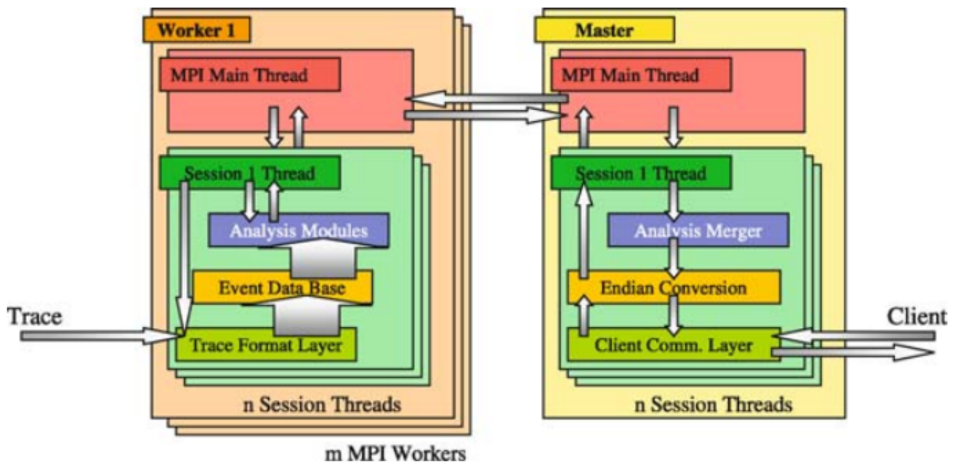


Figure 3.5 – VampirServer architecture with several MPI workers processes in the left, and the master MPI process in the right. Figure adopted from (KNÜPFER et al., 2008).

3.2.2 Paraver

Paraver (PILLET et al., 1995) is a post-mortem, trace-based tool for performance analysis of parallel applications. This intends to graphically display the behavior of a program's execution. Paraver provides different filters to select what is going to be showed and also offers some tools like timing and zooming, which improve the understanding of such programs behavior.

Paraver implements its own trace format without semantics associated. This means that the tool is very dynamic and easy to extend for new performance data and/or programming models. The current run-time measurement system of Paraver [3] is the Extrae (ALONSO et al., 2012) which generates Paraver traces for MPI, OpenMP, pthreads, OmpSs and CUDA.

Paraver Trace Format and Performance Data Generation

The Paraver trace file format (PARAVER, 2001a) is open source and can be summarized in three generic classifications:

- Events: defines a punctual event occurred in a given timestamp that has a pair of integer values to represent its activity. For example, entry and exit points of user functions.
- States: determines intervals of thread status. This type is heavily related with the parallel programming model. For example, the application can be waiting for a

[3]Paraver's website: http://www.bsc.es/computer-sciences/performance-tools/paraver/

message (either MPI or OpenMP), or for a memory transfer.

- Relation/Communication: establishes a relationship between two objects of the trace. This type is frequently used to describe records of the communication between: two processes (in MPI applications), task movement among threads (in OmpSs applications) or memory transfers (in CUDA/OpenCL applications).

The generation of performance data for the Paraver trace format can be done in several ways. The core instrumentation package is the Extrae. It is capable of instrumenting many parallel programming models. Extrae typically generates information like time-stamped events of run-time calls, performance counters and source code references. Furthermore, Extrae provides an API that allows the user to manually instrument the application. Other way to generate Paraver trace files is by using Dimemas (PILLET et al., 1995). This allows the simulation of a program's execution in an abstract machine defined by key factors for network and CPU. Dimemas generates a Paraver trace-file that describes the execution according to the simulation parameters. Therefore the user can compare these results with the real execution. Furthermore, since the trace format is open-source, there are several third-party translators that convert other formats to and from Paraver.

Paraver Analysis Framework

The Paraver analysis framework (LABARTA; GIMENEZ, 2006) provides three views generated from the traces. The first is called graphic view: much similar to the Vampir's master time-line, it consists in a representation of the program's execution behavior over the time. Such view allows the identification of problematic patterns occurred during the execution. The second is a textual view that provides more detailed information. The third is the analysis view: this allows the user to select quantitative data to be displayed.

The Figure 3.6 shows an example of the Paraver's time-line graphic view. This was a survey about the performance of a meteorological model using Paraver, NMMB/BSC-CTM model (MARKOMANOLIS; JORBA; BALDASANO, 2014), and presents one execution hour with 64 cores. The colors have specific semantics, for example: the blue one is computation, the yellow one is MPI message, the red one is MPI Wait and so on.

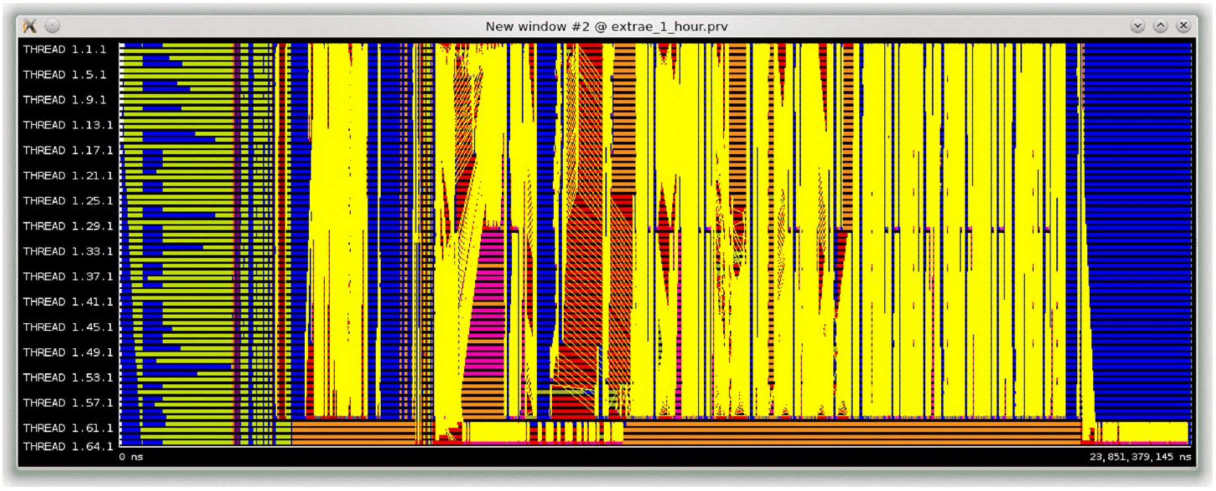


Figure 3.6 – Paraver Graphic view (Time-line) example. One hour of simulation from NMMB/BSC-CTM model. Figure adopted from (MARKOMANOLIS; JORBA; BALDASANO, 2014).

Paraver Performance Architecture

Paraver uses a different approach to gain performance by exploring an intelligent selection of the traced information. This is made by many techniques to reduce the trace file size, like detecting periodic structures, determine sets of events to be captured etc. The Paraver's processing model is divided in three modules: filter, semantic and representation. The filter module, in the top of the Figure 3.7, intends to minimize the trace data size through several selection and aggregation mechanisms. The semantic module, in the middle of the Figure 3.7, generates a semantic value for each represented object which is associated with information like event type/color, event name and so on. Finally, the representation module, in the bottom of the Figure 3.7, translates the traces into visual representations described above.

3.2.3 Jumpshot

Jumpshot (ZAKI et al., 1999) is a performance analysis tool for behavioral understanding of a program's execution. This provides several visualizations based on traces. The primary is a series of time-lines with colored bars for each process. This view can be zoomed and scrolled for examination of specific periods of time. Jumpshot also generates views like state duration histograms and aggregated states process per period of time.

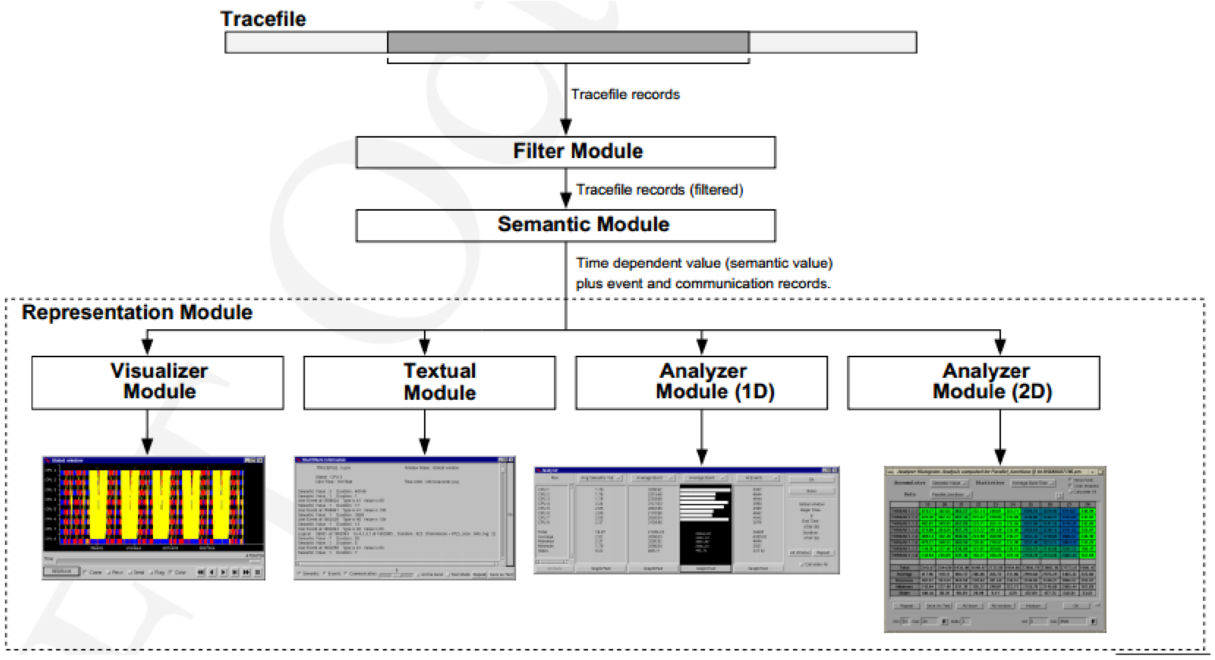


Figure 3.7 – Paraver internal structure. Figure adopted from (PARAVER, 2001b).

Jumpshot has high portability since it was built in Java and can run as an applet for browser. The currently version of Jumpshot, the Jumpshot 4 [4], covers analysis for MPI programs by using CLOG trace format provided by the MPE tracing library.

Jumpshot Analysis Framework and Trace Generation

The Jumpshot tool uses the MPE as tracing library. Such library generates traces in the CLOG trace file format, and Jumpshot needs a SLOG-2 (CHAN; GROPP; LUSK, 2008) log file format as input. So Jumpshot performs this conversion in a sequence of steps, as shown in the Figure 3.8. The first step, in the left, is the program tracing with MPE. The second, in the middle, is the conversion of CLOG traces to SLOG2 log files which stores objects of states, messages and events. Such objects will be further used to create visualizations.

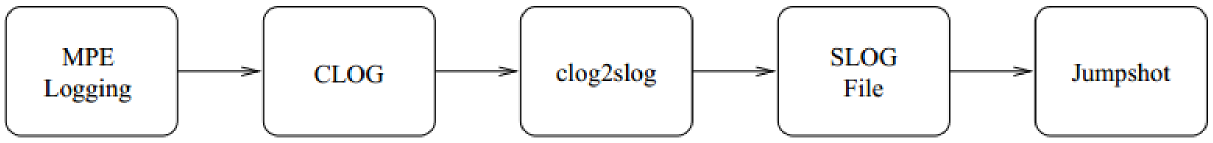
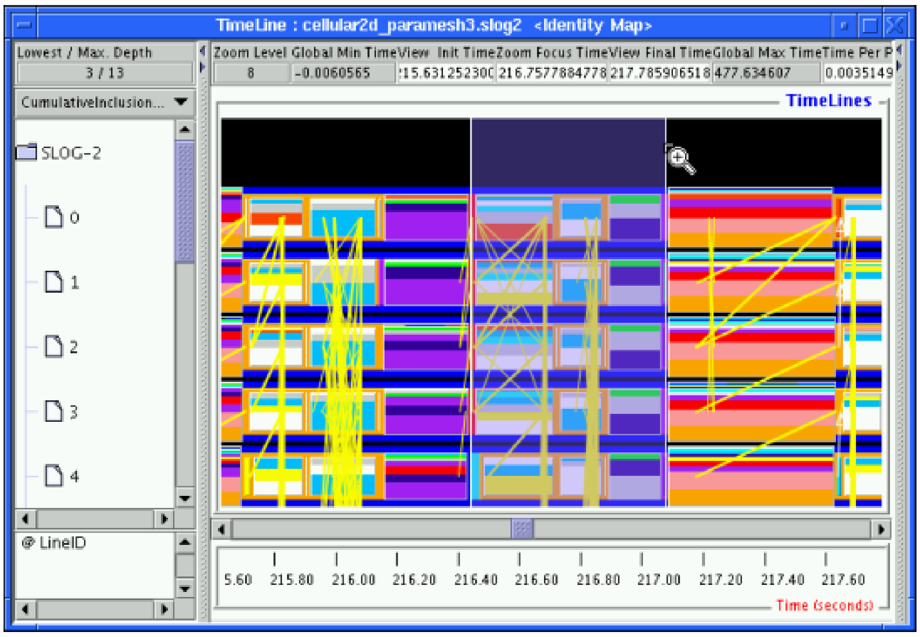


Figure 3.8 – Pipeline of the conversion of CLOG trace file to SLOG2 log file. Figure adopted from (CHAN; GROPP; LUSK, 2008).

[4]Jumpshot's website: http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/

The Jumpshot works like a canvas to display SLOG-2 objects in different forms. The Figure 3.9 shows an example of a Jumpshot time-line canvas. The left side of the Figure 3.9 presents the first 5 processes from a MPI application with a total of 16 processes. The visualization of each object can be manually added or removed by the user. The colors are different states and communication messages. The bottom of the Figure 3.9 displays the current time interval. This can be navigated forward or backward by the scroll bars.


TimeLine : cellular2d_paramesh3.slog2 <Identity Map>

Lowest / Max. Depth: 3 / 13

Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per P
8	-0.0060565	:15.63125230(	216.7577884778	217.785906518	477.634607	0.0035149

Figure 3.9 – Jumpshot preview of five processes time-line from states and messages. Figure adopted from (JUMPSHOT, 2007).

Jumpshot Performance Data Model

The performance of Jumpshot is very attached to the SLOG-2 format and data model. When the trace files are converted to SLOG-2, the events, messages and states become drawable objects organized in a hierarchical structure. This structure is a binary tree that sorts the objects according to the end timestamp of the events and considering the total time interval equals to the application run-time. For example: the root node represents the total time interval $[0, T]$ and the following are children representing time intervals $[t1, 1/2(t1+t2))$ and $[1/2(t1+t2), t2)$ where $t1+t2$ represents the time of its parent node. Thus all objects will find its position in the tree. When Jumpshot performs operations like scrolling forward or backward in time are, the program just needs to read the tree considering the searched end time and display the objects (the nodes) in such time interval.

3.2.4 ViTE

ViTE(Visual Trace Explorer [5]) (COULOMB et al., 2009) is an open-source multi-format trace visualizer for performance analysis. This tool do not have a specific tracing library, however the developers recommend EZTRACE (TRAHAY et al., 2011) or the GTG(Generic Trace Generator) [6]. EZTRACE generates traces in the Pajé format (SCHNORR, 2014a). GTG generates and converts traces in the Pajé and in the previously seen OTF file formats. ViTE intends to be the most compatible as possible with other analysis tools: supports the formats Pajé, OTF and TAU (SHENDE; MALONY, 2006) (although not fully tested yet with the TAU). ViTE provides a graphical interface that allows the user to browse the traces and rendering modules to generate SVG or PNG files depicting traces to export the results.

Traces Generation and Analysis with ViTE

ViTE intends to be a generic tool supporting different trace formats. Thus there are many ways to generate performance data to be processed with ViTE. The recommended libraries are EZTRACE and GTG. EZTRACE was designed to provide an automatic and simple way to trace parallel applications and generate the Pajé format. It provides predefined plugins allowing the tracing of applications that use MPI, OpenMP and Pthreads as well as hybrid implementations. Furthermore user-defined plugins can be developed. The GTG provides a generic way to manually generate traces in various formats such as Pajé and OTF.

The multi-format trace based performance analysis provided by ViTE intends to display graphically the behavior of parallel applications. To process these multiple input formats the tool implements a module architecture as depicted in the Figure 3.10. The set of modules in the bottom of this figures are in charge of parsing the traces and fill the generic data structure. Such structure stores abstract objects defined by features of the trace format. The last modules, in the top, uses the data structure to display the traces as requested by the user (COULOMB et al., 2012).

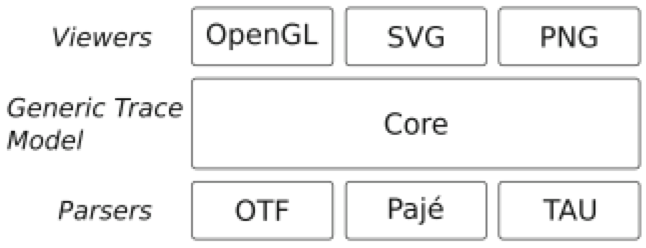
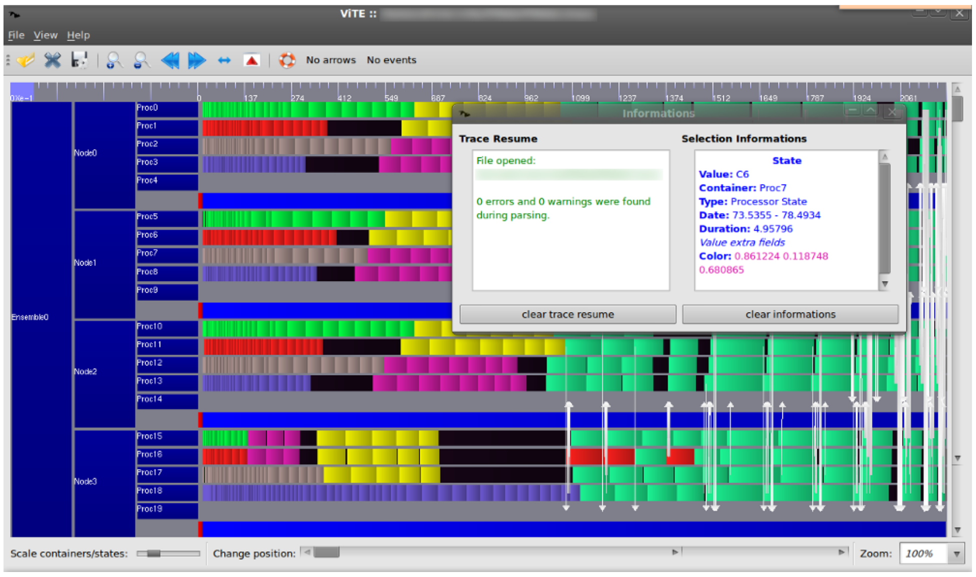


Figure 3.10 – Modules architecture in ViTE. Figure adopted from (COULOMB et al., 2012).

The Figure 3.11 shows an example of a time-line view generated by ViTE. The left side of the Figure 3.11 presents the containers hierarchy from the traced application. This hierarchy was inspired in the Pajé format in which the containers can be any monitored entity like processes, network links, threads etc. In the middle of the Figure 3.11 are the states and events occurred for each container over the time. This view also displays the links/messages between the containers which are represented by the white arrows. The popup demonstrates an user selection of a container, in this case called `proc7`. This action shows detailed information about the selected container.



Trace Resume

File opened:

0 errors and 0 warnings were found during parsing.

Selection Informations

State
Value: C6
Container: Proc7
Type: Processor State
Date: 73.5355 - 78.4934
Duration: 4.95796
Value extra fields
Color: 0.861224 0.118748 0.680865

Figure 3.11 – Example of a time-line view generated by ViTE. Figure taken from ViTE's site.

[5] ViTE's website: http://vite.gforge.inria.fr/index.php
[6] Available under the CeCILL-C license at: http://gtg.gforge.inria.fr/

ViTE Performance

The performance of ViTE is correlated with its efficient data structure and a fast rendering. The data structure is based on binary trees that are built over sorted lists of known size. This construction allows modifications that minimize the data stored, for example: instances of states are recorded just by state changes, thus avoiding store its start and duration. The binary tree structure is also important for rendering the traces, which summarizes portions of the data according to a resolution parameter. This resolution is a portion of time that eliminates items that are too small to be rendered. When applying the zoom-in operation, such resolution decreases and the same process is used. The traces rendering module was built with OpenGL. According to the authors (COULOMB et al., 2012), after benchmarking several rendering solutions, this appear to be the most scalable one.

3.2.5 PajeNG

PajeNG (SCHNORR, 2014b) is a tool-set for trace replay built in C++. The tool is capable to read files that follow the generic Pajé trace format (SCHNORR, 2014a). The tool-set provides four tools with different purposes: `pj_validate`, `pj_dump`, `pajeng`, `libpaje`. The `pj_validate` validates the integrity of the trace data. The `pj_dump` exports the processing results in a CSV file format. The `pajeng` is a visualization tool that generates views with the replayed traces. Finally the `libpaje` is a library to process traces following the Paje format. Furthermore `libpaje` is object oriented and can be easily modified or extended. As it follows, we describe the Paje trace file format, how performance analysis is conducted in the PajeNG framework, and how the tool works to simulate (replay) traces in a sequential manner.

Pajé Trace File Format Description

The Pajé trace file format is a self-defined, textual, and generic format to describe the behavior of the execution of parallel and/or distributed systems. The trace files are composed by two parts. The first part contains a definition for each event type with an unique identification number, like shown in Listing 3.2. The second part, depicted in Listing 3.3, contains one event per line. The first field identifies the event type and the others are separated by spaces or tabs. Those fields must follow the same order presented in the header.

Listing 3.2 – Example of event definition adopted from (SCHNORR, 2014a).

```
% EventDef SendMessage 21
% Time date
% ProcessID int
% Receiver int
% Size int
% EndEventDef

% EventDef UnblockProcess 17
% Time date
% ProcessId int
% LineNumber int
% FileName string
% EndEventDef
```

Listing 3.3 – Example of trace data adopted from (SCHNORR, 2014a).

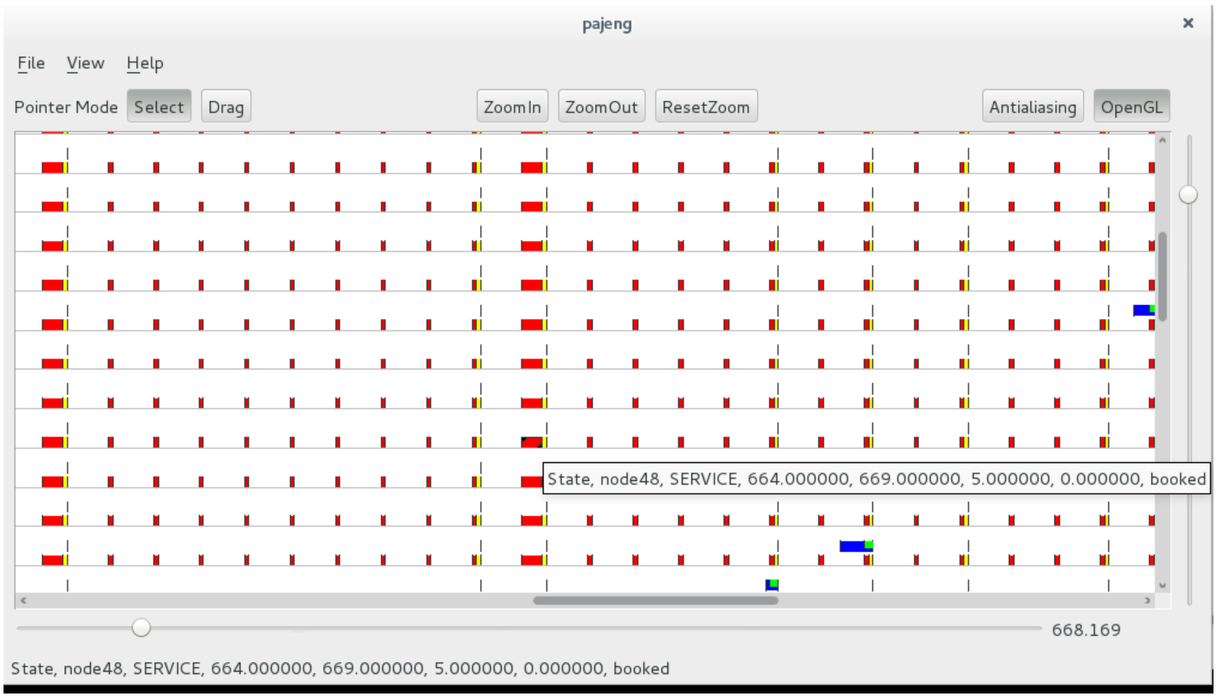
```
21 3.233222 5 3 320
17 5.123002 5 98 sync.c
```

The traced events in Pajé format are defined in five types of conceptual objects: containers, states, events, variables and links. The containers can be any monitored entity like a network link, a process, or a thread. It is the most generic object and the unique that can hold others, including other containers. States represent anything that has a beginning and ending time-stamp. For example: a given container starts blocked and then, after some time, changes to free. This period of time characterises a state. Events are anything remarkable enough to be uniquely identified and contain only one time-stamp. Variables have information about the evolution of the value of a variable along time. Finally, the links are the relation/messages between two containers having a beginning and ending timestamps.

There are many ways to generate traces in the Pajé format: it can be collected directly from the application execution, or converted from other formats. Akypuera is an independent project that provides the tracing library. This library provide a low memory footprint and low intrusion tracing and generates binary traces that are converted to Pajé format. Furthermore Akypuera provide tools to covert traces from several formats to Pajé, like Rastro, TAU, OTF2 and OTF.

Performance Analysis with PajeNG

The PajeNG's framework has two main tools for performance analysis. The `pajeng` is a visualization tool that can generate views such as time-lines, and `pj_dump` that exports the results in a CSV-like format tailored to conduct R analysis (IHAKA; GENTLEMAN, 1996). The Figure 3.12 presents an example of a time-line view generated by using the `pajeng` tool. Each line represents a container. The rectangles are different types of States described by the traces (see trace format description above) and represented by distinct colors. The user can select drawn objects to retrieve more detailed information about that. This case the user have selected a `State` object and received its detailed information in the selected point and in the bottom of the program as shown in Figure 3.12. The tool also provides space-time zoom operations by using the scroll bars in the bottom and in the right side.



State, node48, SERVICE, 664.000000, 669.000000, 5.000000, 0.000000, booked

State, node48, SERVICE, 664.000000, 669.000000, 5.000000, 0.000000, booked

Figure 3.12 – Example of time-line view from `pajeng`.

The Listing 3.4 shows an example of the CSV output text content generated by the `pj_dump` tool. Each line presents one simulated object with its respective features and values. For example, the fifth row of the Listing 3.4 describes a State for the container "node32", of the type "SERVICE", starting time "538 seconds", finish time "548 seconds", duration "10 seconds", imbrication level "0" and with the value "free". This content can be easily used to create several statistics and plots by using R[7] software

[7]R Project's website: https://www.r-project.org/

environment for statistical computing and graphics.

Listing 3.4 – Example of `pj_dump` output data.

```
Container, 0, 0, 0, 1205, 1205, 0
Container, 0, L1, 0, 1205, 1205, node32
State, node32, PM, 1149.000000, 1205.000000, 56.000000, 0.000000, normal
State, node32, SERVICE, 537.000000, 538.000000, 1.000000, 0.000000, reconfigure
State, node32, SERVICE, 538.000000, 548.000000, 10.000000, 0.000000,free
State, node32, SERVICE, 548.000000, 553.000000, 5.000000, 0.000000, booked
Variable, node32, power, 0.000000, 1205.000000, 1205.000000, 1000000000.000000
```

PajeNG's Work-Flow

The `libpaje` does the needed processing for the traces replay. This works with a single trace file each time and does the processing in a sequential way. The `libpaje` follows a work-flow composed by three main components, as depicted in the Figure 3.13 proceeding from the left to the right. The component `PajeFileReader` starts the processing by the trace's reading. This catches chunks of fixed size from the trace file and send them to the second component. The `PajeEventDecoder` receives such chunks, splits them into lines, and transforms each line in a generic event-object according to its definition (see Pajé's trace format description above). The last component, `PajeSimulator`, implements different functions for each event type to stack up each event-object into the correct stack of its respective container.

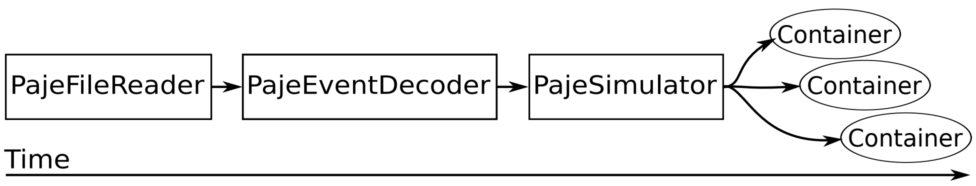


Figure 3.13 – Original libpaje's sequential work-flow.

The Paje structure is strongly connected, that is, the `PajeDecoder` must wait for data provided by the `PajeFileReader`, and the `PajeSimulator` waits for objects coming from the `PajeEventDecoder`. This way, while a chunk of data is being processed, the `PajeFileReader` is stopped. Thus this model has performance problems since the processing time is linearly proportional to the size of the input trace file. The Figure 3.14 shows an example of this problem, where the time proportionally increases to the input size. The `X` axis presents the input trace file size, and the `Y` axis, the minimum run-time collected from 30 executions (these executions were performed in the machine `turing` which is described in Chapter 5). Each line represent a different disk type, HDD or SSD, according to the caption. The results were very similar for both disks, however this was probably due to the `turing`'s SSD that is not much faster than the HDD. Therefore these results demonstrate that a sequential approach is unable to handle high performance parallel applications which produce thousands of Gbytes of trace data when executed. Our work builds on this by attempting to solve this issue with parallelization and a change in the Paje file format. Our solution is described in next chapter.

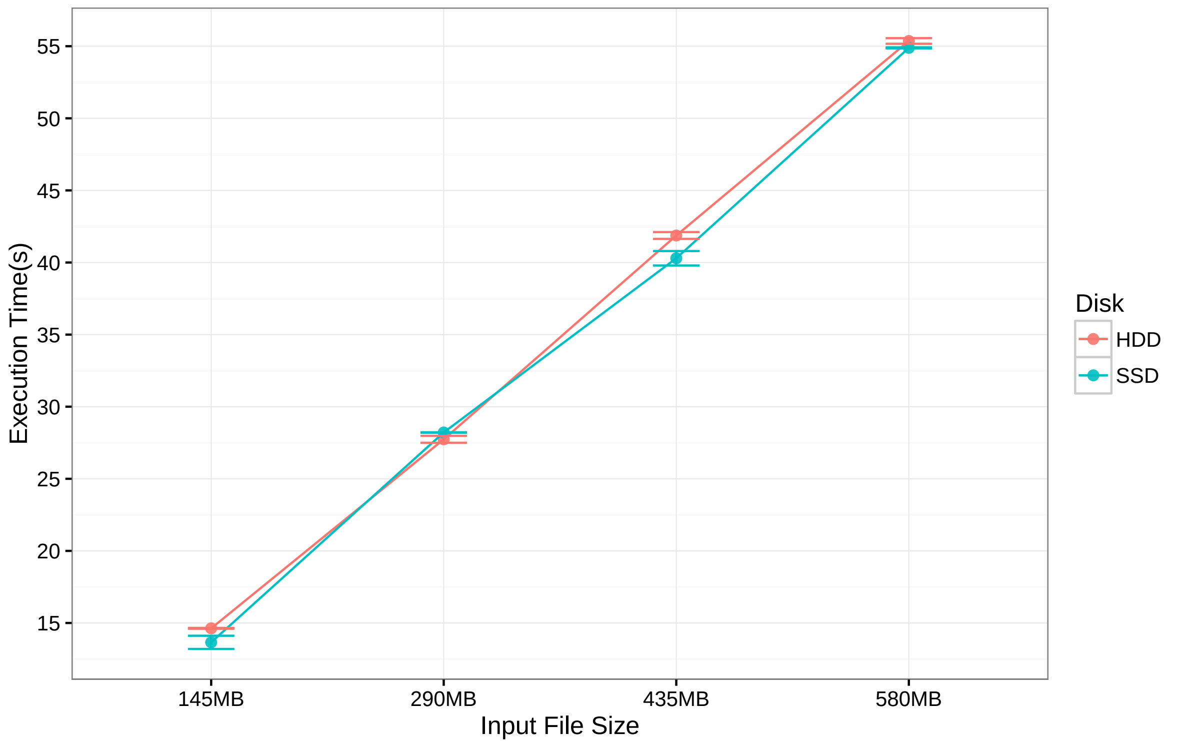


Figure 3.14 – PajeNG's run-time ranging the input size and disk type, HDD vs SSD. This results were collected from the minimum value of 30 executions performed in the machine `turing` (see platform description in Chapter 5).

3.3 Summary and Categorization

The classification focus on the performance structures provided by each tool. Four features were considered to the classification, as shown in Table 3.1. The first is computation, which can be classified in three types: parallel, distributed, or sequential. The second is classified just by yes or no and considers the capability of handle multiple trace files as input. The third determines if the tool automatically does some aggregation/reduction operations to improve the traces processing. This is classified as summary, filter, or nothing which means no reductions in the traces. The last one determines if the tool can be extended to support new programing models, classified as yes, no, or limited.

Table 3.1 – Classification for performance analysis tools.

Tool	**Computation**	**Multiple Traces**	**Trace Reduction**	**Extensible**
Vampir	Parallel/Distr.	Yes	Filter	No
Paraver	Sequential	No	Summary/Filter	Yes
Jumpshot	Sequential	No	Filter	Limited
ViTE	Sequential	Yes	Summary	Yes
PajeNG	Sequential	No		Yes

Vampir 3.2.1 provides high performance traces processing by using a parallel and distributed architecture. Vampir is a commercial tool and can not be modified by third parties. Thus new programming models may not be supported by the tool.

Jumpshot 3.2.3 differs from the others by its great portability since it is implemented with Java. The tool provides an infrastructure that achieves high performance based on its data model attached by the SLOG-2 format. Jumpshot also improves its performance by applying filters to reduce the trace files size, which may lose important data. The tool can be extended, however it is a very complicated task: it uses a trace format that is semantically attached to the collected events. The SLOG-2 trace file format used by Jumpshot may not support new programming models, and so its extensibility is limited.

Paraver 3.2.2 and PajeNG 3.2.5 are tools with a great feature due to their trace file formats. Both tools provide a generic format without associated semantics. Paraver improves its performance by executing several summarization and filtering methods to reduce the trace size. On the other hand, the PajeNG reads the entire trace sequentially, thus losing performance to handle large trace files. The extensible classification for both tools is a great feature, since they are open-source and use generic trace formats, they are more capable to cover new programing models than the other tools.

ViTE 3.2.4 addresses a different idea for performance analysis providing an open-source visualization environment for multi-format traces. The tool supports several trace file formats like Pajé, OTF and TAU, and thus it has a great compatibility with other tools. To achieve high performance ViTE uses an efficient data structure and summarizes parts of the traces. ViTE can be extended and cover many new programming models, since it processes several trace formats, also including the generic format Pajé.

The surveyed tools lead us to further explore the PajeNG due to its great features and capability to handle many types of parallel applications. However its performance degrades the usefulness of the tool, since it takes much time to generate the analysis data. Therewith, in the following Chapter 4 is presented our parallel processing model for PajeNG (`libpaje`). The chapter also includes the description of modifications made in the original Pajé trace format. This model is the main contribution of this work, it was developed and tested showing promising results as is presented in Chapter 5.

4 PARALLEL TRACE REPLAY IN PAJENG

The PajeNG's parallelization strategy proposed by this work aims to maximize the amount of data read for processing. This strategy comes up from our previous attempts to parallelize the PajeNG also presented here. The successful strategy comprises two main changes: one in the input data, and another in the tool's processing behavior. The Pajé trace format was modified to allow the splitting of a trace in multiple files, instead of just a single one with all data. The `libpaje`'s work-flow was adapted to use threads to carry out a parallel processing of the multiple trace files.

This chapter is divided in the following sections. Section 4.1 presents our previous preliminary attempts to improve the PajeNG's performance and how it has guided us to the final parallelization strategy. Section 4.2 describes the modifications made in the Pajé trace file format necessary for our parallelization strategy. Finally, Section 4.3 describes the design and implementation of the parallel PajeNG simulator.

4.1 Preliminary Attempt to Parallelize the PajeNG Simulator

The Figure 4.1 shows the modified work-flow representing the first attempt to improve PajeNG. The modifications have focused on the communication between the PajeSimulator and the containers. This was made by separating the reading of the file from the simulation. In the normal work-flow, the simulator must wait for the processing made by the containers considering each event to move one. This strategy implements a queue of events (tasks) and two threads to process it. This way, the simulator can queue the trace events, while a thread manages the containers processing. We have designed this strategy to verify the performance of two parts of the sequential work-flow: the reading of the trace file and the processing made by the containers.

The Figure 4.2 shows the results obtained with the first parallelization strategy in comparison with the sequential version. The tests represent thirty executions of three trace files, ondes3D with 234MB, scorep-lu with 741MB and scorep-cg with 2006MB. It was made on a machine with four cores, running at 3.1GHz and with 20GB of memory. The X axis represents the size of the trace files and indicates the type of the execution, sequential or parallel, and the Y axis the respective execution time in

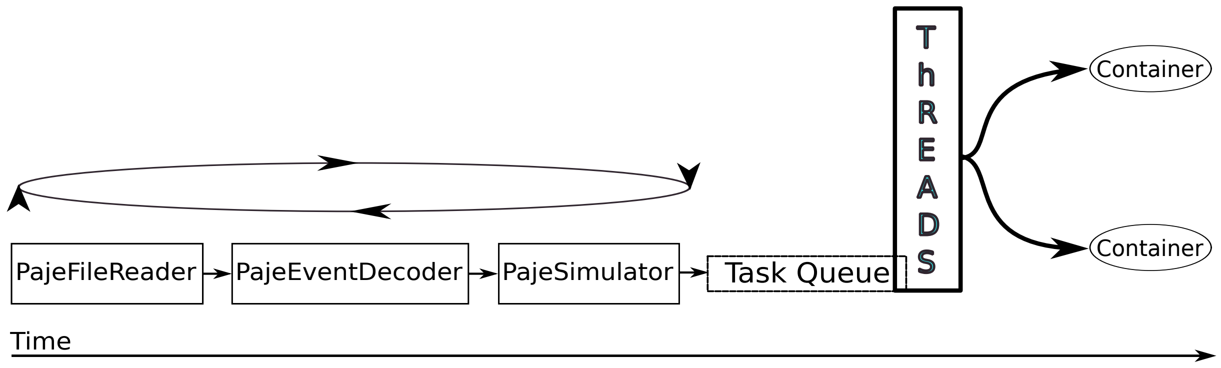


Figure 4.1 – Representation of PajeNG's parallel work-flow with the strategy that has aiming to quickly release the simulator.

seconds. Although not very easily to seen, each rectangle represents the standard deviation and the center lines represent the average time. The experiment shows that we obtain a very little variation in the executions.

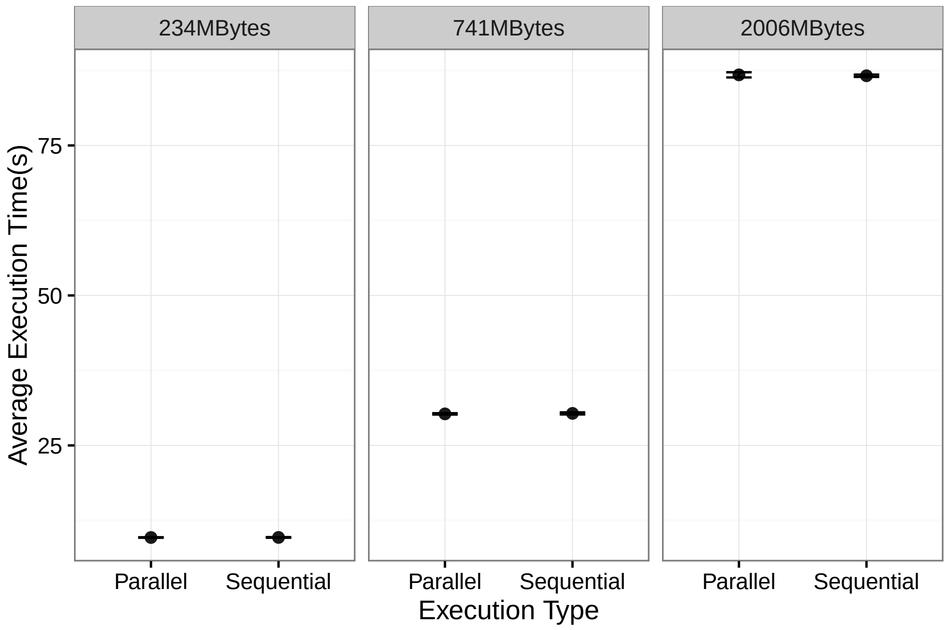


Figure 4.2 – Results obtained with one of the firsts parallelization strategies for the PajeNG.

The execution time of this strategy has remained roughly the same as the sequential. For example, for the file *scorep-lu* with 741MB, the average execution time and standard deviation were respectively, 30.33 and 0.31 seconds for the sequential and 30.25, 0.20 seconds for the parallel. These can not even be considered as performance gain since they are much closer to each other. Based on this results we have noted that the PajeNG's greatest performance bottleneck is not the simulation process but the reading of the trace file. This conclusion has took into account the fact that if the simulation was the bottleneck, when at least two threads are doing its processing we must gain some performance.

We have tried other similar parallelization strategies based on queues and tasks, but none of them lead to acceleration. The most successful solution, designed and implemented, is the one described in the following sections. They radically change the Paje file format providing a parallel reading of the trace file and the event simulation.

4.2 Pajé Trace Format Support for Parallel Reading

The changes made on the `libpaje` must use multiple trace files as input to enable the parallel processing. Therefore the modifications on the Pajé trace format consist in a way to split the old unique trace file in several pieces. Furthermore we have to create a method to link such pieces each other.

The multiple trace files were generated by breaking a single one for each container event found. This approach was chosen due to internal dependencies of the events associated to a container. Thus, this way several trace files are generated, each with full information about one container, and all together being equivalent to the previous single file information. To link the all pieces we have created a new event type called `PajeTraceFile`. This event contains three information: its type, a `Container` and a path/name of one file where the events are registered. Furthermore we have built a descriptor trace file that works as a job producer. Like shown in the Listing 4.1, this file just contains the creation of `Containers` and `PajeTraceFile` events. This case, lines identified by 6 indicates a container creation and lines marked as 26 specify where is the file with the events of some container. This was the better way that we have found to allow the traces processing in parallel. The next Subsection 4.3 will better explain the importance of such modifications for our parallelization strategy and how works such descriptor "job producer" trace file.

Listing 4.1 – Example of the contents from a descriptor - `job producer` - trace file.

```
% EventDef PajeCreateContainer 6
% Time date
% Alias string
% Type string
% Container string
% Name string
% EndEventDef
% EventDef PajeTraceFile 26
% Container string
% Type string
% Filename string
% EndEventDef
6 0.000000000 rank0 PROCESS root "rank0"
26 rank0 PROCESS "traces/containerrank0.trace"
6 0.000000000 rank1 PROCESS root "rank1"
26 rank1 PROCESS "traces/containerrank1.trace"
...
```

4.3 Parallel PajeNG's Processing Work-Flow Description

The Paje Simulator has been modified in two ways: first, to understand the new event type, PajeTraceFile; and second, to use C++11 threads for a parallel processing model. This events are processed by a new component called PajeParallelProcessor. This component implements a queue of tasks and a set of workers. The tasks queue is filled with PajeTraceFile events and the workers are a set of C++11 threads whose size is user defined.

The modifications described above have allowed the construction of a new work-flow for the PajeNG (`libpaje`). The Figure 4.3 illustrates how such work-flow behaves. The main thread is presented in the top of the Figure 4.3. This one follows a similar behavior to the sequential approach: starting by the PajeFileReader (pFR), sending data to the PajeEventDecoder (pED) and finally, reaching the PajeSimulator (pS), like described in the related work Chapter 3, Subsection 3.2.5. The changes start when a PTF event reaches the pS component, like shown in Figure 4.3. When such thing happens, the pS inserts the PTF event in the tasks queue of the PajeParallelProcessor (pPP) component. In our strategy, the main thread will work like a producer by performing the descriptor trace file 4.2. This way such thread fills the pPP's tasks queue with all PTF events in the descriptor. Furthermore the pPP launches its set of workers starting the parallel processing.

The following steps of the parallel work-flow are the workers that consume the pPP's tasks queue. These are illustrated right below of the main thread in the Figure 4.3. Each worker receives a PTF event from the tasks queue until the queue becomes empty. Thus, according to the event information, Container (PC) and file path/-name, the worker starts a new flow in a separated thread. These separated flows work

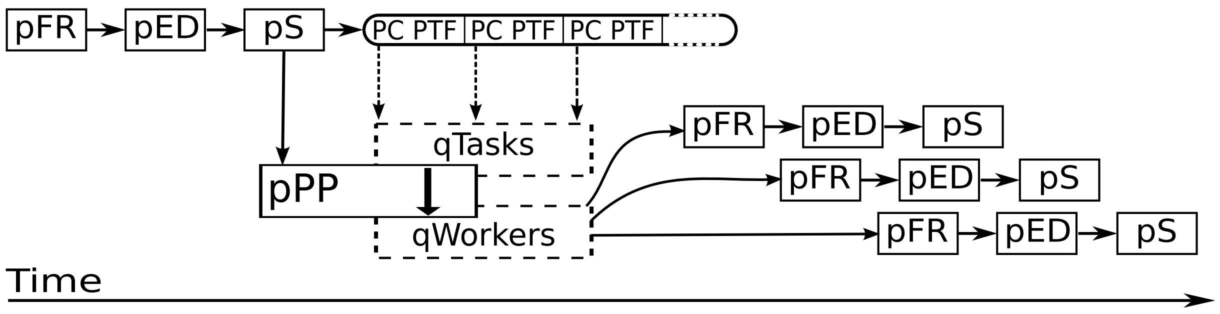


Figure 4.3 – libpaje's parallel work-flow.

as the sequential, however each one processing a piece of the global trace file. The developed model was strongly inspired by the pipeline pattern/technique described in Chapter 3, Subsection 3.1.2. This was the technique that better met our requirements. The next Chapter 5 presents a detailed performance analysis to our strategy intending to prove the performance gains.

5 PERFORMANCE ANALYSIS AND RESULTS

This chapter presents an analysis of the performance achieved by the developed parallelization strategy for the PajeNG's tool-set. Section 5.1 presents the experimental methodology and the two platforms that were used to conduct the experiments. Section 5.2 proposes an overview of the performance obtained with speedup and efficiency graphs. In Sections 5.3 and 5.4, we investigate the reason for the performance drop from a certain level of parallelism. The latter led us to correlate the application behavior with architectural details suggesting changes to our strategy. Finally, the Section 5.5 considers disk measurements, comparing the application performance when the input data are on HDD versus SSD.

5.1 Experimental Platform and Methodology

The settings of the two experimental platforms that were used for the analysis are summarized in Table 5.1. The first is a Dell PowerEdge R910 machine identified as **turing** with 32 physical cores arranged in four Intel(R) Xeon(R) CPU X7550 of 2.00GHz, each with eight cores. The nodes have non-uniform memory access (NUMA), and each processor is in a NUMA node. Cores are identified across NUMA nodes using an offset of 4 by 4. The main memory of the system is 128GBytes. The solid disc is a 64 GBytes Corsair Force 3, while the hard drive is a Hitachi HUC151414CSS60 and Western Digital WD10TPVT-00U configured as RAID 1. The second platform is an SGI C1104G-RP5 machine identified as **bali** with 16 physical cores arranged in two processors Intel(R) Xeon(R) E5-2650 2GHz, each with eight cores. This machine has two NUMA nodes. Cores are identified across NUMA nodes using an offset of 8 by 8. The main memory of the system is 64GBytes. The solid drive is a Samsung SSD 840 500GBytes while the hard drives are two Seagate ST91000640NS 1 TBytes each.

Each experimental combination was executed at least 30 times in order to decrease the possibility that measurement errors disturb the interpretation. The preliminary analysis of each test battery have identified that most of the observations follows a Gaussian distribution indicating normality. The observations that do not fit perfectly into a normal distribution are those where the execution overload and the performance

Table 5.1 – Experimental platform settings.

	turing	**bali**
Brand	Dell PowerEdge R910	SGI C1104G-RP5
Processor	Intel X7550 2GHz	Intel E5-2650 2GHz
Turbo Boost Technology	v1.0	v2.0
Max Turbo Frequency	2.4GHz	2.8GHz
Cache L3 Size	18MB	20MB
Bus Speed	6.4 GT/s QPI	8 GT/s QPI
NUMA Nodes	4	2
Cores/Node	8	8
Total Cores	32	16
NUMA Nodes Config	0: 0, 4, 8, 12, 16, 20, 24, 28 1: 1, 5, 9, 13, 17, 21, 25, 29 2: 2, 6, 10, 14, 18, 22, 26, 30 3: 3, 7, 11, 15, 19, 23, 27, 31	0: 0, 1, 2, 3, 4, 5, 6, 7 1: 8, 9, 10, 11, 12, 13, 14, 15
DMA Disk Controller	Node 0	Node 0
Memory	DDR3	DDR3
Memory Size	128 GBytes	64 GBytes
Memory Speed	1066MHz	1600MHz
Hard drive	RAID Dell PERC H700 4TBytes	2 x Seagate ST91000640NS 2 x 1TBytes
Solid drive	Corsair Forse 3 64GBytes	Samsung SSD 840 500GBytes

bottleneck appear, which this chapter intends to present. The input data for the Sections 5.2, 5.3, 5.4 was a trace file of about 10GBytes divided in 64 pieces of 150MBytes each. For the disk measurements Section 5.5, the input data was a trace file of about 2GBytes split in 64 pieces of 32MBytes each. These differences of input size do not influence the analysis results, this was made just to generate the results more quickly, since the disk experiments need longer periods of time to execute. The experimental methodology is tailored to consider the following factors:

Locality The threads locality, also known as affinity, has two levels of measurement: with free-running threads (*free*), where the system scheduler can migrate these threads to optimize the run-time and memory access, or pinned threads (*pin*), where we define a specific mapping in the beginning of the execution that does not change. In this second case, a linear mapping was done according to the numbering of the cores provided by the operational system.

Quantity The amount of threads was evaluated using five levels comprising: a sequential implementation with only one execution flow, which was used as reference for the speedup calculation, and then a parallel version with two, four, eight and sixteen threads.

Disc Type This factor compares the solid drives and hard drives of the machines. The main goal is to compare the reading performance depending on the disk technology of each platform.

5.2 Scalability Overview

This section presents an overview about the speedup and efficiency achieved by the parallelization strategy of the PajeNG. The used measurements for such overview assumes that the trace data is in memory. This is not the most realistic approach since generally the data will be on the disk, however with that we can explicit the limits of our strategy. The overview is divided in two subsections: the Subsection 5.2.1 with strong scalability 2.1 analysis and Subsection 5.2.2 with weak scalability 2.2 results.

5.2.1 Strong Scalability Overview

This subsection considers strong scalability measurements, that is the run-time versus number of execution flows with a fixed input about 10 GBytes. Besides that, for these experiments we do not have disabled the Linux operation system's cache and the automatic thread scheduling. This means that the files are automatically buffered in memory and the threads are free to migrate among cores according to the operating system rules. So these results consider run-times for *free* threads and with the input data stored in memory independently of the machine's disk type. Figure 5.1 presents the speedup achieved by our strategy on each machine, bali and turing (see platform description in Section 5.1). The X axis presents the number of threads while the Y axis shows the respective speedup. These results come from the minimum value of each experiment. The minimum was chosen because frequently worst results are caused by some intrusion and so the average can not be a good measurement. The grey line illustrates what should be the ideal speedup and the other ones the results for each machine. In the Figure 5.1 we can see that both machines are still speeding up until four threads and then there is a performance drop. The machine bali continues to gain performance until sixteen threads but far away from the ideal speedup. On the other hand, the machine turing drastically slows down with eight and sixteen threads, getting closer to the sequential results. Such observations will be further discussed in the Section 5.3 and 5.4, with more detailed data that can better explain these results.

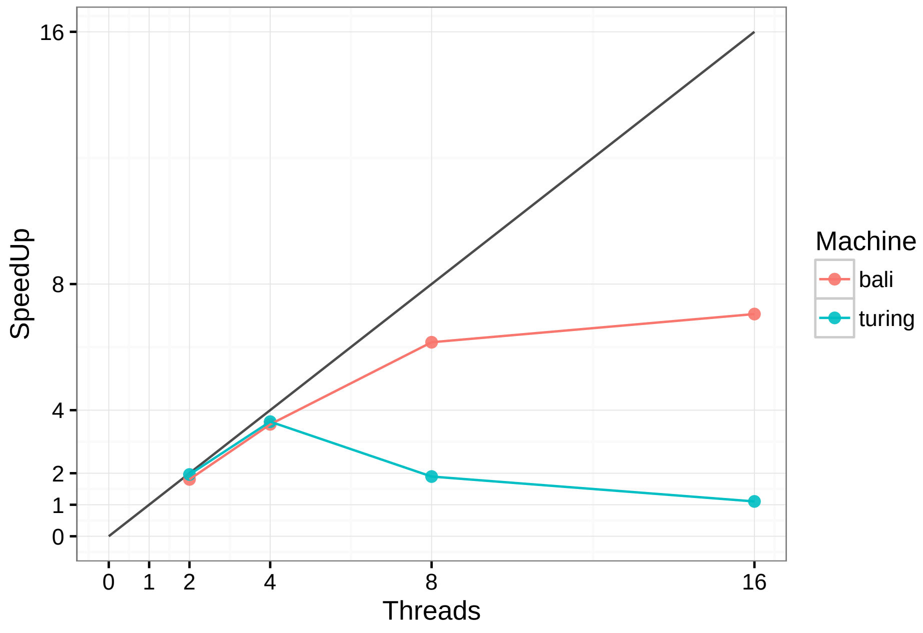


Figure 5.1 – Strong scaling speedup graph for bali and turing machines. Thirty executions for each configuration, all without drop the operational system data cache.

The Figure 5.2 presents the efficiency results of our strategy. Like the speedup (see Figure 5.1), the plotted results consider the minimum value from thirty executions for each configuration. The *X* axis presents the number of threads, while the *Y* axis the respective efficiency. The grey line demonstrates what should be the ideal efficiency and the other ones the results for each machine. Efficiency (see Figure 5.2) shows the same pattern observed for the speedup depicted in Figure 5.1, but in a different manner. It achieves a relatively good efficiency until four threads and then there is an efficiency drop. However this graph 5.2 exposes an interesting behavior for the machines: until four threads the machine turing has achieved a better efficiency than bali, but when the parallelism is increased, turing loses much more performance. Thereby the state changes and bali ends achieving much better efficiency.

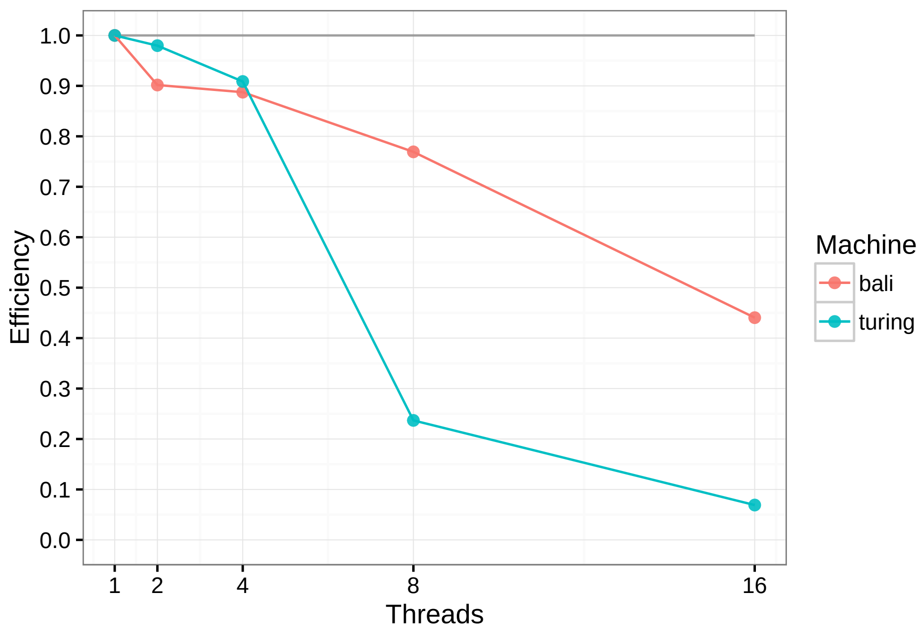


Figure 5.2 – Strong scaling efficiency graph for bali and turing machines. Thirty executions for each configuration, all without drop the operational system data cache.

5.2.2 Weak Scalability Overview

This subsection considers weak scalability experiments, that means increase the problem size proportionally to the number of parallel execution flows. Like the strong scalability 5.2.1 measurements, for this overview we do not disable operation system's disk cache and the automatic thread's scheduling. The Figure 5.3 shows the capability of increase the input size(*sizeup*) achieved by our strategy on each machine, bali and turing. The X axis presents the number of threads while the Y axis shows the respective *sizeup* capability. These results also came from the minimum value of each experiment, all with thirty executions for each configuration. The input size of each experimental setting is displayed over the ideal *sizeup* grey line, starting from 150MBytes for the sequential until 2400MBytes for 16 threads. In the Figure 5.3 we can note that the weak scalability experiments for bali have followed a similar pattern of the strong speedup results as seen in Subsection 5.2.1, scaling up the input size until eight times and then losing performance. On the other hand, the weak measurements for turing have a different behavior, allowing a *sizeup* about eight times. Other interesting observation is that for all experiments, both strong and weak scalability, the machine turing has achieved better results until four threads, but when the parallelism increases, the turing's performance drastically decreases.

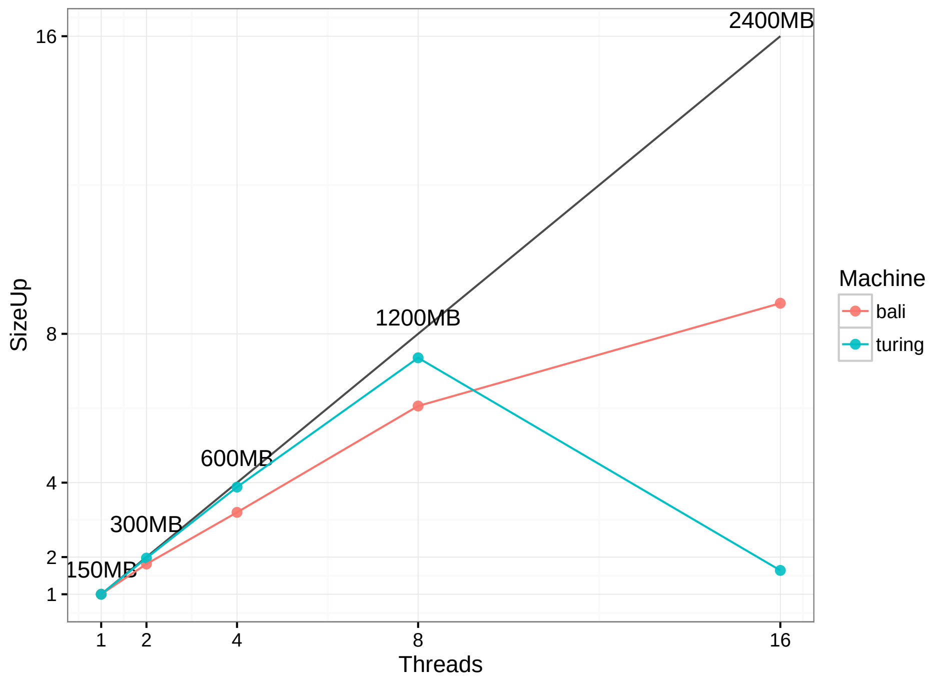


Figure 5.3 – Weak scaling *sizeup* graph for bali and turing machines. Thirty executions for each configuration, all without drop the operational system data cache.

The Figure 5.4 presents the efficiency graph for weak scalability results from both machines bali and turing. The X axis presents the number of threads while the Y axis shows the respective efficiency. These results are also considering the minimum time for each configuration. The input size of each experimental setting is displayed on the top of the Figure 5.4, over the grey line that illustrates the ideal efficiency.

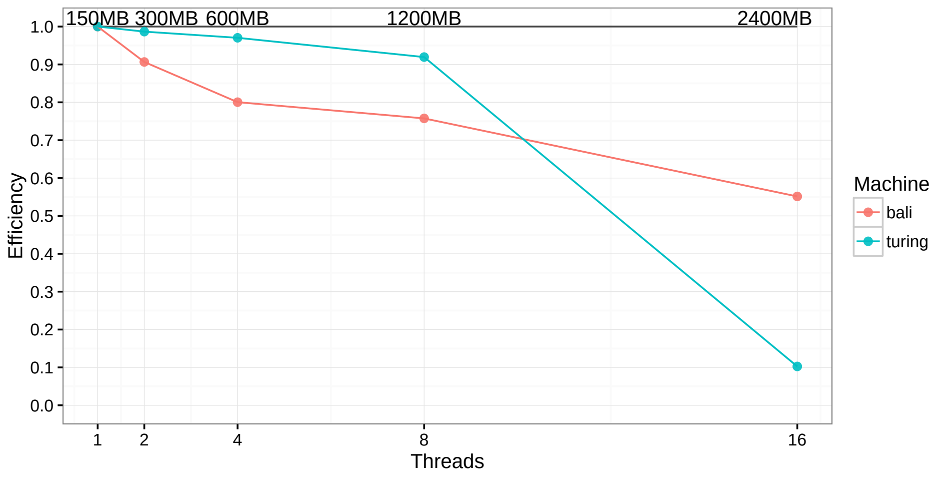


Figure 5.4 – Weak scaling efficiency graph for bali and turing machines. Thirty executions for each configuration, all without drop the operational system data cache.

5.3 Threads Locality Interference

This section deepens the performance analysis and intends to explain why we have obtained the results presented in the overview Section 5.2. For this we have formulated the hypothesis that since the tasks are memory bounded, its migration could be a point of performance loss. To prove such hypothesis we have performed experiments by fixing (*Pin*) each thread on a core, thus preventing the operating system to migrate (*Free*) them. The results for this section still consider that the operational system's data cache is enabled, so the input trace files are kept in memory. Furthermore, for these experiments, the pinning process of the threads was made sequentially, thus ignoring the architectural structure of the machines (see Section 5.1 and Table 5.1 in line NUMA Nodes Config). In other words, each thread was fixed in the first available core identifier, for example: thread 1 in core 1, thread 2 in core 2 and so on.

The Figure 5.5 presents the speedup obtained by the experiments with each thread fixed (*Pin*) on a core, in contrast to the results for *Free* threads. The Figure 5.5 is divided in two plots: in the left are the results for the machine bali and in the right for the turing. The X axis shows the number of threads and the Y axis the respective speedup. The grey lines for both plots are illustrating the ideal speedup, and the other lines, the speedup achieved for each setting: *Free* and pinned (*Pin*) threads. The *Pin* threads version have achieved better speedup results than the *Free* ones. For example: with the *Pin* version for the machine turing we were able to reach a nearly linear speedup until eight threads, while with the *Free* just four. In turn, the machine bali does not present much differences between the versions, just with a little higher speedup for the *Pin* version with sixteen threads. Thereby, these results reinforce our hypothesis that the threads migration are causing performance loss. However the machine turing still presenting huge performance loss after eight threads for both settings, *Pin* and *Free*. Thus, from now on we will focus on turing since it is the bigger machine and have more interesting results. The next Figure 5.6 shows more detailed measurements comparing the data read's time duration between each turing's NUMA node.

The results obtained from turing with *Pin* threads experiments led us to formulate a new hypothesis based on the turing's NUMA nodes organization (see Section 5.1 and Table 5.1 in the line NUMA Nodes Config). The NUMA node 0 is the only one directly connected to the disk IO interface. The remaining NUMA nodes (1, 2, and3)

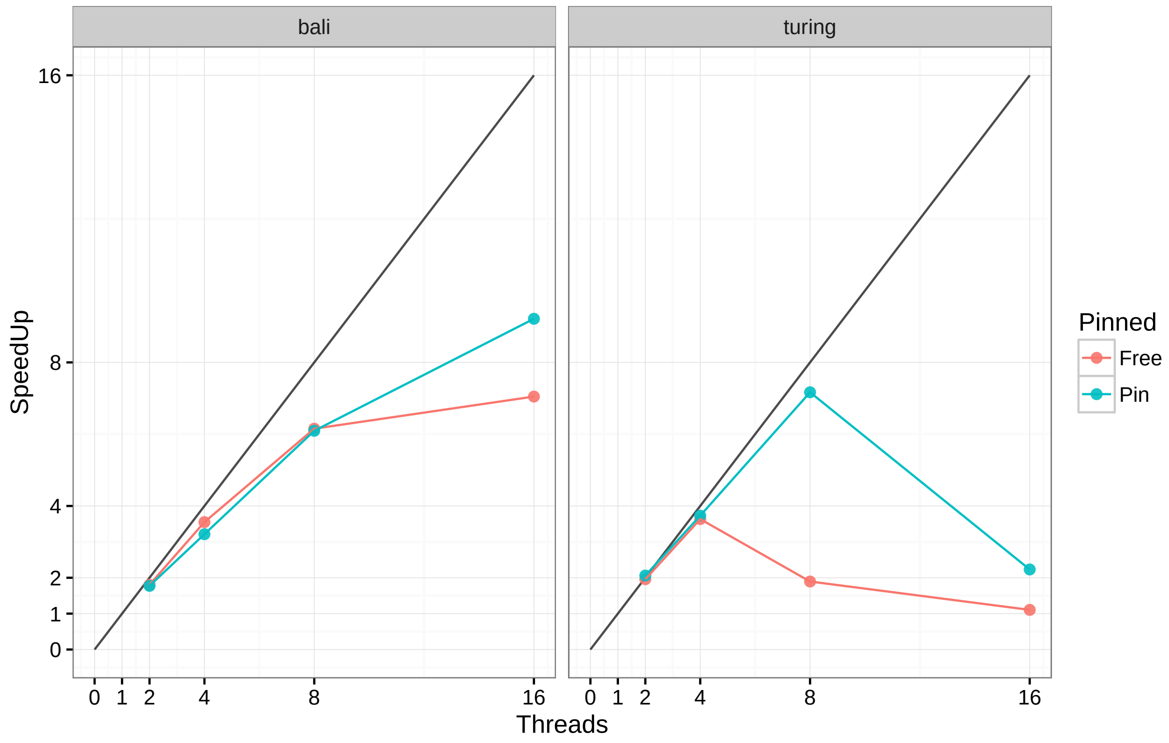


Figure 5.5 – Strong scaling speedup Pin vs Free Threads graph for bali and turing machines. Thirty executions for each configuration, all without drop the operational system data cache.

may require more time to access data, thus causing performance loss. Furthermore, we have pinned the threads sequentially and the cores's identifiers/NUMA nodes are organized in a different way. For example: core 0 is on NUMA 0, core 1 on NUMA 1 and core 2 on NUMA 2, core 3 on NUMA 3, core 4 comes back to NUMA 0 and so on until the counting reaches 16 cores, 8 physical and 8 logical for each of 4 NUMA nodes. Based on this hypothesis we have performed experiments that collect the data's read duration time for each NUMA node intending to show the differences between them. The Figure 5.6 presents the results for such experiment. The X axis shows the NUMA node identifier and the Y axis its respective average data's read duration time. The whiskers represent the standard error of each data-set, calculated as three times the standard deviation divided by the square root of the number of experiments. The average read time duration of the NUMA node 0 is lower than the others. The NUMA node 1 also presents better results than both 2 and 3, and is the only one that comes closer to the node 0. However this probably happens due to a physical proximity of the two nodes. The next Section 5.4 presents some more experiments that will consider the turing's NUMA node configuration, thus trying to achieve better performance.

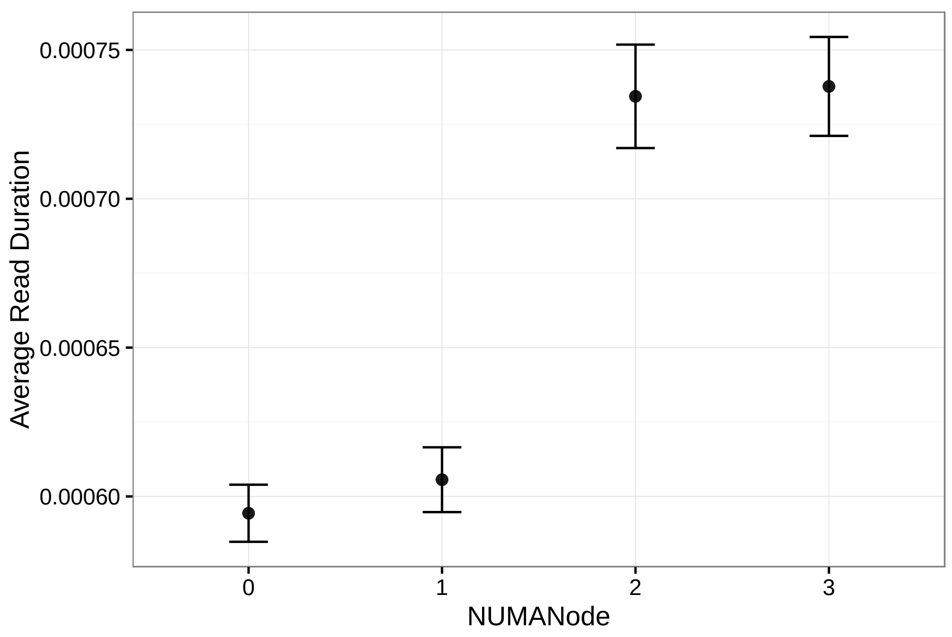


Figure 5.6 – Comparison between turing's NUMA node's average read time duration in seconds. Thirty executions for each configuration, all without drop the operational system's data cache.

5.4 Differences Between NUMA nodes Read Throughput

This section explores a different thread's pinning approach for experiments performed in the turing machine. We have focused on the machine turing since it has produced very strange results and so, we want to explain what is happening and causing the performance drop from 8 to 16 threads. This section takes into account the turing's architecture, more specifically the NUMA nodes/cores distribution, as described in Section 5.1, Table 5.1. The latter consists in the thread's pinning to use only the turing's NUMA nodes 0 and 1. We intend to achieve better performance due to the experiments exposed in the above section by the Figure 5.6, in which the nodes 0 and 1 have presented lower read times.

The Figure 5.7 presents the speedup achieved (on Y) as a function of the number of threads (on X) by the experiments when pinning threads in all NUMA nodes (0, 1, 2, 3) in contrast to only NUMA nodes 0 and 1. The grey line illustrates the ideal speedup, and the other lines, the results obtained for each pinning approach. The Figure 5.7 confirms that the performance increases when we have the threads fixed (pinned) only in the NUMA nodes 0 and 1. On the other hand, the performance dropping from 8 to 16 threads, albeit with less intensity, keeps being observed.

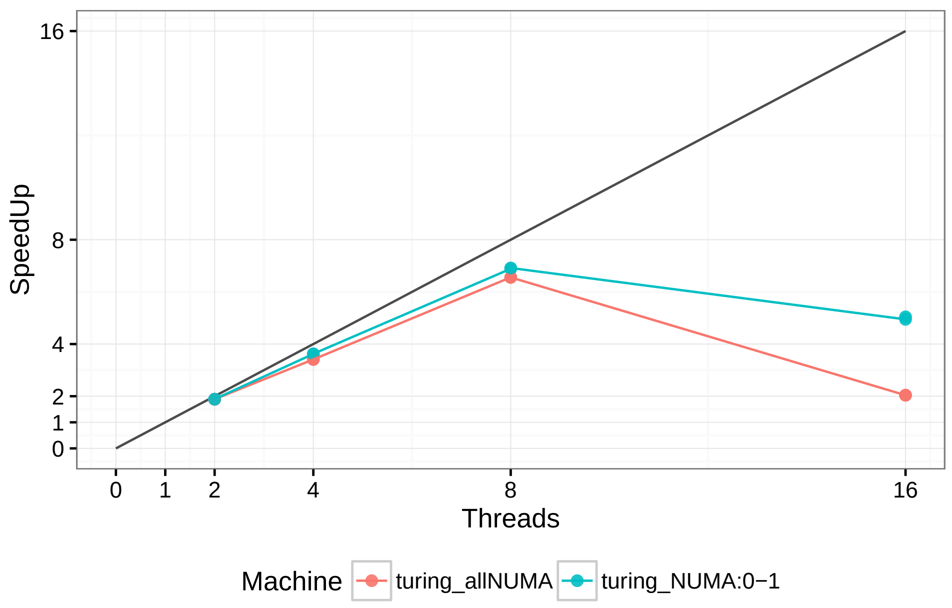


Figure 5.7 – Speedup achieved by pinning threads in all NUMA nodes (0, 1, 2, 3) versus only the nodes 0 and 1.

The experiments performed on turing using only the NUMA nodes 0 and 1 have achieved a little better speedup than the ones that have used all nodes. However the observed pattern of performance drop from 8 to 16 threads occurs in both experiments. Therewith we have decided to compare the average read time duration of the experiments configured to use all NUMA nodes with the ones that use only the nodes 0 and 1. This observation should present lower read times for the experiments with only the nodes 0 and 1. If true, the latter will prove our hypothesis that the nodes 0 and 1 have better read capabilities than the others, and thus they end up being faster. This experiment is depicted in the Figure 5.8. The X axis presents the experiment set name like `turing_allNUMA` (nodes 0, 1, 2, 3) and `turing_NUMA0-1` (nodes 0, 1). The Y axis shows the average read time for each set of experiments. Unfortunately, the Figure 5.8 has showed an unexpected behavior, presenting higher read times for the experiments performed with only the NUMA nodes 0 and 1. The fact is that this results end up refuting our hypothesis for the performance drop in the machine turing.

The exact reason for the performance drop in the machine turing still under study. We have performed some more experiments but none were able to provide the problem. This problem was not observed in the machine bali. Therewith, we consider some differences between the machines intending to formulate a new hypothesis for future studies. All experiments performed so far have considered that the traces are

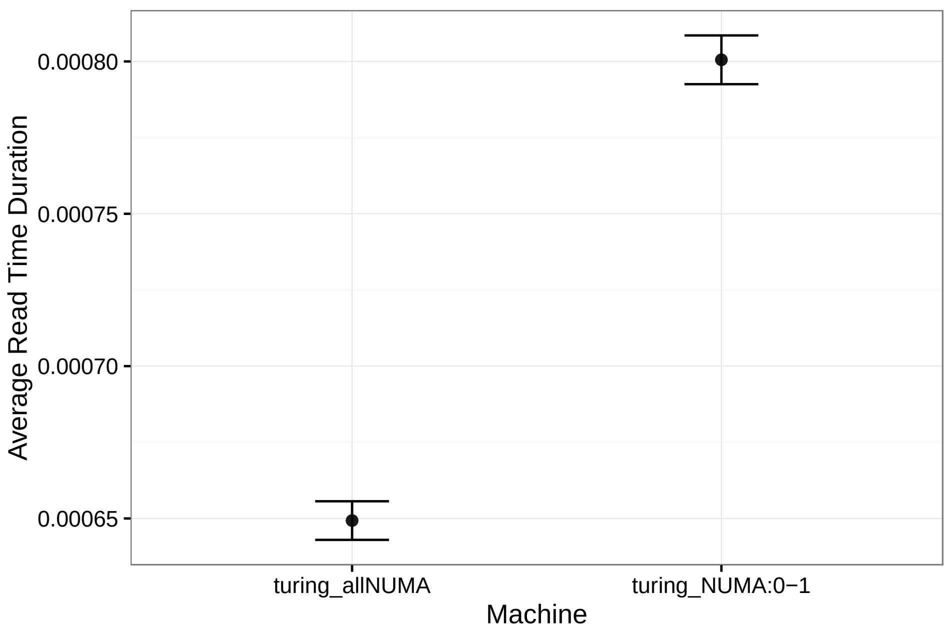


Figure 5.8 – Comparison between the average read time duration in seconds of the experiments with turing's NUMA nodes 0 and 1 versus all nodes.

kept in main memory. Thus, there is a good hypothesis for the bali's better performance, since its memory/bus speeds (1600MHz - 8GT/s QPI) are very faster than the turing ones (1066MHz - 6.4GT/s QPI). Furthermore bali is a little bit newer machine with more cache (20MB versus 18MB on turing) and the turbo boost technology v2.0 that can increase the processor's speed to 2.8GHz, versus the v1.0 present on turing that just achieves 2.4GHz. Therefore, we conclude that since the developed strategy is very memory bounded, the turing's lower memory/bus/processor speeds may be the bottleneck for our approach.

5.5 Disk Analysis

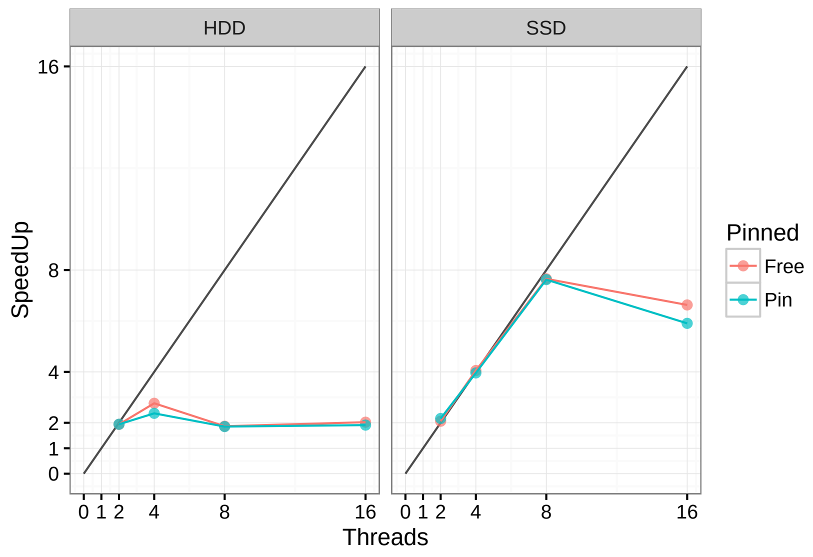
This section presents experiments performed by using the command

```
sysctl vm.drop_caches=3
```

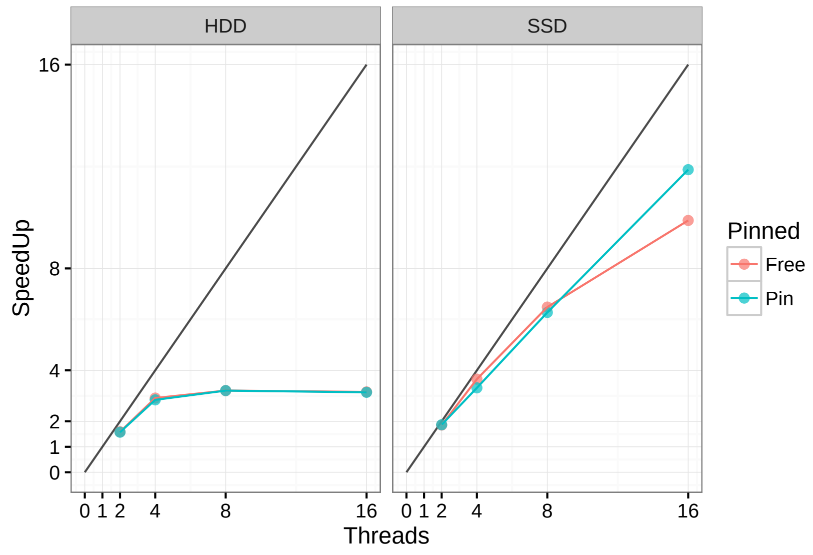
before each experimental execution. This command drops the operational system data cache, thus forcing the data reads directly from the disks where they are kept. The input data for the disk measurements were smaller than the others experiments to decrease the experimental time. Measurements have used traces of about 2GBytes divided in 64 pieces of 32MBytes each. The section discusses the relation between the disk type and the application performance, and also compares the read time duration versus number of threads. The latter considers HDD and SSD measurements for both machines (`bali` and `turing`) and settings (`Pin` and `Free`).

The Figure 5.9 displays the speedup achieved by our strategy assuming that the data were not cached for both machines, for turing (a) and bali (b). In these two plots, (a) and (b), the X axis presents the number of threads, and the Y axis, the speedup obtained. The grey lines represent the ideal speedup and the other lines the results for `Free` or `Pin` threads versions. The left side of the graphs shows the results considering the input data stored on HDD, while the right side considers the results for SSD. These experiments provide information for several conclusions about our strategy. First of all, we can see that the HDD results for both machines were bad, achieving a speedup of only about 4. These results were expected since rarely HDDs can handle with multiple accesses at same time, thus when we increase the number of threads, the concurrence grows degrading the performance. On the other hand, the speedup achieved on the SSDs were good: reaching 8 times in both `Free` and `Pin` settings for the machine turing and about 10 times for bali. The next graphs make a closer look into the behavior of each disk type by considering their read time duration for different number of threads, thus allowing a precise identification of the performance features for each one.

The Figures 5.10 and 5.11 depicts HDD measurements about the thread's read time duration from the machines turing and bali respectively. The results were classified considering three settings: `Pin`, `Free` and `Seq` (sequential). They show the relation between the average read time with an increasing number of threads. In both figures the X axis presents the number of threads, and the Y axis, the respective average read time. These figures also display rectangles which are zoom zones to expose values that are much smaller. For example, in the Figure 5.10 the rectangle comprises read times from 1 to 2 threads, and in the Figure 5.11 from 1 to 4, which are the smaller values of each one. The presented results explain the speedup's behavior for HDDs. As we can see in the Figure 5.9 (HDD, (a) and (b)), the speedup keeps growing until 4 threads, which is exactly the maximum number of threads for which the reads duration is still stable. Besides that, the speedup drops from 4 to 16 threads due to the huge growth of the read time duration from there on. The latter proves the problems from HDDs which can not work with multiple accesses, as observed for both machines.



(a)



(b)

Figure 5.9 – (a) Speedup achieved on turing with the input trace files kept on disk - SSD vs HDD - Pin vs Free; (b) Same but for bali.

The Figures 5.12 and 5.13 are presenting the SSD measurements about the thread's read time duration. These results were also classified considering the settings `Pin`, `Free` and `Seq`. These show the behavior of our strategy when running on SSDs with an increasing number of threads. Like the HDD graphs, the figures expose in the X axis, the number of threads, and in the Y axis, the respective average read time. These figures do not display the zoom rectangles since the measurements for all settings were more stable and closer each others. The results can also explain the performance obtained by our strategy when the traces were stored in SSDs. The speedup graph, Figure 5.9 (SSD, (a) and (b)), shows that the performance gain keeps growing until at least

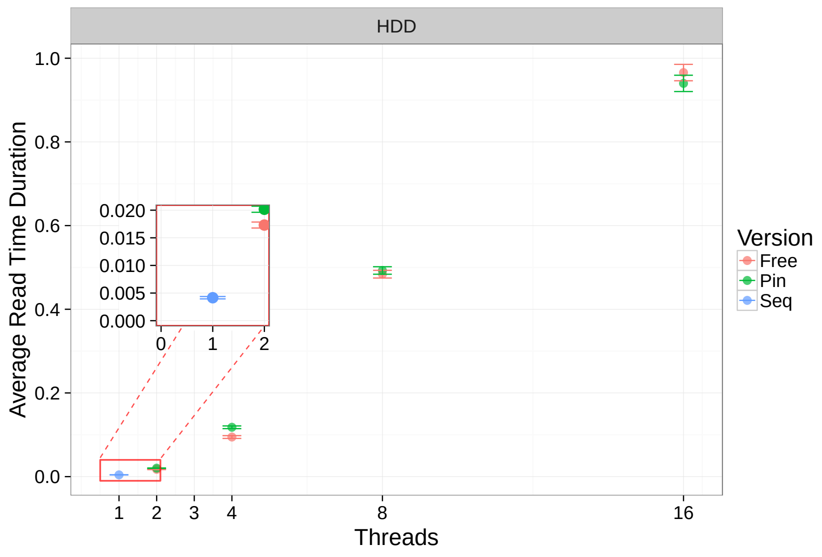


Figure 5.10 – The average read time duration in seconds for HDD as a function of the number of threads for turing.

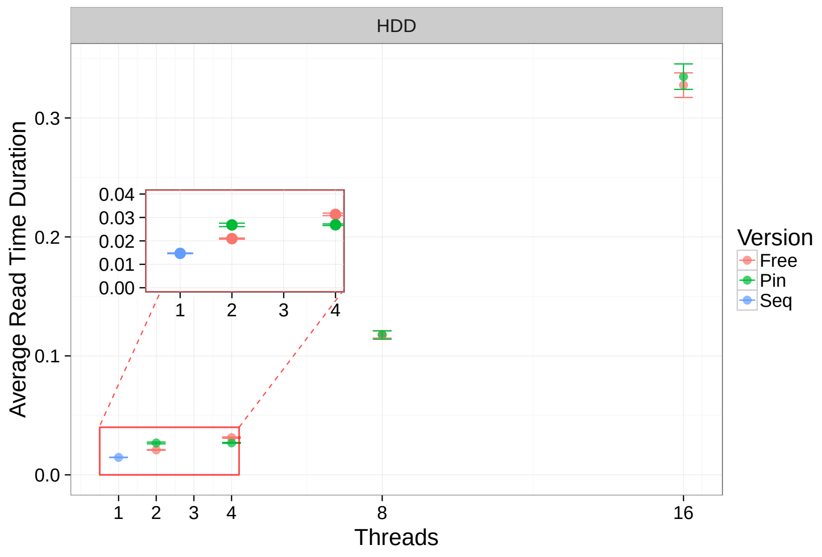


Figure 5.11 – The average read time duration in seconds for HDD as a function of the number of threads for bali.

8 threads in both machines, and then they slow down. This behavior is easily understood when we take a look to the Figures 5.12 and 5.13, which show that the max number of threads with small read times are in fact 8. Furthermore, from 8 to 16 threads the read times increase faster, and thus, the speedup drops. The latter proves the better performance of SSDs when handle with multiple accesses, however it is important to note that the sequential read times are very similar for both disks.

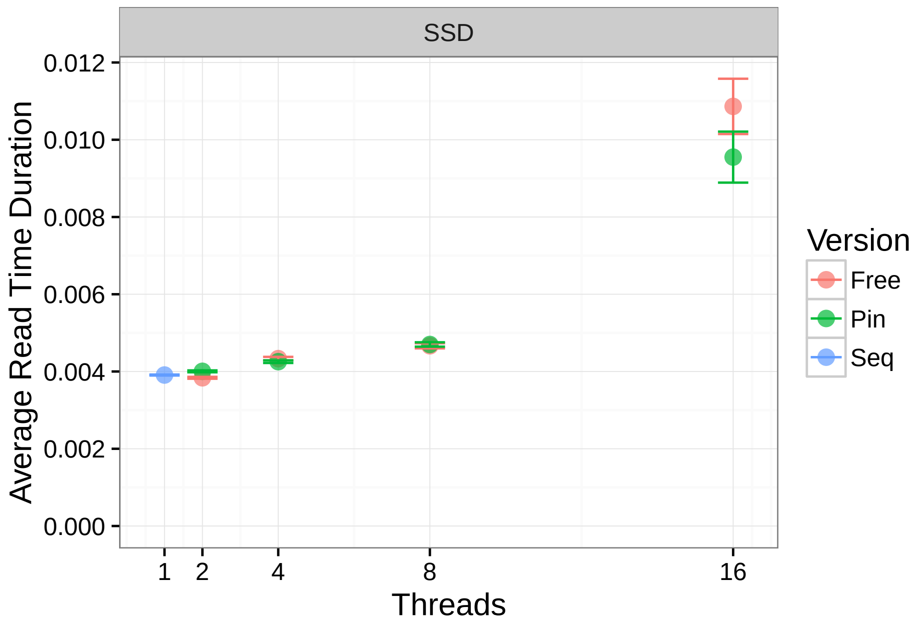


Figure 5.12 – The average read time duration in seconds for SDD as a function of the number of threads for turing.

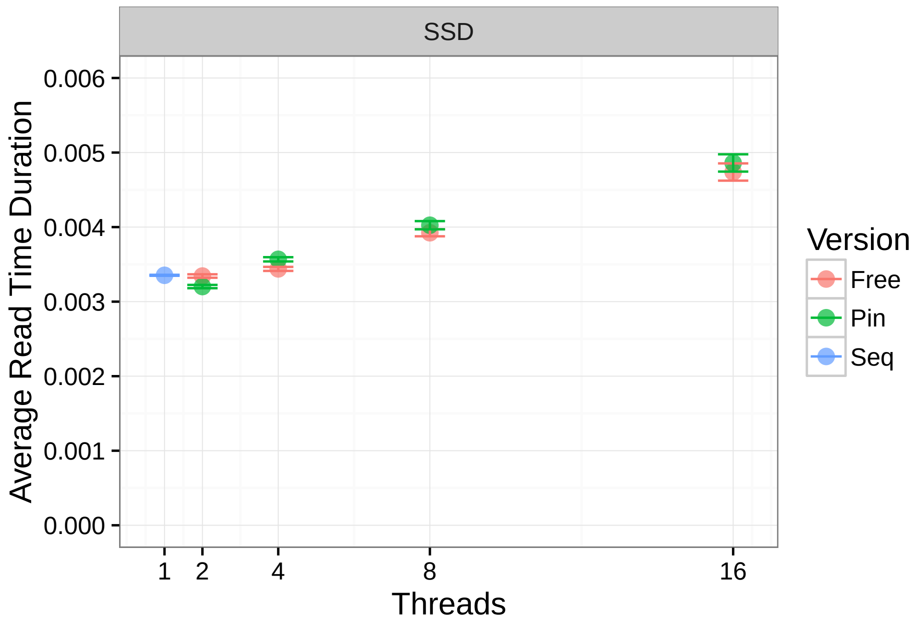


Figure 5.13 – The average read time duration in seconds for SDD as a function of the number of threads for bali.

5.6 Analysis Summary

The performance analysis has focused to provide experimental results showing the gains obtained with the implemented parallel strategy for the PajeNG simulator. The analysis has considered two platforms and several aspects like the number of execution flows (threads), locality and disk type. The Table 5.2 summarizes our results by presenting the maximum speedup obtained for each platform and for each configuration of storage system/threads locality. These results show us that the machines have presented different behavior. The machine bali, in most of the cases, continues gaining speedup until 16 threads, however each time with less efficiency. On the other hand, the machine turing reaches a certain limit with about 8 threads with good efficiency, and then, with more threads, it loses performance. This chapter has also presented an investigation over the application's behavior running on the turing host. We intented to understand the loss of performance detected in this machine. This investigation is still being carried out since the performed experiments have not showed sufficiently conclusive results yet.

Table 5.2 – Experimental results summary

Machine	**Trace Data In**	**Locality**	**Max Speedup**	**Efficiency**
bali	Memory	Free	7.04 (8 t)	44%
	Memory	Pin	9.21 (16 t)	58%
	SSD	Free	9.88 (16 t)	61%
	SSD	Pin	11.87 (16 t)	74%
	HDD	Free	3.20 (8 t)	40%
	HDD	Pin	3.20 (8 t)	40%
turing	Memory	Free	3.63 (4 t)	90%
	Memory	Pin	7.16 (8 t)	89%
	SSD	Free	7.65 (8 t)	95%
	SSD	Pin	7.62 (8 t)	95%
	HDD	Free	2.76 (4 t)	69%
	HDD	Pin	2.37 (4 t)	59%

6 CONCLUSION

The development of efficient parallel and distributed applications that better use the available resources is the way to follow the increasing demand of computing power. This is a very hard task due to the complexity of the current systems that comprise thousands to millions of processing units. There are many attempts intending to help this issue, such as development libraries like OpenMP and MPI, and performance analysis tools that intend to understand the application behavior. The latter was the global focus of this dissertation. This was presented by an extensive survey in the related work Chapter 3 that have considered several features of some of the most important analysis tools like Vampir, Paraver, ViTE and PajeNG.

Performance analysis tools based on execution traces are much helpful for the identification of potential bottlenecks and misused resources of current massive parallel applications. Such applications produce huge amounts of raw data that these tools have to handle. Currently there are some analysis tools that are scalable enough to keep processing this quantity of data. Most of these high performance tools, like Vampir, have a problem due to being strongly proposed to the processing of semantic traces that describe events from specific libraries. This approach prevents that applications built with different libraries to being analyzed, thus, some times, these tools are useless. On the other hand, tools that approach generic trace formats, like Paraver and PajeNG, do not reach the required processing power to analyze great parallel applications.

The main contributions of this work are the design and development of a parallelization strategy for the PajeNG toolset together with its respective performance analysis. The surveyed tools show us that PajeNG has great potential since it uses the generic Paje trace format. However this tool works sequentially and is not able to handle with huge traces. Our improvements made in the PajeNG are inspired by the pipelined multi-threading technique/pattern described in the Chapter 3. As related in the proposal Chapter 4, the developed strategy intends to maximize the amount of input data read by the PajeNG's processing unit. This was made by launching several execution flows similar to the sequential for the processing of several pieces from a trace file. The results are good, but since the strategy is memory bounded, its scalability is limited by the HDDs, SSDs and main memory speeds.

The results presented in the analysis Chapter 5 took into account several aspects that have impacted in the performance of our parallel strategy. The experiments were performed in two platforms as detailed in Table 5.1. The first is a Dell PowerEdge R910 machine identified as **turing** with 32 physical cores arranged in four NUMA nodes of 2.00GHz, each with eight cores. The second platform is a SGI C1104G-RP5 machine identified as **bali** with 16 physical cores arranged in two NUMA nodes of 2GHz, each with eight cores. The results obtained without drop the operational system's data cache have achieved an speedup about 8 for the machine turing with a maximum of 8 threads, and about 11 for the machine bali with 16 threads. We note that the threads migration for the latter case were degrading the strategy performance, so these results consider pinned threads. This behavior have occurred since the experimental platform were NUMA machines which implies in different memory access speed for each node. This way the automatic threads migration end up changing a thread from one node to another which slows the reading process. The machine turing was observed more in details since it presents a strange behavior. It achieves better performance with the NUMA nodes 0 and 1 than with the others nodes. This behavior was not totally understood. Our hypothesis was that since the node 0 is the only one attached to the disk interface, and the node 1 is near from 0, these nodes may perform faster read times. However the experiments comparing the average read time of all NUMA nodes, versus nodes 0 and 1, have showed that we were wrong. In terms of data read duration, the observed results do not show any gains from the nodes 0 and 1. Therewith our final hypothesis is that we have achieved the maximum possible throughput of the turing's memory, and so, even with more threads, the read times increase in a prohibitive way which disturbs the performance gain. The machine bali do not present this behavior since its memory/bus/processors speeds are faster than turing being capable to handle with more threads.

The experiments executed by disabling the operational system's data cache have compared the performance of our strategy when reading the traces directly from the HDDs and SSDs of the machines. As expected, the results obtained from HDDs quickly have reached the maximum amount of concurrent threads accessing the data. These ones have scaled up the PajeNG about 4 times, then, with more threads, the HDDs were not able to provide data efficiently in both machines. Hard disks drives are being optimized from several years, however their mechanic nature physically limits the disks themselves. Solid state drives as storage systems have another proposal that handle muth better with random/parallel accesses (CHEN; KOUFATY; ZHANG, 2009).

The SSDs results have proved such diferences: even with similar bandwidth from the HDDs, the experiments in SSDs have scaled up the PajeNG about 11 times.

As a whole, the work have presented relevant results. The conducted analysis lead us to understand several features of the developed strategy. As future work we intend to further study how to deal with the concurrency from the disk. Furthermore, since the threads run independently and we have created a way to split the trace file in several pieces, our implementation facilitates the development of a version for distributed memory which will not be limited by the storage systems.

REFERENCES

ALLEN, R.; KENNEDY, K. **Optimizing compilers for modern architectures: a dependence-based approach**. [S.l.]: Morgan Kaufmann San Francisco, 2002.

ALONSO, P. et al. Tools for power-energy modelling and analysis of parallel scientific applications. In: IEEE. **Parallel Processing (ICPP), 2012 41st International Conference on**. [S.l.], 2012. p. 420–429.

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: ACM. **Proceedings of the April 18-20, 1967, spring joint computer conference**. [S.l.], 1967. p. 483–485.

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

BARNEY, B. et al. Introduction to parallel computing. **Lawrence Livermore National Laboratory**, v. 6, n. 13, p. 10, 2010.

BRUNST, H. et al. Comprehensive performance tracking with vampir 7. In: MüLLER, M. S. et al. (Ed.). **Tools for High Performance Computing 2009**. [S.l.]: Springer Berlin Heidelberg, 2010. p. 17–29. ISBN 978-3-642-11261-4.

CHAN, A.; GROPP, W.; LUSK, E. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. **Scientific Programming**, IOS Press, v. 16, n. 2, p. 155–165, 2008.

CHEN, F.; KOUFATY, D. A.; ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. Citeseer, 2009.

COOK, J. E.; WOLF, A. L. Discovering models of software processes from event-based data. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM, v. 7, n. 3, p. 215–249, 1998.

CORNELISSEN, B.; MOONEN, L. Visualizing similarities in execution traces. In: **Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis (PCODA)**. [S.l.: s.n.], 2007. p. 6–10.

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

COULOMB, K. et al. **Visual trace explorer (ViTE)**. 2009. <http://vite.gforge.inria.fr/>.

CYTRON, R. Doacross: Beyond vectorization for multiprocessors. In: **ICPP'86**. [S.l.: s.n.], 1986. p. 836–844.

DAGUM, L.; MENON, R. Openmp: An industry-standard api for shared-memory programming. **IEEE Comput. Sci. Eng.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 5, n. 1, p. 46–55, 1998. ISSN 1070-9924.

FERRANTE, J.; OTTENSTEIN, K. J.; WARREN, J. D. The program dependence graph and its use in optimization. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 9, n. 3, p. 319–349, 1987.

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

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

GROPP, W.; LUSK, E. L.; THAKUR, R. **Using MPI-2: Advanced Features of the Message Passing Interface**. Cambridge, Massachusetts, USA: The MIT Press, 1999.

GUSTAFSON, J. L. Reevaluating amdahl's law. **Communications of the ACM**, ACM, v. 31, n. 5, p. 532–533, 1988.

IHAKA, R.; GENTLEMAN, R. R: a language for data analysis and graphics. **Journal of computational and graphical statistics**, Taylor & Francis, v. 5, n. 3, p. 299–314, 1996.

JUMPSHOT. **Jumpshot-4 Users Guide**. <ftp://ftp.mcs.anl.gov/pub/mpi/slog2/js4-usersguide.pdf>, 2007. Acessed in November, 19th, 2014.

JURENZ, M. Vampirtrace software and documentation. **ZIH, Technische Universität Dresden, http://www. tu-dresden. de/zih/vampirtrace**, v. 4, 2006.

KAMINSKY, A. Big cpu, big data: Solving the world's toughest computational problems with parallel computing. **Creative Commons**, 2013.

KERGOMMEAUX, J. C. de; STEIN, B. de O.; BERNARD, P. E. Paje, an interactive visualization tool for tuning multi-threaded parallel applications. **Elsevier Science Parallel Computing**, Elsevier Science, Numath, INRIA Lorraine, BP 101, F-54600 Villers les Nancy, France, v. 26, n. 10, p. 1253–1274, 2000. ISSN 0167-8191.

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

LABARTA, J.; GIMENEZ, J. Performance analysis: From art to science. **Parallel Processing for Scientific Computing. M. Heroux and R. Raghavan and HD Simon Eds. SIAM**, p. 9–32, 2006.

LUNDSTROM, S. F.; BARNES, G. H. A controllable mimd architecture. In: **Proceedings of the 1980 International Conference on Parallel Processing**. [S.l.: s.n.], 1980. p. 19–27.

MARKOMANOLIS, G.; JORBA, O.; BALDASANO, J. Performance analysis of an online atmospheric-chemistry global model with paraver: Identification of scaling limitations. In: IEEE. **High Performance Computing & Simulation (HPCS), 2014 International Conference on**. [S.l.], 2014. p. 738–745.

MATTSON, T. G.; SANDERS, B. A.; MASSINGILL, B. L. **Patterns for parallel programming**. [S.l.]: Pearson Education, 2004.

MCCOOL, M.; REINDERS, J.; ROBISON, A. **Structured parallel programming: patterns for efficient computation**. [S.l.]: Elsevier, 2012.

MÜLLER, M. S. et al. Developing scalable applications with vampir, vampirserver and vampirtrace. In: CITESEER. **PARCO**. [S.l.], 2007. v. 15, p. 637–644.

NAGEL, W. E. et al. Vampir: Visualization and analysis of mpi resources. **Supercomputer 63, Volume XII, Number 1**, p. 69–80, 1996.

OTTONI, G. et al. Automatic thread extraction with decoupled software pipelining. In: IEEE. **Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on**. [S.l.], 2005. p. 105–116.

PARAVER. **Paraver Tracefile Description**. <http://www.bsc.es/media/1370.pdf>, 2001. Acessed in November, 15th, 2014.

PARAVER. **Reference Manual**. <http://www.bsc.es/media/1364.pdf>, 2001. Acessed in November, 15th, 2014.

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

RAMAN, E. **Parallelization techniques with improved dependence handling**. Thesis (PhD) — Princeton University, 2009.

RANGAN, R. et al. Decoupled software pipelining with the synchronization array. In: IEEE COMPUTER SOCIETY. **Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques**. [S.l.], 2004. p. 177–188.

REINDERS, J. **Intel Threading Building Blocks**. First. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007. ISBN 9780596514808.

SCHNORR, L. **Some Visualization Models applied to the Analysis of Parallel Applications**. Thesis (PhD) — INPG; UFRGS, 2009.

SCHNORR, L. M. **Pajé trace file format**. Porto Alegre, Brazil, 2014. <http://paje.sf.net>.

SCHNORR, L. M. **PajeNG – Paje Next Generation**. 2014. <http://github.com/schnorr/pajeng>.

SCHNORR, L. M.; LEGRAND, A.; VINCENT, J.-M. Detection and analysis of resource usage anomalies in large distributed systems through multi-scale visualization. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 24, n. 15, p. 1792–1816, 2012.

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

TARJAN, R. Depth-first search and linear graph algorithms. **SIAM journal on computing**, SIAM, v. 1, n. 2, p. 146–160, 1972.

TRAHAY, F. et al. Eztrace: a generic framework for performance analysis. In: IEEE. **Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on**. [S.l.], 2011. p. 618–619.

ZAKI, O. et al. Toward scalable performance visualization with jumpshot. **International Journal of High Performance Computing Applications**, SAGE Publications, v. 13, n. 3, p. 277–288, 1999.

APPENDIX A — FERRAMENTA DE ALTO DESEMPENHO PARA ANÁLISE DO COMPORTAMENTO DE PROGRAMASPARALELOS BASEADA EM RASTOS DE EXECUÇÃO

A.1 Introdução

A contínua demanda por poder de computação tende ao desenvolvimento de aplicações paralelas eficientes que melhor utilizam os recursos disponíveis. Sistemas modernos de alto desempenho compreendem de milhares a milhões de unidades de processamento conectadas por redes hierárquicas complexas. O desenvolvimento de aplicações para tais sistemas é uma tarefa complexa, afetada por ao menos dois fatores: o primeiro é a alta escalabilidade provida por eles, a qual é difícil de alcançar; o segundo é a falta de uma execução determinística. Como o processamento nos nós é independentes, compreender tais aplicações pode ser muito uma tarefa muito custosa. Algumas bibliotecas como OpenMP e MPI são utilizadas para facilitar a implementação destas aplicações. No entanto, mesmo com o apoio de bibliotecas, o sucesso no desenvolvimento de uma aplicação paralela de alto desempenho depende de um mapeamento correto de seus processos com relação aos recursos disponíveis.

A identificação de recursos não utilizados e potenciais gargalos de processamento requere boas ferramentas para análise de desempenho, capazes de observar muitas entidades durante longos períodos de tempo. Esta observação é muitas vezes iniciada através da coleta de importantes medidas de desempenho através do rastreamento de eventos. Os dados coletados são geralmente em nível de aplicação. Ferramentas de rastreamento comumente registram estados locais e globais do programa, a quantidade de dados transferidos por mensagens e contadores de hardware para funções específicas. Estes dados permitem a identificações de vários padrões complexos, tais como: *late communications*, *costly synchronization* ou *train effects*. Infelizmente o rastreamento de aplicações muitas vezes produz grandes arquivos de rastro, facilmente atingindo a ordem de gigabytes de dados brutos. Portanto, o tratamento desses dados para uma forma legível precisa ser eficiente para permitir uma análise de desempenho menos onerosa, mais rápida e útil.

Ao longo dos anos várias ferramentas baseadas em rastros têm sido desenvolvidas para a análise de desempenho de aplicações paralelas. Os principais desafios dessas ferramentas são a interpretação e o processamento da grande quantidade de dados produzidos por aplicações paralelas. Tais dados são frequentemente utilizados para reproduzir/simular o comportamento da execução do programa. Isto resulta em representações visuais tais como o linhas de tempo, grafos de comunicação entre outros. A maioria das ferramentas existentes, tais como Vampir, Scalasca, TAU, concentram-se no processamento de formatos de rastro com uma semântica fixa e bem definida. Estes formatos de arquivo são normalmente definidos para lidar com aplicações desenvolvidas com bibliotecas populares como OpenMP, MPI, e CUDA. No entanto, algumas vezes, essas ferramentas tonam-se inúteis, tendo em vista que nem todas aplicações paralelas são construídas com tais bibliotecas. Felizmente existem outras ferramentas que apresentam uma abordagem mais dinâmica. Estas apresentam um formato de arquivo de rastreamento aberto, não associado a uma semântica específica. Desta forma, tais ferramentas podem lidar com uma ampla gama de aplicações paralelas. Algumas dessas ferramentas são: Paraver, Pajé e PajeNG. O fato de serem mais genéricas vem com um custo. Estas ferramentas frequentemente apresentam baixo desempenho no processamento de grandes rastros. Algumas destas ferramentas controlam o tamanho dos dados utilizando técnicas de agregação, entretanto tal procedimento pode acabar removendo informações importantes dos dados de rastreamento originais.

Este trabalho se concentra na otimização do conjunto de ferramentas contidas em PajeNG, mais especificamente no simulador Pajé, o qual é responsável por processar os rastros e recriar o comportamento original da aplicação paralela observada. Este conjunto de ferramentas funciona com o formato de rastro Pajé, o qual apresenta uma semântica aberta. PajeNG baseia-se em seu simulador Pajé e oferece ferramentas para análise de desempenho. Exemplos de tais ferramentas são: uma linha de tempo visual provida por pajeng e pj_dump. O último é capaz de exportar os dados de desempenho em um formato textual CSV, os quais podem ser utilizados para análises utilizando o software R. A implementação atual do simulador Pajé é sequencial e, portanto, não escalável para lidar com grandes arquivos de rastro. Esta dissertação apresenta uma estratégia de paralelização para PajeNG, com a intenção de melhorar seu desempenho e escalabilidade.

PajeNG é um conjunto de ferramentas para o processamento e análise de rastros construída em C++. Internamente PajeNG utiliza sua biblioteca chamada libpaje para processar os rastros em formato Pajé. O formato Pajé é auto-definido, textual e genérico. Os arquivos de rastreio neste formato são divididos em duas partes. A primeira parte contém uma definição para cada tipo de evento e um número de identificação único para tal. A segunda parte contém um evento por linha, todas iniciando por seu identificador. Os eventos registrados em formato Pajé são definidos dentro de cinco tipos de objetos conceituais: *Containers*, *States*, *Events*, *Variables* e *Links*. Um *Container* pode ser qualquer entidade monitorável, como um roteador, um processo, ou uma thread. Este é o objeto mais genérico e o único que pode conter outros, incluindo outros containers. *States* representam qualquer coisa que tenha um tempo de início e fim. Por exemplo: um determinado container começa no estado (state) bloqueado e então, depois de algum tempo, volta ao estado livre, ou trabalhando etc. Este período de tempo caracteriza um state. *Events* são qualquer coisa notável o suficiente para ser unicamente identificado e conter apenas uma única medida de tempo. *Variables* registram informações sobre a evolução do valor de uma variável ao longo do tempo. Finalmente, os *Links* são as relações, ou troca de mensagens, entre dois containers, armazenando os tempos de início e fim da comunicação.

A libpaje realiza o processamento necessário para a análise dos rastros. Esta funciona com um arquivo de rastro único contendo todas as informações coletadas e realiza o processamento de forma sequencial. A libpaje segue um fluxo de trabalho constituído por três componentes principais, como representado na Figura A.1, procedendo a partir da esquerda para a direita. O componente PajeFileReader inicia o processo através da leitura do rastro. Este, lê pedaços de tamanho fixo de arquivo e enviá-os para o segundo componente. O componente seguinte, PajeEventDecoder, recebe os pedaços, divide-os em linhas e transforma cada linha em um evento/objeto genérico. O último componente, PajeSimulator, implementa funções diferentes para cada tipo de evento, empilhando cada evento/objeto na pilha correta de seu respectivo container. Este modelo de processamento apresenta sérios problemas de desempenho quando os rastros aumentam muito de tamanho. Um rastro de apenas 1GBytes por exemplo, leva cerca de 1 minuto para ser processado, e um rastro de 10GBytes, já alcança os 10 minutos de processamento. Portanto o o tempo de execução cresce linearmente de acordo com o tamanho da entrada e com isso o PajeNG acaba perdendo utilidade na análise de rastros muito grandes.

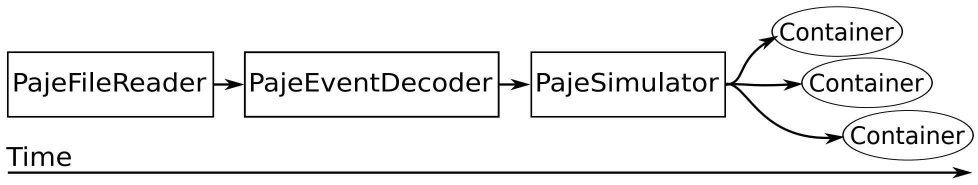


Figure A.1 – Fluxo de execução original da libpaje.

A.2 Proposta e Implementação

A estratégia de otimização do PajeNG proposta por este trabalho trata-se da paralelização do fluxo de trabalho da libpaje visando maximizar a quantidade de dados lidos para processamento. Tal estratégia contempla duas mudanças principais no programa original: uma nos dados de entrada (rastros), e outra no fluxo de execução da ferramenta. O formato rastro Pajé foi modificado para permitir a quebra do arquivo único em vários arquivos menores. O fluxo de trabalho da libpaje foi adaptado para realizar o processamento em paralelo, processando múltiplos arquivos de rastro simultaneamente utilizando threads.

A criação dos múltiplos arquivos de rastro foi feita através da quebra do arquivo único em vários pedaços, cada um contendo um evento container. Desta forma vários arquivos são gerados contendo todas as informações referentes a um container, e todos juntos sendo equivalentes à informação do arquivo único anterior. Para ligar todos os pedaços criamos um novo tipo de evento chamado PajeTraceFile. Este evento contém três informações: seu tipo, um container e um caminho/nome de arquivo, no qual os eventos estão registrados. Além disso, construímos um arquivo de rastreamento descritor. Este, funciona como um produtor de trabalho, contendo apenas a criação de eventos Container e PajeTraceFile. O arquivo descritor será utilizado para inciar a aplicação, inicializando os containers e indicando a localização dos rastros.

Para permitir o processamento paralelo, a libpaje precisou ser modificada em alguns pontos. Primeiramente, o componente PajeSimulator foi modificado para compreender o novo tipo de evento, PajeTraceFile. Além disso, um novo componente chamado PajeParallelProcessor foi implementado. Este é o responsável por tornar o processamento paralelo. O componente PajeParallelProcessor implementa uma fila de tarefas e um conjunto de trabalhadores. O conjunto de trabalhadores é composto por C++11 threads e seu tamanho é definido pelo usuário. A fila de taréfas é preenchida por eventos PajeTraceFile. A Figura A.2 ilustra como estas modificações criam o novo fluxo de trabalho da libpaje. A thread principal da aplicação, topo da Figura A.2, é bem similar ao funcionamento sequencial. As modificações realmente começam quando um evento PajeTraceFile chega ao PajeSimulator. Neste caso, o PajeSimulator coloca tal evento na fila de tarefas do novo componente, PajeParallelProcessor, e inicia o conjunto de trabalhadores, C++11 threads. Deste ponto em diante, os trabalhadores passam a consumir os eventos PajeTraceFile que estão na fila de tarefas. Assim, os trabalhadores recebem cada um, um arquivo de rastro diferente, processando-os em em threads separadas. O fluxo de trabalho dos trabalhadores é idêntico ao sequencial.

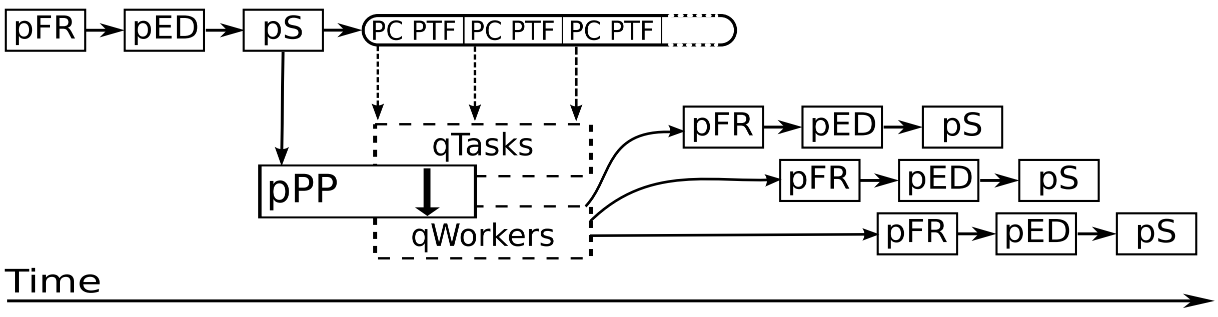


Figure A.2 – Fluxo de execução paralelo da libpaje.

A.3 Resultados

O desempenho alcançado pela estratégia desenvolvida foi bem interessante. A análise realizada levou em consideração várias aspectos, como: localidade das threads, tipo de disco, e quantidade de threads. Os experimentos foram realizados em duas máquinas NUMA (Non Uniform Memory Access) diferentes, turing com 32 núcleos e bali com 16 núcleos. Cada configuração experimental foi executada 30 vezes. A Figura A.3 mostra uma visão geral do desempenho alcançado pela paralelização do PajeNG. Neste gráfico estão representados os resultados de aceleração (Speedup) atingidos em cada máquina, à esquerda bali e à direita turing. O eixo X apresenta o número de threads e no eixo Y a respectiva aceleração alcançada. A linha cinza representa a aceleração ideal, e as outras linhas, os resultados obtidos com duas configurações de localidade de threads diferentes: Free, threads livres para migrar como o sistema operacional decidir e, Pin, cada thread fixada em um processador. Com este gráfico conseguimos observar que para ambas as máquinas o ganho de desempenho com as threads fixadas foi maior. A máquina bali chegando a atingir uma aceleração de cerca de 11 vezes em relação a versão sequencial, e a turing cerca de 8 vezes.

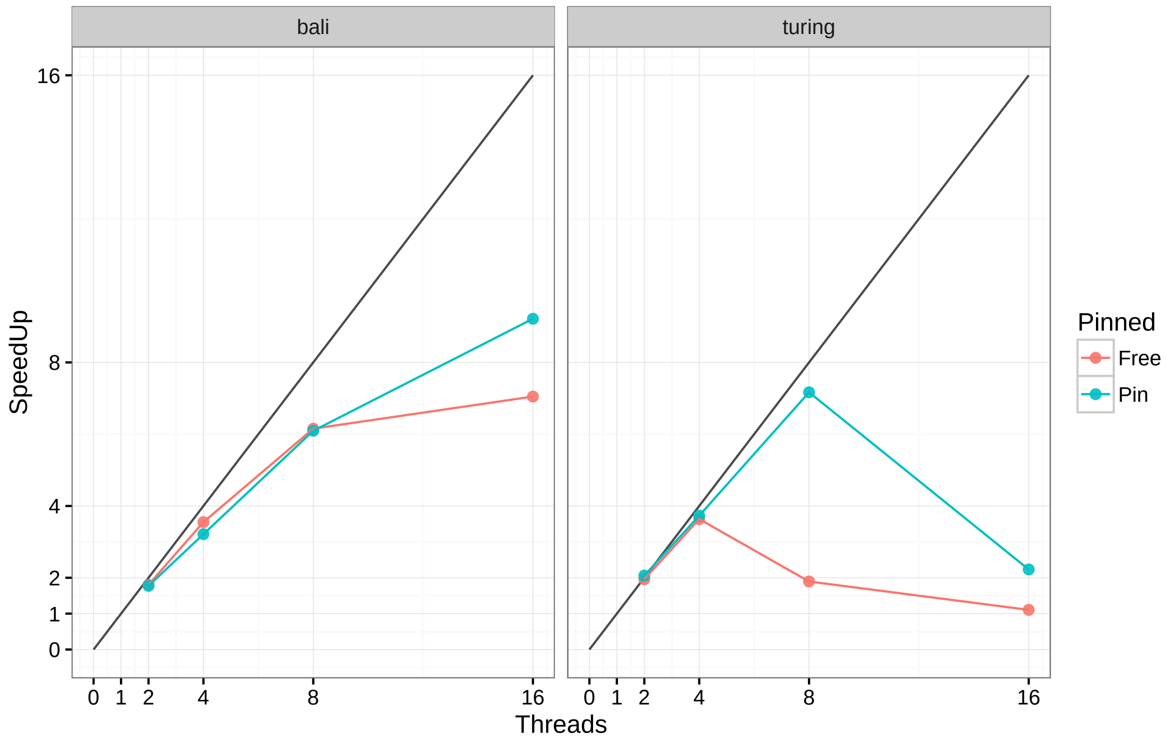


Figure A.3 – Speedup alcançado pela estratégia desenvolvida nas máquinas bali e turing. Gráfico comparando o desempenho alcançado com as threads livres (Free), com as threads fixas (Pin).

A.4 Conclusão e Trabalhos Futuros

O desenvolvimento de aplicações paralelas e/ou distribuídas eficientes, que melhor utilizam os recursos disponíveis, é a forma de suprir a crescente demanda por poder de computação. Isto é uma tarefa muito difícil devido à complexidade dos sistemas atuais, os quais compreendem de milhares a milhões de unidades de processamento. Existem muitas tentativas que pretendem ajudar e facilitar tal desenvolvimento, por exemplo bibliotecas como OpenMP e MPI, e ferramentas para análise de desempenho que pretendem compreender o comportamento de uma aplicação. Estas ferramentas foram o foco global desta dissertação. No capítulo de trabalhos relacionados (related work) é apresentado um extenso levantamento de algumas das ferramentas mais importantes existentes, tais como Vampir, Paraver, ViTE e PajeNG.

As principais contribuições deste trabalho são o projeto e desenvolvimento de uma estratégia de paralelização para o conjunto de ferramentas PajeNG, juntamente com sua respectiva análise de desempenho. As ferramentas pesquisadas nos mostram que PajeNG tem um grande potencial, uma vez que utiliza um formato de rastro genérico. No entanto, o PajeNG original funciona sequencialmente e não é capaz de lidar com grandes rastros. A estratégia desenvolvida maximiza a quantidade de dados de entrada lidos pela unidade de processamento do PajeNG. Isto foi feito através do lançamento de vários fluxos de execução semelhantes ao sequencial, cada um realizando o processamento de um pedaço do arquivo de rastro. Esta abordagem apresentou bons resultados, entretanto sua máxima escalabilidade é limitada pela velocidade dos sistemas de armazenamento, HDDs, SSDs e memória principal.

Como um todo, o trabalho apresenta resultados bem relevantes, alcançando uma aceleração, com relação ao PajeNG original, de aproximadamente 11 vezes em alguns experimentos. Além disso, na versão paralela desenvolvida, cada thread executa de forma independente. Tal independência cria um modelo que facilitará muito o eventual desenvolvimento de uma versão para memória distribuída. A análise de desempenho realizada nos levou a compreender vários aspectos sobre estratégia desenvolvida, como, por exemplo, sua escalabilidade limitada e problemas de acesso concorrente ao disco. Em trabalhos futuros, pretende-se estudar mais a fundo a forma como lidar com tal concorrência e/ou implementar uma versão para memória distribuída, a qual eliminaria este problema.