

FEDERAL UNIVERSITY OF PAMPA

ADRIANO MARQUES GARCIA

**TOWARDS A BENCHMARK FOR PERFORMANCE AND POWER
CONSUMPTION EVALUATION OF PARALLEL PROGRAMMING
INTERFACES**

**Alegrete
2019**

ADRIANO MARQUES GARCIA

**TOWARDS A BENCHMARK FOR PERFORMANCE AND
POWER CONSUMPTION EVALUATION OF PARALLEL
PROGRAMMING INTERFACES**

Master's Thesis submitted to the Graduate Program in Electrical Engineering of Federal University of Pampa in partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

Supervisor: Prof. Dr. Alessandro Gonçalves Girardi

Co-supervisor: Prof. Dr. Claudio Schepke

**Alegrete
2019**

Ficha catalográfica elaborada automaticamente com os dados fornecidos
pelo(a) autor(a) através do Módulo de Biblioteca do
Sistema GURI (Gestão Unificada de Recursos Institucionais) .

G216t Garcia, Adriano Marques

Towards a benchmark for performance and power consumption
evaluation of parallel programming interfaces / Adriano
Marques Garcia.

78 p.

Dissertação(Mestrado)-- Universidade Federal do Pampa,
MESTRADO EM ENGENHARIA ELÉTRICA, 2019.

"Orientação: Alessandro Gonçalves Girardi".


1. Análise de desempenho. 2. Consumo de Energia. 3.
Programação Paralela. I. Título.

Adriano Marques Garcia

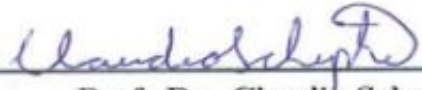
**TOWARDS A BENCHMARK FOR
PERFORMANCE AND POWER CONSUMPTION
EVALUATION OF PARALLEL PROGRAMMING
INTERFACES**

Master's Thesis submitted to the Graduate
Program in Electrical Engineering of Federal
University of Pampa in partial fulfillment of
the requirements for the degree of Master in
Electrical Engineering.

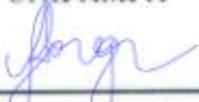
Master's Thesis defended and passed in 25th march 2019
Examination board



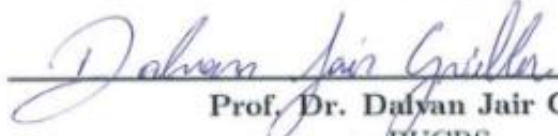
Prof. Dr. Alessandro Gonçalves Girardi
Supervisor
UNIPAMPA



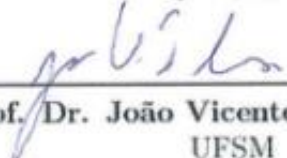
Prof. Dr. Claudio Schepke
Co-supervisor
UNIPAMPA



Prof. Dr. Arthur Francisco Lorenzon
UNIPAMPA



Prof. Dr. Dáyan Jair Griebler
PUCRS



Prof. Dr. João Vicente Ferreira Lima
UFSM

“However difficult life may seem,
there is always something you
can do and succeed at.”
(Stephen Hawking)

RESUMO

Este trabalho apresenta um conjunto de pseudo-aplicações e propõe que sejam utilizadas como um benchmark para avaliar desempenho e consumo de energia de diferentes Interfaces de Programação Paralela (IPPs). O conjunto consiste de 11 pseudo-aplicações implementadas usando as IPPs: PThreads, OpenMP, MPI-1 e MPI-2. Essas IPPs foram escolhidas por serem compatíveis com a maioria das arquiteturas multinúcleo atuais. Estudos anteriores usaram essas pseudo-aplicações para realizar esse tipo de avaliação em diferentes arquiteturas, pois não há outro benchmark que ofereça essa variedade de pseudo-aplicações implementadas em todas essas IPPs, usando diferentes modelos de comunicação (memória compartilhada ou troca de mensagens). Trabalhos relacionados mais recentes que comparam IPPs precisam procurar diferentes alternativas para resolver o problema, pois não há um benchmark que supra essa demanda. O objetivo deste trabalho é propor o uso dessas pseudo-aplicações como um benchmark para avaliar desempenho e consumo de energia de diferentes IPPs. Para alcançar esse objetivo, são analisados o comportamento das pseudo-aplicações e IPPs em relação aos acessos à memória cache, desvios e operações de ponto-flutuante. Os resultados dos experimentos mostraram que existe uma boa distribuição entre pseudo-aplicações que fazem um uso mais ou menos intensivo desses parâmetros. Além disso, é feito um estudo de caso para avaliar o desempenho, o consumo de energia e o consumo de potência (potência dissipada) dessas pseudo-aplicações. Os resultados mostram que as pseudo-aplicações em geral possuem um bom desempenho. Apesar do consumo de energia ser, em alguns casos, 300 vezes maior entre diferentes pseudo-aplicações com MPI por conta das diferentes características e parâmetros de cada aplicação, essa diferença não aparece na potência dissipada. As aplicações e as IPPs mostraram fazer um uso dos recursos de hardware de uma forma bem dinâmica e nossos resultados mostram que elas são capazes de ser representativas em diferentes cenários. Portanto esse conjunto pode sim ser utilizado como um benchmark paralelo.

Palavras-chave: benchmark, desempenho, consumo de energia.

ABSTRACT

This work presents a set of pseudo-applications and proposes them to be used as a benchmark to evaluate the performance and power consumption of different Parallel Programming Interfaces (PPIs). The set consists of 11 algorithms implemented in PThreads, OpenMP, MPI-1, and MPI-2 (spawn) PPIs. These PPIs were chosen because they are compatible with most of the current multi-core architectures. Previous studies have used some of these pseudo-applications to perform this type of evaluation in different architectures since there is no benchmark that offers this variety of PPIs and communication models. Recent related work that compare PPIs have looked for different alternatives to solve the problem since the available parallel benchmarks do not meet this demand. The goal of this work is to propose the use of these pseudo-applications as a benchmark to evaluate the performance and power consumption of different PPIs. To achieve this goal, we analyze the behavior of pseudo-applications and PPIs with respect to cache access, branches, and floating point operations. The results of these experiments showed that there is a good balance among pseudo-applications that make more or less intensive use of these parameters. In addition, we conducted a case study to evaluate the performance, energy consumption, and power consumption (power dissipation) of these pseudo-applications. The results show that the pseudo-applications generally have a good performance. Although the total energy consumption is, in some cases, 300 times greater among different MPI pseudo-applications, this difference does not appear in the power consumption. The PPIs and the pseudo-applications presented to use the hardware resources in a very dynamic way and our results show that they are able to represent different scenarios. Therefore they can be used as a parallel benchmark.

Keywords: benchmark, performance, energy consumption.

LIST OF FIGURES

Figure 1 – Low L3 cache accesses per kilo instructions rate.	48
Figure 2 – Medium L3 cache accesses per kilo instructions rate.	49
Figure 3 – High L3 cache accesses per kilo instructions rate.	50
Figure 4 – Low branch instructions per kilo total instructions rate.	52
Figure 5 – Medium branch instructions per kilo total instructions rate.	52
Figure 6 – High branch instructions per kilo total instructions rate.	53
Figure 7 – Floating-point operations per kilo instructions.	55
Figure 8 – Low floating-point operations per kilo instructions rate.	56
Figure 9 – High floating-point operations per kilo instructions rate.	57
Figure 10 – Energy consumption and performance for each pseudo-application.	59
Figure 11 – Power consumption for CPU-bound pseudo-applications.	62
Figure 12 – Power consumption for memory-bound pseudo-applications.	63
Figure 13 – L2 cache accesses per kilo instructions (A).	77
Figure 14 – L2 cache accesses per kilo instructions (B).	78

CONTENTS

1	INTRODUCTION	15
1.1	Objectives	17
1.2	Structure of the Work	18
2	BACKGROUND	19
2.1	Benchmarks	19
2.2	Parallel Programming Interfaces	20
2.3	Hardware Performance Counters	21
2.4	Related Work	21
2.4.1	Similar Benchmarks	22
2.4.2	Comparison Among Benchmarks	24
3	BENCHMARK PSEUDO-APPLICATIONS	27
3.1	Pseudo-applications Description	27
3.1.1	Numeric Integration	27
3.1.2	PI Calculation	28
3.1.3	Dot Product	29
3.1.4	Harmonic Sums	29
3.1.5	Odd-Even Sort	30
3.1.6	Discrete Fourier Transform	30
3.1.7	Turing Ring	31
3.1.8	Dijkstra	33
3.1.9	Jacobi Method	33
3.1.10	Matrix Multiplication	34
3.1.11	Gram-Schmidt	35
3.2	Parallelizing the Pseudo-Applications	36
3.3	Historic of Classifications	37
3.4	Source Code Improvements	39
4	EXPERIMENTAL RESULTS	45
4.1	Methodology	45
4.2	Cache Memory Accesses	47
4.3	Branch Instructions	51
4.4	Floating-Point Operations	54
4.5	Performance and Energy Consumption	57
4.6	Power Consumption	60
4.7	Results Discussion	64
5	CONCLUSION	67

BIBLIOGRAPHY 69

APPENDIX 75

APPENDIX A – L2 CACHE ACCESSES RESULTS 77

1 INTRODUCTION

Multi-core processors have now become mainstream for both general-purpose and embedded computing (SEZNEC, 2016). In addition, manycore processors are being extensively used in embedded and high-performance computing systems. This transition created a disruptive change: substantial performance gains for applications can no longer be achieved without modifying the underlying source code. The increase in processing power of small processors, the popularization of artificial intelligence, and the growth of IoT will mean that small devices can (and need to) process larger volumes of information. This is called edge computing, where applications do local processing, without having to send all information to be processed in a data center, for example. The trend is that all devices present on this edge will have a multi-core architecture capable of doing parallel processing (ROMAN; LOPEZ; MAMBO, 2018). Sensing devices, for example, often can not wait for the response time of a server (which tends to increase along with the increase of these devices) and will need to make faster decisions. This way, although there are many kinds of applications that cannot be executed in parallel because their nature, many of the future applications should be radically different, they must be parallel.

In the past, the major goal of parallelizing an application was to achieve maximum performance. However, today there is also a growing concern about the energy consumption of these parallel applications. There are two main fronts that motivate this concern. The first one is that many countries are limiting the use of existing supercomputers because of their high energy consumption, such as United States, Japan, and China (Bourzac, Katherine, 2017). The other one is that embedded processors tend to outperform server and general-purpose processors in the multi-core processor market in the following years. Currently, general-purpose processors represent a 53% share of the market, while embedded processors and mobile SoC MPUs account for 47% (IC-INSIGHTS, 2018). This way, in addition to the power limitations imposed on supercomputers, we will soon have this concern for the vast majority of processors.

The embedded processor industry have reshaped by some factors, among them, the slowing of Moore's Law and the realization that most devices will emphasize price and power rather than speed. Most embedded applications do not need leading-edge performance, but they do want the best power efficiency (the ratio of the output power divided by the input power) and the lowest power consumption (BOLARIA; HALFHILL, 2017). Therefore, it is necessary to find efficient ways to identify the best trade-offs between performance and power consumption of these applications.

The performance increase is reached with faster multiple parallel processors. Parallel computing aims to use multiple processors to execute different parts of the same program simultaneously (RAUBER; RÜNGER, 2010). However, processors

should be able to exchange information at a certain point in the execution time. While tasks parallelism makes it possible to increase the performance, the use of more processors and the need for communication among them can lead to an increase in energy consumption.

The parallelism can be explored with different Parallel Programming Interfaces (PPIs), each one having specific characteristics in terms of synchronization and communication. In addition, the performance gain may vary according to processor architecture and hierarchical memory organization, communication model of each PPI, and also with the complexity, problem type, and other characteristics of the application.

So far, software developers have been hesitant to burden themselves with the difficult and error-prone task of parallelizing their programs. This difficulty already begins when it is necessary to choose which PPI or programming language should be used to parallelize a particular application. As parallelism is becoming more popular and a necessity for many applications, more ways to use it will be developed. An example of this is the Threading Building Blocks (TBB) which is a C++ template library recently developed by Intel® for parallel programming on multi-core processors (INTEL, 2018). Another example is the InKS (EJJAOUANI et al., 2018), which is a new programming model to decouple performance from algorithms in HPC codes.

Each PPI has its own characteristics that can behave in different ways in each architecture, according to the application that will run in that system. Finding out which type of parallelization provides the best trade-off between performance and power consumption is still a manual task based on previous experiences or studies. The question here is: how much performance gain outweighs the extra energy expenditure? A benchmark which provides different types of pseudo-applications implemented in several Parallel Programming Interfaces could be useful to answer this question. However, we did not find a benchmark offering a good set of pseudo-applications, fully parallelized, using multiple PPIs and different models of communication between tasks. The most commonly used parallel benchmarks have only partial parallel sets using more than one PPI (e.g. NPB, PARSEC, SPEC) and many researchers need to find other alternatives to fill this lack, as we expose in our related work.

To fill this gap, this work proposes a new benchmark using a set of 11 pseudo-applications developed with the purpose of evaluating the performance and energy consumption in multi-core architectures. These pseudo-applications were parallelized using parallel tasks and classified according to different criteria in previous studies (LORENZON et al., 2015; LORENZON; CERA; BECK, 2015; LORENZON, 2014; GARCIA, 2016). These studies have shown that these pseudo-applications have characteristics distinct enough to represent different scenarios, such as different amount of communication among tasks, and high and low usage of CPU and memory. This work is a continuation of those previous studies that have been cited, from Lorenzon

(2014) to Garcia (2016).

1.1 Objectives

In this work we use a set of 11 parallel pseudo-applications that were used to measure performance and power consumption in multi-core architectures in previous studies. These applications were implemented for specific contexts and architectures and were not ready to be available as a benchmark for the general users. They presented very poor usability and most of them did not work properly with different or new architectures or compilers. These 11 pseudo-applications are implemented in PThreads, OpenMP, MPI-1, and MPI-2 (spawn) PPIs and these PPIs were chosen because they are compatible with most multi-core architectures.

The main objective of this work is to do the first move to turn this set of parallel algorithms into a real benchmark. A benchmark to evaluate the performance and power consumption of different parallel programming interfaces. To achieve this goal we re-coded the pseudo-applications, fixed the errors, and added some new features, such as dynamic parameters, small running interface (bash scripts), results generation, configuration files, and put the set available in a public repository online. We also conduct case studies where we evaluate different parts of the system that can improve or worsen the impact of the pseudo-applications. First we evaluated the cache accesses, to see how memory accesses impact on the performance and power consumption. We also measure the branches to evaluate the branch predictor, and the floating-point operations to evaluate the floating-point unit performance.

In addition, we measured the performance and energy consumption and used them to obtain power consumption (energy consumed over time). It will allow us to observe the scalability of pseudo-applications in the architecture used and the impact of their characteristics on power consumption. To improve the understanding of the results, in the end, we summarize these results in a table which shows an overview of them. The contributions of this work are as follows:

- The set of pseudo-applications did not have a complete formal description. In our work we describe the equations, data structures and algorithms implemented by each pseudo-application;
- The source code of the pseudo-applications had many static parameters, and many errors and bugs. In this way, the pseudo-applications presented very poor usability. So we rewrote many parts of the source code and fixed the problems, increasing portability and usability;
- We extend the work from Lorenzon (2014) on performance, power consumption, and memory access of pseudo-applications by doing a case study of them in

a more robust architecture and also analyzing Floating-Point Operations and Branches.

1.2 Structure of the Work

The remainder of this work is organized as follows. In the chapter 2 we present the PPIs in which the pseudo-applications are implemented, the way we will evaluate the pseudo-applications, and the related work, where we compare our work with similar benchmarks. The chapter 3 presents the set of pseudo-applications and the techniques used to parallelize them, bringing more details about the historic of classifications. At the end of this Section we also show all the improvements and modifications we had apply on the source-codes.

In chapter 4, section 4.1 shows how our experiments were structured and brings some information to a better understanding of the results. section 4.2 presents the results of cache memory accesses, section 4.3 presents the branches results, and FLOPs results are presented in section 4.4. The section 4.5 and section 4.6 brings results of performance and energy, and power consumption respectively. In the section 4.7 we discuss and summarize the results. Finally, in chapter 5 we draw our final conclusions and future works.

2 BACKGROUND

In computing, there has always been a need to compare, classify, or test the performance of different systems for a variety of purposes. With the evolution of these systems, it became impracticable to do this task through simple tests and comparison of the specifications of each architecture (GRAY, 1992). At this point, it was necessary to create a way to standardize a method that would meet this growing need. To supply this need were created benchmarks, which are now used in various areas of computing.

One of the areas that needs to be constantly testing efficiency and comparing system performance is the High Performance Computing area. In this area we aim to increase the performance of a given application, usually through the use of parallel programming. According to Rauber e Runger (2010), the parallelization consists of dividing the tasks of an application and executing them concurrently in order to reduce its total execution time. Its use becomes essential in scientific applications that require high computational power, such as calculations of weather forecasting, calculations of DNA and genome sequences, among other different applications. More recently, with the popularization of multi-core architectures, general-purpose applications (image and sound filters, graphic editors, internet servers, etc.) have also taken advantage of parallel programming.

Parallellising an application may not be an easy task. For this, PPIs are able to make this process less arduous for the programmer. However, each interface has its own characteristics and specifications that must be carefully studied to implement a parallel application. The programmer must follow some basic care, paying attention to which parts of the code will be executed simultaneously and, depending on the interface, must also define how the data will be communicated and synchronized.

This chapter gives a brief description of each parallel programming interface in section 2.2. Section 2.3 briefly discusses how hardware counters are used by similar works. In section 2.4 related work and benchmarks are presented, where a comparison of these benchmarks is made with the benchmark proposed in this work.

2.1 Benchmarks

The evolution of computational architectures has made comparing the performance of different computing systems, only looking at their specifications a difficult task. Historically, manufacturers used to classify the performance of their systems using a variety of different metrics. This generated a lot of confusion among consumers and often this information might not be credible. To solve this problem, a group of manufacturers came together to develop standardized test sets for measurements on these systems, allowing these results to be compared between different (SPEC, 2018) architectures. This process was also intended to educate consumers about the performance of their

products. It was in this context that benchmarks emerged.

In computing, benchmarking is the action of comparing the relative performance of an object or product by running a computer program (GRAY, 1992). To extract correct data on different products and objects, in order to compare them uniformly, a series of standard tests and tests are performed. The term benchmark is commonly used to define the programs developed to run this testing process. Along with this, the term benchmarking is associated with the process of evaluating the characteristics and performance of a computer hardware, such as the performance of the FPU of FPU.

In this work, we will study a benchmark to measure the performance and energy consumption of different PPI in multi-core architectures. These PPIs work in any multi-core architecture. The benchmark currently consists of 11 parallel applications implemented using four different PPIs, PThreads, OpenMP, MPI-1 and MPI-2.

2.2 Parallel Programming Interfaces

There are several computational models used in parallel computing, such as data parallelism, shared memory, message passing, and operations in remote memory. These models differ in several aspects, such as whether the available memory is locally shared or geographically distributed, and volume of communication (GROPP; LUSK; THAKUR, 1999). In this work, the set of pseudo-applications were implemented using two communication models with the four PPIs: PThreads, OpenMP, MPI-1, and MPI-2.

The OpenMP pattern consists of a series of compiler directives, function libraries, and a set of environment variables that influence the execution of parallel programs (RAUBER; RÜNGER, 2010). These directives are inserted into the sequential code and the parallel code is generated by the compiler from them. This interface operates on the basis of the thread fork-join execution model.

Different from OpenMP, in POSIX Threads (PThreads) the parallelism is explicit through library functions. That is, the programmer is responsible for managing *threads*, workload distribution, and execution control (BUTENHOF, 1997). PThreads comprises some subroutines that can be classified into four main groups: thread management, mutexes, condition and synchronization variables.

MPI-1 standard API specifies point-to-point and collective communications operations, among other characteristics. In a program developed using MPI-1, all processes are statically created at the start of the execution. So, the number of processes remains unchanged during program execution. At the start of the program, an initialization function of the execution environment MPI is executed by each process. This function is `MPI_Init()`. A process MPI is terminated by calling the function `MPI_Finalize()`. Each process is identified by a rank, which is the ID of a process inside its group.

Applications deployed with MPI-2 can begin the execution with a single process. Then, the primitive `MPI_Comm_spawn()` can be used for the creation of processes

dynamically. A process of an MPI application, which will be called by the parent, invokes this primitive. This invocation causes a new process, called a child, to be created, which not need to be identical to the parent. After creating a child process, it will belong to an intra-communicator and the communication between parent and child will occur through this intra-communicator. In the child process, the execution of the function `MPI_Comm_get_parent()` is responsible for returning the intercom that links it to the parent. In the parent process, the inter-communicator that binds the child is returned in the execution of the function `MPI_Comm_spawn()`.

2.3 Hardware Performance Counters

To evaluate the pseudo-applications, this work uses Hardware Performance Counters (HPCs) which are available on most current processors. Essentially, hardware performance counters are a tool that is used by software engineers to measure performance and for allowing software vendors to enhance their code such that performance improves (UHSADEL; GEORGES; VERBAUWHEDE, 2008). Hardware performance counters are exposed to the user space on commercial hardware. The performance counters monitor CPU, memory, network and I/O by counting specific events such as cache misses, pipeline stalls, floating point operations, bytes in/out, bytes read/write, and also information about energy consumption (WU; TAYLOR, 2016).

Compared to software profilers, hardware counters provide low-overhead access to a wealth of detailed performance information about the CPU's functional units, caches, main memory etc (UHSADEL; GEORGES; VERBAUWHEDE, 2008). Another benefit of using them is that no source code modifications are needed in general. However, the types and meanings of hardware counters vary from one kind of architecture to another due to the variation in hardware organizations.

Regarding to energy consumption, there are many other previous work on power modeling and estimation that are based on HPCs (LIVELY et al., 2014) (LIVELY et al., 2012) (SONG et al., 2013) (RODRIGUES et al., 2013) (CHETSA et al., 2014). These approaches used HPCs to monitor the system components such as CPU, memory, disk, I/O, and GPU. The methods then correlated these performance counters with the power consumed by each system component to derive a power model for each system component.

2.4 Related Work

Many studies that want to perform comparisons between PPIs need to modify and re-code applications from a parallel benchmark or to use more than one benchmark. It is demonstrated by some previous works. There is a work (ISIDRO-RAMIREZ; VIVEROS; RUBIO, 2015) in which the authors implemented some HPL benchmark

applications in PThreads and Java Threads in order to evaluate the power consumption of these PPIs in an ARM Cortex-A9 processor. Also, in another study Popov et al. (2015) needed to re-implement a set of NAS applications in OpenMP to test the Parallel Codelet Extractor and REplayer (PCERE). In order to make an evaluation between OpenMP and MPI optimized to work with shared memory on a single node (MPI-3), Jain et al. (2018) had to use different benchmarks for the experiments. In another work, Dosanjh et al. (2019) had to modify the MPI Sandia Microbenchmarks (SMBs) (DOEFLER; BARRETT, 2009) to use multithreading. These are some examples that it would be interesting to have a single standard benchmark in order to generate useful data for comparing PPIs in different computational systems.

All the work mentioned above had a major problem to solve and it was necessary to implement or modify existing benchmarks to solve it. However, there is a second group of related work in which building a new benchmark class is part of the major goal. These second group aims to reimplement a whole set of applications, or part of it, into another PPI or programming language. To fill the lack of portability to C++ language of the NAS legacy codes, Griebler et al. (GRIEBLER et al., 2018) described the NAS Kernel applications in C++. They also implemented these new codes using Intel TBB, OpenMP, and FastFlow interfaces in order to compare the performance achieved by different parallel implementations. The PARSEC benchmark originally has parallel implementations with high-level abstraction, it uses pragma-based PPIs such as OpenMP and Intel TBB. Thus, Danelutto et al. (DANELUTTO et al., 2017) identified the lack of low-level mechanisms and implemented P³ARSEC, a subset of this benchmark using POSIX Threads.

The third group of related work is the already consolidated parallel benchmarks. Through a bibliographic study, we searched for benchmarks that have similar purposes and the same target architectures of the benchmark proposed in this work. Therefore, we have considered benchmarks that provide a set of parallel applications for embedded or general-purpose multi-core architectures. In this way, we identify and present in Subsection 2.4.1 the following benchmarks: ALPBench, PARSEC, ParMiBench, SPEC, Linpack, NAS, and Adept Project.

2.4.1 Similar Benchmarks

ALPBench (University of Illinois, 2018) consists of a set of parallelized complex media applications gathered from various sources and modified to expose thread-level and data-level parallelism. It consists of 5 applications parallelized with PThreads. This benchmark is focused on general-purpose processors and has an open source license.

PARSEC (Princeton Application Repository for Shared-Memory Computers) is an open source benchmark suite (Princeton University, 2018). It consists of 11 applications, some parallelized using OpenMP, or PThreads or Intel TBB. The suite

focuses on emerging workloads and was designed to contain a diverse selection of applications that are representative of next-generation shared-memory programs for chip-multiprocessors.

ParMiBench is an open source benchmark that specifically serves to measure performance on embedded systems that have more than one processor (IQBAL; LIANG; GRAHN, 2010). This benchmark organizes its applications into four categories and domains: industrial control and automotive systems, networks, office devices, and security. Its set consists of 7 parallel applications implemented using PThreads.

SPEC (SPEC, 2018) is a closed source benchmark but offers academic licenses. This benchmark is intended for general purpose architectures, but is subdivided into several groups with specific target architectures, and can be used for several purposes, such as Java servers, file systems, high-performance systems, CPU tests, among others. We consider the following groups of SPEC: SPEC MPI2007, SPEC OMP2012, and SPEC Power. They were chosen because they are the ones who have at least one kind of parallel applications on their sets. SPEC MPI2007 is a set of 18 applications deployed in MPI focused on testing high-performance computers. SPEC OMP2012 uses 14 scientific applications implemented in OpenMP, offering optional energy consumption metrics based on SPEC Power. Finally, SPEC Power tests the energy consumption and performance of servers using CPU/Memory-Bound applications implemented in C and Fortran.

HPL consists of a software package that solves arithmetic dual floating-point precision random linear systems in high-performance architectures (PETITET,). It runs a testing and timing program to quantify the accuracy of the solution obtained, as well as the time it took to compute. HPL code is open and consist of 7 applications form a collection of subroutines in Fortran, mostly CPU-Bound. Parallel implementations use MPI. HPL is the benchmark that makes up the so-called *High-Performance Computing Benchmark Challenge*, which is a list of the 500 fastest high-performance computers in the world.

The NAS Parallel Benchmarks (BAILEY et al., 1991) is a small set of open source programs that serve to evaluate the performance of parallel supercomputers. The benchmark is derived from physical applications of fluid dynamics and consists of four cores and three pseudo-applications. It is an open source benchmark and the pseudo-applications are implemented with MPI and OpenMP. Some applications are also implemented in HPF, UPC, Java, Titanium, TBB etc.

The Adept Benchmark (Adept-Project, 2018) is used to measure the performance and energy consumption of parallel architectures. Its code is open and is divided into 4 sets: Nano, Micro, Kernel and Application. The Micro suite, for example, consists of 12 sequential and parallel applications with OpenMP, focusing on specific aspects of the system, such as process management, caching, among others. On the other hand,

Table 1: Comparison of our benchmark with the similar ones

Rating criteria	ALPBench	PARSEC	ParMiBench	SPEC	HPL	NAS	Adept	Our benchmark
Number of applications	5	11	7	14-18	7	7	10-12	11
Number of PPIs	1	3	1	2	1	2	3	4
Number of communication models	1	1	2	1	1	2	2	2
Set of applications implemented in multiple PPIs						X		X
Open source	X	X	X		X	X	X	X

Source: by the author.

the Kernel set has 10 applications implemented sequentially and parallel with OpenMP, MPI and one of them in UPC (Unified Parallel C).

2.4.2 Comparison Among Benchmarks

The benchmark addressed in this work consist of 11 pseudo-applications implemented in C and their complexities range from $O(n)$ to $O(n^3)$. All pseudo-applications are parallelized in 4 PPIs: PThreads, OpenMP, MPI-1 and MPI-2. These PPIs are the target of this work because they are the most widespread in the academic field and also because they are supported by most multi-core architectures, both embedded and general purpose. Therefore, the purpose of this benchmark is to provide the user with a tool to evaluate the performance and energy consumption of different PPIs in multi-core architectures.

We analyze the main characteristics of the related parallel benchmarks and compare to the benchmark we propose in this work in Table 1. In regarding to the benchmarks, some use only one PPI while others use more than one. However, some of those who use more than one PPI do not have the whole set of applications paralleled by all PPIs. They implement parts of the set with one PPI and other parts with another PPI. NAS, for example, uses several other PPIs. However, virtually no pseudo-application is implemented in all PPIs. Many of them are not supported by any multi-core architecture (e.g. ARM Architecture, as showed by Lorenzon (2014)). Three of the benchmarks use PThreads, five of them use OpenMP, and four use MPI. ALPBench also uses Intel TBB and Adept uses UPC.

Thus, even if some of these benchmarks implement three different PPIs, none

of them allow an efficient comparison between these PPIs and between different communication models. Also, they do not exploit the parallelism with dynamic process creation that MPI-2 offers. In this way, we do not find any other benchmark that uses different PPIs, different communication models and a completely parallelized set of applications. The exception is the NAS, but it only offers two PPIs. Therefore, none of them meets the objective of comparing parallel programming interfaces, which is the main objective of the benchmark we are proposing in this work.

3 BENCHMARK PSEUDO-APPLICATIONS

This chapter presents the pseudo-applications¹ that compose the benchmark in detail, as well as the methods and techniques of parallelization adopted. In section 3.1 are presented the sequential algorithms and their main characteristics. The strategies used in the parallelization of each of the pseudo-applications are detailed in section 3.2. Finally, section 3.3 shows how these pseudo-applications have been used and classified since their implementations.

3.1 Pseudo-applications Description

These pseudo-applications were developed with the purpose of establishing a relationship between performance and energy consumption in multi-core architectures. The pseudo-applications described here were based in the ones originally developed by Lorenzon (2014). The main features of each are presented in this section. Each subsection below presents the pseudo-applications in their sequential versions and shows details about their complexities, their algorithms, and formulas.

3.1.1 Numeric Integration

The integral of a function was originally created to determine the area under a curve in the Cartesian plane by means of approximation techniques. The process of calculating the integral of a function is called integration (STEWART, 2001). This pseudo-application integrates the function $f(x)$ (3.3) in the $[a, b]$ interval. The basic method involved in this approximation is called numerical quadrature and is expressed by the equation 3.1, where α_i is real coefficient (weight of the function) and x_i , is a sampling point of $[a, b]$ defined by equation 3.2. The algorithm 1, which has a complexity of $O(n)$, represents this pseudo-application. It is ideal for evaluating the FPU because of its low complexity and high number of floating point operations.

$$\int_b^a f(x)dx \simeq \sum_{i=0}^n \alpha_i f(x_i) \quad (3.1)$$

$$x_i = \frac{(n - i - 1) \cdot (a + (i \cdot b))}{(n - 1)} \quad (3.2)$$

¹ Source codes available at <https://github.com/adrianomg/PAMPAR>

$$f(x) = \frac{50}{\pi \cdot (2500 \cdot (x^2 + 1))} \quad (3.3)$$

Algorithm 1: Numeric Integration

```

1 begin
2   Set the number of iterations
3   Set the limits  $a$  e  $b$ 
4   for (From zero to number of iterations) do
5     Calculate  $f(x)$  from  $a$  to  $b$ 
6   end
7 end

```

3.1.2 PI Calculation

Pi (π) is an irrational value that establishes a numerical relationship between the perimeter of a circumference and its diameter (ROY, 1990). Pi is commonly represented with 52 decimal places when it is necessary to perform high precision calculations. However, this accuracy can be increased by computational algorithms. There are several methods for calculating the value of Pi, these involve approximations, successive approximations, and infinite series of sums, multiplications, and divisions. In this pseudo-application, we used the Gregory-Leibniz method that is established by the equation 3.4 (ANDREWS; ROY, 1999).

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n + 1} = \frac{\pi}{4} \quad (3.4)$$

Although the Gregory-Leibniz method is considered inefficient for a few iterations, the precision increases as the iterations of n are increased. It generates Pi with an accuracy of 5 decimal places after 500 thousand iterations and 10 decimal places with 5×10^9 iterations (BORWEIN; BORWEIN; DILCHER, 1989). The implementation of this algorithm is quite simple. It starts by defining the number of iterations to be calculated and in the sequence a loop applies the equation 3.4. The output consists of an approximate value for Pi. This algorithm is defined by the pseudo-code 2. It has complexity of $O(n)$ and can represent applications with low complexity with average number of floating-point operations.

Algorithm 2: Pi Calculation

```

1 begin
2   | Set the number of iterations
3   | for (From zero to number of iterations) do
4   |   | Apply Gregory-Leibniz method
5   | end
6 end

```

3.1.3 Dot Product

The dot product is an operation between two vectors whose result is a real number (also called scalar). The dot product is used in Euclidean geometry and is a special case of inner product (BANCHOFF; WERMER, 2012). This pseudo-application calculates the dot product between a sequence of ordered values and another one of reverse ordering. This algorithm can be seen in the pseudo-code and is defined by the equation 3.5. It can represent applications with simple integer calculations and low complexity ($O(n)$).

$$a \cdot b = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n \quad (3.5)$$

Algorithm 3: Dot Product

```

1 begin
2   | Set the number of iterations
3   | for (From zero to number of iterations) do
4   |   | Calculate the dot product
5   | end
6 end

```

3.1.4 Harmonic Sums

The Harmonic Sums or Harmonic Series is a finite series that calculates the sum of arbitrary precision after the decimal point (GOLDSTON; YILDIRIM, 2001). This mathematical sequence has this name because it has similar proportions to the wavelengths of a vibrating string. This sequence diverges slowly, as can be seen from the equation 3.6. This algorithm represents pseudo-applications with average number of branches and high number of FLOPs.

$$\sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} \quad (3.6)$$

This pseudo-application contains a vector that stores the precision sum for each i of the algorithm. Its main processing consists of successive operations of division and module calculation for each value of i . The output consists of the sum of these operations. This algorithm is described in the pseudo-code 4 and has complexity of $O(n \times d)$, where n is the number of iterations and d is the size of the vector.

Algorithm 4: Harmonic Sums

```

1 begin
2   Set the number of iterations
3   for (From zero to number of iterations) do
4     for (Iterate through the vector) do
5       Calculate the Harmonic Sum
6       Store the precision sum
7     end
8   end
9   for (Iterate through the vector) do
10    Decimal precision adjustment
11  end
12 end

```

3.1.5 Odd-Even Sort

Odd-Even sort is a sorting algorithm that compares all the indexed (odd-even) pairs of adjacent elements in the list. If a pair is in the wrong order (the first is greater than the second), it swaps the elements. The next step repeats this for the indexed (even-odd) pairs of adjacent elements. It then toggles between (odd-even) and (even-odd) steps until the list is sorted (KNUTH, 1998). The output of this algorithm is the ordered input vector. Its implementation is represented by the algorithm 5 and is based on bubble-sort. Its complexity is $O(n^2)$ and this pseudo-application is suitable for evaluating the PPI performance under the presence of a high number of branches.

3.1.6 Discrete Fourier Transform

The discrete Fourier transform (DFT) converts a finite sequence of equally-spaced samples of a function into an equivalent-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency (SMITH; STEVEN et al., 1997). This way, the resulting frequency values are integer multiples of a fundamental frequency whose period corresponds to the length of the sampling interval. This function is widely used in digital signal processing and is defined by the equation 3.7, where: N is the number of samples; n

Algorithm 5: Odd-Even Sort

```

1 begin
2   Set number of elements ( $n$ )
3   for (Iterate through the vector) do
4     for (from 0 to  $n - 1$  step 2) do
5       if (Element at current position is higher than element at next position)
6         then
7           Swap elements
8         end
9       end
10      for (From 1 to  $n - 1$  step 2) do
11        if (Element at current position is higher than element at next position)
12          then
13            Swap elements
14          end
15        end
16      end
17   end

```

is the current sample (from 0 to $N - 1$); x_n is the signal level at time n ; k is the current frequency (from 0 Hz to $N - 1$ Hz); and X_k is the level of the frequency k on the signal.

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot \left(\cos \left(2\pi k \frac{n}{N} \right) + j \sin \left(2\pi k \frac{n}{N} \right) \right), \quad n \in \mathbb{Z} \quad (3.7)$$

The algorithm has two main loops to calculate the sum of the real and imaginary terms that compose the equation. These calculations are performed separately and summed at the end, as can be seen in the pseudo-code 6. The algorithm has $O(n^2)$ complexity and is suitable for representing CPU-bound applications with high number of FLOPs.

3.1.7 Turing Ring

Alan Turing analyzed the interaction of two chemicals in a ring of cells using two differential equations coupled to describe a prey/predator system (TURING, 1952). It is a space system in which predators and prey interact in the same environment. The system simulates the iteration and evolution between prey and predator through the use of two differential equations 3.8 and 3.9. $r_{x,y}$ are the birth rates, $c_{a,b}$ represent local interactions, $\mu_{x,y}$ are migration rates between neighbouring cells and the predator and prey populations in location i are represented by X_i, Y_i . Evolution is defined according

Algorithm 6: Discrete Fourier Transform

```

1 begin
2   for (Iterate through the vector) do
3     for (Iterate through the vector) do
4       Calculate the real part of the function
5     end
6     for (Iterate through the vector) do
7       Calculate the imaginary part of the function
8     end
9     Sum the two parts and store the result in the solution vector
10  end
11 end

```

to the neighbouring cells (PAUDEL; AMARAL, 2011).

$$\frac{dX_i}{dt} = X_i(r_x + c_{xx}X_i + c_{xy}Y_i) + \mu_x(X_{i+1} + X_{i-1} - 2X_i) \quad (3.8)$$

$$\frac{dY_i}{dt} = Y_i(r_y + c_{yx}X_i + c_{yy}Y_i) + \mu_y(Y_{i+1} + Y_{i-1} - 2Y_i) \quad (3.9)$$

The algorithm consists of an input matrix containing in each position the number of predators and prey, which will interact for n evolutions producing an output (matrix) society. This algorithm is represented by the pseudo-code 7 and has complexity of $O(m \times n^2)$, where m is the number of evolutions of the society and n the size of the matrix. This application has a high number of branches.

Algorithm 7: Turing Ring

```

1 begin
2   Set the number of evolutions
3   for (From zero to the number of evolutions) do
4     for (Iterate through the matrix rows) do
5       for (Iterate through the the matrix Columns) do
6         Simulate the evolution by applying the differential equations Store
          the solution in an auxiliary matrix
7       end
8     end
9     for (Iterate through the matrix rows) do
10      for (Iterate through the matrix columns) do
11        Store the auxiliary array data in the original array
12      end
13    end
14  end
15 end

```

3.1.8 Dijkstra

The Dijkstra algorithm, designed by the Dutch computer scientist Edsger Dijkstra, solves the shortest path problem in a directed or non-directed graph with non-negative weight edges. Given a source vertex in the graph, the algorithm finds the least cost path between this vertex and any other vertex (DIJKSTRA, 1959).

Our implementation uses an adjacency matrix of size $N \times N$. Considering n being the number of vertices, the complexity of the algorithm is $O(n^2)$. However, our pseudo-application considers the smallest path from n to n vertices, bringing its total complexity to $O(n^3)$. The output of this pseudo-application is a vector containing the minimum distance between each of the vertices as output. It is represented in the pseudo-code 8. This pseudo-application do only integer operations and has an average number of branches.

Algorithm 8: Dijkstra

```

1 begin
2   for (Travel vertices) do
3     for (Iterate through the distances vector) do
4       Set the maximum possible distance to all vertices
5       Set each vertex as unvisited
6     end
7     Set NULL distance to the source vertex
8     for (Iterate through the vertices) do
9       for (Iterate through the vertices) do
10        if (If the vertex is a neighbor and has not yet been visited) then
11          Update current vertex in the array
12        end
13      end
14      Set current vertex as visited for (Iterate through the distances vector)
15      do
16        if (Check if it is the shortest path) then
17          Update the path in the distances vector
18        end
19      end
20    end
21 end

```

3.1.9 Jacobi Method

Jacobi method is a classical method dating back to the late eighteenth century. Iterative techniques are rarely used to solve linear systems of small dimensions, since the time required to obtain a minimum precision exceeds that required by direct techniques like the Gaussian Elimination (BURDEN; FAIRES; REYNOLDS, 2001). However, for

large systems, with a high percentage of null inputs (sparse systems), these techniques appear as more efficient alternatives. Considering a linear system $Ax = b$, where A is the matrix of the coefficients $m \times n$, x is the vector of variables, x^k is the k^{th} iteration of x , and b the vector of the constant terms. The objective of the method is to find an approximate result for x through the convergence of vectors (PRESS et al., 2007). The equation 3.10 represents this method.

The Jacobi method is an algorithm for determining the solutions of a diagonally dominant system of linear equations (BURDEN; FAIRES; REYNOLDS, 2001). It is a classical method dating back to the late eighteenth century. Iterative techniques are rarely used to solve linear systems of small dimensions, since the time required to obtain a minimum precision exceeds that required by direct techniques like the Gaussian Elimination (BURDEN; FAIRES; REYNOLDS, 2001). However, for large systems, with a high percentage of null inputs (sparse systems), these techniques appear as more efficient alternatives. Considering a square linear system $Ax = b$, where A is the $n \times n$ coefficients matrix, x is the vector of variables and b is the vector of the constant terms, the objective of the method is to find an approximate result for x through the convergence of vectors (PRESS et al., 2007) by using equation 3.10. x^k is the k^{th} iteration of x .

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), i = 1, 2, \dots, n. \quad (3.10)$$

The program consists of a repetition loop in which each iteration scrolls the target matrix and applies the Jacobi method. The algorithm performs operations on a single matrix and this generates dependence on data. To work around this problem, an auxiliary matrix is used, which will contain the correct value at the end of each computation. This algorithm is described in the pseudo-code 9. It is a $O(n^3)$ memory-bound pseudo-application and has few branches and FLOPs.

3.1.10 Matrix Multiplication

There are several methods for performing matrix multiplication. The pseudo-application that we use in our work implements the most common method (PRESS et al., 2007). It consists of the multiplication of matrix A rows elements by the matrix B columns elements, storing the result in a matrix C . This is an $O(n^3)$ algorithm and it represents applications with high number of memory accesses. It is represented by the pseudo-code 10.

$$C = (AB)_{ij} = \sum_{k=1}^n a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \dots + a_{in} b_{nj}. \quad (3.11)$$

Algorithm 9: Jacobi Method

```

1 begin
2   Set the number of iterations
3   for (From zero to number of iterations) do
4     for (Iterate through the matrix rows) do
5       for (Iterate through the matrix columns) do
6         Apply Jacobi Method
7         Store the result in an auxiliary matrix
8       end
9     end
10    Copy the data from the auxiliary matrix to the solution matrix
11  end
12 end

```

Algorithm 10: Matrix Multiplication

```

1 begin
2   for (Iterate through the rows) do
3     for (Iterate through the columns) do
4       for (Iterate through the elements) do
5         Multiply rows elements and columns elements
6         Store the result in the solution matrix
7       end
8     end
9   end
10 end

```

3.1.11 Gram-Schmidt

The Gram-Schmidt algorithm is a method for orthonormalising a set of vectors in an inner product space (CHENEY; KINCAID, 2009). It is a $O(n^3)$ algorithm and is representative for applications with high number of memory accesses, branches, and FLOPs. This algorithm receives a finite and linearly independent set of vectors $S = u_1, \dots, u_n$ and returns an orthonormal set $S' = v_1, \dots, v_n$ which generates the same initial subspace S . It performs a series of projection operations between the input

vectors. Each vector v_i is calculated as:

$$\begin{aligned}
 v_1 &= u_1 \\
 v_2 &= u_2 - \frac{\langle u_2, v_1 \rangle}{\langle v_1, v_1 \rangle} v_1 \\
 v_3 &= u_3 - \frac{\langle u_3, v_1 \rangle}{\langle v_1, v_1 \rangle} v_1 - \frac{\langle u_3, v_2 \rangle}{\langle v_2, v_2 \rangle} v_2 \\
 &\vdots \\
 v_n &= u_n - \frac{\langle u_n, v_1 \rangle}{\langle v_1, v_1 \rangle} v_1 - \frac{\langle u_n, v_2 \rangle}{\langle v_2, v_2 \rangle} v_2 - \dots - \frac{\langle u_n, v_{n-1} \rangle}{\langle v_{n-1}, v_{n-1} \rangle} v_{n-1}
 \end{aligned} \tag{3.12}$$

The inner product between u and v can be calculated as:

$$\langle v, u \rangle = v_1 u_1 + \dots + v_n u_n \tag{3.13}$$

The algorithm has as its input an array of vectors. After successive computations in n steps a new output matrix is generated. These steps are computed one after another, where the computation of $n + 1$ depends on the result of n . This algorithm is represented by the pseudo-code 11 and its complexity is $O(n^3)$.

Algorithm 11: Gram-Schmidt

```

1 begin
2   for (Iterate through the matrix rows) do
3     for (Iterate through the matrix columns) do
4       Calculate the inner product
5     end
6     for (Iterate through the matrix columns) do
7       Calculate the resulting vector norm
8     end
9     for (Iterate through the matrix columns) do
10      for (Iterate through the matrix elements) do
11        Normalize the vector and store in the solution matrix
12      end
13    end
14  end
15 end

```

3.2 Parallelizing the Pseudo-Applications

Parallelize a sequential program can be done in several ways. However, inappropriate techniques can negatively impact the performance of an algorithm. To

minimize this problem, all parallel implementations in this work were based on statements from (FOSTER, 1995; BUTENHOF, 1997; GROPP; LUSK; THAKUR, 1999; RAUBER; RÜNGER, 2010). Rauber e Rüniger (2010) propose that the parallelization must be done in a systematic way. According to them, there are three fundamental steps for the parallelization of a sequential algorithm, which are: computation decomposition; assigning tasks to processes/threads; and mapping processes/threads into physical processing units.

The decomposition of the computation and assignment of tasks to processes/threads occurred explicitly in the parallelization with PThreads and MPI-1 and -2, in order to obtain the best workload balancing. Also, message exchange functions between processes were included, as well as the dynamic creation of processes in MPI-2. For parallelization with OpenMP, parallel loops with thin and coarse granularity were used. According to (FOSTER, 1995; RAUBER; RÜNGER, 2010), this technique is most appropriate for parallelizing algorithms that perform iterative calculations and traverse contiguous data structures (e.g. matrix, vector, etc.). For each data structure, a specific parallelization model was adopted.

3.3 Historic of Classifications

The set of pseudo-applications that compose the benchmark have already been investigated in previous works. The pseudo-applications were used in these works to analyze performance and energy consumption on embedded systems and general purpose processors. In (LORENZON; CERA; BECK, 2015; LORENZON, 2014; LORENZON; CERA; BECK, 2014) the authors classified the pseudo-applications between CPU-Bound, Weakly Memory-Bound and Memory-Bound, according to the following criteria:

1. **Reads/writes to memory** - represents the number of accesses to the shared and private memory addresses of the processor, considering read and write operations for each pseudo-application;
2. **Data dependence** - means that at least one thread/ process can only start its execution when the computation result of one or more threads/ processes is over. This shows the existence of communication between threads/processes;
3. **Synchronization points** - determine that at certain times during the execution of an pseudo-application, all threads/processes will need to be synchronized before a new task starts.
4. **Thread-Level Parallelism** - shows how busy the processor is during pseudo-application execution;

5. **Communication rate** - represents the volume of communication required by threads/processes during pseudo-application execution.

In (LORENZON et al., 2015), the authors used the number of data exchange operations as a criterion for classification. In the PPI target, these operations represent barriers, locks/unlocks and threads/processes creation or termination. Using this criterion, the pseudo-applications were divided between High and Low Communication. The main problem with both classifications is that they were not done uniformly with all pseudo-applications. The first classification used some pseudo-applications with a specific interface, while another configuration was used to do a second classification. Hence, the four PPIs were never evaluated together. TLP, for example, was collected only for 9 pseudo-applications and using only PThreads.

Already using the first criterion (access read/write to shared memory), the pseudo-applications were classified between CPU-Bound and Memory-Bound. However, this type of data does not indicate how much CPU was actually used by a particular pseudo-application. An pseudo-application that performs many accesses to shared memory could also have a high CPU usage. In the opposite case, an pseudo-application with few accesses to memory and previously classified as CPU-Bound could also make less use of CPU in regarding to the other pseudo-application classified as Memory-Bound.

After that, in (GARCIA; SCHEPKE, 2018; GARCIA, 2016) the authors investigated the impact of each PPI on the use of CPU and memory. In these studies, the authors classified the pseudo-applications in such a way that all scenarios analyzed contained at least one pseudo-application with: high CPU usage and high memory usage; high CPU usage and low memory usage; low CPU usage and high memory usage; or low CPU and memory usage. Finally, (LORENZON; CERA; BECK, 2016) used some of these pseudo-applications to verify the best performance and energy consumption in different multi-core architectures. All the previous research using these pseudo-applications went at this point. Here starts our contributions.

Gathering all these previous studies, it was concluded that this set contained pseudo-applications diverse enough to characterize a benchmark. After all, they were being used as a benchmark, but an effort was needed to unify them, to analyze the whole set together and prepare them for use by others. The first thing we had to do was fixing the source codes to ensure the correctness of the pseudo-applications outputs. Many of them presented errors when it has some of its parameters modified. Second problem we had to care about was the usability of these pseudo-applications. For the user it could be very confusing to understand how to use and set all the parameters needed for running them.

3.4 Source Code Improvements

The original benchmark suite, provided by Lorenzon (2014), needed some modifications in order to perform the experiments and make the suite closer to a benchmark. We made updates on these pseudo-applications to be used in a wide range of scenarios. The size of the input problem was somewhat static and not compatible with the declared data type. Many pseudo-applications stored values greater than the capacity of the designated data type and this led to erroneous outputs. The GCC compiler, which was used with the original pseudo-applications, is able to work around this problem in some cases, but other compilers may not. So we had to sort this out.

Another problem was that many parallel pseudo-applications had static workload balancing for at most 8 parallel tasks only. So we had to implement dynamic workload balancing that works for all pseudo-applications and for any number of parallel tasks. In addition, this workload balancing was defined for a single input problem size. In the Listing 3.1 there is an example of what the original `defineChunk` function was like.

```
1 void defineChunk(int nTasks) {
2   if(nTasks == 2) {
3     firstEven[0]= 0;      lastEven[0]= 74999;
4     firstOdd[0] = 1;      lastOdd[0] = 74998;
5     firstEven[1]= 75000; lastEven[1]= N-1;
6     firstOdd[1] = 74999; lastOdd[1] = N-1;
7   }else if(nTasks == 3) {
8     firstEven[0]= 0;      lastEven[0]= 49999;
9     firstOdd[0] = 1;      lastOdd[0] = 49998;
10    firstEven[1]= 50000; lastEven[1]= 99999;
11    firstOdd[1] = 49999; lastOdd[1] = 99998;
12    firstEven[2]= 100000; lastEven[2]= N-1;
13    firstOdd[2] = 99999; lastOdd[2] = N-1;
14  }else if(nTasks == 4) {
15    :
16  }else if(nTasks == 8) {
17    :
18  }
19  :
20 }
```

Listing 3.1: Static function to set the size and amount of chunks.

In this Listing 3.1 the chunks are assigned statically to different numbers of tasks. This is a code piece extracted from the Odd-Even Sort application, and the global variable `N` represents the size of the input vector (this works just for 150000 elements). The function shows that for each number of parallel tasks it needs to indicate the start and end of each chunk statically one-to-one. That is, if the user wants to run the

application with an input vector of 150001 elements this will lead to inconsistent results. Also, if the user wants to run with 6 parallel tasks, it will not be possible either. So to improve the original implementation we implemented a new `defineChunk` function (Listing 3.2). With this new function it is possible to define any size of input problem and it will also work for any number of parallel tasks.

```

1 void defineChunk(int nTasks) {
2     int i;
3     int aux = N/nTasks;
4     for(i = 0; i < nTasks; i++){
5         firstEven[i] = i*aux;
6         lastEven[i] = ((i+1)*aux)-1;
7         if(i == 0) {
8             firstOdd[i] = (i*aux)+1;
9             lastOdd[i] = ((i+1)*aux)-2;
10        } else if(i == nTasks-1) {
11            firstOdd[i] = (i*aux)-1;
12            lastOdd[i] = ((i+1)*aux)-1;
13        } else if(i != 0 && i != nTasks-1) {
14            firstOdd[i] = (i*aux)-1;
15            lastOdd[i] = ((i+1)*aux)-2;
16        }
17    }
18 }

```

Listing 3.2: Dynamic function to set the size and amount of chunks.

These modifications needed to be made in many applications, including some with complex `defineChunk` functions that took a long time to construct the dynamic equations. The static function to distribute chunk to 2, 3, 4 and 8 tasks has 75 code lines (symbolized by the vertical ellipsis), while the new dynamic function was written in 18 code lines. So we can say that certainly implementing this kind of improvement is a big step towards improving usability and decreasing the complexity of understanding the application by the user.

Using the Odd-Even Sort application as an example, we address another type of problem that often appeared in the original MPI-1 applications. This application divides an interleaving vector between odd and even positions and sorts each part using Bubble Sort. In the original application for each number of parallel processes created there was a different `bubbleSort` function to execute the `MPI_Send` and `MPI_Irecv` commands statically. This can be seen in Listing 3.3.

```

1 void bubbleSort2Tasks() {
2     :
3 }
4 void bubbleSort4Tasks() {
5     int i, source;

```

```

6   for (i=0; i<(N/2)+1; i++) {
7       sortEven(0);
8       MPI_Send(&vector[37499], 1, MPI_INT, 0, 99, interComm);
9       MPI_Irecv(&vector[37499], 1, MPI_INT, 0, 99, interComm, &request[0]);
10      sortOdd(0);
11      MPI_Wait(&request[0], MPI_STATUS_IGNORE);
12  }
13  MPI_Irecv(&vector[37499], 37500, MPI_INT, 0, 99, interComm, &request[0]);
14  MPI_Irecv(&vector[74999], 37500, MPI_INT, 1, 199, interComm, &request[1]);
15  MPI_Irecv(&vector[112499], 37501, MPI_INT, 2, 299, interComm,
16          &request[2]);
16  for (i=0; i<3; i++) {
17      MPI_Waitany(3, request, &source, MPI_STATUS_IGNORE);
18  }
19 }
20 void bubbleSort8Tasks() {
21     :
22 }
23 :

```

Listing 3.3: Static function to define chunk parameters.

The approach used in the original `bubleSort` function was appalling. However it was easily solved. We need to take care of the case where there are only two parallel processes, because this case has a different flow. All other cases followed the same structure with each other and this was solved in few code lines in the `bubleSort` function improved, as seen in Listing 3.4.

```

1 void bubbleSort(int nTasks) {
2     int i, source = 0, aux = N/nTasks, id = 99;
3     for (i=0; i<(N/2)+1; i++) {
4         sortEven(0);
5         MPI_Send(&vector[aux-1], 1, MPI_INT, 0, id, interComm);
6         MPI_Irecv(&vector[aux-1], 1, MPI_INT, 0, id, interComm, &request[0]);
7         sortOdd(0);
8         MPI_Wait(&request[0], MPI_STATUS_IGNORE);
9     }
10    if (nTasks==2) {
11        MPI_Recv(&vector[aux-1], (N-(aux-1))+1, MPI_INT, 0, id, interComm,
12              MPI_STATUS_IGNORE);
13    } else {
14        for (i=0; i<nTasks-1; i++)
15            MPI_Irecv(&vector[((i+1)*aux)-1], aux, MPI_INT, i, id+(i*100),
16                  interComm, &request[i]);
17        for (i=0; i<nTasks-1; i++)

```

```

16     MPI_Waitany(nTasks-1, request, &source, MPI_STATUS_IGNORE);
17 }
18 }

```

Listing 3.4: Dynamic function to define chunk parameters.

Static functions to control interprocess communication was another problem often found in the original MPI applications. As can be seen in Listing 3.5, the `startIrecv` function runs the `MPI_Irecv` command statically for 2, 3, 4, or 8 processes. Any other amount of parallel processes that the user sets will not work for this function. To solve this, we re-think this function so that the application works with any number of parallel processes. The current and improved version is in the Listing 3.6.

```

1 void startIrecv(int rank, int nTasks, ringStruct *game, int *vecEnd, int *
   vecInit, int N){
2   if(nTasks == 2){
3     if(rank == 0){
4       MPI_Irecv(&game[(vecInit[1]*N)], N, MPI_INT, 1, 199, MPI_COMM_WORLD,
   &req[0]);
5     }else{
6       MPI_Irecv(&game[(vecEnd[0]-1)*N], N, MPI_INT, 0, 99, MPI_COMM_WORLD,
   &req[0]);
7     }
8   }else if(nTasks == 3){
9     if(rank == 0){
10      MPI_Irecv(&game[(vecInit[1]*N)], N, MPI_INT, 1, 199, MPI_COMM_WORLD,
   &req[0]);
11    }else if(rank == 1){
12      MPI_Irecv(&game[(vecInit[2])*N], N, MPI_INT, 2, 299, MPI_COMM_WORLD,
   &req[0]);
13      MPI_Irecv(&game[(vecEnd[0]-1)*N], N, MPI_INT, 0, 99, MPI_COMM_WORLD,
   &req[1]);
14    }else if(rank == 2){
15      MPI_Irecv(&game[(vecEnd[1]-1)*N], N, MPI_INT, 1, 399, MPI_COMM_WORLD,
   &req[0]);
16    }
17  }else if(nTasks == 4){
18    :
19  }else if(nTasks == 8){
20    :
21  }
22  :
23 }

```

Listing 3.5: Static function to start `MPI_Irecv`.

```

1 void startIrecv(int rank, int nTasks, ringStruct *game, int *vecEnd, int *
   vecInit, int N){

```

```
2  if(rank == 0){
3      MPI_Irecv(&game[(vecInit[rank+1]*N)], N, MPI_INT, rank+1, ((rank+1)*
4      100)+49, MPI_COMM_WORLD, &req[0]);
5  }else if(rank == (nTasks-1)){
6      MPI_Irecv(&game[(vecEnd[rank-1]-1)*N], N, MPI_INT, rank-1, ((rank-1)*
7      100)+99, MPI_COMM_WORLD, &req[0]);
8  }else{
9      if(rank%2!=0){
10         MPI_Irecv(&game[(vecEnd[rank-1]-1)*N], N, MPI_INT, rank-1, ((rank-1)*
11         100)+99, MPI_COMM_WORLD, &req[0]);
12         MPI_Irecv(&game[(vecInit[rank+1]*N)], N, MPI_INT, rank+1, ((rank+1)*
13         100)+49, MPI_COMM_WORLD, &req[1]);
14     }else{
15         MPI_Irecv(&game[(vecInit[rank+1]*N)], N, MPI_INT, rank+1, ((rank+1)*
16         100)+49, MPI_COMM_WORLD, &req[0]);
17         MPI_Irecv(&game[(vecEnd[rank-1]-1)*N], N, MPI_INT, rank-1, ((rank-1)*
18         100)+99, MPI_COMM_WORLD, &req[1]);
19     }
20 }
21 }
```

Listing 3.6: Dynamic function to start MPI_Irecv.

In addition to the code problems and dynamism, some modifications were necessary to improve more the usability of the pseudo-applications. The only way to change almost any execution parameter was by rewriting the source code, which was an inefficient way of doing this. So we've made all modifiable parameters dynamic and also added basic information on how to use them. In addition, we make the pseudo-applications available in an online repository² where we will add new updates and improve the source code documentation in the future.

² Source codes available at <https://github.com/adrianomg/PAMPAR>

4 EXPERIMENTAL RESULTS

In this chapter, we present our experimental results. The methodology we used for the experiments is described in Section 4.1. To identify which pseudo-applications make more accesses to the memory, in Section 4.2 we present cache memory accesses per instruction by pseudo-application. In Section 4.3 we organise the pseudo-applications according to the number of branch instructions. Section 4.4 presents the amount of floating-point operations each pseudo-application do per instruction. We carry out a case study in Section 4.5, where we evaluate the performance and energy consumption of the pseudo-applications. At the end, in Section 4.7, we discuss and summarise the results.

4.1 Methodology

The results presented in this chapter are the average of 30 executions. The executions are interleaved among the pseudo-applications. This number of executions was established as indicated in (HUNOLD; CARPEN-AMARIE, 2016). In this study, the authors perform experiments that show that the minimum number of executions is MPI in order to obtain statistically acceptable results. Following the indications of this study, the results in MPI-1 and MPI-2 showed an average absolute deviation below 3.7% in the worst cases. It means that the resulting data were very similar in each execution. OpenMP and PThreads showed an average absolute deviation below 1.3% in all cases. During the experiments, the computer remained locked to ensure that other pseudo-applications did not interfere actively with the results.

We use hardware counters to gather data about total instructions, cache accesses, branch instructions, and floating-point operations. Hardware counters are a set of special-purpose registers built into modern microprocessors to store the counts of hardware-related activities within computer systems. To access these hardware counters we use the PAPI (Performance Application Programming Interface) (TERPSTRA et al., 2010).

PAPI works by inserting directives within the application code. These directives use PAPI events, which are aliases for the native events identified by a hexadecimal value that points to a specific hardware counter. A big problem when using PAPI, or any other profiler, with parallel applications is to be able to pin all parallel tasks. This can be a challenge for a single parallel application, even using only one PPI. So to evaluate multiple applications and PPIs, and ensure reliable results, this can take a lot of time. In addition, writing directives in the pseudo-application codes is a not very efficient option to evaluate this amount of pseudo-applications.

To work around this problem, we decided to use an external C program that allows PAPI to be used without having to rewrite the pseudo-application code. This

Table 2: Details about the applications

Data Structures	Problem Size	Acronym	Pseudo-Application	Complexity	
Unstructured data	10^9	NI	Numerical Integration	$O(n)$	
	4×10^9	PI	PI Calculation		
	15×10^9	DP	Dot Product		
Vector	10^5	HA	Harmonic Sums	$O(n \times d)$	
	25×10^4	OE	Odd-Even Sort		
	32768	DFT	Discrete Fourier Transf.		$O(n^2)$
Matrix	1920×1080	HS	Histograms Similarity	$O(m \times n^2)$	
		GL	Game of Life		
	TR	Turing Ring			
	2048×2048	DJ	Dijkstra		$O(n^3)$
		JA	Jacobi Method		
		MM	Matrix Multiplication		
GS		Gram-Schmidt			

Source: by the author.

program invokes the PAPI library using the `LD_PRELOAD` environment variable and captures the pseudo-application that is currently running with the `/proc/self/exe` link. This way it is possible to pin and monitor each thread or process created by a specific pseudo-application. We ran several tests using this new technique and the traditional technique and we were able to obtain the same results, confirming its effectiveness and that the external C program do not interfere with the results.

The toolkit Intel® Performance Counter Monitor (PCM) 2.0 was used to measure energy consumption. It has a tool to monitor the power states of the processor and DRAM memory. For the execution time, the time at the beginning and at the end of the main function of each pseudo-application was measured and the difference between these values was used. It was done using the function `gettimeofday` from the `time.h` library.

To set a workload that is equivalent to all applications is not a simple thing to do when comparing different parallel applications. We prepared three sets of problem sizes: small, medium, and large. However, in this work, we use the medium set for tests, since they presented the characteristics of each pseudo-application in a more explicit way. For what we are looking for, the large set produces redundant results and increases too much the duration of the experiments. Table 2 shows the data structure and problem size corresponding to the medium set inputs, the acronym used to identify each pseudo-application in the following sections, and their complexities.

Next experiments were carried out on a computer equipped with 2 Intel® Xeon® E5-2650 v3 (Q3'14) Haswell processor. Each processor has 10 physical cores and 10

virtual cores operating at the standard 2.3 GHz frequency and a turbo frequency of 3 GHz. Its memory system consists of three levels of cache: a 32 kB cache L1 and a 256 kB cache L2 for each core. Level L3 has a 25 MB cache for each processor using Smart Cache technology. The main memory (RAM) is 128 GB in size and DDR3 technology. The operating system is Ubuntu 18.04.2 and Linux version 4.15.0-45 using Intel® ICC 18.0.1 compiler with default optimization flags.

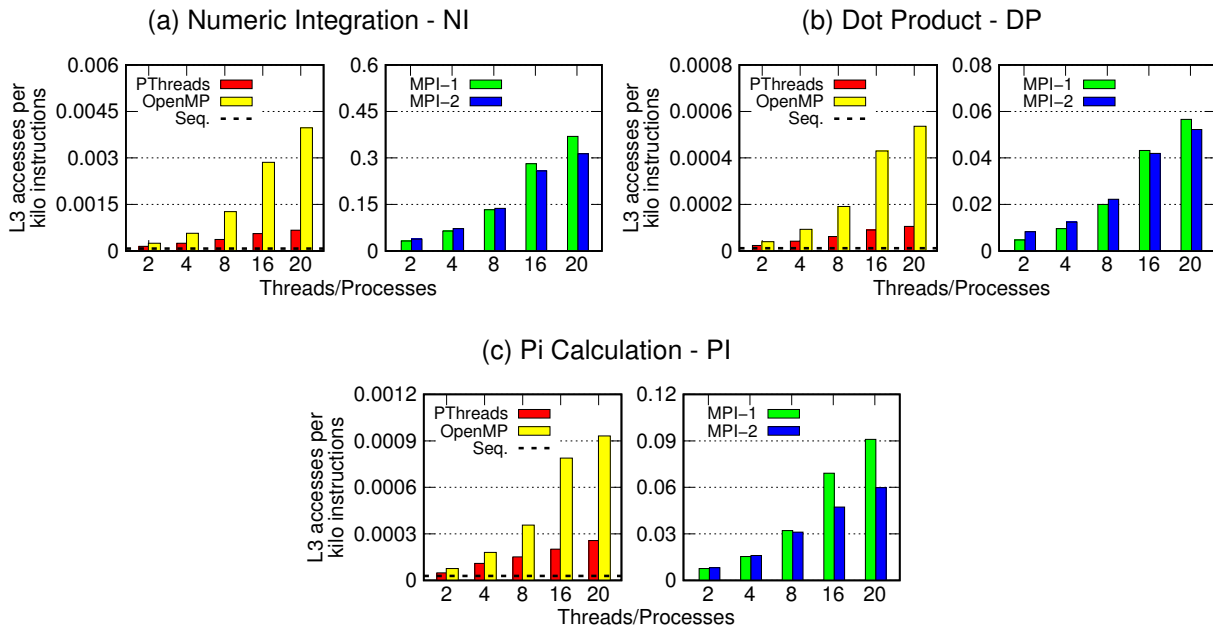
To achieve statistically reliable results it would also be necessary to configure processor affinity. This is mainly because we are evaluating cache accesses and using a two-processor architecture, so assigning parallel tasks to the processor cores non-uniformly can have a big impact on the results. We tried to configure the system in different ways, but we were not able to make the affinity work for all pseudo-applications and PPIs simultaneously. We can configure affinity for one case at a time. However, in our experiments we run the pseudo-applications and the PPIs interleaved, which means having to reset and reconfigure the affinity thousands of times. Even then, we tried to do it that way, but we could not set affinity for MPI-2 reliably. MPI-2 launches its child processes at runtime and these children are created in different ways in each pseudo-application. So it is a very complex task to ensure that processor affinity would work in all cases. In this way, we decided to go ahead with the experiments leaving the responsibility of the system itself to manage the scheduling of threads/processes in the processor.

There is no easy way to compare the number of accesses to the cache, branches instructions, and FLOPs of different pseudo-applications simply by looking at the total values. Each pseudo-application handles differently with the size of the input problem and this is not something comparable. To make this data more correlated and reliable, we normalized it using the total instructions of each pseudo-application. To have greater readability of the results, we made this relation for every thousand instructions, what we call kilo instructions. In this way, the results presented in the Sections 4.2, 4.3, and 4.4 are the corresponding data per kilo instructions. In these Sections, each pseudo-application is represented in a different chart. In these charts, there are 5 sets of bars, each one corresponding to a different amount of parallel tasks. All PPIs are represented by distinct colors in each set. Also, the dashed line is the pseudo-application using no PPI, a single task execution.

4.2 Cache Memory Accesses

This section presents the cache accesses results. We collect data from cache L3 and L2 levels because that is what the hardware counters make available on memory accesses in the architecture used. In this Section, we show and discuss the results of the L3 cache. The results of the L2 cache leads to the same conclusions and are presented in Figures 13 and 14 in Appendix A. The results here presented are arranged

Figure 1: Low L3 cache accesses per kilo instructions rate.



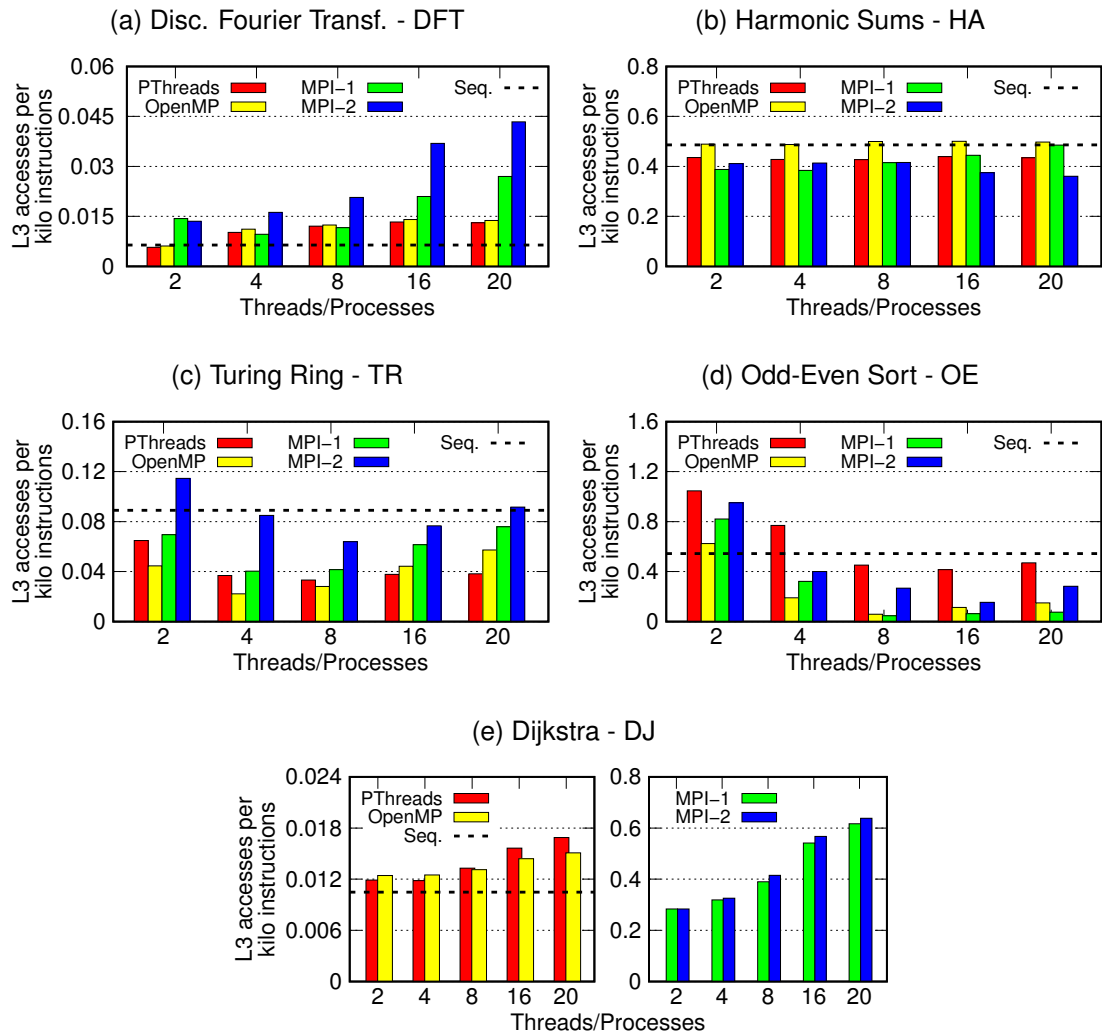
Source: by the author.

in 3 figures. Figure 1 shows the pseudo-applications that do not do any significant cache access. In the 2 figure are the graphs of the pseudo-applications that make only a few accesses. And the figure 3 shows the result of the pseudo-applications that do many accesses to the cache compared to the total of instructions.

The pseudo-applications that almost do not access cache are DP, PI, and NI. This is exactly the expected behavior. These are the 3 pseudo-applications that use only simple data and do not allocate any more complex data structure in memory. The other results in the graphs of Figure 1 are the accesses made by the PPIs. They show that OpenMP adds a higher cache overhead than PThreads. This may possibly be caused by the copy of private variables in memory that OpenMP needs to do, among other reasons. OpenMP has an advantage in usability, but it does not make it transparent to the programmer what its directives do exactly. On the other hand, we have MPI 1 and 2, two PPIs that have an overhead in the accesses to the cache about a hundred times greater than OpenMP or PThreads. Both MPI PPIs use buffers to store data in communication among tasks. In addition, each process has its own memory allocation region. Therefore in these pseudo-applications, it is expected that all the PPIs present an increase in the rate of access to memory and that this rate increases proportionally to the number of parallel tasks.

After the pseudo-applications that make almost no access to the cache, there are the pseudo-applications that make few accesses. For this, we are considering sequential pseudo-applications that make at least 0.01 access per kilo instruction: DFT,

Figure 2: Medium L3 cache accesses per kilo instructions rate.

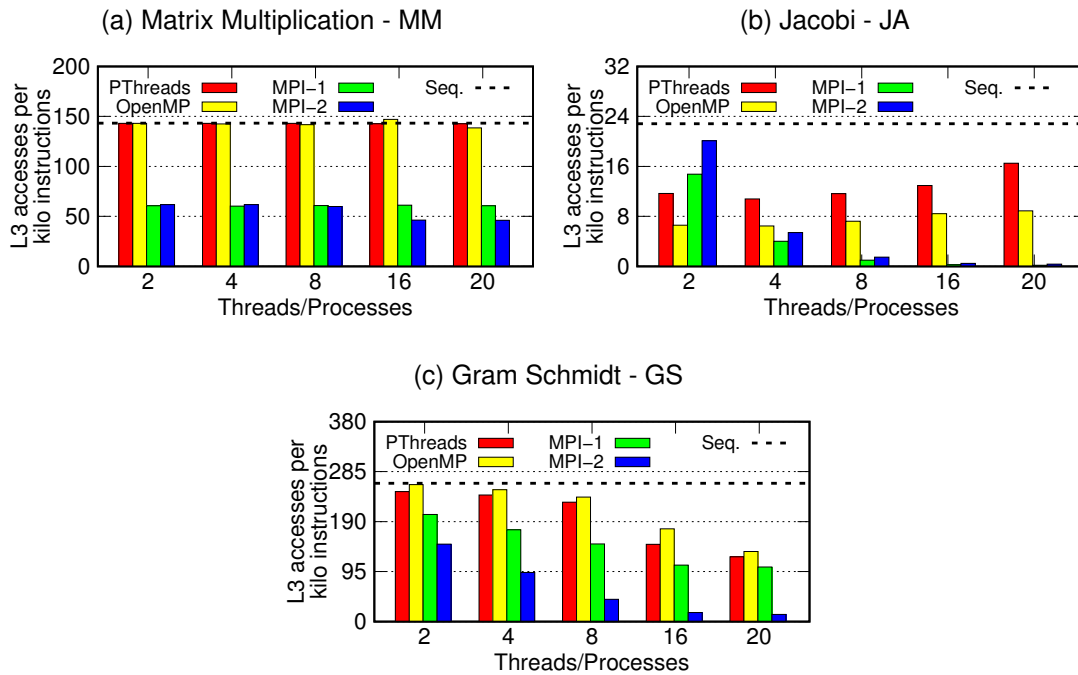


Source: by the author.

HA, TR, OE, DJ (Figure 2). This rate is hundreds of times greater than the sequential NI, DP, and PI rates. Among these pseudo-applications, we have TR and DJ that use matrices while HA, OE, and DFT use vectors. The vectors used are small enough that their pieces fit into the L2 cache. This can be seen if we compare the results of accesses to L2 and L3 of OE that uses a vector of 250,000 elements, the largest of them all. The rate of access per instruction from OE to L2 in Figure 13 (b) (Appendix A) is tens of times greater than the accesses to L3 in Figure 2 (d). Therefore, these pseudo-applications are expected to make very few access to L3.

Besides the pseudo-applications that use vectors are the two that use matrices. The matrices in both cases are 2048×2048 in size and the pseudo-applications would necessarily need to fetch on L3. However, these are sparse matrices and DJ and TR are pseudo-applications that do few operations in the matrices compared to the others. These two pseudo-applications, as we will see in Section 4.3, are the two with

Figure 3: High L3 cache accesses per kilo instructions rate.



Source: by the author.

the largest branches per kilo instructions rate. These branches cause the array not to be accessed as often as it is done with other cases of pseudo-applications that use matrices.

Finally, we have in Figure 3 the three pseudo-applications that make more access to the cache memory: MM, JA, and GS. These pseudo-applications need to access matrix data much more often than DJ and TR. It can be concluded by looking at the characteristics of these applications in section 3.1. Among all benchmark pseudo-applications, GS is the one that performs the larger access to the cache. One of the reasons for this can be found by looking at the Algorithm 11, in which we see that GS accesses the elements of the array columns three times each loop. In addition to GS, MM also made many accesses to memory. This is also expected because this pseudo-application allocates and accesses three matrices at the same time. The differential of MM is that the PPIs show a stable behavior with the task variation for both L3 and L2 caches (Figure 13 (g)). What happens here is not that MPI 1 and 2 make fewer accesses to the cache, but rather that these PPIs use much more instructions than PThreads and OpenMP. This is why they exhibit exactly the same behavior in the L3 and L2 results.

Regarding PPIs, besides MM only HA showed a non-variable behavior with the increase of parallel tasks in all PPIs. For the other cases there are two situations: either the access rate increased because of overhead, or the access rate decreased

because of the increase in the number of total instructions. Both cases are directly linked to communication among tasks. The cases in which the access rate increases occurs in pseudo-applications that make few or almost no access to the cache. In these pseudo-applications, any small overhead caused by communication among tasks will greatly impact the results. On the other hand are the pseudo-applications in which the access rate decreases as the number of tasks increases (mainly MPI). These are the pseudo-applications that do a lot of communication between tasks. JA and GS, for example, are the pseudo-applications that present the biggest drop in the access rate. Using 20 parallel tasks, JA and GS with MPI-2 run about 2×10^{12} of total instructions, against 500×10^9 for MPI-1 and about 15×10^9 for OpenMP and PThreads. As Lorenzon, Cera e Beck (2016) shows, these pseudo-applications are among those that do a lot of communication among tasks and in MPI this communication is very costly, which increases the number of instructions.

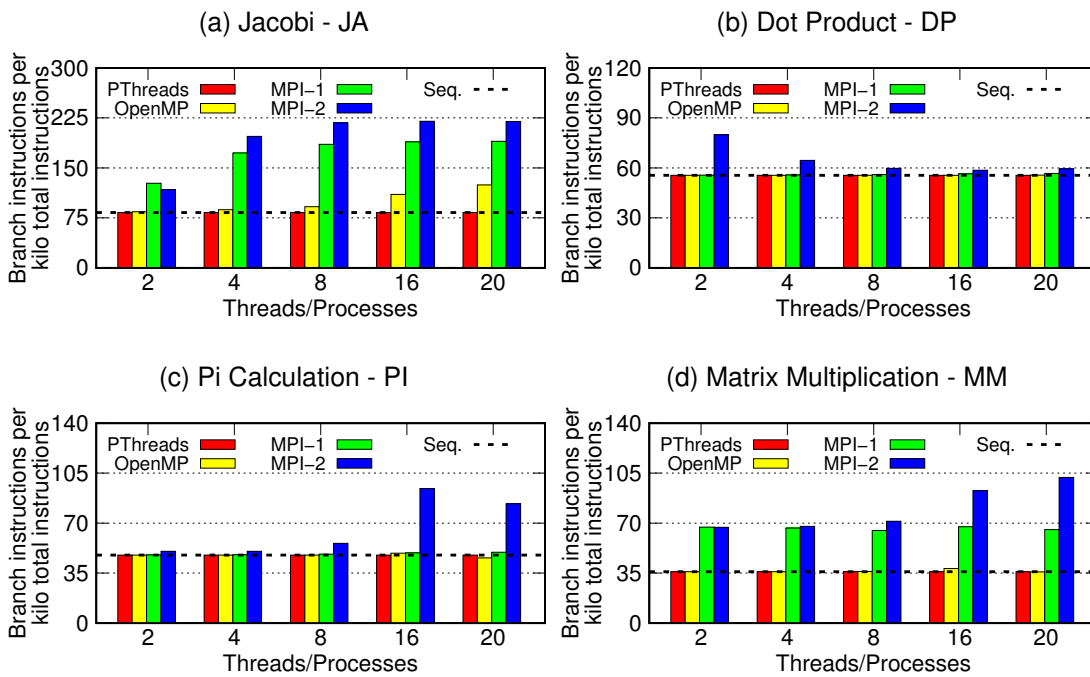
However, it is not only the number of instructions that goes up when there is a lot of communication among processes. Although not in the same proportion, this communication also causes a large increase in the number of accesses to memory. In our case, MPI communicating ranks are on the same machine, so the data transfer is through shared memory. This happens because MPI ranks are independent processes, they do not have access to each other's memory. Instead, the MPI processes on the same node set up a shared memory region and use it to transfer messages. So sending a message involves copying the data twice: the sender copies it into the shared buffer, and the receiver copies it out into its own address space. Therefore all this operation will cause overhead in the memory accesses. For pseudo-applications that already need to do a lot of communication or do not need to make too many memory accesses, this overhead is diluted. However, it can be clearly seen in pseudo-applications that do little communication by parallel tasks (Figure 2), and the more tasks are used the more overhead it is.

4.3 Branch Instructions

In this section, we evaluate how many branches each pseudo-application has. For this, we measure the number of branch instructions and divide by total kilo instructions. The results presented here are arranged in three figures. In Figure 4 are the pseudo-applications that least executed branch instructions (less than 100 branches per kilo instruction). The pseudo-applications that ran between 100 and 200 branch instructions per kilo instructions are in Figure 5 and rates over 200 in Figure 6.

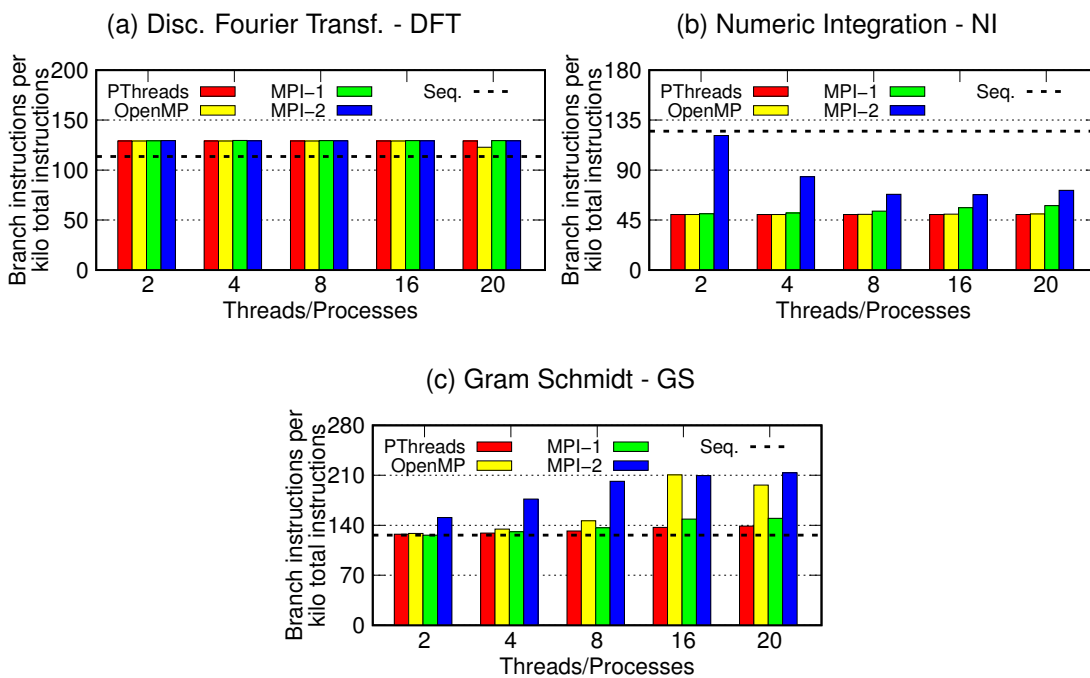
In general, all pseudo-applications run at least dozens of branch instructions. There are no cases of pseudo-applications that almost take no branches, which occurs with accesses to memory (previous section) and also occurs with floating-point operations, as we will see in the next Section. In Figure 4 are the pseudo-applications JA,

Figure 4: Low branch instructions per kilo total instructions rate.



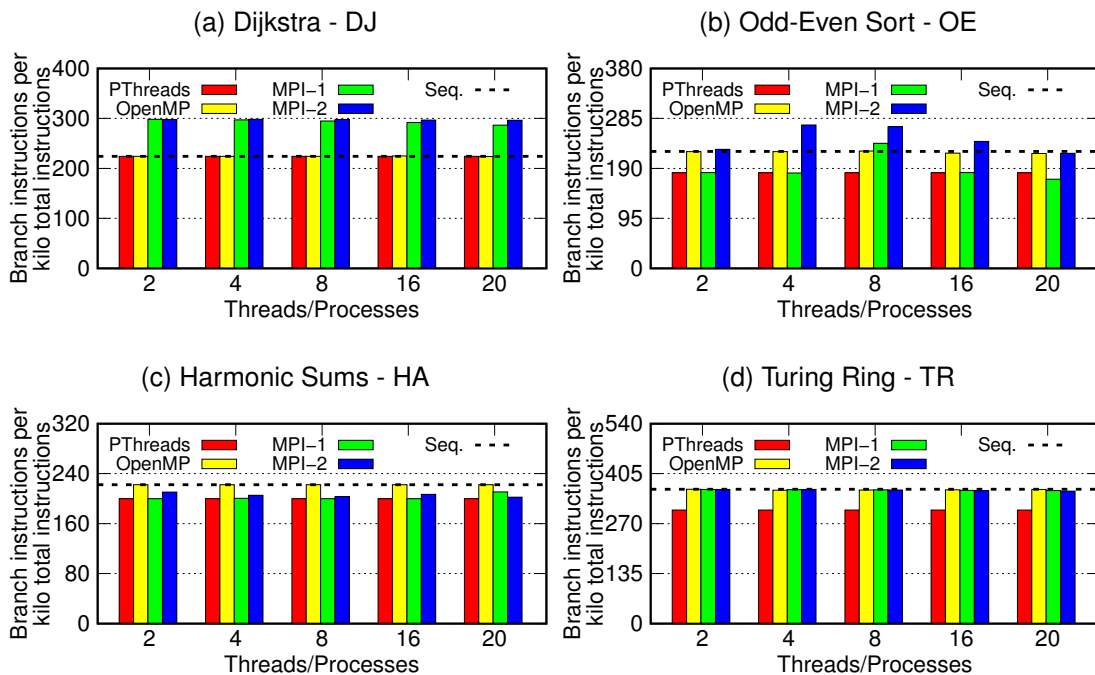
Source: by the author.

Figure 5: Medium branch instructions per kilo total instructions rate.



Source: by the author.

Figure 6: High branch instructions per kilo total instructions rate.



Source: by the author.

DP, PI, and MM which are those that execute less than 100 branch instructions per kilo total instruction. The first point to note in these charts is that PThreads and OpenMP do not add any overhead. In DP and PI the same occurs with MPI-1, and even in the other two pseudo-applications, MPI-1 adds an overhead but with a stable threshold. On the other hand, MPI-2 shows a slightly different behavior, increasing and decreasing the overhead according to the number of parallel processes. In these pseudo-applications, the total of branch instructions almost matches with the total of executed loops.

We divided the results into high, medium and low rate classes just in order to better present the results. These classification are not meant to be used out of this work context. Pseudo-applications that execute more than 100 and less than 200 branch instructions per kilo instruction we consider to have a medium branch rate. These pseudo-applications are DFT, NI and GS and their respective charts are in Figure 5. DFT presents a simple behavior where all PPIs add a small non-variable overhead. On the other hand, NI exhibits unexpected behavior. NI is a pseudo-application that does not use complex data structures and all computation is done within a simple loop. Therefore NI was expected to have similar results to DP and PI, and indeed with PThreads, OpenMP, and MPI-1, it is similar. However the sequential has a branch rate almost triple that expected. MPI-2 does the same but decreases using more parallel processes. Finally, GS presents an increasing overhead with all PPIs at different levels.

In the last group of pseudo-applications (Figure 6) are DJ, OE, HA, and TR.

These are pseudo-applications with a high branch rate per kilo instruction (over 200). In all four cases, PPIs do not present as much difference to sequential pseudo-applications. MPI 1 and 2 show a continuous overhead in DJ and variable in OE. In other cases, PPIs do not show many variations. The highlight is TR, which is the pseudo-application that performs the highest rate of branch instructions per kilo total instructions among all. It also features the highest total amount of branch instructions, about 70 billion (sequential version). However this was expected, Turing Ring has several rules that define the interaction between prey and predator and for each rule there is a conditional branch that controls it.

Thus, the results of this Section show us that PPIs impact less on conditional branches than on cache accesses or floating-point operations, as presented in the next Section. Some particular cases have a larger overhead with MPI 1 and 2, but even so, it is not a very pronounced difference. The fact is that since no pseudo-application made less than 1 branch per kilo instruction, the overhead does not appear as much and ends up being diluted in most cases. Therefore the results show that PThreads and OpenMP practically do not execute extra branch instructions, while MPI 1 and 2 do.

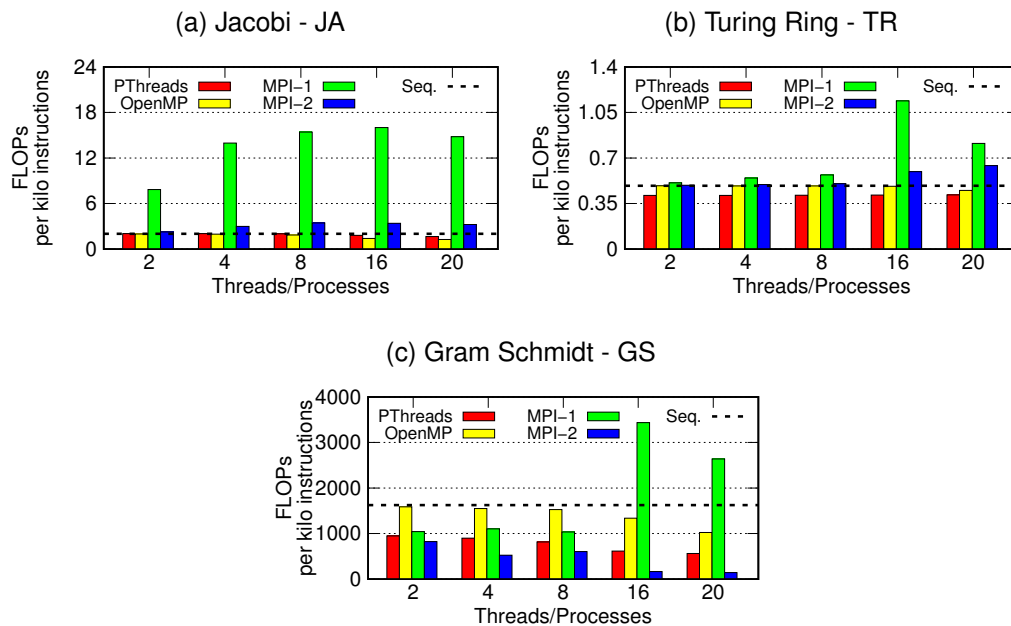
4.4 Floating-Point Operations

Regarding floating-point operations (FLOPs) the pseudo-applications show quite varied results (we use the term FLOPs, careful not to confuse with FLOPS). To facilitate the analysis of the results, we will only evaluate the sequential pseudo-applications first. Many pseudo-applications do hundreds of FLOPs per kilo instructions, one of them up to thousands. On the other hand, four of them (OE, DP, MM, and DJ) do almost no floating-point operations (Figure 8). Dot-Product and Odd-Even Sort are pseudo-applications that do simple operations between integer vectors. The same thing is applicable for MM integer arrays. Dijkstra uses a common adjacency matrix and also performs only simple calculations between integer values. In addition to these cases, TR and JA are pseudo-applications that do only a few operations of this type. JA does about 2 operations per kilo instructions and TR only 0.5 approximately (Figure 7).

A particular case is when an pseudo-application has many floating-point operations on its code but a low rate of these operations per instruction. What explains this is that these operations are repeated only a few times. Turing Ring (TR), for example, has dozens of such operations with each loop. However, this is the pseudo-application with the largest number of branches, as seen in the previous section. In this way, it is clear that most of these operations are not performed with each loop. So this is the main reason for this pseudo-application which has many floating-point operations in its code making only 0.5 FLOPs per kilo instructions, on average.

The pseudo-application that presents a more distinct behavior among the others is GS (Figure 7 (c)). This is the only pseudo-application that does more floating-point

Figure 7: Floating-point operations per kilo instructions.



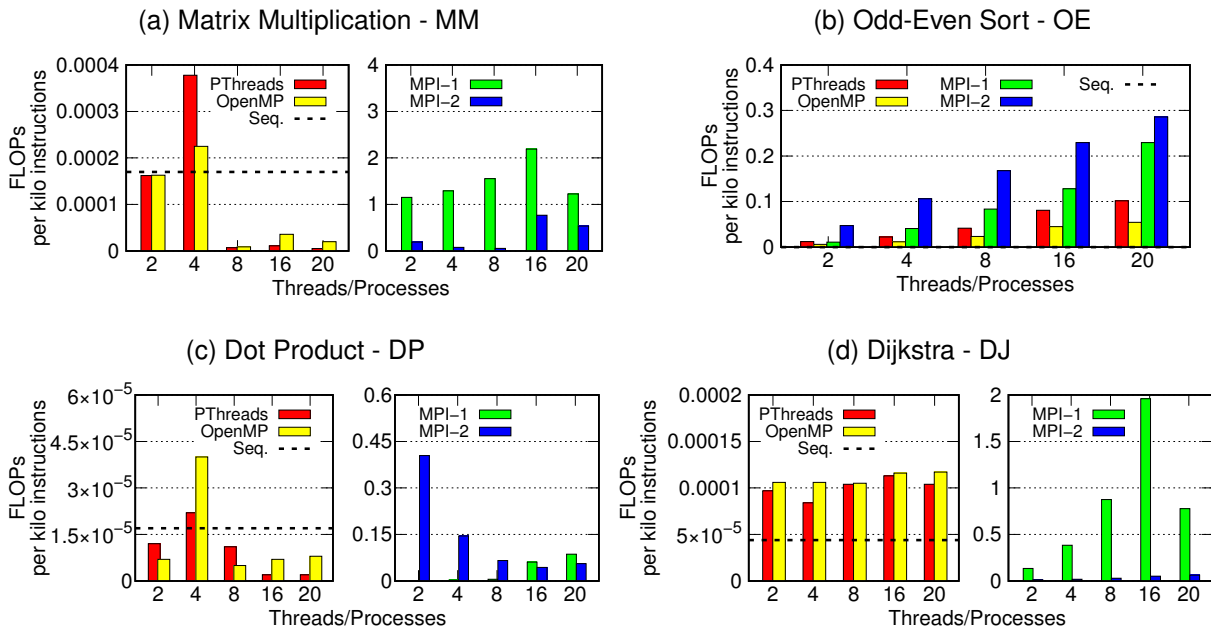
Source: by the author.

operations than total instructions. This pseudo-application has several nested loops in sequence with no conditional branches, as can be seen in the algorithm 11. In this way, this pseudo-application can exploit the SIMD (Single Instruction, Multiple Data) processing capacity for vector operations, which means that the processor can perform the same operation on multiple data points simultaneously. The Intel[®] Xeon[®] E5-2650 processor architecture includes instruction sets that allow this type of operation between floating-point vectors, such as AVX and SSE.

Disregarding the exceptional case of GS, pseudo-applications with a rate of hundreds of FLOPs are DFT, HA, NI, and PI. If we look at the respective equations (3.7, 3.6, 3.1, and 3.4) it becomes clear to see the characteristic that leads to this. All of them have at least one division operation within a summation, which indicates the presence of floating-point values. Looking at just the size of the loops and the type of operation, PI and NI could have a rate of FLOPs per kilo instructions as large as GS. However, these pseudo-applications do not use any data structure that can take advantage of vector processing. Even so, NI makes more than 400 FLOPs per kilo instructions. Therefore we can consider that in architectures without a vector processing unit, NI would take advantage.

Analyzing PPI results now, there are cases where the rate of FLOPs per kilo instructions is higher and other cases where it is lower than the sequential pseudo-application. Same behavior observed with cache accesses and branches. Again the fact that we did not set core affinity can impact that. Also, as we have explained in previous

Figure 8: Low floating-point operations per kilo instructions rate.

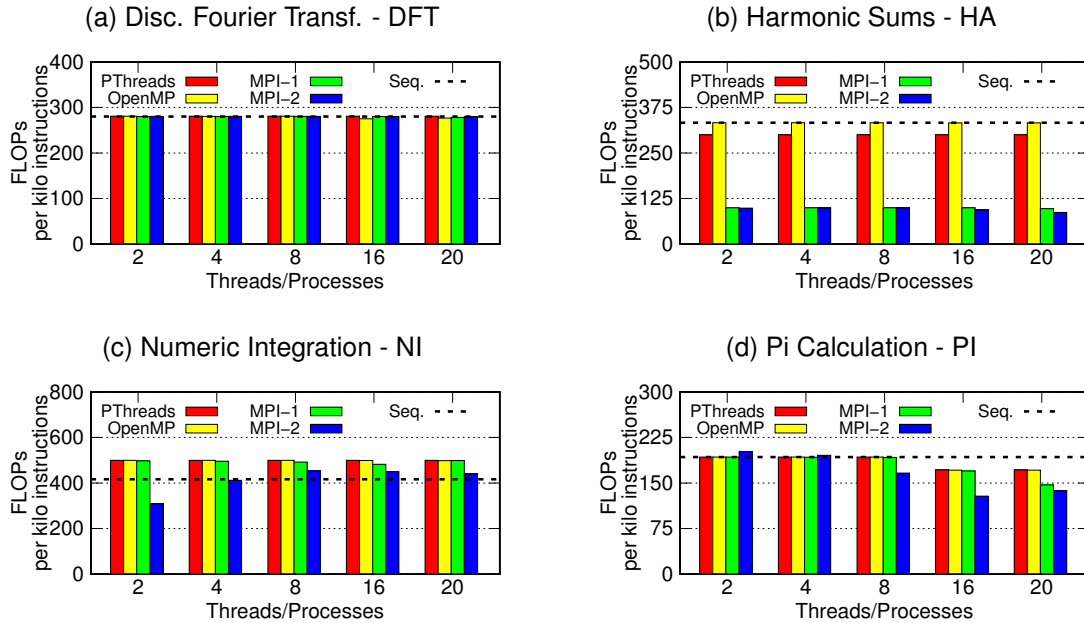


Source: by the author.

sections, total instructions play a significant role in these fluctuations in PPI results. For the pseudo-applications in Figure 8 that do not present floating-point operations, the behavior of the PPIs is the same seen with respect to the cache accesses. PThreads and OpenMP present a small overhead. In addition, MPI 1 and 2 show an overhead that exceeds the sequential results of TR and JA pseudo-applications.

All cases where the FLOPs rate decreases as the number of parallel tasks increases happen because the tasks rise the number of executed instructions, and not because the total number of FLOPs has decreased. So when looking at the raw data resulting from the executions, we notice that all PPIs insert FLOPs overhead in pseudo-applications always, even when the rate per instruction decreases. Part of this overhead comes from the fact that PPIs run parallel threads/processes in different cores, and each core has its own FPU. So this certainly causes different results than expected when running a single thread/process in a single core. In addition, one thing that stands out is some pseudo-applications in which MPI-1 has a large overhead always with 16 processes. We are not able to explain exactly what causes this behavior, but total FLOPs grow steeply while total instructions remain stable. This occurs in different pseudo-applications with no relation at all, so further study would need to be done to find answers.

Figure 9: High floating-point operations per kilo instructions rate.



Source: by the author.

4.5 Performance and Energy Consumption

The results of energy consumption and performance are arranged on the same charts. Bars show the total energy consumption for all processors in joules. It corresponds to the y-axis values on the left. The time in seconds is represented by the marker \times . It is aligned to the y-axis on the right. Each chart displays the results by pseudo-application individually. These results refer to running using 2, 4, 8, and 16 parallel threads/processes for each PPI. We did not get results using 20 parallel tasks because these were the first experiments and not all pseudo-applications were prepared to do load balancing for numbers that are not a power of two at that time (GARCIA; SCHEPKE; GIRARDI, 2018a). They are currently able to run with any number of threads/processes, but we were not able to redo those experiments on time.

In addition, the first result of each graph represents the sequential execution of the respective pseudo-application. The remaining sets refer to each of the PPIs, nominated at the top of each graph. In all the charts in Figure 10, the scales of the two y-axes were adjusted so that both energy and performance top values for the sequential pseudo-application were aligned at the same point. This allows for easier visualization of the impact of varying the number of threads/processes.

Our initial hypothesis was that higher use of the processor and memory system should cause an increase in energy consumption in proportion to the number of parallel threads/processes. But in addition, reducing the execution time of each pseudo-

application should reduce its energy consumption in proportion to the performance achieved over the sequential pseudo-application. However, if the sequential results were proportionally aligned in a chart, we should note that this proportion does not appear in the parallel versions. The energy consumption does not decrease in the same proportion as the execution time. For all cases, there is an overhead in energy consumption that can be caused by the increase in the number of accesses to memory, branches, and FLOPs, as seen in previous sections. There are also other factors that impact on energy consumption, such as the need for communication among tasks and the increase in complexity of control structures that the OS has to deal with.

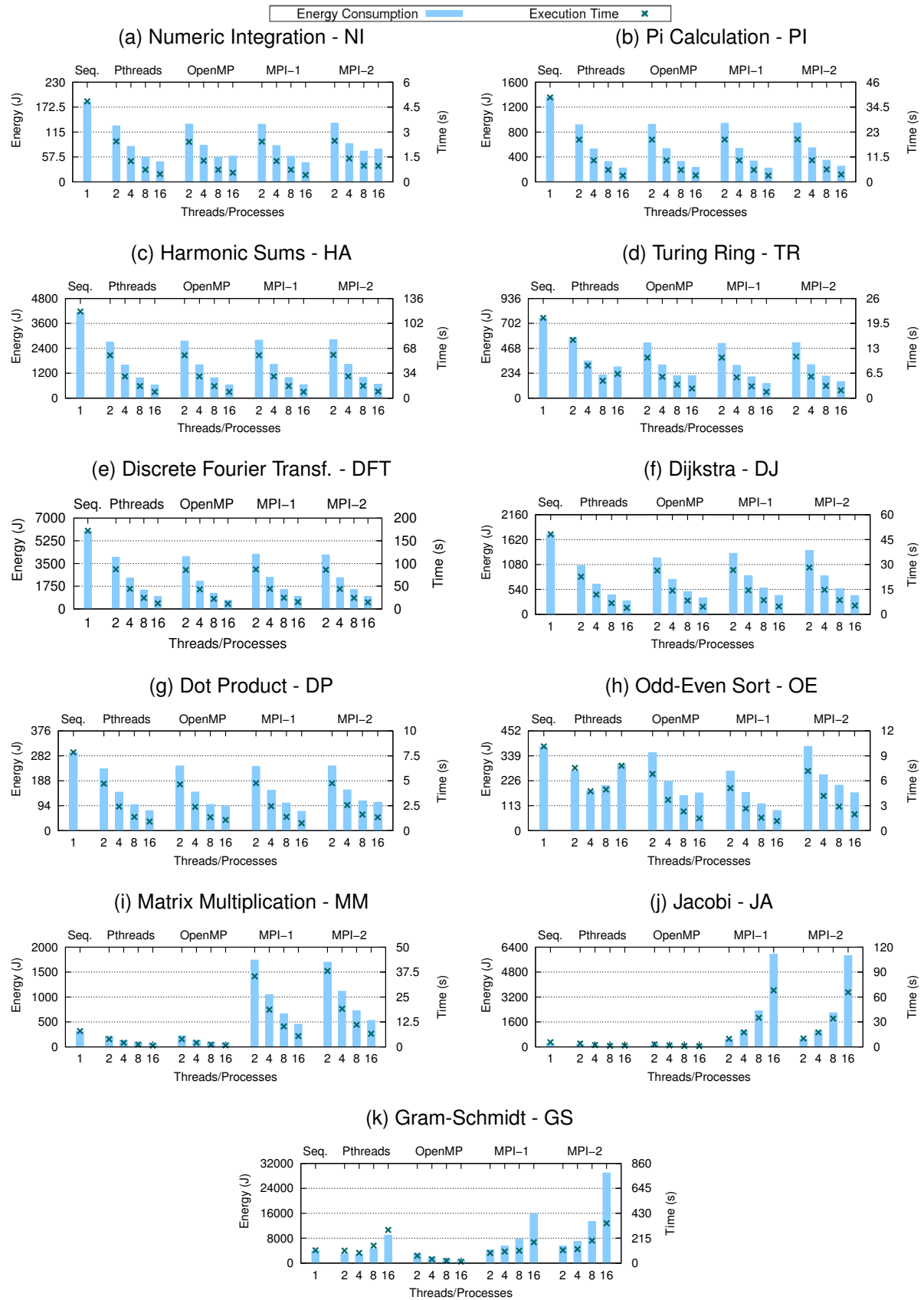
In Figure 10, the pseudo-applications between (a) and (i) in general show a result as expected in our initial hypothesis. However, the results show that the energy consumption of MPI-1 and MPI-2 is slightly higher in most cases. In addition, in pseudo-applications that do more communication between tasks, such as GS and JA, energy consumption and runtime were about ten times higher for both MPI PPIs than the others. The results in the previous sections are evaluated by instruction, so the overhead caused by MPI 1 and 2 is not so evident. But what happens is that both the total of instructions and the other parameter analyzed grow proportionally. But by looking at the power consumption and runtime of these pseudo-applications independently, it is clear that the total value observed with MPI in the previous sections was very different from other PPIs.

The TR (Figure 10-d) pseudo-application showed a different behavior when using PThreads. Except when using 16 threads, in the other cases, this PPI had an increase in execution time compared to the others. However, the energy consumption did not increase in a proportional way to the time, remaining close to the other PPIs results. A similar behavior, but with worse scalability with PThreads, can be seen in OE. This behavior of PThreads in OE can be explained by the increase in the rate of accesses to memory, in Figure 2 we can see that PThreads presented the highest rate among PPIs.

OpenMP showed that its best trade-off between performance and energy consumption occurs using 8 threads. Observing the results of NI, TR, DP, and OE, we can see that although there is a slight reduction in execution time with 16 threads, the energy consumption does not follow this reduction and it is the same scene with 8 threads. If we compare this larger consumption with 16 threads to the accesses to the cache memory, we can observe the same behavior. With TR and OE that make more access, these accesses decrease with up to 8 threads and rise again with 16. In addition, all OpenMP results with 8 threads were smaller than the others in almost all cases.

Regarding Odd-Even Sort (Figure 10-h), the results show that we obtained performance gains in all cases with both MPI-1 and MPI-2. However, with 16 OpenMP threads, there was no performance gain or PThreads with 16 and 8 threads. What we have concluded, is that the average workload initially set, is not large enough for all

Figure 10: Energy consumption and performance for each pseudo-application.



Source: by the author.

cases. OE is a memory-bound pseudo-application, so the overhead of communication/synchronization between threads begins to impact negatively earlier in these cases. With MPI, performance only begins to converge after 32 processes for this workload, but this result is not included in this work.

It is expected that the more robust the architecture, the better the MPI results can be. In this way, it was possible to observe good scalability of MPI in the first cases. However, considering the last 3 cases, we can conclude that MPI is the worst case overall. In these three cases, MPI did not have good scalability and presented worse results than the sequential version. Considering only MPI-1 and MPI-2, the second one was the one with the worst results. Similar behavior was observed by the authors in (LORENZON; CERA; BECK, 2016). Additionally, the pseudo-applications which presented the worst results for MPI are the only ones in (GARCIA, 2016) and (GARCIA; SCHEPKE, 2018) that showed a higher percentage of memory usage than CPU usage during execution. In this way, we can conclude that memory accesses have a strong negative impact on energy consumption.

The worst results among all the pseudo-applications were obtained with GS. Not considering OpenMP, the other PPIs did not present superior scalability compared to the sequential version. It was also the pseudo-application that obtained the highest energy consumption among all, with a peak of about 30,000 joules with 16 processes in MPI-2, but execution time does not increase that much proportionally. One of the reasons that MPI uses more energy than OpenMP accessing memory for the GS case is not that MPI is just inherently less efficient at accessing memory, but that using distributed memory requires redundant memory accesses. This way, more memory accesses are made.

The difference in these PPIs can be explained in the context of threads and processes. Threads are often referred to a lighter type of process for the system, while processes are heavier. A thread shares with other threads its code area, data, and operating system resources. Because of this sharing, the operating system needs to deal with less scheduling costs and thread creation, when compared to context switching by processes. All of these factors impact on performance and consequently on power consumption.

4.6 Power Consumption

All the differences in energy consumption and execution time seen in the section above do not show the power consumption of the pseudo-applications. The purpose of these data was to show the amount of energy each pseudo-application and PPI spent through a given problem size and how they perform varying the number of parallel tasks. We need to use energy and time to calculate the average power consumption (or power dissipation) of the pseudo-applications, according to Equation 4.1 (GARCIA; SCHEPKE; GIRARDI, 2018b). Where P is the power in Watts (W), E is the consumed energy in

Joules (J) and t is the spent time in seconds (s). Here we are considering both static and dynamic processor power consumption together. Static power refers to the power required to keep the processor in idle mode and dynamic when it is active.

$$P = \frac{E}{t} \quad (4.1)$$

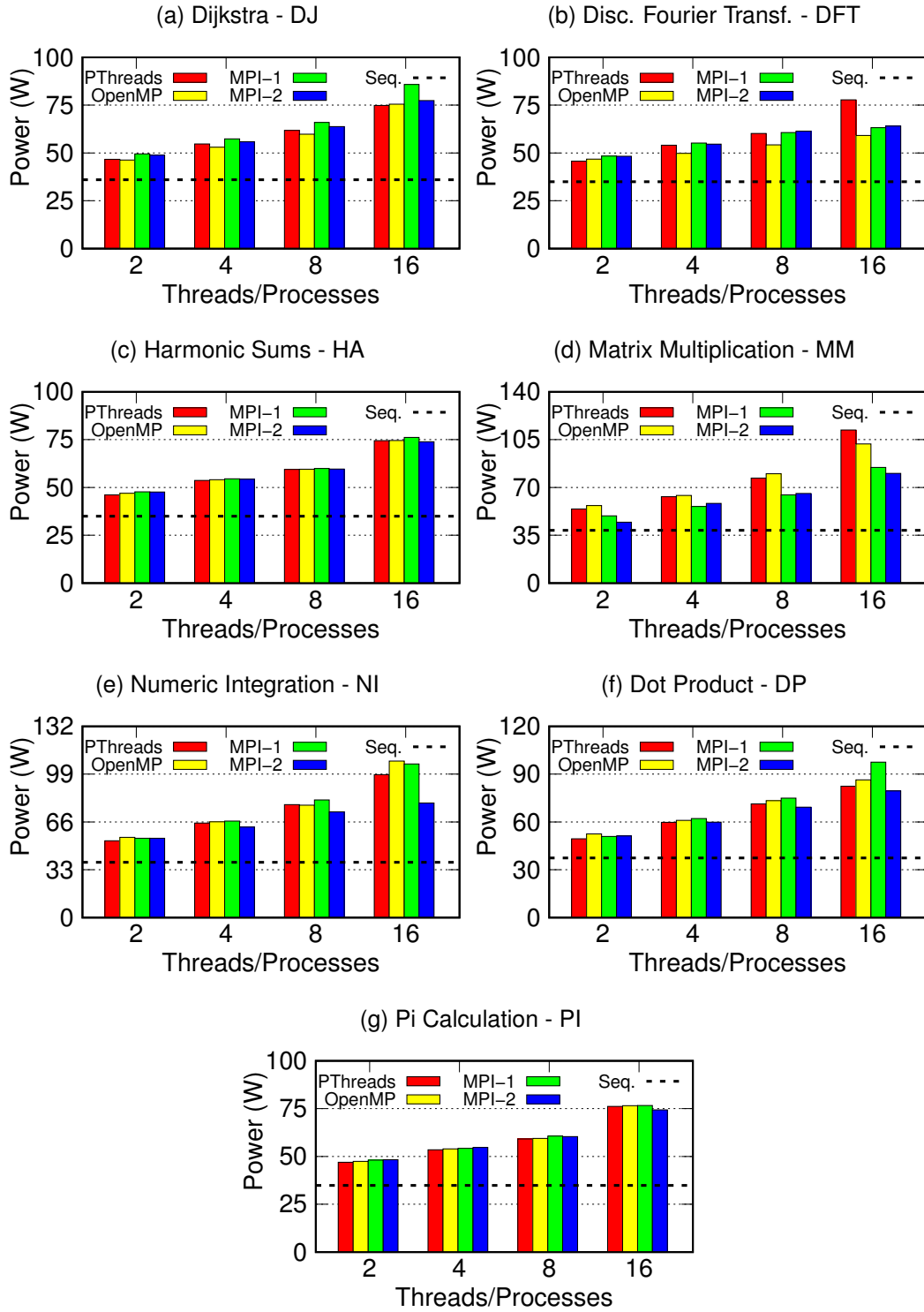
The charts in Figure 11 and Figure 12 show the power consumption in Watts for each PPI. Each chart displays the results by pseudo-applications individually. These results refer to running using 2, 4, 8, and 16 parallel threads/processes for each case. In addition, a dashed horizontal line represents the sequential result of the respective pseudo-application. In Figure 11 are presented the results for the pseudo-applications previously classified as CPU-bound. These results show that the power consumption of MPI-1 and MPI-2 is slightly higher when using 2 and 4 parallel tasks. However, for 8 and 16 processes, MPI 1 and 2 varies and have a power consumption equal to or lower than PPIs that use threads.

The DFT pseudo-application shows a different PThreads behavior compared to the other PPIs when the number of parallel tasks increases. With 2 threads PThreads follows the pattern that is seen in most CPU-Bound pseudo-applications. However, using more threads, this consumption exceeds the consumption of other PPIs up to 20%. If we look at Figure 10-e, we can see that the difference between the execution time and the energy consumption of PThreads in DFT with 16 threads is higher regarding the other PPIs. These other PPIs keep a pattern, where OpenMP consumes less power than MPI-1 and MPI-2.

MM and DFT show a similar behavior of PThreads in both pseudo-applications. The consumption of this PPI is lower than OpenMP with 2 threads but grows at a higher rate than the other PPIs as the number of threads increases. The difference in MM is that OpenMP also consumes more power than both MPI PPIs, but the rate does not increase as high as PThreads. In this case, PThreads consumes twice as much power with 16 threads as when running with 2 threads. In addition, using MPI-2 with 2 processes this pseudo-application was the one that most approached the consumption of the sequential version among the CPU-Bound pseudo-applications.

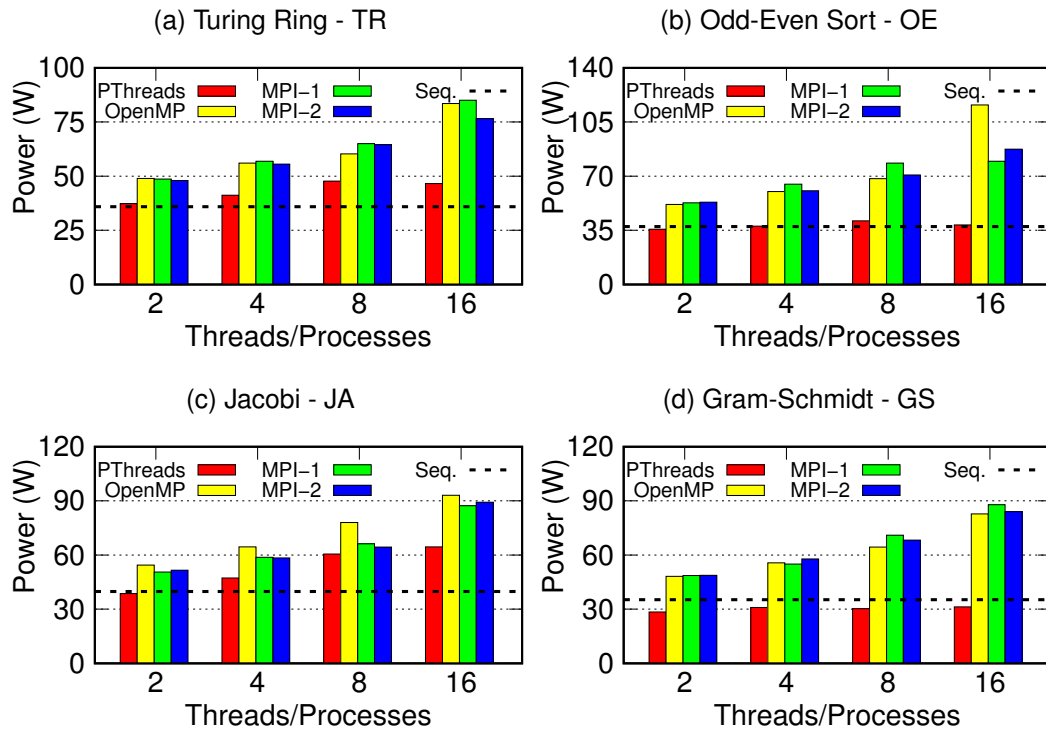
The low power consumption by PThreads in memory-bound pseudo-applications does not represent that the total power consumed was lower in this PPI. As seen in the section 4.5, runtime and energy consumption is higher than OpenMP. This low power consumption means that the pseudo-application consumed less energy over time, but that time was higher than OpenMP. This means that PThreads has a lower overhead caused by parallelization over OpenMP (Figure 1 shows that clearly). Therefore, the execution of PThreads takes more time but the use of hardware in this period is less intense compared to the other PPIs, which implies in lower consumption of energy over time. This increase in runtime can be caused by busy waiting for PThreads.

Figure 11: Power consumption for CPU-bound pseudo-applications.



Source: by the author.

Figure 12: Power consumption for memory-bound pseudo-applications.



Source: by the author.

In the memory-bound pseudo-applications (Figure 12), OE with OpenMP using 16 threads reached the highest power consumption. OpenMP achieves good performance but the energy consumption keeps the same when scaling to 16 threads. This leads to an increase in power consumption. OE is a weakly memory-bound pseudo-application, so the overhead of communication/synchronization among threads begins to impact negatively earlier for small input problems. With MPI-1 and MPI-2 the results obtained are similar to the results of CPU-bound pseudo-applications. The growth of power consumption as the number of parallel processes increases follows the same pattern previously observed. What is perceived is that MPI-2 has a lower consumption than MPI-1 in most cases for both CPU and memory-bound. This small difference may possibly be caused by dynamic process creation. This causes processes to be created later in MPI-2.

Another observed factor is that PThreads access less the memory system during synchronization. This means that for memory-bound programs parallelized using PThreads, this more robust processor we used is a good choice since it provides considerable performance improvements at the same price in energy consumption. For CPU-bound programs, the power consumption for each PPI is very similar. As the pseudo-applications use more CPU, the impact of particular characteristics of each communication model on the memory system is reduced. A characteristic we can

observe is that all sequential pseudo-applications consume about 35 Watts. In addition, all parallel pseudo-applications have an overhead, although PThreads and OpenMP cause a lower power consumption overhead in memory-bound pseudo-applications. This behavior is a consequence of all the variations between the PPIs observed in the previous sections.

4.7 Results Discussion

The experiments performed in this chapter were intended to show that the pseudo-applications exploit the architecture used in different ways. Our proposal is to have pseudo-applications that are representative for the greatest number of scenarios. For this, we analyze the number of cache accesses, branch instructions, and floating-point operations, and normalize this data to the total instructions. By doing this we reduce the impact of architecture on results. In addition, we measured the execution time to show the performance of the pseudo-applications and the energy consumption, where making a relation between both we obtained the power consumption that shows the energy efficiency.

For the results of accesses to cache, branches, and FLOPs, we are able to classify pseudo-applications that show a high, medium or low rate of these values per total instructions. This classification cannot be made with regard to run-time and energy consumption results since both data are presented individually on the same chart. On power consumption, there are also no reasons to perform this classification, because they are made based on sequential pseudo-applications and all sequential pseudo-applications show approximately 35 Watts of power consumption.

In the Table 3 we present the results of the cache accesses/branches/FLOPs per total instruction in a summarized way for easy visualization. In this table we classify these rates of sequential pseudo-applications in: high rate ($\star\star\star$), average rate ($\star\star$), low rate (\star), and minimal rate (no symbol). In addition, each PPI has a classification compared to the sequential version of the pseudo-application itself, based on a mean between the executions with different parallel tasks. For this classification we used $+$ symbols to represent how much overhead the PPI showed in regard to the sequential pseudo-application, applying the same previous rule. Similarly, we use $-$ symbols to indicate a drop in the PPI rate. When PPI showed results very close to the sequential version we used the $=$ symbol. Finally, we use the symbol $?$ for cases where there is much fluctuation among executions with different tasks.

As we can see in Table 3, for all cases of access to the cache, branches, and FLOPs there is at least one pseudo-application which represents one of the different categories (1, 2, and 3 stars). Among all pseudo-applications, GS was the one that presented the highest rates, with three stars for L2 and L3 cache accesses and FLOPs and two stars for branches. On the other hand is the DP pseudo-application, which

received only a single star for branches and no stars on the other parameters. In addition to these two extremes, there is an pseudo-application with three stars for cache access and no stars for FLOPs (MM), an pseudo-application with three stars for FLOPs and no star for cache access (NI), an pseudo-application with three stars for branches and only single stars in the other parameters (TR), and other cases with several other types of patterns.

The PPIs also presented a dynamic behavior among the analyzed parameters. The PPI that presented the least overhead cases upon sequential pseudo-application was PThreads, closely followed by OpenMP. Most + + + cases occurred with MPI-2, so it is the PPI that adds the largest overhead, followed closely by MPI-1. However, the MPI PPIs are also the ones that make the largest decrease (– – –) of the parameter rates (GS, HA, JA, and MM). Therefore, it is safe to say that MPI 1 and 2 also have the largest variation between increase and decrease in results. In addition, both PThreads and OpenMP and MPI-1 resulted in such extreme variations of FLOPs that they needed to be represented by the "?" symbol. Thus, in the same way, that sequential pseudo-applications represent several scenarios in the table, PPIs also present different scenarios within each pseudo-application.

This table shows that the pseudo-applications are well diversified regarding the use of analyzed parts. The objective of this study case is to show that the pseudo-applications are able to be representative in the tested scenario and our results show that they are. In addition, almost all of them presented good performance and scalability, even though we have evaluated limited cases (single input problem size and single machine). The power consumption analysis showed that the sequential pseudo-applications present similar results among themselves and that the characteristics of each PPI and each pseudo-application have a great impact on the final result. The parallel pseudo-applications have higher power consumption but lower energy consumption in most cases. All of these different features end up either improving or worsening performance in certain aspects. So it's up to the programmer to choose what works best for him.

There is plenty of room for improving the quality of our results. Setting threads/affinity processes would bring greater reliability to our analysis. It would also be better if we had results with 20 parallel tasks for power and runtime. Running the pseudo-applications in another architecture or in a distributed system could bring very different results. Relating the data in other ways would lead to different conclusions. Finally, using other values of parallel tasks and input sizes could also present different results. However, it is not easy to deal with so many pseudo-applications and PPIs at the same time and the preparation of experiments and mainly bug fixes took up most of the time of our study.

Table 3: Summary of results. Each \star represents an increasing in the respective rate, and $+$ or $-$ represent more or less overhead added by the PPIs.

		L3 accesses	L2 acceses	Branches	FLOPs
Discrete Fourier Transform	Sequential		\star	$\star\star$	$\star\star\star$
	PThreads	$+$	$=$	$+$	$=$
	OpenMP	$+$	$++$	$+$	$=$
	MPI-1	$++$	$-$	$+$	$=$
	MPI-2	$+++$	$+$	$+$	$=$
Dijkstra	Sequential	\star		$\star\star$	
	PThreads	$+$	$+$	$=$	$+$
	OpenMP	$+$	$+$	$=$	$+$
	MPI-1	$+++$	$+++$	$++$	$++$
	MPI-2	$+++$	$+++$	$++$	$+++$
Dot Product	Sequential			\star	
	PThreads	$+$	$+$	$=$	$--$
	OpenMP	$++$	$++$	$=$	$--$
	MPI-1	$+++$	$+++$	$=$	$+++$
	MPI-2	$+++$	$+++$	$+$	$+++$
Gram-Schmidt	Sequential	$\star\star\star$	$\star\star\star$	$\star\star$	$\star\star\star$
	PThreads	$-$	$=$	$=$	$--$
	OpenMP	$-$	$-$	$++$	$-$
	MPI-1	$--$	$--$	$=$	$?$
	MPI-2	$---$	$---$	$++$	$---$
Harmonic Sums	Sequential	\star	\star	$\star\star$	$\star\star\star$
	PThreads	$-$	$-$	$-$	$-$
	OpenMP	$=$	$=$	$=$	$=$
	MPI-1	$-$	$++$	$-$	$---$
	MPI-2	$--$	$+$	$-$	$---$
Jacobi Method	Sequential	$\star\star$	$\star\star$	\star	\star
	PThreads	$-$	$+$	$=$	$=$
	OpenMP	$---$	$=$	$+$	$-$
	MPI-1	$---$	$---$	$++$	$+++$
	MPI-2	$---$	$---$	$+++$	$+$
Matrix Multiplication	Sequential	$\star\star\star$	$\star\star\star$	\star	
	PThreads	$=$	$=$	$=$	$?$
	OpenMP	$=$	$=$	$=$	$?$
	MPI-1	$---$	$---$	$++$	$+++$
	MPI-2	$---$	$---$	$+++$	$++$
Numeric Integration	Sequential			$\star\star$	$\star\star\star$
	PThreads	$+$	$+$	$---$	$+$
	OpenMP	$++$	$++$	$---$	$+$
	MPI-1	$+++$	$+++$	$---$	$+$
	MPI-2	$+++$	$+++$	$---$	$=$
Odd-Even Sort	Sequential	\star	$\star\star$	$\star\star$	
	PThreads	$=$	$--$	$-$	$++$
	OpenMP	$---$	$---$	$=$	$+$
	MPI-1	$---$	$---$	$-$	$+++$
	MPI-2	$-$	$--$	$+$	$+++$
Pi Calculation	Sequential			\star	$\star\star$
	PThreads	$+$	$+$	$=$	$=$
	OpenMP	$++$	$++$	$=$	$=$
	MPI-1	$+++$	$+++$	$=$	$-$
	MPI-2	$+++$	$+++$	$+$	$--$
Turing Ring	Sequential	\star	\star	$\star\star\star$	\star
	PThreads	$-$	$=$	$-$	$-$
	OpenMP	$---$	$=$	$=$	$=$
	MPI-1	$-$	$++$	$=$	$?$
	MPI-2	$=$	$+++$	$=$	$+$

Source: by the author.

5 CONCLUSION

In this work, we propose a set of pseudo-applications to be used as a benchmark. The main purpose of this benchmark is to analyze energy consumption and performance of different parallel programming interfaces in multi-core architectures. We first did a study on related work that showed the need to have such a benchmark. So, we compared our benchmark to the main parallel benchmarks that are currently used for the same purpose. This comparison showed that there is no benchmark that meets the proposed goal: to offer a simpler way to compare PPIs.

In addition, we did a study of the history of the pseudo-applications, where we showed that there were authors using them for the same purpose. This fact meant that there was no other benchmark that would effectively meet this demand. So it was necessary to create one from scratch.

Our experimental results showed that the pseudo-applications use the dynamically evaluated hardware resources in this specific architecture. These results showed that some PPIs in the same pseudo-application are capable of increasing or decreasing the rate of a given parameter, only by varying the number of parallel tasks. The execution time results showed that the pseudo-applications generally have a good performance gain. In addition, energy consumption showed a behavior proportional to the number of threads/processes used in parallel.

Some pseudo-applications in our benchmark still require a few more adjustments and deep analysis, as it can be seen in the results. However, regarding runtime, in most cases we have achieved near optimal performance, reducing runtime twice with 2 threads, 4 times with 4 threads, and so on. In addition, energy consumption has also been reduced in the same proportion for this majority of cases. In the end, even the pseudo-applications that made high energy consumption showed a power consumption similar to the others.

The objectives of this work have been completed. We were able to collect different data parts of the system for each pseudo-application and PPI. Thus, we can summarize these results and make explicit the differences and similarities between the pseudo-applications. In this way, whoever uses the benchmark can quickly see the possible behavior of an pseudo-application. In addition, we update the pseudo-applications and from now they will be available in an online repository, where we intend to add more features to improve usability. The pseudo-applications have shown that they represent rather different scenarios and that they can be perfectly used as a benchmark for evaluating parallel programming interfaces.

As future work, we intend to improve the quality of the experiments by solving the problems discussed at the end of the previous chapter. We will also need to create appropriate documentation for the pseudo-applications and update them in the online

repository. In addition, we have two new pseudo-applications (Histogram Similarity and Game of Life) that are already implemented and we are making final adjustments to include them in the benchmark. We also consider including more PPIs such as Intel TBB, UPC or Intel Cilk. To build this benchmark, our proposal is to have pseudo-applications that are representative for the greatest number of scenarios. There is still a long way to go, but there are other developers already implementing the pseudo-applications in other PPIs, which are important steps for the success of this project.

BIBLIOGRAPHY

- Adept-Project. **Adept: addressing energy in parallel technologies**. 2018. Available at: <http://www.adept-project.eu>. Accessed in: 26 Nov. 2018. Cited on page 23.
- ANDREWS, G. E.; ROY, R. **Special Functions**. Cambridge, UK; New York, NY, USA: Cambridge University Press, 1999. ISBN 9780521623216. Cited on page 28.
- BAILEY, D. H. et al. The NAS Parallel Benchmarks. **International Journal of High Performance Computing Applications**, SAGE Publications, v. 5, n. 3, p. 63–73, 1991. Cited on page 23.
- BANCHOFF, T.; WERMER, J. **Linear algebra through geometry**. Crinan Street, London UK: Springer Science & Business Media, 2012. Cited on page 29.
- BOLARIA, J.; HALFHILL, T. R. **A Guide to Multicore Processors**. Chesley Avenue Mountain View, CA, 2017. Available at: http://www.linleygroup.com/report_detail. Accessed in: 10 Oct. 2018. Cited on page 15.
- BORWEIN, J. M.; BORWEIN, P. B.; DILCHER, K. Pi, Euler Numbers, and Asymptotic Expansions. **The American Mathematical Monthly**, Mathematical Association of America, v. 96, n. 8, p. pp. 681–687, 1989. ISSN 00029890. Cited on page 28.
- Bourzac, Katherine. **Supercomputing poised for a massive speed boost**. Crinan Street, London UK: Springer Nature International Journal of Science, 2017. Cited on page 15.
- BURDEN, R. L.; FAIRES, J. D.; REYNOLDS, A. C. **Numerical analysis**. Pacific Grove, CA: Brooks, 2001. Cited 2 times on pages 33 and 34.
- BUTENHOF, D. R. **Programming with POSIX threads**. Boston, Massachusetts USA: Addison-Wesley Professional, 1997. Cited 2 times on pages 20 and 37.
- CHENEY, W.; KINCAID, D. Linear algebra: Theory and applications. **The Australian Mathematical Society**, p. 654, 2009. Cited on page 35.
- CHETSA, G. T. et al. Exploiting performance counters to predict and improve energy performance of hpc systems. **Future Generation Computer Systems**, Elsevier, v. 36, p. 287–298, 2014. Cited on page 21.
- DANELUTTO, M. et al. P³arsec: towards parallel patterns benchmarking. **Proceedings of the Symposium on Applied Computing**, p. 1582–1589, 2017. Cited on page 22.
- DIJKSTRA, E. W. A note on two problems in connexion with graphs. **Numerische Mathematik**, v. 1, p. 269–271, 1959. Cited on page 33.
- DOEFLER, D.; BARRETT, B. W. Sandia mpi microbenchmark suite (smb). **Technical report, Sandia National Laboratories**, 2009. Cited on page 22.
- DOSANJH, M. G. et al. Tail queues: A multi-threaded matching architecture. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, 2019. Cited on page 22.
- EJJAOUANI, K. et al. Inks, a programming model to decouple performance from algorithm in hpc codes. **Reengineering for Parallelism in Heterogeneous Parallel Platforms (REPARA)**, p. 1–12, 2018. Cited on page 16.

- FOSTER, I. **Designing and building parallel programs**. [S.l.]: Addison Wesley Publishing Company, 1995. Cited on page 37.
- GARCIA, A. M. **Classificação de um benchmark paralelo para arquiteturas multinúcleo**. 2016. Monography (Bachelor of Computer Science), UNIPAMPA (Federal University of Pampa), Alegrete, Brazil. Cited 4 times on pages 16, 17, 38, and 60.
- GARCIA, A. M.; SCHEPKE, C. Uma proposta de benchmark paralelo para arquiteturas multicore. **XVIII Escola Regional de Alto Desempenho**, p. 285–289, 2018. Cited 2 times on pages 38 and 60.
- GARCIA, A. M.; SCHEPKE, C.; GIRARDI, A. G. A new parallel benchmark for performance evaluation and energy consumption. **13th International Meeting on High Performance Computing for Computational Science (VECPAR)**, 2018. Cited on page 57.
- GARCIA, A. M.; SCHEPKE, C.; GIRARDI, A. G. Power consumption of parallel programming interfaces in multicore architectures: A case study. **Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)**, p. 149–159, 2018. Cited on page 60.
- GOLDSTON, D.; YILDIRIM, C. Y. **On the second moment for primes in an arithmetic progression**. 2001. 85–104 p. Cited on page 29.
- GRAY, J. **Benchmark handbook: for database and transaction processing systems**. Burlington, Massachusetts USA: Morgan Kaufmann Publishers Inc., 1992. Cited 2 times on pages 19 and 20.
- GRIEBLER, D. et al. Efficient nas benchmark kernels with c++ parallel programming. **26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)**, p. 733–740, 2018. Cited on page 22.
- GROPP, W.; LUSK, E.; THAKUR, R. **Using MPI-2: Advanced features of the message-passing interface**. One Rogers Street, Cambridge USA: MIT press, 1999. Cited 2 times on pages 20 and 37.
- HUNOLD, S.; CARPEN-AMARIE, A. Reproducible mpi benchmarking is still not as easy as you think. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 27, n. 12, p. 3617–3630, 2016. Cited on page 45.
- IC-INSIGHTS. **The McClean Report**. Scottsdale, Arizona USA, 2018. Available at: <http://www.icinsights.com/services/mcclean-report>. Accessed in: 9 Sep. 2018. Cited on page 15.
- INTEL. **Parallel Programming and Heterogeneous Computing at Your Fingertips with Threading Building Blocks**. 2018. Available at: <https://software.intel.com/en-us/intel-tbb>. Accessed in: 19 Sep. 2018. Cited on page 16.
- IQBAL, S.; LIANG, Y.; GRAHN, H. ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems. **Computer Architecture Letters**, v. 9, n. 2, p. 45–48, Feb 2010. ISSN 1556-6056. Cited on page 23.

ISIDRO-RAMIREZ, R.; VIVEROS, A. M.; RUBIO, E. H. Energy consumption model over parallel programs implemented on multicore architectures. **International Journal of Advanced Computer Science and Applications (IJACSA)**, v. 6, n. 6, 2015. Cited on page 21.

JAIN, S. et al. Parallelizing mpi using tasks for hybrid programming models. **International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**, p. 1303–1312, 2018. Cited on page 22.

KNUTH, D. E. **The art of computer programming: sorting and searching**. [S.l.]: Pearson Education, 1998. v. 3. Cited on page 30.

LIVELY, C. et al. E-amom: an energy-aware modeling and optimization methodology for scientific applications. **Computer Science-Research and Development**, Springer, Salmon Tower Building, New York City, v. 29, n. 3-4, p. 197–210, 2014. Cited on page 21.

LIVELY, C. et al. Power-aware predictive models of hybrid (mpi/openmp) scientific applications on multicore systems. **Computer Science-Research and Development**, Springer, v. 27, n. 4, p. 245–253, 2012. Cited on page 21.

LORENZON, A. **Avaliação do desempenho e consumo energético de diferentes interfaces de programação paralela em sistemas embarcados e de propósito geral**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 2014. Cited 6 times on pages 16, 17, 24, 27, 37, and 39.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Performance and Energy Evaluation of Different Multi-Threading Interfaces in Embedded and General Purpose Systems. **Journal of Signal Processing Systems**, Springer, v. 80, n. 3, p. 295–307, 2014. Cited on page 37.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Optimized use of parallel programming interfaces in multithreaded embedded architectures. **IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**, p. 410–415, 2015. Cited 2 times on pages 16 and 37.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy. **Journal of Parallel and Distributed Computing**, Elsevier, v. 95, p. 107–123, 2016. Cited 3 times on pages 38, 51, and 60.

LORENZON, A. F. et al. The Influence of Parallel Programming Interfaces on Multicore Embedded Systems. **Computer Software and Applications Conference (COMPSAC)**, v. 2, p. 617–625, 2015. Cited 2 times on pages 16 and 38.

PAUDEL, J.; AMARAL, J. N. Using the Cowichan Problems to Investigate the Programmability of X10 Programming System. **Proceedings of the 2011 ACM SIGPLAN X10 Workshop**, ACM, New York, NY, USA, p. 4:1–4:10, 2011. Cited on page 32.

PETITET, A. **HPL-a portable implementation of the high-performance Linpack benchmark for distributed-memory computers**. Available at: www.netlib.org/benchmark. Accessed in: 4 Feb. 2019. Cited on page 23.

POPOV, M. et al. Pcere: Fine-grained parallel benchmark decomposition for scalability prediction. In: **Parallel and Distributed Processing Symposium**. Chicago, USA: IPDPS, 2015. p. 1151–1160. Cited on page 22.

PRESS, W. H. et al. **Numerical Recipes 3rd Edition: The Art of Scientific Computing**. Cambridge, England: Cambridge University Press, 2007. Cited on page 34.

Princeton University. **PARSEC: A Unity of Measure**. 2018. Available at: <http://parsec.cs.princeton.edu>. Accessed in: 15 Oct. 2018. Cited on page 22.

RAUBER, T.; RÜNGER, G. **Parallel programming: For multicore and cluster systems**. Crinan Street, London UK: Springer Science & Business Media, 2010. Cited 4 times on pages 15, 19, 20, and 37.

RODRIGUES, R. et al. A study on the use of performance counters to estimate power in microprocessors. **IEEE Transactions on Circuits and Systems II: Express Briefs**, IEEE, v. 60, n. 12, p. 882–886, 2013. Cited on page 21.

ROMAN, R.; LOPEZ, J.; MAMBO, M. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. **Future Generation Computer Systems**, Elsevier, v. 78, p. 680–698, 2018. Cited on page 15.

ROY, R. The Discovery of the Series Formula for pi by Leibniz, Gregory and Nilakantha. **Mathematics Magazine**, Mathematical Association of America, v. 63, n. 5, p. pp. 291–306, 1990. ISSN 0025570X. Cited on page 28.

SEZNEC, A. **Defying amdahl's law-dal**. Domaine de Voluceau Rocquencourt, France, 2016. Available at: <https://team.inria.fr/alf/members/andre-seznec/defying-amdahls-law-dal>. Accessed in: 2 Dec. 2018. Cited on page 15.

SMITH; STEVEN, W. et al. **The scientist and engineer's guide to digital signal processing**. San Diego, California USA: California Technical Pub. San Diego, 1997. Cited on page 30.

SONG, S. et al. A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In: **Proceedings of 27th International Symposium on Parallel & Distributed Processing (IPDPS)**. Rio de Janeiro: IPDPS, 2013. p. 673–686. Cited on page 21.

SPEC. **Standard Performance Evaluation Corporation**. 2018. Available at: <http://www.spec.org>. Accessed in: 4 Feb. 2019. Cited 2 times on pages 19 and 23.

STEWART, J. Cálculo, vol. 1. **Pioneira Thomson Learning**, 2001. Cited on page 27.

TERPSTRA, D. et al. Collecting performance data with papi-c. In: **Tools for High Performance Computing 2009**. Salmon Tower Building New York City: Springer, 2010. p. 157–173. Cited on page 45.

TURING, A. M. The chemical basis of morphogenesis. **Phil. Trans. Roy. Soc.**, <http://www.bibsonomy.org/bibtex/28607650779098f64c92839e48fe2908c/bronckobuster>, v. 237, p. 37, 1952. Cited on page 31.

UHSADEL, L.; GEORGES, A.; VERBAUWHEDE, I. Exploiting hardware performance counters. **5th Workshop on Fault Diagnosis and Tolerance in Cryptography**, p. 59–67, 2008. Cited on page 21.

University of Illinois. **ALP: All Levels of Parallelism for Multimedia**. 2018. Available at: <http://rsim.cs.uiuc.edu/alp/alpbench>. Accessed in: 8 Jun. 2018. Cited on page 22.

WU, X.; TAYLOR, V. Utilizing hardware performance counters to model and optimize the energy and performance of large scale scientific applications on power-aware supercomputers. In: **Proceedings of Parallel and Distributed Processing Symposium Workshops**. London, UK: IPDPS, 2016. p. 1180–1189. Cited on page 21.

Appendix

APPENDIX A – L2 CACHE ACCESSES RESULTS

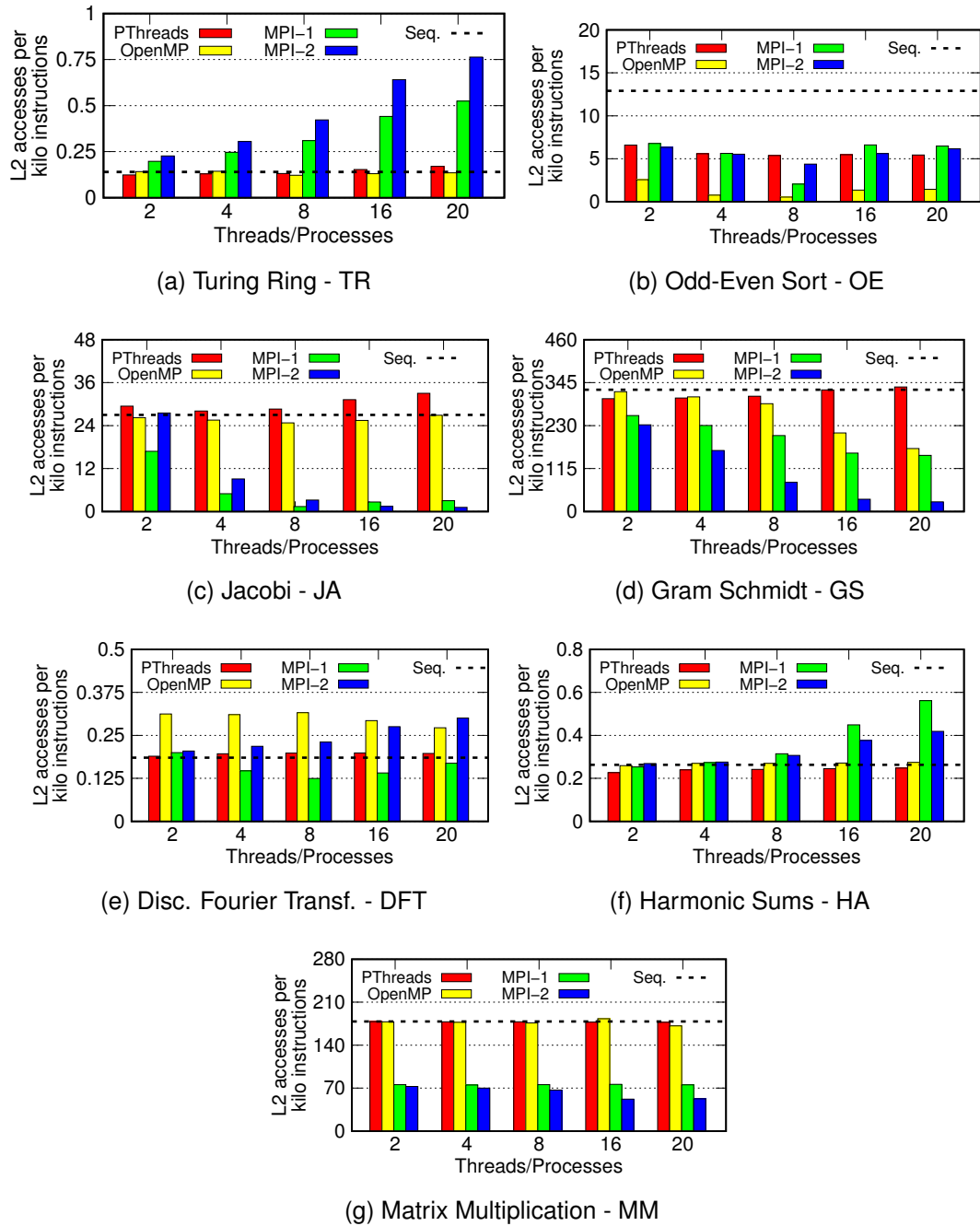


Figure 13: L2 cache accesses per kilo instructions (A).

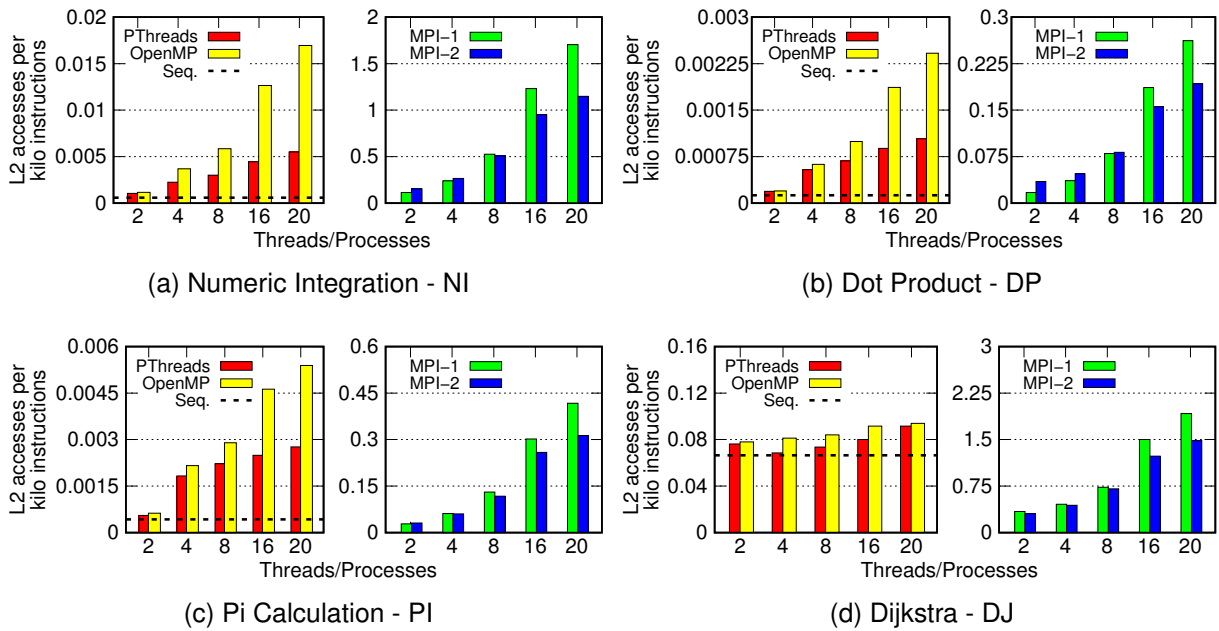


Figure 14: L2 cache accesses per kilo instructions (B).