PAMPAR: A new parallel benchmark for performance and energy consumption evaluation

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

PAMPAR - A New Parallel Benchmark for Performance and Energy Consumption Evaluation.

Adriano Marques Garcia*[1,2] | Claudio Schepke[1] | Alessandro Girardi[2]

[1]Advanced Studies Laboratory, Federal University of Pampa, Alegrete, Brazil
[2]Computer Architecture and Microelectronics Group, Federal University of Pampa, Alegrete, Brazil

Correspondence
Adriano Marques Garcia, Federal University of Pampa, Alegrete - 97546-550, RS, Brazil.
Email: adriano1mg@gmail.com

Summary

This paper presents PAMPAR, a new benchmark to evaluate the performance and energy consumption of different Parallel Programming Interfaces (PPIs). The benchmark is composed of 11 algorithms implemented in PThreads, OpenMP, MPI-1, and MPI-2 (spawn) PPIs. 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. In this work we measure the energy and performance of each pseudo-application in a single architecture, varying the number of threads/processes. We also organize the pseudo-applications according to their memory accesses, floating-point operations, and branches. The goal is to show that this set of pseudo-applications has enough features to build a parallel benchmark. The results show that there is no single best case that provides both better performance and low energy consumption in the presented scenarios. Also, the pseudo-applications usage of the system resources are different enough to represent different scenarios and be efficient as a benchmark.

KEYWORDS:
Parallel benchmark, performance, energy consumption, power consumption, OpenMP, PThreads, MPI.

1 | INTRODUCTION

Multicore processors have become the dominant form of general purpose processors. In addition, manycore processors are being extensively used in embedded and high-performance computing. This transition created a disruptive change: substantial performance gains for applications can no longer be achieved without modifying the underlying source code[1]. Future applications must 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[2]. The other one is that embedded processors tend to outperform server and general-purpose processors in the multicore 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%[3]. 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 has been 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 feeds and speeds. Most embedded applications do not need

*This is a preprint version from the authors of Garcia A. M., Schepke C, Girardi A. PAMPAR: A new parallel benchmark for performance and energy consumption evaluation. Concurrency Computat Pract Exper. 2020; 32:e5504.
https://doi.org/10.1002/cpe.5504

leading-edge performance, but they do demand the best power efficiency and the lowest power consumption[4]. 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 even faster multiple parallel processors. Parallel computing aims to use multiple processors to execute different parts of the same program simultaneously[5]. However, processors should be able to exchange information at a certain point in 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 peculiarities 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 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 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[6]. Another example is the InKS[7], 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 is executing in the 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. A benchmark which provides different types of applications implemented in several Parallel Programming Interfaces could be useful to draw a relationship between performance and power consumption. However, there is not a consolidated benchmark that offers a good set of 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.

To fill this gap, this work proposes PAMPAR, 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 developed and classified according to different criteria in previous studies[8,9,10,11]. These studies have shown that these pseudo-applications have characteristics that are distinct enough to be representatives in different scenarios.

The main objective of this work is to use these 11 pseudo-applications to build a benchmark to evaluate the performance and power consumption of different parallel programming interfaces. The benchmark is composed of 11 pseudo-applications implemented in POSIX Threads (PThreads), OpenMP, MPI-1, and MPI-2 (spawn) PPIs. These PPIs were chosen because they are compatible with most multicore architectures. To achieve our goals we conduct case studies where we evaluate different parts of the system. We evaluated the behavior of the pseudo-applications in relation to cache accesses, branch instructions and floating-point operations. In addition, we evaluated the performance and energy consumption and related them to obtain power consumption. It allow us to observe the scalability of pseudo-applications and the impact of their characteristics on power consumption. To improve the understanding of the results, in the end, we summarize the results in a table which shows an overview of our results. The contributions of this work are as follows:

- the set of pseudo-applications did not have a 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;
- previous studies used these pseudo-applications to do general analysis over this set and the focus has never been on analyzing and comparing the pseudo-applications as a set, but rather architectures or PPIs. In our analyses we present results, relating and comparing the pseudo-applications among themselves.

The remainder of this work is organized as follows. In the Section 2 we present a background for this work, we describe the PPIs in which the pseudo-applications are implemented, how we gather data from them, and our related work. Section 3 presents the set of pseudo-applications and the techniques used to parallelize them, bringing more details about the historic of classifications. Section 4 presents our experimental results and the methodology applied. Finally, Section 5 draws the final considerations.

2 | BACKGROUND

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 architectures [12]. 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 [13]. 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 [14]. 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 [5]. 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 [15]. 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 [16]. 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 [17].

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[16]. 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[18 19 20 21 22]. 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. An example is the work of Isidro-Ramirez et al.[23], 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.[24] 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), Jang et al.[25] had to use different benchmarks for the experiments. In another work, Dosanjh et al.[26] had to modify the MPI Sandia Microbenchmarks (SMBs)[27] 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 works in which building a new benchmark class is a part of the major goal. This 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.[28] 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.[29] identified the lack of low-level mechanisms and implemented P^{3}ARSEC, 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[30] 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[31]. 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[32]. 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[12] 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

TABLE 1 Comparison of PAMPAR with the similar benchmarks

Rating criteria	ALPBench	PARSEC	ParMiBench	SPEC	HPL	NAS	Adept	PAMPAR
Number of applications	**5**	**13**	**7**	**32**	**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**

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[33]. 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[34] 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[35] 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, the Kernel set has 10 applications implemented sequentially and parallel with OpenMP, MPI and one of them in UPC (Unified Parallel C).

PAMPAR is the benchmark we present in this work and consists of 11 applications implemented in C. All applications are parallelized in 4 PPIs: PThreads, OpenMP, MPI-1 and MPI-2. These PPIs are the target of this research 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 analyse the main characteristics of the related parallel benchmarks above and compare to the PAMPAR in Table 1. Some of theses benchmarks 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 parallelized for all PPIs. They implement parts of the set with one PPI and other parts with another PPI.

SPEC, for example, has 32 parallel applications divided into two different sets. One of them uses OpenMP and the other one uses MPI, so it does not provide a set implemented with two or more PPIs. NAS, on the other hand, uses several other PPIs. However, most of applications are not implemented in all PPIs. Many of them are not supported by any multicore architecture. 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 of these PPIs and different communication models (message passing or shared memory). 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 NAS, but it only offers two PPIs. Therefore, none of them meets the objective of comparing parallel programming interfaces, which is the main goal of PAMPAR.

3 | PAMPAR PSEUDO-APPLICATIONS

This Section describes the 11 pseudo-applications that are included by PAMPAR. All pseudo-applications were developed with the purpose of establishing a relationship between performance and energy consumption in embedded systems and general purpose architectures[36].

- **Odd-Even Sort** - Odd-Even is a sorting algorithm that compares all indexed (odd-even) pairs of adjacent elements in a list. If a pair is in the wrong order (the first element is greater than the second), the elements are swapped. The next step repeats the same procedure for the indexed (even-odd) pairs of adjacent elements. It then toggles between odd-even and even-odd steps until the list is sorted. This pseudo-application is suitable for evaluating the PPI performance under the presence of a high number of branches and has $O(n^2)$ complexity.

- **Dijkstra** - It finds a minimal cost path between nodes in a graph with non-negative edges. This pseudo-application finds a path from each of n nodes to each of $n-1$ remaining nodes. Our implementation uses a $N \times N$ adjacency matrix. Considering n being the number of vertices, the complexity of the algorithm is $O(n^2)$. However, as our pseudo-application considers the smallest path from n to n vertices it brings its total complexity to $O(n^3)$. The output is a vector containing the minimum distance between each pair of vertices. This application performs only integer operations and has an average number of branches.

- **Gram-Schmidt** - The Gram-Schmidt algorithm is a method for orthonormalising a set of vectors in an inner product space[37]. It is an $O(n^3)$ algorithm and is representative for pseudo-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:

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

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

$$\langle v, u \rangle = v_1 u_1 + \cdots + v_n u_n \tag{2}$$

- **Matrix Multiplication** - This algorithm multiplies the lines of a matrix A by the columns of a matrix B and stores the results in a matrix C, according to the equation 3. This is an $O(n^3)$ algorithm and it represents pseudo-applications with high number of memory accesses.

$$C = (AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}. \tag{3}$$

- **Jacobi Method** - The Jacobi method is an algorithm for determining the solutions of a diagonally dominant system of linear equations[38]. 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[38]. 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 [39] by using equation 4. x^k is the k^{th} iteration of x. This is a $O(n^3)$ memory-bound pseudo-application and performs few branches and FLOPs.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), i = 1, 2, \ldots, n \tag{4}$$

- **Dot Product** - The dot product is an algebraic operation that multiplies two equal-length sequences of numbers and is defined by the equation 5. The dot product is used in Euclidean geometry and is a special case of inner product [40]. It can represent pseudo-applications with simple integer calculations and low complexity ($O(n)$).

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \tag{5}$$

- **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 [41]. 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 6, where: N is the number of samples; n 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. The algorithm has $O(n^2)$ complexity and is suitable for representing CPU-bound pseudo-applications performing a high number of FLOPs.

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot \left(cos\left(2\pi k \frac{n}{N}\right) + j sin\left(2\pi k \frac{n}{N}\right) \right), \; n \in \mathbb{Z} \tag{6}$$

- **Harmonic Sums** - The Harmonic Sums or Harmonic Series is a finite series that calculates the sum of arbitrary precision after the decimal point [42]. 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 7. This algorithm represents pseudo-applications with average number of branches and high number of FLOPs. It has complexity of $O(n \times d)$, where n is the number of iterations and d is the size of the vector.

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \tag{7}$$

- **PI Calculation** - It applies the Gregory-Leibniz method (equation 8) to find π. Although this method is considered inefficient for a few iterations, precision increases with more iterations [43]. It generates π with an accuracy of 5 decimal places after 500 thousand iterations and 10 decimal places with 5×10^9 iterations. This algorithm can represent pseudo-applications with low complexity ($O(n)$) and average number of floating-point operations.

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4} \tag{8}$$

- **Numerical 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 [44]. This pseudo-application integrates the function $f(x)$ (equation 11) in the $[a, b]$ interval. The basic method

involved in this approximation is called numerical quadrature and is expressed by the equation 9, where α_i is real coefficient (weight of the function) and x_i, is a sampling point of $[a, b]$ defined by equation 10. This pseudo-application has $O(n)$ complexity and is ideal for evaluating the FPU because of its low complexity and high number of floating point operations.

$$\int_b^a f(x)dx \simeq \sum_{i=0}^{n} \alpha_i f(x_i) \tag{9}$$

$$x_i = \frac{(n-i-1)\cdot(a+(i\cdot b))}{(n-1)} \tag{10}$$

$$f(x) = \frac{50}{\pi\cdot(2500\cdot(x^2+1))} \tag{11}$$

- **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[45]. It is a space system in which predator (x) and prey (y) interact in the same environment. The system simulates the iteration and evolution between prey and predator through the use of two differential equations 12 and 13. r is the birth rate, c represent local interactions, μ is the migration rate between two neighbouring cells and the predator and prey populations in location i are represented by X_i and Y_i. Evolution is defined according to the neighbouring cells[46]. This pseudo-application has a high number of branches and has $O(m \times n^2)$ complexity, where m is the number of evolutions of the society and n the size of the matrix.

$$\frac{dX_i}{dt} = X_i(r_x + c_{xx}X_i + c_{xy}Y_i) + \mu_x(X_{i+1} + X_{i-1} - 2X_i) \tag{12}$$

$$\frac{dY_i}{dt} = Y_i(r_y + c_{yx}X_i + c_{yy}Y_i) + \mu_y(Y_{i+1} + Y_{i-1} - 2Y_i) \tag{13}$$

These algorithms are used in the most diverse computing areas. Four of them are directly related to linear algebra. However, some other areas are also represented, such as molecular dynamics, electromagnetism, digital signal processing, image processing, mathematical optimization, among others.

3.1 | Pseudo-Applications Complexity

Table 2 shows the data structure, the problem size, the acronym used to identify each pseudo-application in the Section 4, and their complexities. The last column of Table 2 presents the algorithmic complexity by pseudo-application. Only the serial pseudo-applications were used for this complexity analysis, which is based on the arithmetic operations of the algorithm. The complexity analysis for parallel pseudo-applications is mainly based on execution time. However, several factors influence the complexity of parallel pseudo-applications, such as load balancing and parallelization model[47].

The analysis of sequential complexities shows that the set of pseudo-applications range from $O(n)$ to $O(n^3)$, with several other intermediate complexities. In this way, the benchmark presents enough diversity in this aspect to evaluate performance in different architectures.

3.2 | Parallelizing the Pseudo-Applications

Parallelizing a sequential program can be done in several ways. However, inappropriate techniques can produce a negative impact in the performance of an application. To minimize this problem, all parallel implementations in this work were based on statements from[48,15,14,5], which 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 application, which are: computation decomposition; tasks to processes/threads assignment; processes/threads to physical processing units mapping.

TABLE 2 Details about the pseudo-applications

Data Structure	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	$O(n^2)$
	32768	DFT	Discrete Fourier Transf.	
Matrix	2048×2048	TR	Turing Ring	$O(m \times n^2)$
		DJ	Dijkstra	$O(n^3)$
		JA	Jacobi Method	
		MM	Matrix Multiplication	
		GS	Gram-Schmidt	

The decomposition of the computation and assignment of tasks to processes/threads occur explicitly in the parallelization with PThreads and MPI 1 and 2, in order to obtain the best workload balancing. We also included message exchange functions among processes, as well as the dynamic creation of processes in MPI-2. The workload balancing was done by breaking down the problem into chunks using coarse granularity and equally distributing them to the tasks. This technique is most appropriate for parallelizing applications that perform iterative calculations and traverse contiguous data structures (e.g. matrix, vector, etc.) [48,5].

3.3 | Pseudo-Applications History

The set of pseudo-applications that compose the benchmark have already been investigated in previous works. The pseudo-applications were used to analyze performance and energy consumption on embedded systems and general purpose processors. In [9,10,36] the authors classified the pseudo-applications in 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 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 application execution;
5. **Communication rate** - represents the volume of communication required by threads/processes during application execution.

Lorenzon et al. [8] used the number of data exchange operations as a criterion for classification. In the target PPI, 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.

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 application.

An application that performs many accesses to shared memory could also have a high CPU usage. In the opposite case, an application with few accesses to memory and previously classified as CPU-Bound could also make less use of CPU in relation to the other application classified as Memory-Bound.

After that, Garcia et al.[11] 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 et al.[49] used some of these pseudo-applications to verify the best performance and energy consumption in different multi-core architectures.

Gathering all these previous studies, we conclude that this set contains pseudo-applications diverse enough to characterize a benchmark. After all, they are being used as a benchmark, but an effort is necessary to unify them, to analyze the whole set focusing on the pseudo-applications (not only on the PPIs, as has been done so far) and to prepare them for use by others. This is one of the main objectives of this work.

4 | EXPERIMENTAL RESULTS

In this chapter, we present our results. The original benchmark suite, provided by Lorenzon et al.[10], needed some modifications in order to perform the experiments and make the suite closer to a benchmark. We have 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 fixed 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 jobs.

In addition to the code problems, some modifications were necessary to improve 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 made all modifiable parameters dynamic and also added basic information on how to use them. We also created a script to easy configure and run some tests. In addition, we make the pseudo-applications available in an online repository[†] where we will add new updates and improve the source code documentation in the future.

The methodology we used for the experiments is described in Section 4.1. To identify which pseudo-applications makes more accesses to the memory, in Section 4.2 we present cache memory accesses per instruction by pseudo-application. In Section 4.3 we organize the pseudo-applications according to the number of branch instructions. Section 4.4 presents the amount of float 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 summarize the results.

4.1 | Methodology

The results presented in this Section are the average of 30 executions. The executions are interleaved among the pseudo-applications. This number of executions was established as indicated in[50]. In this study, the authors perform experiments that show that this is the minimum number of executions of MPI in order to obtain statistically acceptable results. By following the indications of this study, the MPI results showed an average absolute deviation bellow 3.7% in the worst cases. 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 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)[51].

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

[†]Source codes available at https://github.com/adrianomg/PAMPAR

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 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 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 does not add noise in 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 runtime, 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.

To set a workload that is equivalent to all pseudo-applications is not simple to deal when comparing different parallel pseudo-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.

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. No specific thread/process scheduling to cores was configured.

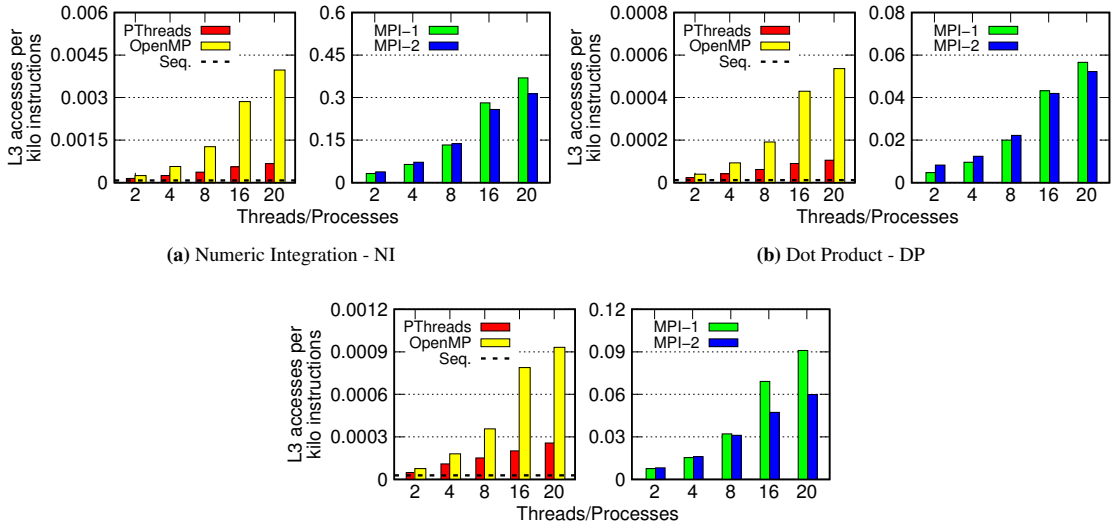
There is no easy way to compare the number of accesses to the cache, branches instructions, and FLOPs of different applications simply by looking at the total values. Each application handles differently with the size of the input problem and this is not something comparable. To make this data more correlated and reliable, we relate it to 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 not presented. The results here presented are arranged 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 in relation to the total of instructions.

The pseudo-applications that almost do not access cache are DP, PI, and NI. This is exactly the expected behaviour. 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.

Beyond the pseudo-applications that make almost no access to the cache 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, HA, TR, OE,



(a) Numeric Integration - NI

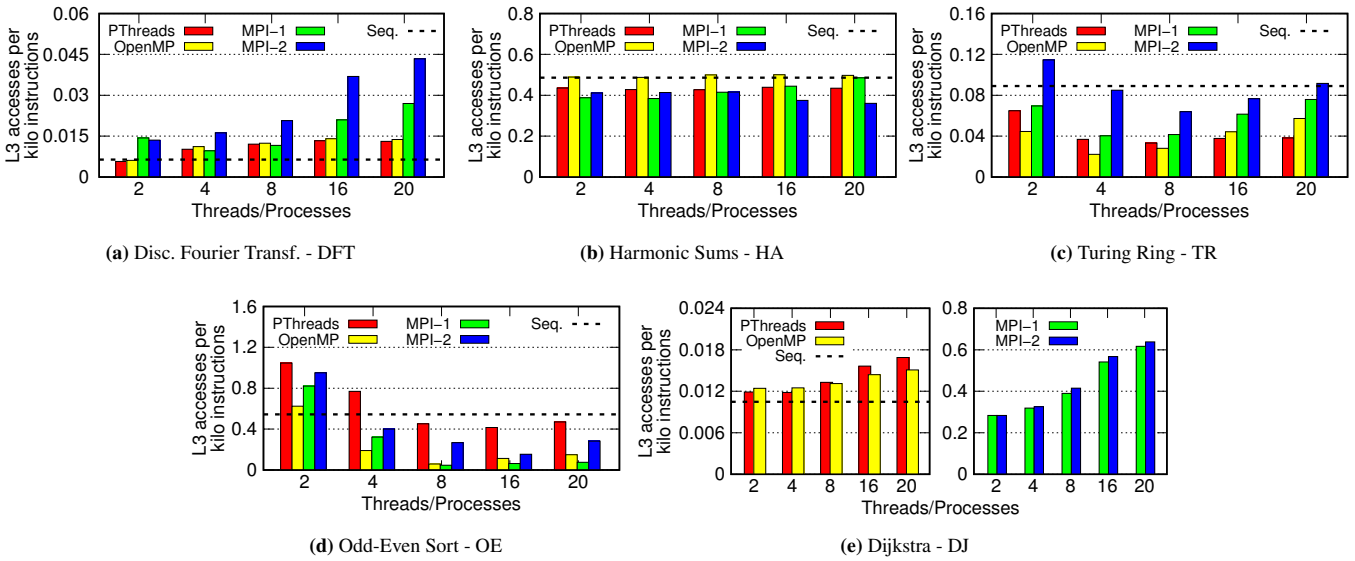
(b) Dot Product - DP

(c) Pi Calculation - PI

FIGURE 1 Low L3 cache accesses per kilo instructions rate.

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 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 in relation to the others. These two pseudo-applications, as we will see in Section 4.3, are the two with 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.



(a) Disc. Fourier Transf. - DFT

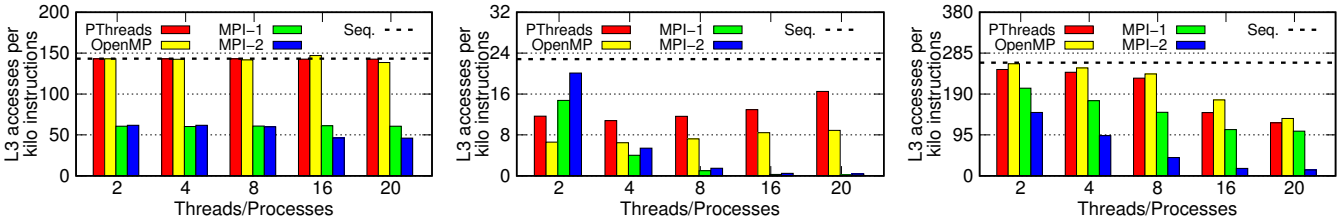
(b) Harmonic Sums - HA

(c) Turing Ring - TR

(d) Odd-Even Sort - OE

(e) Dijkstra - DJ

FIGURE 2 Medium L3 cache accesses per kilo instructions rate.



(a) Matrix Multiplication - MM **(b)** Jacobi - JA **(c)** Gram Schmidt - GS

FIGURE 3 High L3 cache accesses per kilo instructions rate.

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. Among all benchmark pseudo-applications, GS is the one that performs the larger access to the cache. One of the reasons for this is 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 behaviour with the task variation for cache accesses. 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 behaviour in the L3 and L2 results.

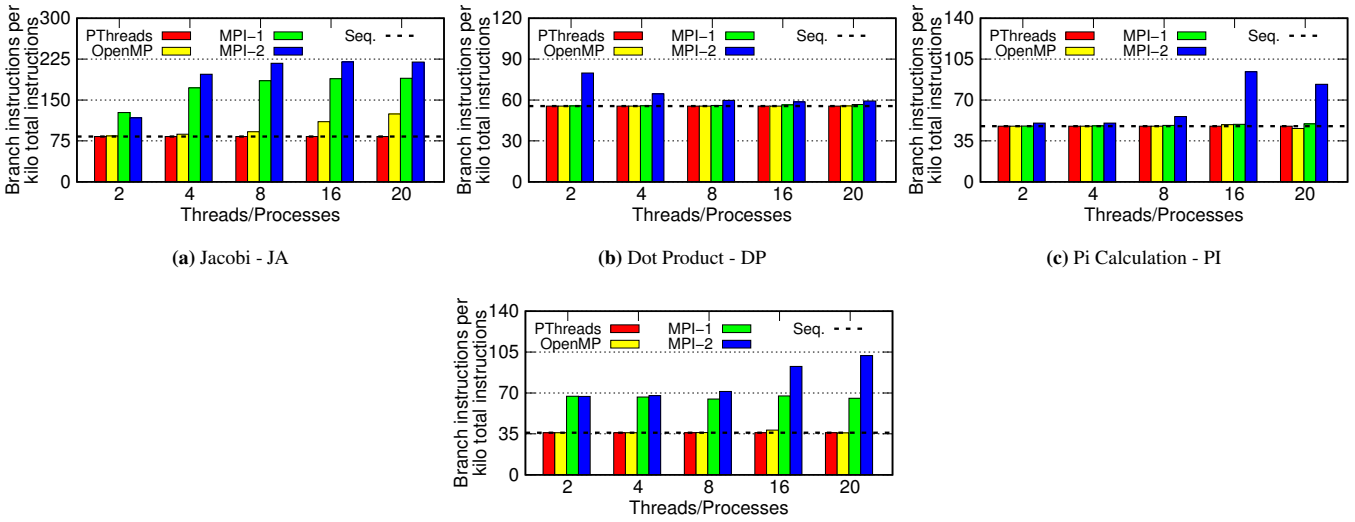
Regarding PPIs, besides MM only HA showed a non-variable behaviour with the increase of parallel tasks in all PPIs. For the other cases there are two situations: either the access rate increased because of overhead, or the access rate decreased because of the increase in the number of total instructions. Both cases are directly related 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 et al.[49] 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 others 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, 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



(a) Jacobi - JA **(b)** Dot Product - DP **(c)** Pi Calculation - PI

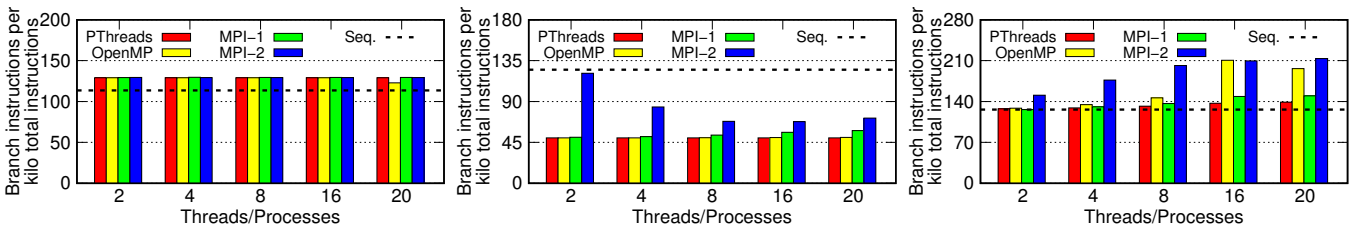
(d) Matrix Multiplication - MM

FIGURE 4 Low branch instructions per kilo total instructions rate.

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 behaviour, increasing and decreasing the overhead according to the number of parallel processes. In these pseudo-applications, the total of branch instructions coincides with the total of executed loops.

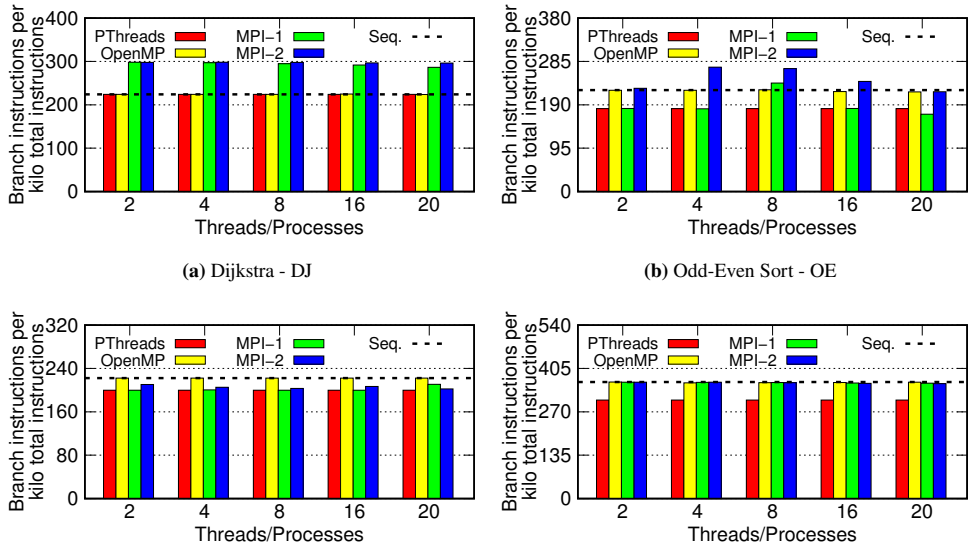
Pseudo-applications that execute more than 100 and less than 200 branch instructions per kilo instruction we consider to have a medium branch rate. These pseudo-applications are DFT, NI and GS and their respective charts are in Figure 5. DFT presents a simple behaviour where all PPIs add a small non-variable overhead. On the other hand, NI exhibits unexpected behaviour. NI is an 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.



(a) Disc. Fourier Transf. - DFT **(b)** Numeric Integration - NI **(c)** Gram Schmidt - GS

FIGURE 5 Medium branch instructions per kilo total instructions rate.



(a) Dijkstra - DJ

(b) Odd-Even Sort - OE

(c) Harmonic Sums - HA

(d) Turing Ring - TR

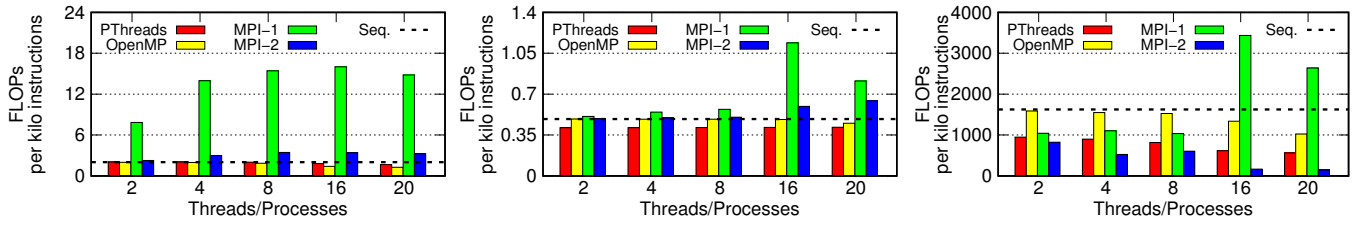
FIGURE 6 High branch instructions per kilo total instructions rate.

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,



(a) Jacobi - JA

(b) Turing Ring - TR

(c) Gram Schmidt - GS

FIGURE 7 Floating-point operations per kilo instructions.

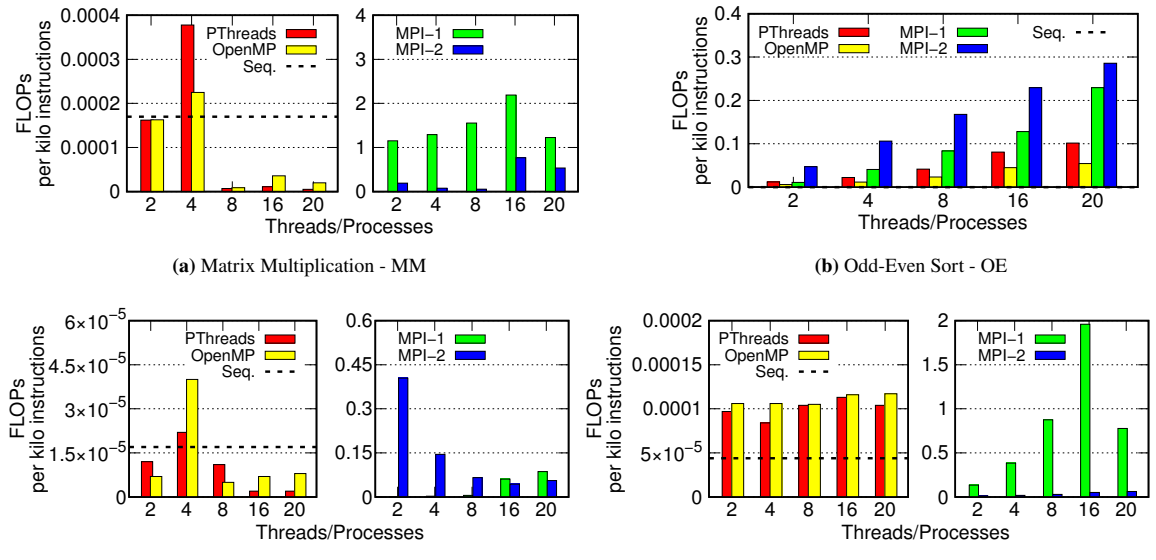
has dozens of such operations with each loop. However, this is the pseudo-application with the largest number of branches, as seen in the previous section. In this way, it is clear that most of these operations are not performed with each loop. So this is the main reason for this pseudo-application which 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 behaviour among the others is GS (Figure 7 (c)). This is the only pseudo-application that does more floating-point operations than total instructions. This pseudo-application has several nested loops in sequence with no conditional branches. 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 (Figure 9). If we look at the respective equations (6, 7, 9, and 8) 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 behaviour 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 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 behaviour of the PPIs is the same seen in relation 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.



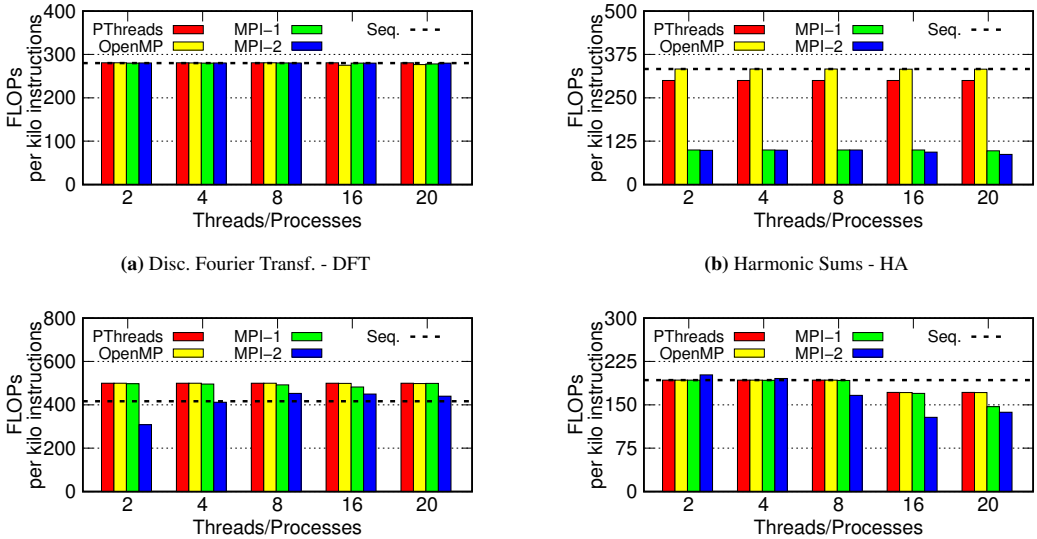
(a) Matrix Multiplication - MM

(b) Odd-Even Sort - OE

(c) Dot Product - DP

(d) Dijkstra - DJ

FIGURE 8 Low floating-point operations per kilo instructions rate.



(a) Disc. Fourier Transf. - DFT

(b) Harmonic Sums - HA

(c) Numeric Integration - NI

(d) Pi Calculation - PI

FIGURE 9 High floating-point operations per kilo instructions rate.

We are not able to explain exactly what causes this behaviour, 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.

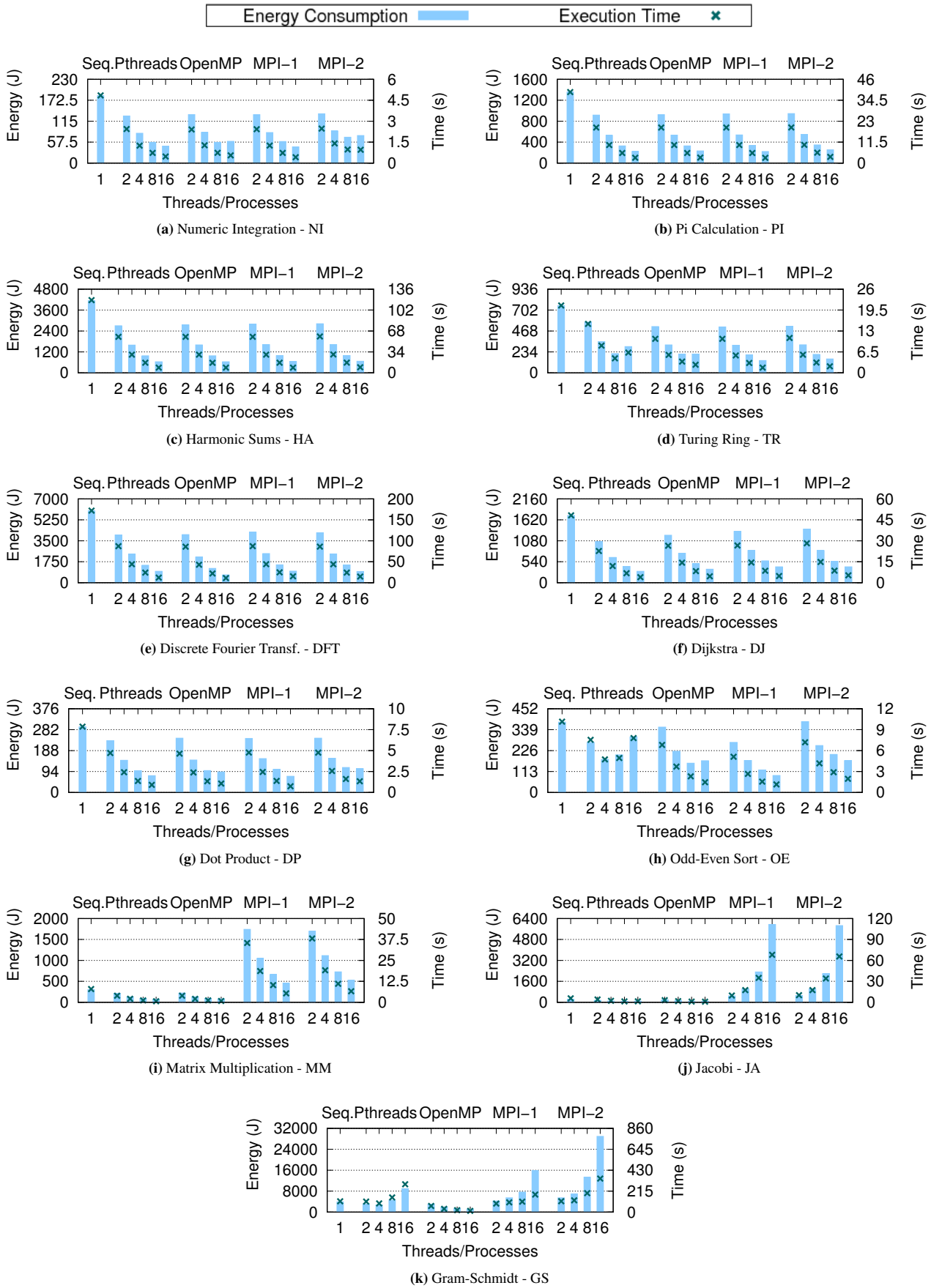
4.5 | Performance and Energy Consumption

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 ×. 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[52]. 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



(a) Numeric Integration - NI

(b) Pi Calculation - PI

(c) Harmonic Sums - HA

(d) Turing Ring - TR

(e) Discrete Fourier Transf. - DFT

(f) Dijkstra - DJ

(g) Dot Product - DP

(h) Odd-Even Sort - OE

(i) Matrix Multiplication - MM

(j) Jacobi - JA

(k) Gram-Schmidt - GS

FIGURE 10 Energy consumption and performance for each pseudo-application.

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 behaviour when using PThreads. Except when using 16 threads, in the other cases, this PPI had an increase in execution time in relation 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 behaviour, but with worse scalability with PThreads, can be seen in OE. This behaviour 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 relate this larger consumption with 16 threads to the accesses to the cache memory, we can observe the same behaviour. 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 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 behaviour was observed by the authors in [49]. Additionally, the pseudo-applications which presented the worst results for MPI are the only ones who have a higher percentage of memory usage than CPU usage during execution [11]. 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 in relation 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 energy efficiency of the pseudo-applications. The purpose of these data was to show the amount of energy each pseudo-application spent trough a given problem size and how they perform varying the number of parallel tasks. To see how efficient they are we need to relate energy and time to calculate power consumption, according to Equation 14. Here, P is the power consumption in watts (W), E is the consumed energy in joules (J) and t is the spent time in seconds (s). 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.

$$P = \frac{E}{t} \tag{14}$$

The charts in Figure 12 and Figure 11 show the power consumption 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 12 we presented the

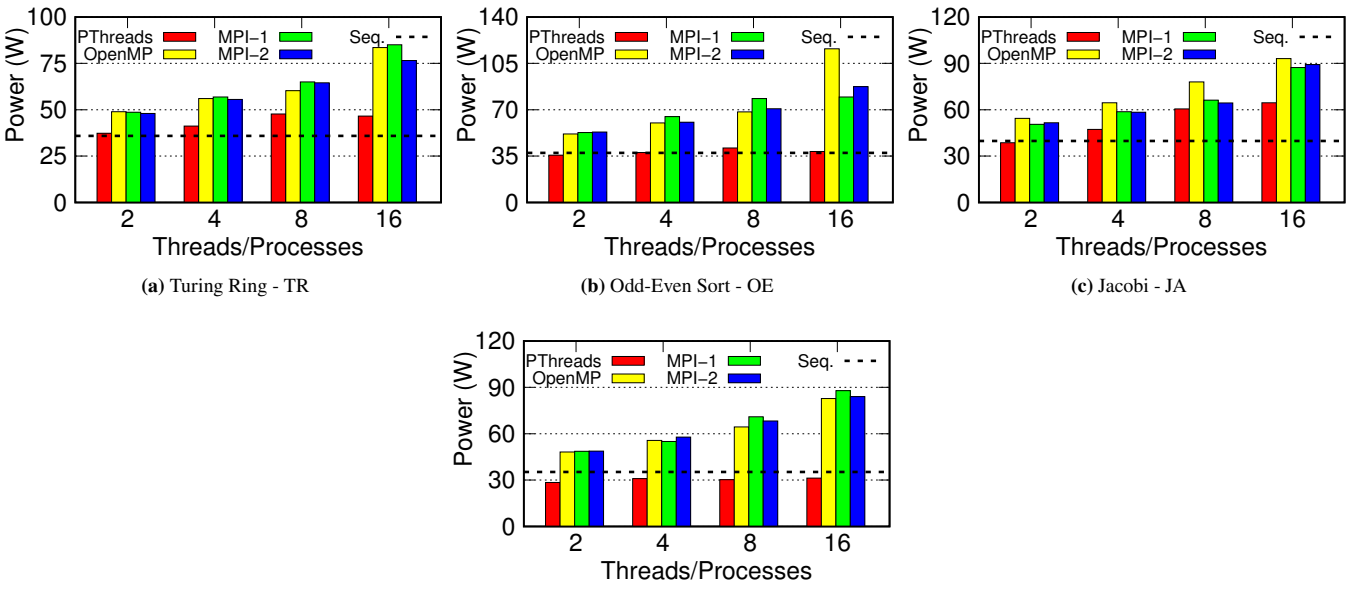
results for the pseudo-applications classified as CPU-bound. These results show that the power consumption of MPI-1 and MPI-2 is slightly higher when using 2 and 4 parallel tasks. However, for 8 and 16 processes, MPI 1 and 2 varies and have a power consumption equal to or lower than PPIs that use threads.

The DFT pseudo-application shows a different PThreads behaviour in relation 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 in relation to 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 behaviour 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-1 and 2, 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 power 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 subsection 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 in relation 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.

In the memory-bound pseudo-applications (Figure 11), 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/synchronisation 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 power 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.



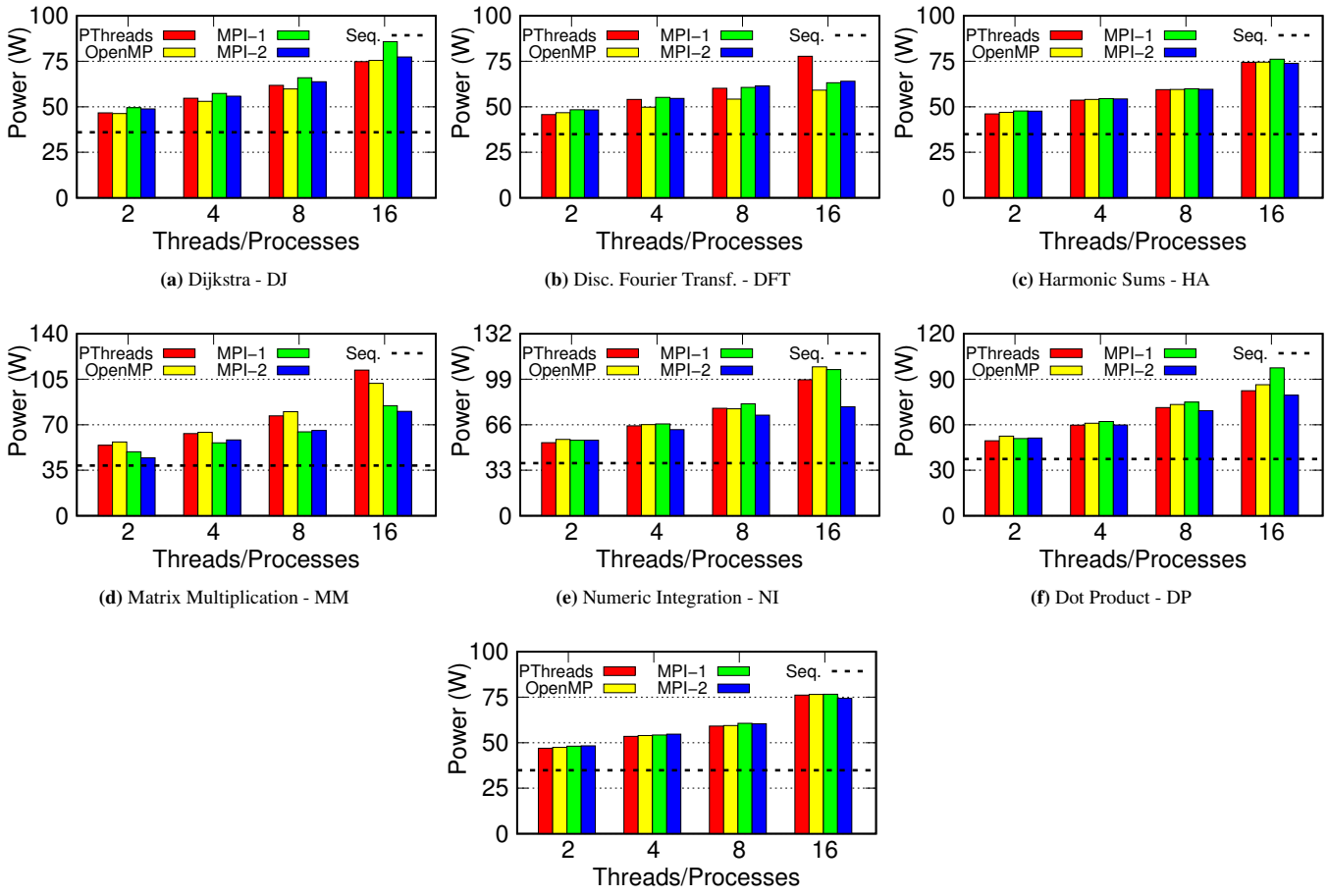
(a) Turing Ring - TR

(b) Odd-Even Sort - OE

(c) Jacobi - JA

(d) Gram-Schmidt - GS

FIGURE 11 Power consumption for memory-bound pseudo-applications.



(a) Dijkstra - DJ

(b) Disc. Fourier Transf. - DFT

(c) Harmonic Sums - HA

(d) Matrix Multiplication - MM

(e) Numeric Integration - NI

(f) Dot Product - DP

(g) Pi Calculation - PI

FIGURE 12 Power consumption for CPU-bound pseudo-applications.

Another observed factor is that PThreads accesses less the memory system during synchronisation. 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 W. 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 behaviour 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 analyse the number of cache accesses, branch instructions, and floating-point operations, and relate this data to the total instructions. By doing this we reduce the impact of architecture on results.

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 in relation 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 W of power consumption.

In the Table 3 we present the results of the cache accesses/branches/FLOPs per total instruction in a summarised way for easy visualisation. 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 in relation 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 relation 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 too much fluctuation among executions with different tasks, and no pattern can be extracted.

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 behaviour among the analysed 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 PTreads 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 in relation to the use of analysed parts. The objective of this work is to show that the pseudo-applications are able to be representative in different scenarios 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 either improve or worse performance in certain aspects. So it is an option to the programmer to choose what works best for him.

5 | CONCLUSION AND FUTURE WORK

In this work, we propose a set of pseudo-applications to be used as a benchmark. The main purpose of this benchmark is to analyse 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. We presented several studies that had to solve this problem locally, showing that there is a lack of such a benchmark. We also find initiatives that are trying to solve the same problem through modifications to existing benchmarks, but often these changes are not included in the main benchmark and are dispersed. 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 runtime results showed that the pseudo-applications generally have a good performance gain. In addition, energy consumption showed a behaviour 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

TABLE 3 Summary of results. Each ⋆ represents an increasing in the respective rate, and + or − represent more or less overhead added by the PPIs.

		L3 accesses	L2 acesses	Branches	FLOPs
Discrete Fourier Transform	Sequential		⋆	⋆⋆	⋆⋆⋆
	PThreads	+	=	+	=
	OpenMP	+	++	+	=
	MPI-1	++	−	+	=
	MPI-2	+++	+	+	=
Dijkstra	Sequential	⋆		⋆⋆	
	PThreads	+	+	=	+
	OpenMP	+	+	=	+
	MPI-1	+++	+++	++	++
	MPI-2	+++	+++	++	+++
Dot Product	Sequential			⋆	
	PThreads	+	+	=	−−
	OpenMP	++	++	=	−−
	MPI-1	+++	+++	=	+++
	MPI-2	+++	+++	+	+++
Gram-Schmidt	Sequential	⋆⋆⋆	⋆⋆⋆	⋆⋆	⋆⋆⋆
	PThreads	−	=	=	−−
	OpenMP	−	−	++	−
	MPI-1	−−	−−	=	?
	MPI-2	−−−	−−−	++	−−−
Harmonic Sums	Sequential	⋆	⋆	⋆⋆	⋆⋆⋆
	PThreads	−	−	−	−
	OpenMP	=	=	=	=
	MPI-1	−	++	−	−−−
	MPI-2	−−	+	−	−−−
Jacobi Method	Sequential	⋆⋆	⋆⋆	⋆	⋆
	PThreads	−	+	=	=
	OpenMP	−−	=	+	−
	MPI-1	−−−	−−−	++	+++
	MPI-2	−−−	−−−	+++	+
Matrix Multiplication	Sequential	⋆⋆⋆	⋆⋆⋆	⋆	
	PThreads	=	=	=	?
	OpenMP	=	=	=	?
	MPI-1	−−−	−−−	++	+++
	MPI-2	−−−	−−−	+++	++
Numeric Integration	Sequential			⋆⋆	⋆⋆⋆
	PThreads	+	+	−−−	+
	OpenMP	++	++	−−−	+
	MPI-1	+++	+++	−−−	+
	MPI-2	+++	+++	−−	=
Odd-Even Sort	Sequential	⋆	⋆⋆	⋆⋆	
	PThreads	=	−−	−	++
	OpenMP	−−	−−−	=	+
	MPI-1	−−	−−	−	+++
	MPI-2	−	−−	+	+++
Pi Calculation	Sequential			⋆	⋆⋆
	PThreads	+	+	=	=
	OpenMP	++	++	=	=
	MPI-1	+++	+++	=	−
	MPI-2	+++	+++	+	−−
Turing Ring	Sequential	⋆	⋆	⋆⋆⋆	⋆
	PThreads	−	=	−	−
	OpenMP	−−	=	=	=
	MPI-1	−	++	=	?
	MPI-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 summarise 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 behaviour 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 PAMPAR goal is offer parallel pseudo-applications that are representative for the greatest number of scenarios. 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. 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 the PAMPAR project [‡].

References

1. Bienia C. *Benchmarking Modern Multiprocessors*. PhD thesis. Princeton University, 2011.
2. Bourzac, Katherine . *Supercomputing poised for a massive speed boost.* Springer Nature International Journal of Science . 2017.
3. IC-Insights . The McClean Report. tech. rep., http://www.icinsights.com/services/mcclean-report/: 2018.
4. Bolaria J, Halfhill TR. A Guide to Multicore Processors. tech. rep., The Linley Group; http://www.linleygroup.com/report_detail: 2017.
5. Rauber T, Rünger G. *Parallel programming: For multicore and cluster systems.* Springer Science & Business Media . 2010.
6. Intel®. Parallel Programming and Heterogeneous Computing at Your Fingertips with Threading Building Blocks. 2018.
7. Ejjaaouani K, Aumage O, Bigot J, et al. InKS, a Programming Model to Decouple Performance from Algorithm in HPC Codes. *4th International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (REPARA)* 2018: 1–12.
8. Lorenzon AF, Sartor AL, Cera MC, Beck ACS. The Influence of Parallel Programming Interfaces on Multicore Embedded Systems. *Computer Software and Applications Conference (COMPSAC)* 2015; 2: 617–625.
9. Lorenzon AF, Cera MC, Beck ACS. Optimized use of parallel programming interfaces in multithreaded embedded architectures. *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* 2015: 410–415.
10. Lorenzon A. Avaliação do desempenho e consumo energético de diferentes interfaces de programação paralela em sistemas embarcados e de propósito geral. Master's thesis. Universidade Federal do Rio Grande do Sul. 2014.
11. Garcia AM, Schepke C. Uma Proposta de Benchmark Paralelo para Arquiteturas Multicore. *XVIII Escola Regional de Alto Desempenho* 2018: 285-289.
12. SPEC . Standard Performance Evaluation Corporation. 2018.
13. Gray J. *Benchmark handbook: for database and transaction processing systems.* Morgan Kaufmann Publishers Inc. . 1992.
14. Gropp W, Lusk E, Thakur R. *Using MPI-2: Advanced features of the message-passing interface.* MIT press . 1999.
15. Butenhof DR. *Programming with POSIX threads.* Addison-Wesley Professional . 1997.
16. Uhsadel L, Georges A, Verbauwhede I. Exploiting hardware performance counters. *5th Workshop on Fault Diagnosis and Tolerance in Cryptography* 2008: 59–67.

[‡] www.researchgate.net/project/PAMPAR-A-Parallel-Benchmark-for-Performance-and-Energy-Consumption-Evaluation-of-Parallel-Programing-Interfaces

17. 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. *Parallel and Distributed Processing Symposium Workshops* 2016: 1180–1189.

18. Lively C, Taylor V, Wu X, et al. E-amom: an energy-aware modeling and optimization methodology for scientific applications. *Computer Science-Research and Development* 2014; 29(3-4): 197–210.

19. Lively C, Wu X, Taylor V, et al. Power-aware predictive models of hybrid (MPI/OpenMP) scientific applications on multicore systems. *Computer Science-Research and Development* 2012; 27(4): 245–253.

20. Song S, Su C, Rountree B, Cameron KW. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. *27th International Symposium on Parallel & Distributed Processing (IPDPS)* 2013: 673–686.

21. Rodrigues R, Annamalai A, Koren I, Kundu S. A study on the use of performance counters to estimate power in microprocessors. *IEEE Transactions on Circuits and Systems II: Express Briefs* 2013; 60(12): 882–886.

22. Chetsa GT, Lefèvre L, Pierson JM, Stolf P, Da Costa G. Exploiting performance counters to predict and improve energy performance of HPC systems. *Future Generation Computer Systems* 2014; 36: 287–298.

23. Isidro-Ramirez R, Viveros AM, Rubio EH. Energy consumption model over parallel programs implemented on multicore architectures. *International Journal of Advanced Computer Scienceand Applications (IJACSA)* 2015; 6(6).

24. Popov M, Akel C, Conti F, Jalby W, Oliveira Castro dP. Pcere: Fine-grained parallel benchmark decomposition for scalability prediction. *Parallel and Distributed Processing Symposium (IPDPS)* 2015: 1151–1160.

25. Jain S, Zheng G, Garzaran M, Cownie JH, Doodi T, Wilmarth TL. Parallelizing MPI Using Tasks for Hybrid Programming Models. *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* 2018: 1303–1312.

26. Dosanjh MG, Grant RE, Schonbein W, Bridges PG. Tail queues: A multi-threaded matching architecture. *Concurrency and Computation: Practice and Experience* 2019.

27. Doefler D, Barrett BW. Sandia MPI microbenchmark suite (SMB). *Technical report, Sandia National Laboratories* 2009.

28. Griebler D, Loff J, Mencagli G, Danelutto M, Fernandes LG. Efficient NAS benchmark kernels with C++ parallel programming. *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* 2018: 733–740.

29. Danelutto M, De Matteis T, De Sensi D, Mencagli G, Torquati M. P^3ARSEC: towards parallel patterns benchmarking. *Proceedings of the Symposium on Applied Computing* 2017: 1582–1589.

30. University of Illinois . ALP: All Levels of Parallelism for Multimedia. 2018.

31. Princeton University . PARSEC: A Unity of Measure. 2018.

32. Iqbal S, Liang Y, Grahn H. ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems. *Computer Architecture Letters* 2010; 9(2): 45–48. doi: 10.1109/L-CA.2010.14

33. Petitet A. HPL-a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. 2004.

34. Bailey DH, Barszcz E, Barton JT, et al. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications* 1991; 5(3): 63–73.

35. Adept-Project . Adept: addressing energy in parallel technologies. 2018.

36. Lorenzon AF, Cera MC, Beck ACS. Performance and Energy Evaluation of Different Multi-Threading Interfaces in Embedded and General Purpose Systems. *Journal of Signal Processing Systems* 2014; 80(3): 295–307.

37. Cheney W, Kincaid D. Linear algebra: Theory and applications. *The Australian Mathematical Society* 2009: 654.

38. Burden RL, Faires JD. Numerical Analysis, Brooks. *Cole, Belmont, CA* 1997.

39. Press WH, Teukolsky SA, Vetterling WT, Flannery BP. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press . 2007.

40. Banchoff T, Wermer J. *Linear algebra through geometry*. Springer Science & Business Media . 2012.

41. Smith , Steven W, others . *The scientist and engineer's guide to digital signal processing*. California Technical Pub. San Diego . 1997.

42. Goldston D, Yildirim CY. On the second moment for primes in an arithmetic progression. 2001.

43. Borwein JM, Borwein PB, Dilcher K. Pi, Euler Numbers, and Asymptotic Expansions. *The American Mathematical Monthly* 1989; 96(8): pp. 681–687.

44. Stewart J. *Calculus: Concepts and contexts*. Cengage Learning . 2009.

45. Turing AM. The chemical basis of morphogenesis. *Phil. Trans. Roy. Soc.* 1952; 237: 37.

46. Paudel J, Amaral JN. Using the Cowichan Problems to Investigate the Programmability of X10 Programming System. *Proceedings of the 2011 ACM SIGPLAN X10 Workshop* 2011: 4:1–4:10.

47. Mikloško J, Kotov VE. Complexity of parallel algorithms. In: Springer. 1984.

48. Foster I. *Designing and building parallel programs*. Addison Wesley Publishing Company . 1995.

49. Lorenzon AF, Cera MC, Beck ACS. Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy. *Journal of Parallel and Distributed Computing* 2016; 95: 107–123.

50. Hunold S, Carpen-Amarie A. Reproducible MPI benchmarking is still not as easy as you think. *IEEE Transactions on Parallel and Distributed Systems* 2016; 27(12): 3617–3630.

51. Terpstra D, Jagode H, You H, Dongarra J. Collecting performance data with PAPI-C. In: Springer. 2010 (pp. 157–173).

52. Garcia AM, Schepke C, Girardi AG. A New Parallel Benchmark for Performance Evaluation and Energy Consumption. *13th International Meeting on High Performance Computing for Computational Science (VECPAR)* 2018.