Power Consumption of Parallel Programming Interfaces in Multicore Architectures: A Case Study

Adriano Marques Garcia, Claudio Schepke, Alessandro Gonçalves Girardi, Sherlon Almeida da Silva
Graduate Program of Electrical Engineering – Federal University of Pampa – Alegrete, Brazil
adriano1mg@gmail.com, {claudio.schepke,alessandro.girardi}@unipampa.edu.br, sherlonalmeidadasilva@gmail.com

***Abstract*—This paper evaluates the power consumption of different parallel programming interfaces (PPI) in a multicore architecture. These PPIs are: PThreads, OpenMP, MPI-1 and MPI-2 (spawn). We measure the total energy and execution time of 11 applications in a single architecture, varying the number of threads/processes. The goal is to show that these applications can be used as a parallel benchmark to evaluate the power consumption of different PPIs. The results show that PThreads has the lowest power consumption among the interfaces, consuming less than the sequential version for memory-bound applications.**

***Index Terms*—parallel benchmark, parallel programing interfaces, MPI, OpenMP, POSIX Threads, energy.**

I. Introduction

Nowadays, many countries are limiting the use of existing supercomputers because of their high energy consumption [1]. This shows that energy consumption is currently a concern in many different computer systems. The increasing of the applications complexity and data size has required extensive research into both computational performance and energy efficiency. The popularization of Green500, which lists computers from the TOP500 list of supercomputers in terms of energy efficiency, shows that reducing energy consumption is one of the directions of high-performance computing [1]. So the challenge should not only be to increase performance, but also to consume less energy.

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 [2]. 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 need for communication among them and the more extensive use of the processor can lead to an increase in power 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.

Although parallelism allows performance gains, this can lead to higher power consumption. This power consumption grows mainly according to the amount of processors that are used in parallel and the volume of communication among them. On the other hand, the reduction in execution time allowed by the parallelization causes the decrease in the total energy consumption in some cases. Parallel benchmarks can be used to define which parallelization strategy compensates for the increase in power consumption in a particular architecture. However, there is not a benchmark that offers a good set of applications, fully parallelized, using multiple PPIs and different models of communication by tasks. The most commonly used parallel benchmarks have only partial parallel sets using more than one PPI.

To fill this gap, this work studies a set of 13 applications developed with the purpose of evaluating the performance and energy consumption in multi-core architectures. These applications were developed and classified according to different criteria in previous studies [3]–[6]. These studies have shown that these applications have characteristics that are distinct enough to represent different scenarios. The objective of this work is to show the impact of these distinct characteristics on the power consumption of different applications and also the impact of the implementations using different PPIs.

The remainder of this work is organized as follows. In the section II we present the PPIs in which the applications are implemented. The related works are discussed in section III, where we compare our work with similar benchmarks. The section IV presents the set of applications and the techniques used to parallelize them, bringing more details about the historic of classifications. Section V shows how our experiments were structured and brings some information to a better understanding of the results. The section VI discusses the results and, finally, section VII draws the final considerations and future works.

II. Parallel Programming Interfaces

There are several forms of parallelism that can be applied into a program, such as: data parallelism, shared memory, exchange of messages, 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 [7]. In this work, the set of 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

Supported by CAPES

that influence the execution of parallel programs [2]. 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 [8]. 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. This means that 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.

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 child, to be created, which does 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 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 intercom that binds the child is returned in the execution of the function `MPI_Comm_spawn()`.

III. Related Work

There are several benchmarks developed to serve different purposes. 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 the following benchmarks: ALPBench, PARSEC, ParMiBench, SPEC, Linpack, NAS and Adept Project.

A. Similar Benchmarks

ALPBench 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 open source license.

PARSEC (Princeton Application Repository for Shared-Memory Computers) is an open source benchmark suite. It consists of 13 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. 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 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, and others. We consider the following groups of SPEC: SPEC MPI2007 and SPEC OMP201. SPEC MPI2007 is a set of 18 applications deployed in MPI focused on testing high-performance computers. SPEC OMP2012 uses 14 scientific applications implemented in OpenMP, offering optional energy consumption metrics based on SPEC Power.

HPL consists of a software package that solves arithmetic dual floating-point precision random linear systems in high-performance architectures. 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 is open-source and consist of 7 applications that 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 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 applications are implemented with MPI and OpenMP. Some applications are also implemented in HPF, UPC, Java, Titanium, TBB etc.

The Adept Benchmark 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, and 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).

B. Comparison of Benchmarks

The benchmark addressed in this work consist of 11 applications implemented in C and their complexities range from

TABLE I: Comparison of our benchmark with the similar ones

Rating criteria	ALPBench	PARSEC	ParMiBench	SPEC	HPL	NAS	Adept	Our Benchmark
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

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

We analyze the main characteristics of the related parallel benchmarks and compare to the benchmark we propose in this work in Table I. In relation to the benchmarks, some use only one PPI while others use more than one. However, some of those who use more than one PPI do not have the whole set of applications paralleled by all PPIs. They implement parts of the set with one PPI and other parts with another PPI.

SPEC, for example, has 32 parallel applications divided into two different sets. One of them using OpenMP and the other one using MPI, so it has no set implemented with two or more PPIs. NAS, on the other hand, uses several other PPIs. However, most of application are not implemented in all PPIs. Many of them are not supported by any multicore architecture (ARM 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 the NAS, but it only offers two PPIs. Therefore, none of them meets the objective of comparing parallel programming interfaces, which is the main objective of the benchmark we are proposing in this work.

IV. Benchmark Applications

This section presents the 11 applications of the benchmark. They were developed with the purpose of establishing a relation between performance and energy consumption in embedded systems and general purpose architectures [9]. Late, the proposal to use this set of applications to form a parallel benchmark was given by [10]. All the applications used in this work have detailed descriptions in [6], where the authors provide the algorithms, a detailed description of each application and also the mathematical equations that were used in some implementations. Below is a list of these applications with a brief description of each.

- **Gram-Schmidt**- The Gram-Schmidt process is a method for orthonormalising a set of vectors in an inner product space.
- **Matrix Multiplication** - This algorithm multiplies the lines of a matrix A by the columns of a matrix B.
- **Dot Product** - The dot product is an algebraic operation that multiplies two equal-length sequences of numbers.
- **Odd-Even Sort** - It is a comparison sort algorithm related to bubble sort.
- **Dijkstra** - It finds a minimal cost path between nodes in a graph with non-negative edges.
- **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.
- **Jacobi Method** - The Jacobi method is an algorithm for determining the solutions of a diagonally dominant system of linear equations.
- **Harmonic Sums** - The Harmonic Sums or Harmonic Series is a finite series that calculates the sum of arbitrary precision after the decimal point.
- **PI Calculation** - It applies the Gregory-Leibniz formula to find π.
- **Numerical Integration** - This algorithm integrates an $f(x)$ function in a given interval, using approximation techniques to define an area.
- **Turing ring** - It is a space system in which predators and prey interact in the same place. The system simulates the iteration and evolution of preys and predators through the use of differential equations.

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, as: molecular dynamics, electromagnetism, digital signal processing, image processing, mathematical optimization, and others.

A. Parallelizing the Applications

Parallelize a sequential program can be done in several ways. However, inappropriate techniques can negatively im-

TABLE II: Details about the applications

Data Structures	Problem Size	Acronym	Application	Complexity
Unstructured data	1 billion	NI	Numerical Integration	$O(n)$
	4 billion	PI	PI Calculation	
	15 billion	DP	Dot Product	
Vector	100000	HA	Harmonic Sums	$O(n \times d)$
	150000	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	

pact the performance of an application. To minimize this problem, all parallel implementations in this work were based on statements from [2], [7], [8], [11]. [2] 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; assigning tasks to processes/threads; mapping processes/threads into physical processing units.

The decomposition of the computation and assignment of tasks to processes/threads occurred explicitly in the parallelization with PThreads and MPI 1 and 2 in order to obtain the best workload balancing. Also message exchange functions among processes were included, as well as the dynamic creation of processes in MPI-2. For Parallelization with OpenMP, parallel loops with thin and coarse granularity were used. According to [2], [11], this technique is most appropriate for parallelizing applications that perform iterative calculations and traverse contiguous data structures (e.g. matrix, vector, etc.). For each data structure, a specific parallelization model was adopted.

V. Methodology

The results presented in section VI are the average of 30 executions disregarding the extreme values. This number of executions was established as indicated in [12]. In this study, the authors perform experiments that show that the minimum number of executions in order to obtain statistically acceptable results in MPI is 30. Following the indications of this study, the results in MPI-1 and MPI-2 showed a standard deviation below 0.5 in the worst cases. OpenMP and PThreads showed a standard deviation below 0.1 in all cases. During the experiments, the computer remained locked to ensure that other applications did not interfere 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 application was measured and the difference of these values was used. We used these data to calculate power consumption, according to Equation 1. Where W is the power in Watts, J is the total energy in Joules and s is the execution time in seconds.

$$W = \frac{J}{s} \tag{1}$$

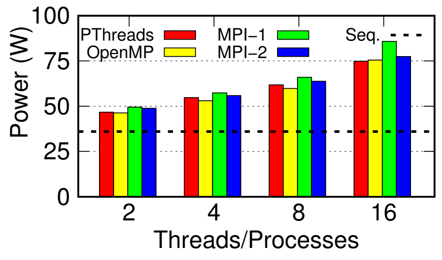
The Table II shows the size of the inputs used for each application, the acronym used to identify each application in the following section, and their complexities.

VI. Results

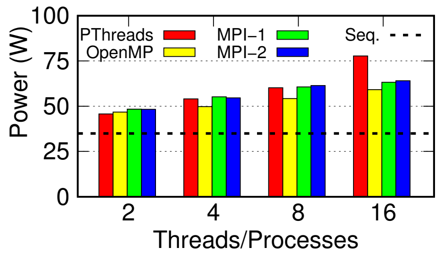
Next experiments were carried out on a computer equipped with 2 Intel® Xeon® E5-2650 v3 processor. Each processor has 10 physical cores and 10 virtual cores operating at the standard 2.3 GHz frequency and a turbo frequency of 3 GHz. Its memory system consists of three levels of cache: a 32 KB cache L1 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 Linux version 4.4.0-128 using Intel® ICC 18.0.1 compiler with default optimization flags.

In Figure 1 and Figure 2 bars show the power consumption in watts for each PPI. Each chart displays the results by 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 application. In Figure 1 are presented the results for the CPU-bound applications. 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 MPI-1 and MPI-2 processes, they have a power consumption equal to or lower than PPIs that use threads.

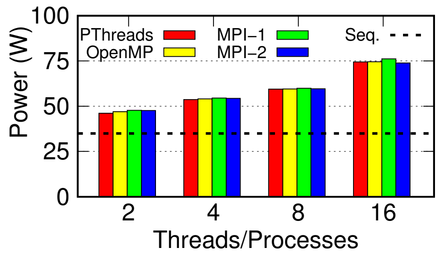
The DFT application shows a different PThreads behavior in relation to the other PPIs when the number of parallel tasks increases. With 2 threads PThreads follows the pattern that is seen in most CPU-Bound applications. However, using more threads, this consumption exceeds the consumption of other PPIs incrementally. The other three PPIs follow the pattern by increasing the number of threads, where OpenMP consumes



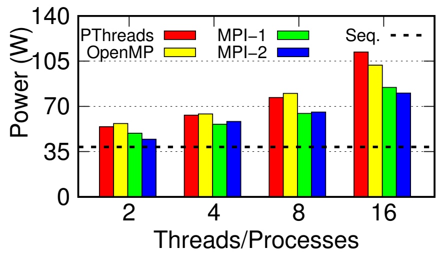
(a) Dijkstra - DJ



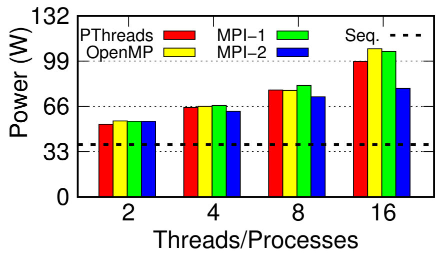
(b) Disc. Fourier Transf. - DFT



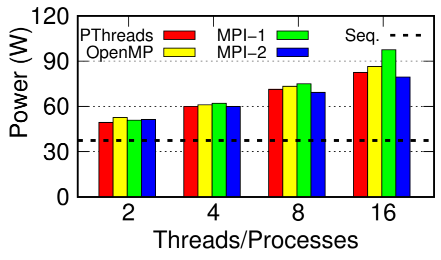
(c) Harmonic Sums - HA



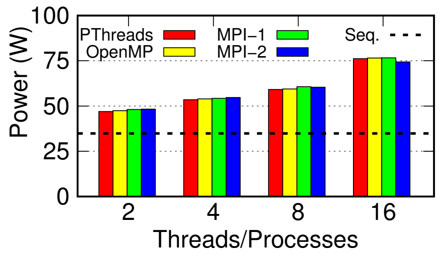
(d) Matrix Multiplication - MM



(e) Numeric Integration - NI



(f) Dot Product - DP



(g) Pi Calculation - PI

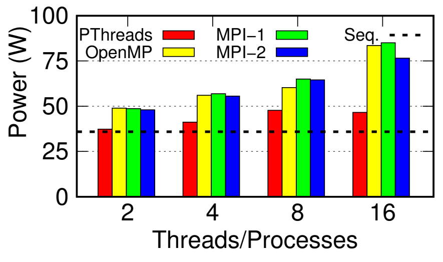
Fig. 1: Power consumption for CPU-bound applications.

less power than MPI-1 and MPI-2, have similar results for this application.

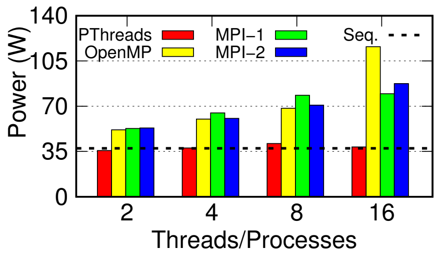
Comparing MM with DFT shows similar behavior of PThreads in both applications. The consumption of this PPI is lower than OpenMP with 2 threads but grows at a higher rate than the other PPIs as the number of threads increases. The difference in this application is that OpenMP also uses more power than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

The low power consumption by PThreads in memory-bound applications does not represent that the total power consumed was lower in this PPI. As seen by [13] the runtime and energy consumption is higher than OpenMP. This low power consumption meant that the 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. In fact, for all memory-bound applications OpenMP uses about 2 and 3 times more memory than PThreads with 8 and 16 threads respectively. On the other hand, PThreads have approximately 10 times more cache misses than other PPIs in memory-bound applications. In this way 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 a lower consumption of energy over time. This increase in runtime can be caused by busy waiting for PThreads.

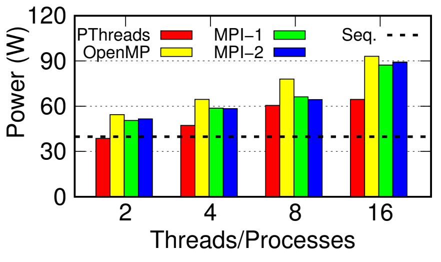
In the memory-bound applications (Figure 2), OE with OpenMP using 16 threads reached the higher power consumption for these cases. What we have concluded, is that the average workload initially set, is not large enough for all cases. Besides that, as seen by [13] OpenMP achieves good performance but the energy consumption keeps the same when scaling to 16 threads, and this leads to an increase in the power consumption. OE is a memory-bound application, so the overhead of communication/synchronization among threads begins to impact negatively earlier in these cases. With MPI-1 and MPI-2 the results obtained are very similar to the results of CPU-bound applications. The growth of the



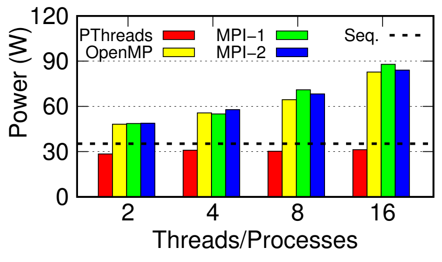
(a) Turing Ring - TR



(b) Odd-Even Sort - OE



(c) Jacobi - JA



(d) Gram Schmidt - GS

Fig. 2: Power consumption for memory-bound applications.

power consumption as the increase in the number of parallel processes follows the same pattern previously observed. What is perceived is that MPI-2 has a lower consumption than MPI-1 in most cases for both CPU and memory-bound. This small difference may possibly be caused by dynamic process creation. This causes processes to be created later in MPI-2.

Our initial hypothesis was that a 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 application should reduce its consumption in proportion to the performance achieved over the sequential application. But what happens in practice is that the energy consumption does not decrease in the same proportion as the execution time, as investigated by [13]. In this way, this generates an increase in energy consumption. This is because there are 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.

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

The difference in these PPIs can be explained in the context of threads and processes. Threads are often referred as 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.

VII. Conclusions and Future Work

In this paper, we present a set of applications that can be used as a benchmark. The main purpose of this benchmark is to analyze power consumption and performance of different parallel programming interfaces in a multi-core architecture. We first compared our studied benchmark with the main parallel benchmarks that are currently used for the same purpose. This comparison showed that there is no benchmark that meets one of our goals: to offer a simpler way to compare PPIs. In addition, we did a study of the history of the applications, where we showed that there were already 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 power consumption have increasing rate proportional to the number of threads/processes used in parallel. An important thing to note is that the way each application uses memory causes a high impact on power consumption. These memory access features of each application and PPI should be far better investigated to trace a relationship between the increase in the number of parallel tasks and energy consumption in an upcoming work. With this more detailed analysis we will be able to observe possible problems and points of improvement in the codes.

In future works, we intend to verify how the distribution of threads/processes to different cores and processors affects our experiments. We should also repeat the experiments using another compiler, such as gcc without optimisation flags. Next step is to check the scalability of our applications, so we will increase the number of threads/processes by varying the size of the workload and execute in a distributed system. In addition, we have two new applications (Histogram Similarity and Game of Life) that are already implemented and we are making final adjustments to include them in the benchmark. We also consider including more PPIs such as Intel TBB, UPC or Intel Cilk. Finally, to make the benchmark available, we will check the percentage of floating point operations, integer, loads, store, etc. In this way we have data defining each application independently of the architecture used.

References

[1] Bourzac, Katherine, *Supercomputing poised for a massive speed boost*. Springer Nature International Journal of Science, 2017.

[2] T. Rauber and G. Rünger, *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media, 2010.

[3] A. F. Lorenzon, A. L. Sartor, M. C. Cera, and A. C. S. Beck, "The Influence of Parallel Programming Interfaces on Multicore Embedded Systems," in *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, vol. 2. IEEE, 2015, pp. 617–625.

[4] A. F. Lorenzon, M. C. Cera, and A. C. S. Beck, "Optimized use of parallel programming interfaces in multithreaded embedded architectures," in *VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on*. IEEE, 2015, pp. 410–415.

[5] A. F. Lorenzon, "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.

[6] A. M. Garcia, "Classificação de um benchmark paralelo para arquiteturas multinúcleo," p. 109, 2016.

[7] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999.

[8] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[9] A. F. Lorenzon, M. C. Cera, and A. C. S. Beck, "Performance and Energy Evaluation of Different Multi-Threading Interfaces in Embedded and General Purpose Systems," *Journal of Signal Processing Systems*, vol. 80, no. 3, pp. 295–307, 2014.

[10] A. M. Garcia and C. Schepke, "Uma proposta de benchmark paralelo para arquiteturas multicore," *XVIII Escola Regional de Alto Desempenho*, pp. 285–289, 2018.

[11] I. Foster, *Designing and building parallel programs*. Addison Wesley Publishing Company, 1995.

[12] S. Hunold and A. Carpen-Amarie, "Reproducible mpi benchmarking is still not as easy as you think," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3617–3630, 2016.

[13] A. M. Garcia, C. Schepke, A. G. Girardi, and S. A. Silva, "A new parallel benchmark for performance evaluation and energy consumption," *13th International Meeting on High Performance Computing for Computational Science (VECPAR)*, 2018.