A New Parallel Benchmark for Performance Evaluation and Energy Consumption

A New Parallel Benchmark for Performance Evaluation and Energy Consumption *

Adriano Marques Garcia[1][0000−0003−4796−773X], Claudio Schepke[1][0000−0003−4118−8831], Alessandro Gonçalves Girardi[1][0000−0002−0074−7855], and Sherlon Almeida da Silva[1][0000−0001−6124−9350]

Laboratório de Estudos Avançados em Computação (LEA)
Universidade Federal do Pampa (UNIPAMPA), Alegrete, Brazil
adriano1mg@gmail.com,
{claudio.schepke,alessandro.girardi}@unipampa.edu.br,
sherlonalmeidadasilva@gmail.com

Abstract. This paper presents a new benchmark to evaluate 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 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 application in a single architecture, varying the number of threads/processes. The goal is to show that this set of applications has enough features to form 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. However, PThreads and OpenMP achieve the best trade-offs between performance and energy in most cases.

Keywords: Benchmark · Performance · Energy Consumption.

1 Introduction

In recent years, the increase in the complexity of applications and data size has demanded a great search for computational and energetic efficiency. Moreover, 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 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.

* Supported by CAPES

*This is a preprint verion from the authors of Garcia, A.M., Schepke, C., Girardi, A.G., da Silva, S.A. (2019). A New Parallel Benchmark for Performance Evaluation and Energy Consumption. In: Senger, H., et al. High Performance Computing for Computational Science – VECPAR 2018. VECPAR 2018. Lecture Notes in Computer Science(), vol 11333. Springer, Cham. https://doi.org/10.1007/978-3-030-15996-2_14

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 [14]. 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.

The scenario here presented shows that, although parallelism allows performance gains, this can lead to higher energy consumption. This energy consumption grows mainly according to the amount of processors that are used in parallel and the volume of communication between 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. It is fundamental to use adequate benchmarks to define which parallelization strategy compensates for the increase in energy 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 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 a set of 11 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 [10,8,9,4]. 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 performance and energy consumption of different applications and also the impact of the implementations using different PPIs.

To achieve this goal, applications were run on a multi-core machine. For each application, the execution time and the total power consumption of the processor were measured. Different numbers of threads/processes were used for the parallelization. The data were analyzed side by side with the result obtained by the sequential version of each application.

The remainder of this work is organized as follows. In the section 2 we present the PPIs in which the applications are implemented. The related works are discussed in section 3, where we compare our work with similar benchmarks. The section 4 presents the set of applications and the techniques used to parallelize them, bringing more details about the historic of classifications. The section 5 discusses the results and, finally, section 6 draws the final considerations and future works.

2 Parallel Programming Interfaces

There are several computational models used in parallel computing, 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 [6]. 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 that influence the execution of parallel programs [14]. 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 [2]. 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()`.

Traditionally, applications deployed with MPI-2 begin the execution with a single process. The primitive `MPI_Comm_spawn()` is 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 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()`.

3 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.

3.1 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 composed of multi-threaded programs. 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 is 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 organize 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, among others. We consider the following groups of SPEC: SPEC MPI2007, SPEC OMP2012 and SPEC Power. 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 power consumption metrics based on SPEC Power. Finally, SPEC Power tests the power 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. 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 (NPB) is a set of open source programs generally used 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 in the original "pencil-and-paper" specification (NPB 1). It is an open source benchmark and the main set of applications is implemented with MPI and OpenMP.

Table 1: Comparison of our benchmark with the similar ones

Rating criteria	ALPBench	PARSEC	ParMiBench	SPEC	HPL	NPB	Adept	Our benchmark
Number of applications	**5**	**11**	**7**	**14-18**	**7**	**12**	**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**

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, 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).

3.2 Comparison Between the Benchmarks

The benchmark addressed in this work consist of 11 applications implemented in C and their complexities range from $O(n)$ 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 a tool to evaluate the performance and power consumption of different PPIs in embedded and general purpose multi-core architectures.

We analyze the main characteristics of the related parallel benchmarks and compare to the benchmark we propose in this work in Table 1. In 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. Three of these benchmarks use PThreads, five of them use OpenMP, and four use MPI. ALPBench also uses Intel TBB and Adept uses UPC.

Thus, even if some of these benchmarks implement three different PPIs, none of them allow an efficient comparison between these PPIs and between

different communication models. Also, they do not exploit the parallelism with dynamic process creation that MPI-2 offers. In this way, we do not find any other benchmark that use different PPIs, different communication models and a completely parallelized set of applications. The exception is the NPB, 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.

4 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 [11]. Below are listed the applications detailing briefly 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 between 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, some of them are: molecular dynamics, electromagnetism, digital signal processing, image processing, mathematical optimization, among others.

4.1 Parallelizing the Applications

Parallelize a sequential program can be done in several ways. However, inappropriate techniques can negatively impact the performance of an application. To minimize this problem, all parallel implementations in this work were based on statements from [3,2,6,14]. [14] propose that the parallelization 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 included were message exchange functions between processes, 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 [3,14], this technique is most appropriate for parallelizing applications that perform iterative calculations and traverse contiguous data structures (eg matrix, vector, etc.). For each data structure a specific parallelization model was adopted.

4.2 Applications History

The set of applications that compose the benchmark have already been investigated in previous works. The applications were used in these works to analyze performance and energy consumption on embedded systems and general purpose processors. In [8,9,11] the authors classified the applications between CPU-Bound, Weakly Memory-Bound and Memory-Bound, according to the following criteria:

1. **Reads/writes to memory** - represents the number of accesses to the shared and private memory addresses of the processor, considering read and write operations for each 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.

In [10], the authors used the number of data exchange operations as criteria for classification. In the PPI target, these operations represents barriers, locks/unlocks and threads/processes creation or termination. Using this criteria,

the applications were divided between High and Low Communication. The main problem with both classifications is that they were not done uniformly with all applications. The first classification used some 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 applications and using only PThreads.

Already using the first criterion (access read/write to shared memory), the 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, in [5,4] the authors investigated the impact of each PPI in the use of CPU and memory. In these studies the authors classified the applications in such a way that all scenarios analyzed contained at least one 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, [12] used some of these applications to verify the best performance and energy consumption in different multi-core architectures.

Thus, gathering all these previous studies, it was concluded that this set contained applications diverse enough to characterize a benchmark. After all, they were already being used as a benchmark, but an effort was needed to unify them, to analyze the whole set together and prepare them for use by others. This is one of the main objectives of this work

5 Results

This section presents the methodology for evaluate the applications, presenting its complexity and the result for energy consumption and performance.

5.1 Methodology

The results presented in this section are the average of 30 executions disregarding the extreme values. This number of executions was established as indicated in [7]. In this study the authors perform experiments that show that the minimum number of executions is MPI in order to obtain statistically acceptable results. Following the indications of this study, the results in MPI-1 and MPI-2 showed 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 actively with the results.

The toolkit Intel® Performance Counter Monitor (PCM) 2.0 was used to measure energy consumption. It has a tool to monitor the power states of the

Table 2: 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	$O(n * d)$
	150000	OE	Odd-Even Sort	$O(n^2)$
	32768	DFT	Discrete Fourier Transf.	
Matrix	2048×2048	TR	Turing Ring	$O(m * n)$
		DJ	Dijkstra	$O(n^3)$
		JA	Jacobi Method	
		MM	Matrix Multiplication	
		GS	Gram-Schmidt	

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 between these values was used.

One thing that is not simple to do when comparing different parallel applications is to set a workload that is equivalent to all of them. That way, we tested applications with small, medium, and large inputs. However, we choose to present in this work only the data referring to the medium sized inputs, since they allowed a better visualization of the results for most cases and are the same ones used in the previous works.

The Table 2 shows the size of the inputs used for each application, as well as the acronym used to identify each application in the following sections.

5.2 Complexity

The last column of the Table 2 presents the algorithmic complexity by application. Only the serial applications were used for this complexity analysis, which is based on the arithmetic operations of the algorithm. The complexity analysis for parallel applications is mainly based on execution time. However, several factors influence the complexity of parallel applications, such as load balancing and parallelization model, as [13] explain in their works.

The analysis of sequential complexities showed that the set of applications range from $O(n)$ to $O(n^3)$, with several other intermediate complexities. In this way, it is possible to conclude that the benchmark has enough diversity in this aspect to evaluate performance in different architectures.

5.3 Performance and Energy Consumption

The 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 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 126 GB in size and DDR3 technology. The operating system is Linux version 4.4.0-128 using Intel® ICC 18.0.1 compiler with optimization flags.

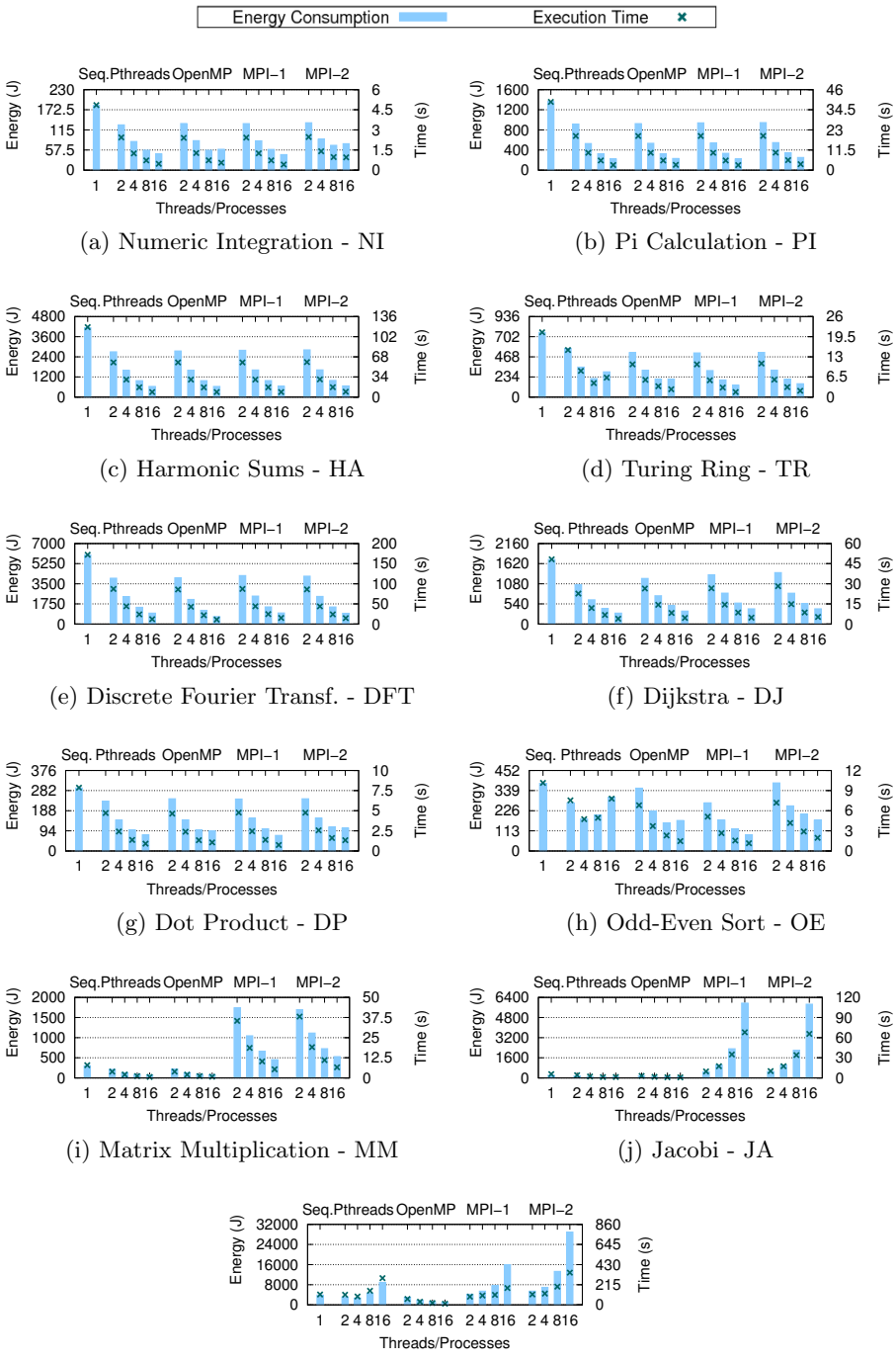
The results of energy consumption and performance are arranged in the same graphs. Bars show the energy consumption in joules and correspond to the y-axis values on the left. The time in seconds is represented by the marker X and is aligned to the y-axis on the right. Each chart displays the results of each application individually. These results refer to running using 2, 4, 8, and 16 parallel threads/processes for each PPI. In addition, the first result of each graph represents the sequential execution of the respective application. The other sets refer to each of the PPIs, nominated at the top of each graph.

In all the graphs in Figure 1, the scales of the two y-axes were adjusted so that both energy consumption and performance values for the sequential application were aligned at the same point. This allows an easier visualization of the impact of variation on the number of threads/processes.

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. However if the sequential results were proportionally aligned in a graph, we should note that this proportion does not appear in the parallel versions. This is because there are other factors that impact on energy consumption, such as the need for communication between tasks and increasing the complexity of the control structures that the operating system has to deal with.

In Figure 1, the 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 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. Following, the TR (Figure 1-d) application showed a different behavior when using PThreads. Except when using 16 threads, in the other cases this PPI had an increase in execution time in relation to the others. However, the energy consumption did not increase in a proportional way to the time, remaining close to the result of the other PPIs. A similar behavior, but with a worse scalability with PThreads, can be seen in OE.

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



(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

Fig. 1: Energy consumption and performance graphs for each application.

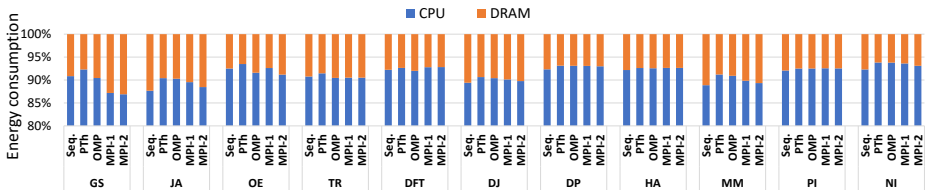


Fig. 2: Percentage of energy consumed by DRAM memory in relation to consumption by the CPU using 8 threads/processes.

16 threads, the energy consumption does not follow this reduction and it is the same seen with 8 threads. If we compare the three applications that only do iterative computation (DP, PI and NI), it can be seen that the execution time of DP and NI is smaller than PI (which still scaling with 16 threads). Therefore, the loss of scalability in the other two applications is certainly related to the size of the problem that was not adequate. In addition, all OpenMP results with 8 threads were smaller than the others in almost all cases. The easy structuring of parallelism with OpenMP ends up playing a fundamental role in these results when compared to manual parallelism using PThreads.

Regarding Odd-Even Sort (Figure 1-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 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, because we are still implementing efficient workload distribution for many tasks in some applications.

It is expected that the more robust the architecture, the better the MPI results can be. In this way it was possible to observe a 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 a 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. A similar behavior was observed by the authors in [12]. Additionally, the applications that presented the worst results for MPI are the only ones in [4] and [5] that showed a higher memory usage than CPU usage during execution. In this way we can conclude that a high memory use has a strong negative impact on energy consumption.

The worst results among all the applications were obtained with GS. Not considering OpenMP, the other PPIs did not present a superior scalability in relation to the sequential version. It was also the application that obtained the highest energy consumption among all, with a peak of about 30.000 joules with 16 processes in MPI-2, but with a much shorter execution time. 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.

Finally, we present in Figure 2 the energy consumption results divided by CPU consumption and memory consumption. For all amounts of threads/processes the results did not have much difference between them, therefore we only present the graph for 8 threads/processes. In this graph we can see that DRAM energy consumption is stable among PPIs in some cases, representing on average 10% of total consumption. In other cases, MPI-1 and MPI-2 always have a larger portion of energy destined for memory. In addition, the highest memory energy consumption occurs with MPI in GS. We will have to carry out another study to analyze the relation of this with the results obtained in Figure 1.

The difference between 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. 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 among processes. All of these factors impact on performance and consequently on energy consumption.

6 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 energy consumption and performance of PPIs in multi-core architectures. We first compared our proposed benchmark with 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 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 efficiently meet this demand, so it was necessary to create one from scratch.

Our experimental results showed that the applications generally have a good performance gain. In addition, the overhead caused by the energy consumption showed a behavior proportional to the number of threads/processes used in parallel. The exceptions are two applications (JA and GS) that demonstrated an unexpected behavior, but equal with MPI-1 and MPI-2. Both should be further investigated in the course of our study to verify the relationship of these results to the fact that they have been classified as highly memory-bound applications in previous studies.

Some applications in our benchmark still require a few more adjustments, as you could see in the results. However most applications showed a pattern in relation to gain performance. In many cases we have achieved near optimal performance, reducing runtime twice with 2 threads, 4 times with 4 threads, and so on. In addition, energy consumption has also been reduced in the same

proportion for this majority of cases. We conclude that after these adjustments, the applications will already be ready to be made available to the community for use in other studies.

As future works, we intend to verify how the distribution of threads/processes between different cores and processors affects our experiments. We should also repeat the experiments using another compiler, such as gcc without optimization flags. The 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 (some preliminary tests have shown that MPI gets better trade-offs in these cases). Finally, we also consider including more PPIs such as Intel TBB, Cilk or UPC.

References

1. Bourzac, Katherine: Supercomputing poised for a massive speed boost. Springer Nature International Journal of Science (2017)
2. Butenhof, D.R.: Programming with POSIX threads. Addison-Wesley Professional (1997)
3. Foster, I.: Designing and building parallel programs. Addison Wesley Publishing Company (1995)
4. Garcia, A.M.: Classificação de um benchmark paralelo para arquiteturas multinúcleo (2016)
5. Garcia, A.M., Schepke, C.: Uma proposta de benchmark paralelo para arquiteturas multicore. XVIII Escola Regional de Alto Desempenho pp. 285–289 (2018)
6. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced features of the message-passing interface. MIT press (1999)
7. Hunold, S., Carpen-Amarie, A.: Reproducible mpi benchmarking is still not as easy as you think. IEEE Transactions on Parallel and Distributed Systems **27**(12), 3617–3630 (2016)
8. Lorenzon, A.F., Cera, M.C., Beck, A.C.S.: Optimized use of parallel programming interfaces in multithreaded embedded architectures. In: VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on. pp. 410–415. IEEE (2015)
9. Lorenzon, A.F.: 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)
10. Lorenzon, A.F., Sartor, A.L., Cera, M.C., Beck, A.C.S.: The Influence of Parallel Programming Interfaces on Multicore Embedded Systems. In: Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual. vol. 2, pp. 617–625. IEEE (2015)
11. Lorenzon, A.F., Cera, M.C., Beck, A.C.S.: Performance and Energy Evaluation of Different Multi-Threading Interfaces in Embedded and General Purpose Systems. Journal of Signal Processing Systems **80**(3), 295–307 (2014)
12. Lorenzon, A.F., Cera, M.C., Beck, A.C.S.: Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy. Journal of Parallel and Distributed Computing **95**, 107–123 (2016)
13. Mikloško, J., Kotov, V.E.: Complexity of parallel algorithms. In: Algorithms, software and hardware of parallel computers, pp. 45–63. Springer (1984)
14. Rauber, T., Rünger, G.: Parallel programming: For multicore and cluster systems. Springer Science & Business Media (2010)