UNIVERSIDADE FEDERAL DO PAMPA

Sandro Matheus Vila Nova Marques

Transparent Optimization of OpenMP Applications via Thread Throttling and Boosting Techniques

Alegrete
2021

Sandro Matheus Vila Nova Marques

Transparent Optimization of OpenMP Applications via Thread Throttling and Boosting Techniques

Final paper presented to the Undergraduate Course in Software Engineering at the Federal University of Pampa as a partial requirement to obtain the Bachelor's degree in Software Engineering.

Supervisor: Prof. Dr. Arthur Francisco Lorenzon

Alegrete
2021

SERVIÇO PÚBLICO FEDERAL

MINISTÉRIO DA EDUCAÇÃO

Universidade Federal do Pampa

SANDRO MATHEUS VILA NOVA MARQUES

TRANSPARENT OPTIMIZATION OF OPENMP APPLICATIONS VIA THREAD THROTTLING AND BOOSTING TECHNIQUES

Monografia apresentada ao Programa de Engenharia de Software da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Monografia defendida e aprovada em: 07, de Maio de 2021.

Banca examinadora:

_____

Prof. Dr. Arthur Francisco Lorenzon

Orientador

Unipampa

_____

Prof. Dr. Marcelo Caggiani Luizelli

Unipampa

Prof. Dr. Antonio Carlos Schneider Beck Filho

UFRGS

Prof. Dr. Antoni Navarro Munoz

Barcelona Supercomputing Center

Assinado eletronicamente por **ARTHUR FRANCISCO LORENZON**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 07/05/2021, às 11:23, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Antonio Carlos Schneider Beck Filho**, **Usuário Externo**, em 10/05/2021, às 15:31, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **MARCELO CAGGIANI LUIZELLI**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 12/05/2021, às 14:18, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0516618** e o código CRC **C4A8E37D**.

Para meus amados pais.
Obrigado por terem me dado apoio e segurança durante todos esses anos.

"The dictionary is the only place that success comes before work." (Suits)

ABSTRACT

The growing number of cores in modern multicore architectures has brought together the need for better use of hardware resources. Consequently, two techniques have become widely used to optimize the performance and energy consumption of these environments: *Dynamic Concurrency Throttling* (DCT) and *Boosting*. On the one hand, DCT adjusts the number of threads in parallel regions to minimize the effects of intrinsic characteristics of the applications that impact performance and energy consumption (e.g., data synchronization and communication). On the other hand, *Boosting* techniques focus on making the performance reach its maximum level during all phases of the application by increasing the processor frequencies while respecting the Thermal Design Power (TDP). One of the main challenges is that each region of a parallel application can behave differently (i.e., memory access behavior and degree of parallelism) which makes using both techniques combined not a straightforward task. Choosing the wrong number of *threads* and enabling/disabling boosting frequencies in the incorrect phases can lead to increasing the energy consumption and performance degradation. To solve this problem, this work presents two strategies that apply DCT and *Boosting* to improve the trade-off between performance and energy consumption (represented by the *energy-delay product* - EDP): PFG, a strategy that optimizes each region of a given application, individually; and PCG that considers the combination of parallel and sequential regions during optimization. Both strategies are transparent, automatic, and deeply integrated into the OpenMP parallel programming interface, so no code modification or recompilation is necessary. By executing twelve well-known benchmarks in three multicore systems, PFG and PCG improve EDP by up to, respectively, 95.3% and 95.5% compared to standard OpenMP execution, 90.9%, and 94.8% on Varuna-PM and 80.5% and 83.7% against the Core Packing technique. We also show that PFG is more suitable for applications with high variability in the CPU workload, while PCG is better when there is low workload variability.

Key-words: High Performance Computing. Thread-level Parallelism. Turbo Boosting. Energy efficiency.

RESUMO

O número crescente de núcleos em arquiteturas multicore modernas trouxe consigo A necessidade de melhor uso dos recursos de hardware. Consequentemente, duas técnicas têm se tornado amplamente utilizadas para otimizar o desempenho e o consumo de energia desses ambientes: o *Dynamic Concurrency Throttling* (DCT) e *Boosting*. Por um lado, o DCT ajusta o número de threads em regiões paralelas para minimizar os efeitos das características intrínsecas das aplicações que afetam o desempenho e o consumo de energia (e.g., comunicação e sincronização de dados). Por outro lado, as técnicas de *Boosting* focam em fazer o desempenho atingir seu nível máximo durante todas as fases da aplicação, por meio do aumento das frequências do processador, respeitando o Thermal Design Power (TDP). Um dos principais desafios é que cada região de uma aplicação paralela pode se comportar de forma diferente (e.g., comportamento de acesso à memória e grau de paralelismo), o que torna o uso de ambas as técnicas combinadas tarefa complicada. Escolher o número errado de *threads* e habilitar/desabilitar as frequências de boost nas fases erradas pode levar ao aumento do consumo de energia e degradação do desempenho. Para resolver este problema, este trabalho apresenta duas estratégias que aplicam DCT e *Boosting* para melhorar o trade-off entre desempenho e consumo de energia (representado pelo *energy-delay product* - EDP): PFG, uma estratégia que otimiza cada região de um determinada aplicação, individualmente; e PCG que considera a combinação de regiões paralelas e sequenciais durante a otimização. Ambas as estratégias são transparentes, automáticas e profundamente integradas à interface de programação paralela OpenMP, portanto, nenhuma modificação ou recompilação de código é necessária. Por meio da execução de doze benchmarks amplamente conhecidos em três sistemas multicore, PFG e PCG melhoram EDP em até, respectivamente, 95,3% e 95,5% em comparação com a execução OpenMP padrão, 90,9 % e 94,8 % em Varuna-PM e 80,5% e 83,7% contra a técnica Core Packing. Também mostramos que o PFG é mais adequado para aplicações com alta variabilidade na carga de trabalho da CPU, enquanto o PCG é melhor quando há baixa variabilidade da carga de trabalho.

Palavras-chave: Computação de Alto Desempenho. Paralelismo a nível de *Threads*. Turbo Boosting. Eficiência Energética.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ACPI Advanced Configuration and Power Interface

DCT Dynamic Concurrency Throttling

DVFS Dynamic Voltage and Frequency Scaling

EDP Energy-Delay Product

HISTO Saturating Histogram

HPCG High Performance Conjugate Gradient

JA Jacobi Method Iteration

LUD LU Decomposition

MPI Message Passing Interface

NN Nearest Neighbor

OpenMP Open Multi-Processing

PPI Parallel Programming Interface

PThreads Posix Threads

QS Quicksilver

SC Streamcluster

SP Scalar Penta-diagonal Solver

SPMV Sparse-matrix Dense-vector Multiplication

TBB Threading Building Block

TDP Thermal Design Power

TLP Thread-Level Parallelism

TPACF Two Point Angular Correlation Function

TR Turing Ring

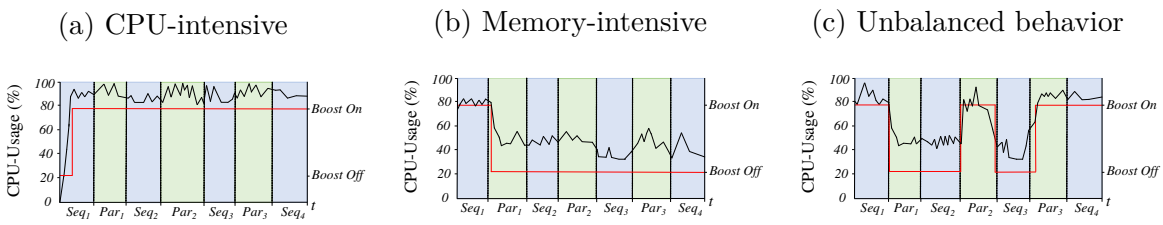
CONTENTS

1 INTRODUCTION

The growing use of virtualization and software-as-a-service (SaaS) in cloud systems has been increasing the need for using Dynamic Concurrency Throttling (DCT) and Dynamic Voltage and Frequency Scaling (DVFS) to make better use of hardware resources and keep the costs low. DCT artificially decreases the number of executing threads of a parallel application according to its available scalability, while DVFS exploits the processor idleness (usually caused by I/O operations or memory requests) to decrease the operating frequency of the core with a corresponding reduction in the supply voltage (SUEUR; HEISER, 2010). The clever employment of DVFS not only further improves the energy efficiency of such systems but also benefits big data centers and HPC servers. However, DVFS may also be applied in the opposite direction by using the so-called boosting techniques (e.g., Intel's Turbo CORE and AMD's Turbo Boost) (WAMHOFF; DIESTELHORST; FETZER,). In this mode, the operating frequency of a group of cores dynamically increases to maximize performance, while the remaining cores either run at a lower operating frequency or are powered down, respecting the limits of the Thermal Design Power (TDP) (CHARLES et al., 2009; WAMHOFF; DIESTELHORST; FETZER,).

Boosting techniques have been applied to speed up the execution of sequential and parallel phases in applications from many domains. This can be done by activating/deactivating boosting frequencies according to the characteristics of the applications, as shown in Figure 1. While boosting techniques are usually turned on to further accelerate the execution of CPU-intensive applications (Figure 1a), boosting frequencies are disabled for memory-intensive ones, to save energy, as there will be low CPU-usage (Figure 1b). However, many parallel applications have a mix of both behavior, i.e., some regions are CPU-intensive while other are Memory-intensive. On top of that, a great number of parallel applications are unbalanced and do not scale as the number of threads increases due to software and hardware issues (e.g., data-synchronization and off-chip bus saturation (SULEMAN; QURESHI; PATT, 2008; Lorenzon et al., 2019)). This unbalancing results in threads waiting for the others to finish (active waiting or in idleness), wasting resources, and spending extra energy with only marginal gains in performance.

Figure 1 – General CPU-Usage behavior of parallel applications



Source – The author

For this reason, DCT techniques have been used to bring back this balance by employing different strategies to artificially reduce the number of threads.

Given the considerations above, applying DCT together with the boosting techniques, besides bringing extra speedups to the sequential code, can also speed up some unbalanced threads to achieve even more equilibrium to the parallel execution, further improving the trade-off between performance and energy consumption. When both knobs are applied together, a key design choice is to decide the best strategy to boost the CPU frequencies in the regions of an application: fine or coarse grain. In the former, the decision to activate the boosting considers the behavior of each sequential and parallel region, individually. This scenario benefits the applications where the CPU workload changes between regions (Par_2, Seq_3, and Par_3 in Figure 1c). However, it implies changing the CPU boosting frequencies whenever a region changes, which may add an overhead to the application, as more computation must be done to understand the behavior of each region individually. On the other hand, the optimization at coarse-grain considers that the parallel and its next sequential region have similar workload behavior (Par_1 and Seq_2 regions in Fig. 1c). In this case, boosting frequencies are applied considering the behavior of both regions combined, decreasing the overhead of the fine-grain strategies, with costs in adaptability

1.1 Objectives

Considering the aforementioned scenario, the main objective of this work is to design and evaluate two different strategies to fulfill the challenging task of finding the ideal combination of degree of Thread-Level Parallelism (TLP) and boosting operating mode that optimizes the trade-off of performance and energy consumption (i.e., Energy-Delay Product - EDP) of OpenMP applications. The first strategy, **Poseidon Fine-Grain (PFG)**, performs the optimization at fine-grain. It aims to find the ideal configuration for each parallel (number of threads and boosting mode) and sequential region (boosting mode only), individually. On the other hand, in the second approach, **Poseidon Coarse-Grain (PCG)**, the optimization is executed at coarse-grain. Therefore, it considers that a region of interest to be optimized comprises a parallel region and its subsequent sequential one, as we are going to explain later in this work.

We can decompose this objective into specific ones as follows:

1. Characterize OpenMP applications by analyzing the memory access behavior and degree of thread-level parallelism.

2. Study the difference of the state-of-the-art boosting techniques (i.e., Intel Turbo Boost and AMD Turbo Core) and how they are managed in modern processors.

3. Design PFG and PCG learning algorithms to find the ideal number of threads and boosting operating mode for OpenMP applications.

4. Integrate both frameworks into the OpenMP library (*libgomp*) to provide transparency to the end-user and online characteristics to the search algorithm. This is important because the objective of both strategies is to provide an effortless optimization of OpenMP applications at runtime.

5. Analyze how the proposed algorithms perform in the chosen applications and provide guidelines on when to use each optimization strategy.

1.2 Organization of this work

The remainder of this work is organized as follows:

In Chapter 2 we present the fundamental concepts used in this work. First, an introduction about parallel architectures, parallel programming, and OpenMP are presented. Second, an overview of DVFS and boosting techniques and how the OS performs such features. Finally, the main points of this Chapter are outlined.

In Chapter 3 the related work is presented. It is divided into four sections: First, the works that explore optimization opportunities on boosting techniques are discussed. Second, the studies that aim to improve the degree of TLP exploitation are described. Third, the works that aim to optimize both the degree of TLP and boosting techniques combined are presented. Finally, the contributions of this work are compared with the related works.

We present in Chapter 4 the main contribution of this work: Poseidon Fine-grain and Poseidon Coarse-grain, two dynamic transparent frameworks to automatically improve EDP on OpenMP applications. In this part, the main mechanisms of both strategies, their transparent integration with the OpenMP environment, and the methodology used to validate them are highlighted.

In Chapter 5 the results from the validation of Poseidon Fine-grain and Poseidon Coarse-grain using different OpenMP applications are presented. The results are compared with different scenarios discussed in the methodology.

Chapter 6 concludes the work developed and presented in this manuscript. We also discuss promising ideas for extending our strategies and the publications regarding the scope of this work.

2 BACKGROUND

In this chapter, we introduce the background regarding the topics covered in this work. Firstly, an introduction of parallel architectures, parallel programming, and the OpenMP interface are presented. Then, DVFS and boosting techniques are explained. We focus on the implementation of both techniques and how they manage the hardware resources. Finally, we conclude this chapter with a summary of the presented topics.

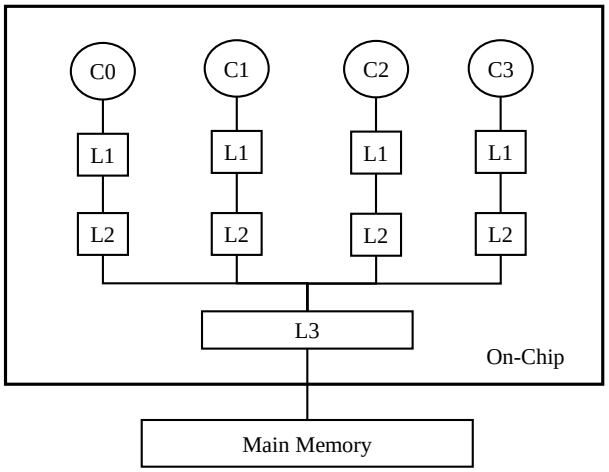
2.1 Parallel Architectures

A parallel architecture is a system that contains multiple processing units (commonly called cores) and a memory system in a single integrated circuit. Each core has its resources (e.g., registers and cache memories) and can communicate with each other using shared memory regions (PATTERSON; HENNESSY, 2016). Figure 2 shows a common organization of a multicore processor with four cores. As one can observe, each core (C0, C1, C2, and C3) has its own private memories (L1 and L2 caches) and share the L3 cache memories.

When executing a parallel application on a multicore processor, each core will execute a piece of the code and access/modify shared data by load and store operations. One of the main challenges of this approach is to maintain the coherence between these data, since this information is modified by different cores and stored in different cache levels. To tackle this problem, cache coherence protocols are used (PATTERSON; HENNESSY, 2016).

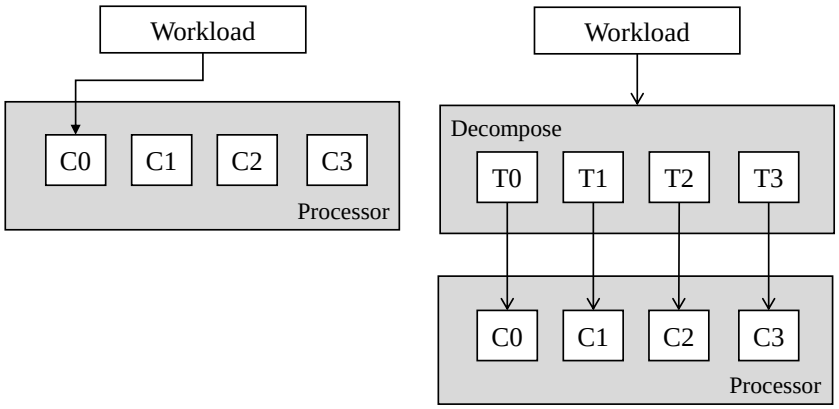
Cache coherence protocols ensure that the information used by different cores are always the most updated. They can be divided into two classes: snooping and directory-based. On the former, each cache has a copy of a given shared-data and its state (i.e.,

Figure 2 – Basic multicore organization in a four cores processor.



Source – The author

Figure 3 – Example of parallel computing



(a) Sequential execution (b) Parallel execution

Source – The author

modified, shared, or invalid), then once a core updates this data on its cache it invalidates the copies of other cache-levels. In this protocol, the cores must keep tracking the memory operations. On the latter, the directory-based protocols, the status of the data is maintained in just one location, called directory. The directory is responsible for the update or the invalidation of the copies in other caches-levels when the data is modified (PATTERSON; HENNESSY, 2016). However, the programmers just need to know about how the data exchange is performed because OpenMP Parallel Programming Interface (PPI) (discussed in Section 2.1.2) and the OS provides an abstraction with respect to cache coherence protocols to facilitate the development of parallel applications.

2.1.1 Parallel Programming

Parallel programming is a paradigm that exploits TLP of an application by splitting its workload into smaller parts (threads) that will be executed concurrently in multiple cores and exchanging data through shared variables or message-passing (PATTERSON; HENNESSY, 2016). It is widely used in applications that require high performance, such as scientific and cloud applications (BAILEY et al., 1991), (KANNADASAN et al., 2018), (CIORBA et al., 2019). However, as the multicore processors became highly available in the market, general-purpose applications (i.e., financial applications, social media applications, etc) have been applying parallelism as an approach to improve performance.

To better understand how TLP exploitation works, Figure 3 shows a comparison of parallel and sequential execution of an application's workload. On the sequential one, the workload is assigned for just one core, as illustrated in Figure 3a. On the other hand, in the parallel execution (Figure 3b), the workload is firstly decomposed in small parts to be divided into the threads (T0, T1, T2, T3), which are assigned to different cores (C0, C1, C2, C3) and computed concurrently.

When parallelizing a sequential code the programmer must know how the program will be divided, how many threads will execute, and how the communication between these threads will happen. Therefore, using a PPI make this job easier and faster. There are many widely known PPIs, such as OpenMP, Posix Threads (PThreads), Message Passing Interface (MPI), and Threading Building Block (TBB). However, in this work, we will focus on OpenMP PPI, because it is widely used, efficient, and very appealing to the programmers (as discussed next).

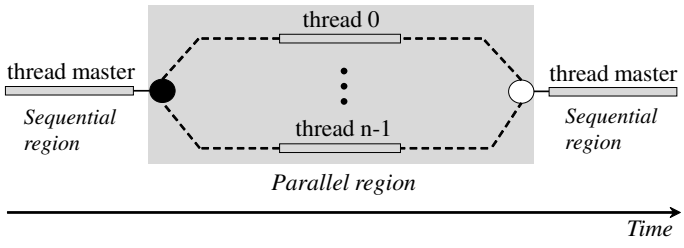
2.1.2 Open Multiprocessing

Open Multiprocessing is a parallel programming interface for shared memory that works in C/C++ and FORTRAN. It comprises a set of compiler directives, library functions, and environment variables that specify how the parallelism will occur (CHAPMAN; JOST; PAS, 2007). For this reason, it is well known for providing an easier way to design parallel applications when compared with other PPIs (e.g., PThreads, MPI, TBB, etc). Therefore, OpenMP is widely used and there are several different benchmarks: NAS Parallel Benchmark (SEO; JO; LEE, 2011a), Parboil (STRATTON et al., 2012), Rodinia (CHE et al., 2009), PARSEC (BIENIA et al., 2008), RAJA Performance Suite (BECKINGSALE et al., 2019), etc.

In OpenMP, the parallelism is exploited by indicating which parts of the code must be parallelized through the insertion of compiler directives. There are three main directives used to perform such indications: sections, tasks, and parallel-for. Sections are used to specify several different regions of the code to be executed by one single thread (i.e., similar to PThreads). Tasks are used in recursive algorithms. Both directives are used in very specific cases. On the other hand, parallel-for is used to distribute the iterations of a loop to multiple threads, in applications that use multidimensional data structures (i.e., vector, matrix, etc.). It is the most employed structure in OpenMP and all the benchmarks used in this work apply it.

A parallel application is composed of both sequential and parallel regions as depicted in Figure 4. In this case, the application is sequentially executed by the thread master (sequential region) until it finds a parallel region (e.g., indicated by a parallel-for

Figure 4 – A parallel application's execution structure



Source – The author

directive). In this phase, the workload is divided to be computed in multiple threads (as explained in the previous section), then once the computation is done, the threads are joined and the thread master continues the execution. Hence, we denote that parallel applications are composed of P parallel regions and $P + 1$ sequential ones.

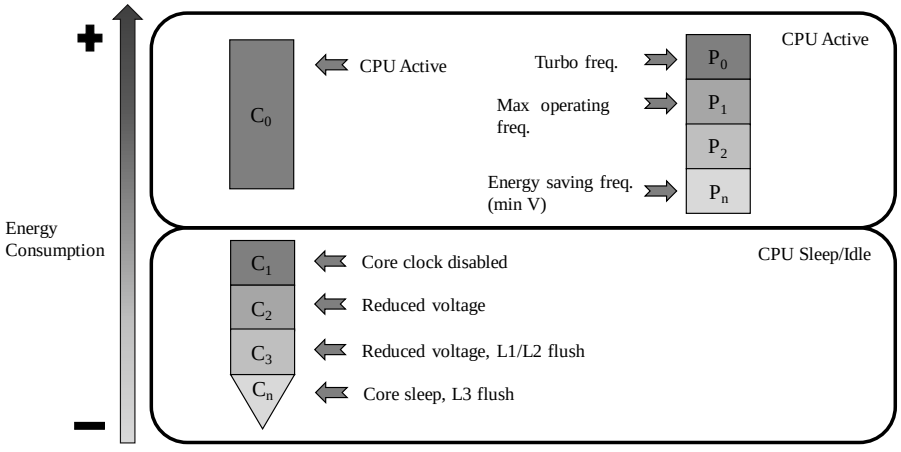
2.2 Dynamic Voltage and Frequency Scaling and Boosting Techniques

DVFS is a technique that aims to reduce the dynamic power consumption of the system (CARDOSO; COUTINHO; DINIZ,). This technique takes advantage of processor idleness (e.g., memory accesses), to decrease the frequency and voltage levels, thus minimizing the energy consumption. On the other hand, the so-called boosting techniques also apply DVFS but in a different direction. They opportunistically increase the frequency of a group of cores to maximize the performance, while the remaining ones are throttled down (CHARLES et al., 2009).

In order to allow the power management of hardware components, modern processors implement the Advanced Configuration and Power Interface (ACPI). It defines power and performance states (*C-states* and *P-states*, respectively). The *C-states* are used to reduce power, and consequently, the energy consumption, by powering down subsystems (Figure 5). On the other hand, *P-states* allow changing the frequency and voltage operating points of the active cores in order to either improve the performance or reduce the power consumption.

Figure 5 shows an abstract organization of the P-states and C-states. As one can observe, the basic *C-states* are defined by ACPI as follows: C_0, in which the CPU/Core is active and executing instructions; and from C_1 to C_n, where the CPU/Core is set to reduce the energy consumption by cutting the clock signal and power from unused cores, and switching off hardware components. In such cases, the more units switched off (the

Figure 5 – Abstract ACPI's P-states and C-states organization.



,

Source – The author

greater the value of C_n), the less energy is spent. On the other hand, *P-states* allow changing the frequency and voltage operating points of the CPU/core in order to either improve the performance or reduce the power consumption when it is active (C_0). As one can note in Figure 5, there is a set of *P-states* that correspond to different operating points (i.e., pairs of frequency-voltage), and a given *P-state* refers to one specific point ranging from P_0 to P_n (HAJ-YAHYA et al., 2018).

In processors that implement boosting technology and use the ACPI standard for power management, the P_0-*state* represents the highest operating frequency reached when the boosting is turned on. When it is turned off, the P_1-*state* is set, which represents the maximum base operating frequency without the interference of any boosting technique. Following this logic, the higher the value of P_n, the lower the operating frequency and voltage level. ACPI implements methods that allow the OS to change the *P-state* level (e.g., through the use of DVFS governors). In such cases, the OS selects an operating frequency while the voltage is set automatically by the processor. The most common governors are *performance* and *powersave*, in which the CPU frequency is set statically to the highest and lowest frequency, respectively; and *ondemand*, where the CPU frequency is set depending on the current system load (ALJUHNI et al., 2018) (SUEUR; HEISER, 2010).

However, it is important to emphasize that even though ACPI defines a pattern for the performance state organization, it varies according to the microarchitecture. For instance, the state-of-the-art Intel and AMD processors, propose some modification on the ACPI states organization. Intel inverts the model, which means that the highest frequencies levels are located in the *P-states* with the highest number (e.g., P_n-*state*). On the other hand, since AMD processors have a lower number of available operating frequencies, the number of *P-states* is also lower than the proposed by ACPI model (WAMHOFF; DIESTELHORST; FETZER,).

When comparing the most used boosting techniques, AMD Turbo Core and the Intel Turbo Boost, they show similar behavior. Both of the techniques increase the processor's cores frequency over the base operating frequency, considering the number of active cores and the TDP (WAMHOFF; DIESTELHORST; FETZER,). However, they have some differences related to their implementation. Unlike the AMD Turbo Core, the activation of the Turbo Boost depends on not only the TDP but the temperature of the packages as well (HAJ-YAHYA et al., 2018). For that reason the Intel's technique allows the processor to exceed the TDP using the thermal capacitance of the package.

In both AMD and Intel techniques, the boosting frequencies are managed at the hardware-level. To provide to the end-user the control of these techniques, ACPI implements the widely used *acpi-cpufreq* driver. It contains a file that can be modified at OS-level to inform the hardware whether the cores are allowed to scale to the boosting frequencies.

2.3 Summary

In order to stick with all the information presented in this chapter, we now briefly summarize the content of all its sections.

Parallel Architectures are systems with multiples processing units and a memory system (divided into shared and private memories) that allow the parallel execution of applications. These architectures apply cache coherence protocols to keep the information coherence and consistency among the multiple memory levels. However, the OS and PPI's environment make these protocols transparent to the programmers.

Parallel Programming is a paradigm to exploit TLP, which consists of dividing the execution of a given application's workload into multiple threads that will execute in different processing units concurrently. In order to employ such a paradigm in an easy and productive way, PPI's are used.

Open Multiprocessing is a PPI for shared-memory that comprises a set of compiler directives, library functions, and environment variable to specify how the parallelism shall occur, which provides an effortless interface to program in parallel. It has three main directives to indicate where the code must be parallelized: parallel-for, tasks, and sections. However, parallel-for (used in multidimensional structures) is the most used in OpenMP applications, and all the ones used in this work use it.

Dynamic Voltage and Frequency Scaling and Boosting Techniques. DVFS is a technique that takes advantage of processor idleness and decreases the frequency/voltage levels to minimize energy consumption. Oppositely, boosting techniques (i.e., AMD Turbo Core and Intel Turbo Boost) apply DVFS to increase the frequencies of a group of cores to improve performance, respecting the TDP. Both techniques use the ACPI's *P-states* and *C-states*, which are mechanisms to manage the frequency and voltage scaling of the modern processors. While DVFS can be controlled through governors, boosting frequencies are hardware controlled. However, to tackle this problem ACPI implements the *acpi-cpufreq*, a widely used driver that contains a file to allow the end-user to control the boosting status (i.e., active or inactive) at OS-level.

3 RELATED WORK

In this chapter, the related work of this work is discussed in chronological order. First, works that optimize the use of boosting techniques or the number of threads are described. Then, works that optimize both knobs. Finally, the contributions and the comparison with the related work are presented.

3.1 Boosting Techniques

Many works perform boosting optimization to improve the performance, energy consumption, or processor lifetime. Juan et al. (JUAN et al., 2013) propose a machine-learning algorithm to improve the performance and energy consumption of parallel applications under power or throughput constraints. The algorithm works as follows: given a power or throughput limit, it executes the applications multiple times and applies a model to learn how different frequency levels impact the performance and energy consumption. Then, the results generated are validated using a method called LOOCV, which tests the accuracy of the model. Finally, if the results are accurate, an Optimization Engine applies the optimal frequencies. Autoturbo (LO; KOZYRAKIS, 2014) is a daemon that predicts the characteristics of the application to activate/deactivate the boosting. The proposal comprises three steps: (*i*) It applies a learning algorithm that executes the application multiple times to classify it based on the memory accesses patterns; (*ii*) using this classification, Autoturbo applies a heuristic that determines if the boosting feature benefits a given metric; and (*iii*) At run time, the daemon uses the result of the heuristic to choose the state of boosting.

Wamhoff et al., (WAMHOFF; DIESTELHORST; FETZER,) propose a library called TURBO that allows the control of the processor's states (i.e., *P-states* and *C-states*) and a model to guide software designers in the use of the boosting technique. The model calculates when the transitions to the higher *P-states* (i.e., Turbo Frequencies) payoff regarding performance. It considers different frequency levels and the time to transit between these frequencies. The authors apply the TURBO library to apply this model. This library simplifies the end-user to accesses the processors' frequencies by using a set of layers with different levels of abstraction to communicate with hardware components and drivers (*acpi-cpufreq*, MSRs, etc). Maiti et al. (MAITI; KAPADIA; PASRICHA, 2015) propose a framework that aims to improve energy consumption under performance limitations and throughput under a TDP threshold. The framework works as follows: It captures run-time information (e.g., instructions per cycle, cache misses) for each core of the processor. Based on such information and a given budget (i.e., performance or TDP), the framework selects the group of cores and maps the threads on them. Finally, using a heuristic that estimates the frequency/voltage optimal points, the framework chooses the frequency for each selected core.

Verner et al. (Verner; Mendelson; Schuster, 2017) present an extension of Amdahl's

law to predict the speedup and energy savings on applications using the boosting feature. The machine used in the tests consists of an 8-core Intel Xeon 2.9 GHz and a 12-core Intel Xeon 2.2GHz. GuardBoost (KHDR; AMROUCH; HENKEL, 2018) aims to improve the processors boosting performance considering aging effects. It comprises of two different approaches to decrease both long and short-term aging. First, GuardBoost calculates a power constraint using a model to reduce the maximum voltage that the processor can scale, which diminishes the power dissipation that in turn reduce the long-term aging. On the other hand, it also dynamically estimates near-optimal guardbands using a heuristic to decrease the occurrences of voltage downscale and then reduce the short-term aging.

Adaptive Turbo Boosting (KONDGULI; HUANG, 2018) is a hardware architecture to improve performance via boosting. While executing the application in the main thread, the architecture uses an extra thread in an idle core to predict and execute the next instructions. It allows the main thread to stay more time processing data and executing a lower number of load/stores, thus improving the use of boosting frequencies.

Michaud., (MICHAUD, 2020) proposes an open-source boosting technique for homogeneous and heterogeneous systems. The algorithm implements a thermal model to estimate the maximum power dissipation the processor can reach without exceeding a temperature threshold. Then, it chooses the ideal boosting frequency.

3.2 Dynamic Concurrency Throttling

Many works have focused on performance and energy consumption optimization through selecting the ideal TLP degree. Feedback-Driven Threading (FDT) (SULEMAN; QURESHI; PATT, 2008) optimizes the performance of OpenMP applications. FDT uses a specific compiler that analyzes the synchronization and communication by inserting instructions at the beginning and end of critical sections. Based on these analyses, the application is reexecuted with the adjusted number of threads. ACTOR (CURTIS-MAURY et al., 2008) uses an artificial neural network to predict the performance of different parallel regions of an application. It comprises three steps: Before the execution, artificial neural networks (ANNs) are trained to analyze the hardware counter events and the performance in different degrees of parallelism with a multivariate regression algorithm. Then, at runtime, ACTOR uses the ANN analysis to predict the performance of each parallel region. Finally, it executes each parallel region with the estimated number of threads. Thread Reinforcer (PUSUKURI; GUPTA; BHUYAN, 2011) determines a near-optimal number of threads for an application. It consists of two steps: (*i*) the application is ran multiple times for a short duration, while the framework searches for the appropriate number of threads; and (*ii*) Once the number is found the application is fully reexecuted with the degree of parallelism determined in the first step.

Sridharan et al. (SRIDHARAN; GUPTA; SOHI, 2013) propose ParallelismDial (PD), a system that continuously monitors the system efficiency and adapts the degree of

parallelism over the execution. PD uses an algorithm based on a hill-climbing search to find the best number of threads that comprises three steps. In the first one, the parallel region runs in sequential mode to establish a baseline. On the second step, the region is executed with three different degrees of parallelism (low, medium, and high). Finally, the search refines the best number of threads and continues the executions until it finds the best solution. Varuna (SRIDHARAN; GUPTA; SOHI, 2014) extends PD. It is composed of an analytical engine and a manager. The former continuously monitors changes in the system behavior, using hardware counters, models of execution, and determines the ideal number of threads. The latter, apply in the execution of the optimal degree of parallelism found by the analytical engine.

OpenMPE (ALESSI et al., 2015) is an OpenMP extension for energy management. It automatically adjusts the number of threads by the insertion of a series of directives that indicate which parts of the code must be optimized. Ju et al., (JU et al., 2015) propose a prediction model to optimize the number of threads of parallel applications that run in heterogeneous systems. Mariani et al. (MARIANI et al., 2017) propose a methodology to analyze exascale systems. The proposal comprises two steps: (*i*) it applies a learning algorithm to classify and cluster the threads that have the same behavior; and (*ii*) executes an application multiple times with a small degree of parallelism to analyze the classification found the previous step and use an extrapolation model to infer the behavior with a large number of threads. Rughetti et al. (RUGHETTI et al., 2017) propose a framework to optimize the performance of software transactional memory. The framework is composed of three components: (*i*) a Statistics Collector that collects samples of the memory accesses of the workload; (*ii*) Neural-Network that uses the samples to predict the variation of the throughput with different degrees of parallelism; and (*iii*) Control Algorithm which uses the predictions to manage the parallelism.

LAANT (Lorenzon; Dellagostin Souza; Schneider Beck, 2017) optimizes the execution of OpenMP applications by selecting the ideal number of threads for each parallel region at runtime. It is composed of three functions that are inserted in the code to indicate to the LAANT runtime system, which parallel regions must be optimized. LAANT uses a finite state machine to implement a hill-climbing algorithm. In each state of the algorithm, it executes a given parallel region with a different number of threads until it finds the ideal result. Aurora (Lorenzon et al., 2019) is an extension of LAANT that comprises a refined algorithm, a large number of optimization metrics (performance, energy, aging), and full integration with the OpenMP environment. BOPPETG (KHOKHRIAKOV; MANUMACHU; LASTOVETSKY, 2020) is a method for determining the Pareto-optimal solution considering the energy and performance of parallel applications. The method is expressed in three steps: (*i*) the application is re-implemented in a way that allows the modification of the threadgroups (chunk size) and the number of threads; (*ii*) the applications is exhaustively executed with all configurations of threadgroups and number of

threads to capture the energy consumption and execution time; and (*iii*) all the results of energy and time are compared, and the Pareto-optimal solutions returned.

3.3 DCT and Boosting Techniques

The aforementioned works only consider one part of the optimization problem (i.e., boosting or TLP degree). More complete solutions take into consideration both knobs. Raghavan et al. (RAGHAVAN et al., 2012) propose a computational sprinting to improve the responsiveness of embedded applications. It continuously monitors the degree of TLP and power dissipation of the processor through hardware performance counters. When the search algorithm observes that there are more running threads than active cores, it wake-ups idle cores distributes the threads, and applies boosting on those cores.

seBoost (PAGANI et al., 2015) is a boosting algorithm that improves the performance of multiple applications running on heterogeneous systems. It assumes that each application has an input table with the power profile for every thread. Then, seBoost splits the applications into two groups: the ones that run on the boosted cores and the applications that run on the remaining cores. With that, seBoost maximizes the frequencies of the boosted cores, while minimizing the frequencies in the non-boosted ones.

Core Packing (RICHINS et al., 2018) applies boosting in parallel regions with a low degree of TLP to improve the performance and energy consumption of Hadoop-based applications. It continuously monitors the TLP behavior of each application and migrates threads with a low workload to the most active cores and boosts the frequency of such cores.

3.4 Contributions

When comparing the previous work, the strategies discussed in Section 3.1 and Section 3.2 try to optimize only the boosting feature or the degree of TLP. On the other hand, our strategies are more complete solutions as they are capable of optimizing both knobs at run-time. When comparing PFG and PCG to more complete strategies discussed in Section 3.3, which consider both degree of TLP and boosting techniques, one can describe the following differences:

- Our strategies are the only ones that learn at run-time the ideal configuration of the degree of TLP and boosting technique for each parallel/sequential region. All the other strategies only consider the execution of the application with different degrees of TLP but do not optimize it at runtime.

- Poseidon Fine-Grain and Poseidon Coarse-Grain do not rely on any sort of hardware modification or the operating system, neither need extra tools to monitor the application's behavior as needed by the strategies presented by (RAGHAVAN et

al., 2012; PAGANI et al., 2015; RICHINS et al., 2018). Instead, they use the information provided by the HW counters and common tools available in modern architectures and Linux OS.

- Our strategies are transparent (i.e., no changes in the source nor binary codes are necessary): the user can enable them through the use of one environment variable in Linux OS.

4 OPTIMIZING THE USE OF BOOSTING TECHNIQUES AND DEGREE OF TLP

In this chapter, the main contribution of this work is presented. First, in Section 4.1, we explain how our proposals are integrated into the OpenMP environment. Second, we explain how our algorithms are organized, in Section 4.2. Then in Section 4.3 and Section 4.4, the different phases of the search algorithm on both approaches are discussed. Finally, the validation methodology is presented in Section 4.5.

4.1 OpenMP Integration

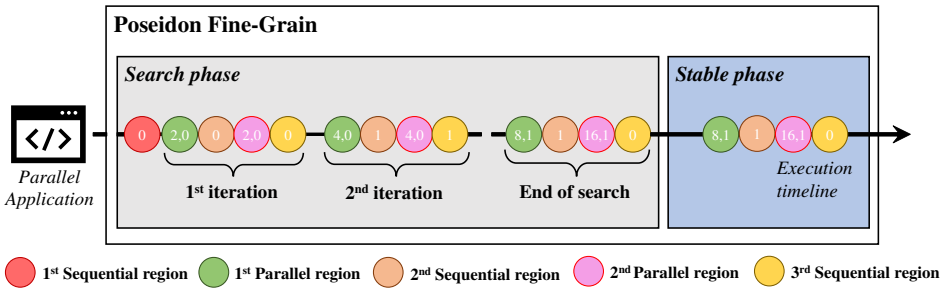
To provide an effortless approach (i.e., no code modification or recompilation), our strategies are deeply integrated with the GNU Offloading and Multiprocessing Runtime library (*libgomp*). This library implements the OpenMP environment. To use our approaches, the user has to replace the original OpenMP *libgomp* library with the PCG or PFG ones. Our modified *libgomp* includes all the original OpenMP functions plus PCG/PFG functionalities. This effortless approach works because the *libgomp* library is dynamically linked to the OpenMP applications. Hence, any modification of its original code is transparent for the user. When the environment variable *OMP_POSEIDON* is set *TRUE* in the OS, the OpenMP thread management environment is replaced for the implemented by PCG/PFG. Otherwise, the application executes without any influence of both techniques. Furthermore, besides not requiring any modifications in the original user binary, any changes in the OS (e.g., package installation, kernel recompilation, or superuser permission) are also not necessary because our strategies use the native tools from the OS and OpenMP environment. The boosting control is done by using the internal files of the *acpi-cpufreq* driver. On the other hand, to calculate the EDP of each region, PCG and PFG retrieve the execution time via the *omp_get_wtime* function and energy consumption from the hardware counters through the Running Average Power Limit library (HäHNEL et al., 2012).

4.2 Algorithms' Organization

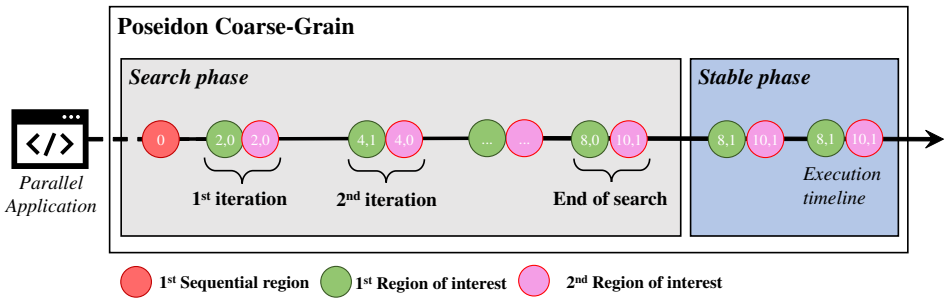
A parallel application that applies the fork-join model is composed of P parallel regions and S sequential regions, where after each parallel region, there is a sequential one. Our strategies consider different granularity to optimize these parallel/sequential regions. On the one hand, Poseidon Fine-Grain applies the optimization for every single parallel and sequential region. On the other hand, Poseidon Coarse-gain considers the combination of a parallel and its subsequent serial region.

Nevertheless, despite the difference in the granularity level, both techniques are divided into two phases: learning (i.e., search) and stable, as illustrated in Figure 6. Each strategy employs its learning algorithm (described in the next Sections) over each

Figure 6 – Illustration of the search and stable phase of each search strategy. The values inside each region (circle) represents the configuration: <#threads, boosting mode> for the parallel region; and <boosting mode> for the sequential region.



(a) Poseidon Fine-Grain



(b) Poseidon Coarse-Grain

Source – The author

region of interest towards the solution. Every time the execution of a region of interest is repeated, a new configuration is evaluated and compared, to assess the region's scalability. At the end of the search phase, each region is assigned with an individual configuration of boosting mode (parallel and sequential regions) and number of threads (parallel regions). Once the search converges to a configuration (i.e., the learning process has finished), the stable phase for that region is activated. From this moment on, the region is executed with the assigned configuration until the end of the execution.

4.3 Poseidon Fine-Grain

Considering that sequential regions can behave differently than parallel regions w.r.t. the CPU workload, this strategy optimizes each parallel and sequential region individually. Hence, as shown in Figure 6a, it searches for: (*i*) an ideal combination of TLP degree and boosting mode for each parallel region; and (*ii*) an ideal state of boosting mode for each sequential region.

The Poseidon Fine-Grain (***PFG***) is given an application A with m independent parallel regions $P = \{P_1, P_2, \ldots P_m\}$ and n independent sequential regions $S = \{S_1, S_2, \ldots S_n\}$. Each region in P can be executed by at most c distinct cores $C =$

Algorithm 1 Poseidon Fine-Grain: Search Algorithm

Input: $C \leftarrow \{C_1, C_2, \ldots, C_c\}$: set of threads/cores
$T \leftarrow \{Off, On\}$: set of boosting states
$P \leftarrow \{P_1, P_2, \ldots, P_m\}$: parallel regions of application A
$S \leftarrow \{S_1, S_2, \ldots, S_n\}$: sequential regions of application A
γ: Time to write the boosting controller file
α: Initial number of threads
β: Increasing number of threads

1: $t \leftarrow T\{1\}$: initial state of turbo feature
2: $\phi' \leftarrow \infty$: minimum metric measurement found so far
3: **while** application executes **do**
4: **for** every parallel region $P_i \in P$ **do**
5: $bestTLP = DCT\_Search(P_i, \alpha, \beta, t)$
6: $Boost\_Search(P_i, bestTLP, \gamma, T\{2\})$
7: **end for**
8: **for** every sequential region $S_i \in S$ **do**
9: $Boost\_Search(S_i, 1, \gamma)$
10: **end for**
11: **end while**

$\{C1, C2, \ldots, C_c\}$ and the boosting feature t in $T = \{Off, On\}$. On the other hand, each region in S can be executed by only one core (as it is a sequential region) and the boosting feature t in $T = \{Off, On\}$. We denote by $A(p, s)$ the set of regions in an application A. The optimization problem we are interested in seeks an assignment of regions P and S to a subset of threads/cores in C and the boosting configuration t in T. We denote by C^* the set of all subsets of threads/cores in C, that is, $C^* = \cup_c \binom{C}{c}$. A feasible assignment can be defined as a function of $\phi : P_i \rightarrow C^* \times T(\forall P_i \in P)$ that assigns a subset of threads/cores and boosting mode to a region $P_i \in P$. Hence, we denote by $M : (P \times C^* \times T) \rightarrow R^+$ the measured EDP of executing the parallel region P_i on c distinct cores and t boosting mode. We say an assignment ϕ is optimized if $\sum_{P_i \in P} M(P_i, \phi(P_i))$ is minimum.

The learning algorithm implemented by the PFG strategy is shown in Algorithm 1. It receives as input the set of cores C, the set of states of boosting T, and the set of regions P and S for a given application A. Also, it considers three parameters that are obtained from the target architecture during its initialization: the initial number of threads α, the increasing factor of the number of threads β (both are obtained by factorizing the number of cores of the target architecture), and the time to write the boosting controller file γ. Furthermore, the search algorithm keeps track of the current state of each region P_i and S_i in the application's execution. As the application executes, for every parallel region $P_i \in P$, it first applies a search for an ideal number of threads (*DCT_Search* – Algorithm 2) and then for the boosting mode. As the search space in the threads domain is more significant than in the boosting domain, as a design choice, it first searches in the thread domain.

Therefore, the procedure *DCT_Search* starts increasing exponentially (w.r.t. β) the number of threads α given to P_i (lines 2-5), while minimizing score function $M(\phi) = M'$. Upon reaching a local minimum with respect to function M, the algorithm performs a modified hill-climbing search and lateral movements (lines 8-19) in the interval where the algorithm has found a minimum value for M. Finding the ideal number of threads to

Algorithm 2 DCT Search Algorithm

Input: r: region of interest
α: Initial number of threads
β: Increasing number of threads
t: initial state of Boosting feature

1: $M' \leftarrow M(r, \alpha, t)$
2: **while** $M' \leq \phi'$ and $\alpha \leq totalCores$ **do**
3: $\phi' \leftarrow M'$ and $\phi \leftarrow (r, \alpha, t)$
4: $\alpha \leftarrow \alpha \times \beta$ and $M' \leftarrow M(r, \alpha, t)$
5: **end while**
6: **if** $M' \geq \phi'$ and $\alpha \leq totalCores$ **then**
7: $upper \leftarrow \alpha$ and $lower \leftarrow \alpha/\beta$
8: **while** $lower \leq upper$ **do**
9: $\alpha' \leftarrow (upper + lower)/2$ and $M' \leftarrow M(r, \alpha', t)$
10: **if** $M' \geq \phi'$ **then**
11: $upper \leftarrow \alpha'$ and $lower \leftarrow \alpha/\beta$
12: **end if**
13: **if** $M' \leq \phi'$ **then**
14: $\phi \leftarrow (r, \alpha', t)$ and $upper \leftarrow \alpha$ and $lower \leftarrow \alpha'$
15: **end if**
16: **if** $|upper - lower| = 2$ **then**
17: *lateral movement*
18: **end if**
19: **end while**
20: **end if**

Algorithm 3 Boosting Search Algorithm

Input: $T \leftarrow \{Off, On\}$: set of boosting states
r: region of interest to be optimized
γ: Time to write the boosting controller file
nt: number of threads running the region

1: **if** $time(\phi') > \gamma$ **then**
2: $\Omega \leftarrow T\{2\}$ and $M' \leftarrow M(r, nt, \Omega)$
3: **if** $M' \leq \phi'$ **then**
4: $\phi \leftarrow (r, nt, \Omega)$
5: **end if**
6: **end if**
7: **return** ϕ

execute any parallel region can be considered a convex optimization problem. It means that there will be one specific number of threads that deliver the best EDP result. Hill Climbing algorithms are very suitable for this kind of problem and are also known for having low complexity ($\Theta(\log n^2)$) and being fast, which is essential to reduce the technique overhead.

After converging to an ideal or near-ideal number of threads, it will search in the boosting domain through the function *Boost_Search* (Algorithm 3). Due to the overhead for managing the boosting mode, the search algorithm only changes the boosting status when it brings benefits to the execution, i.e., if the time to write the boosting controller file –γ– is lower than the execution of P_i itself. Also, the search algorithm only writes into the boosting controller file when the state between regions P_i and the previous sequential region changes, avoiding writing the same information in the file, which could very likely lead to overheads. Furthermore, every time a new sequential region $S_i \in S$ is executed, the search algorithm will perform the *Boost_Search* procedure to find the ideal boosting configuration for S_i.

Algorithm 4 Poseidon Coarse-Grain: Search Algorithm

Input: $C \leftarrow \{C_1, C_2, \ldots, C_c\}$: set of threads/cores
$T \leftarrow \{Off, On\}$: set of boosting states
$R \leftarrow \{R_1, R_2, \ldots, R_r\}$: regions of interest of application A
γ: Time to write the boosting controller file
α: Initial number of threads
β: Increasing number of threads

1: $t \leftarrow T\{1\}$: initial state of turbo feature
2: $\phi' \leftarrow \infty$: minimum metric measurement found so far
3: **while** application executes **do**
4: **for** every region of interest $R_i \in R$ **do**
5: $bestTLP = DCT\_Search(R_i, \alpha, \beta, t)$
6: $Boost\_Search(R_i, bestTLP, \gamma, T\{2\})$
7: **end for**
8: **end while**

4.4 Poseidon Coarse-Grain

Considering that managing the boosting mode in each region individually (parallel or sequential) can take more time than the execution of the region itself[1], this strategy considers that the region of interest $R_i \in R$ comprises the combination of the parallel region $P_i \in P$ and its subsequent serial region.

The Poseidon Coarse-Grain (***PCG***) strategy is given an application A with r independent regions of interest $R = \{R_1, R_2, \cdots, R_r\}$. Each region in R can be executed by at most c distinct cores $C = \{C_1, C_2, \cdots, C_c\}$ and the boosting feature t in $T = \{Off, On\}$. We denote by $A(r)$ the set of regions in an application A. The optimization problem we are interested in seeks an assignment of regions R (of a given application A) to a subset of threads/cores in C and the boosting configuration t in T. We denote by C^* the set of all subsets of threads/cores in C, that is, $C^* = \cup_c \binom{C}{c}$. A feasible assignment can be defined as a function of $\phi : R_i \rightarrow C^* \times T(\forall R_i \in R)$ that assigns a subset of threads/cores and boosting mode to a region $R_i \in R$. Hence, we denote by $M : (R \times C^* \times T) \rightarrow R^+$ the measured EDP of executing the region of interest on c distinct cores and t boosting mode. We say an assignment ϕ is optimized if $\sum_{R_i \in R} M(R_i, \phi(R_i))$ is minimum.

The learning algorithm is shown in Algorithm 4. It receives three parameters as input: the set of cores C, the set of states of boosting T, and the set of regions R. Also, it considers the same three parameters already explained for the *PFG strategy*: α, β, and γ. For every region of interest R_i in R, the learning algorithm searches for the configuration that includes the ideal degree of TLP (line 5) and boosting state (line 6). The algorithms used to search for the best number of threads (*DCT_Search*) and boosting (*Boost_Search*) are the same used by the *PFG* strategy. Therefore, the only difference between both strategies is where the optimization is applied.

[1] Through an extensive set of experiments we have found that the time to perform the boosting control is, on average, 0.074ms, 0.136ms, and 0.145ms on the AMD 24-core, Intel 24-core, and Intel 88-core systems, respectively.

4.5 Methodology

4.5.1 Benchmarks

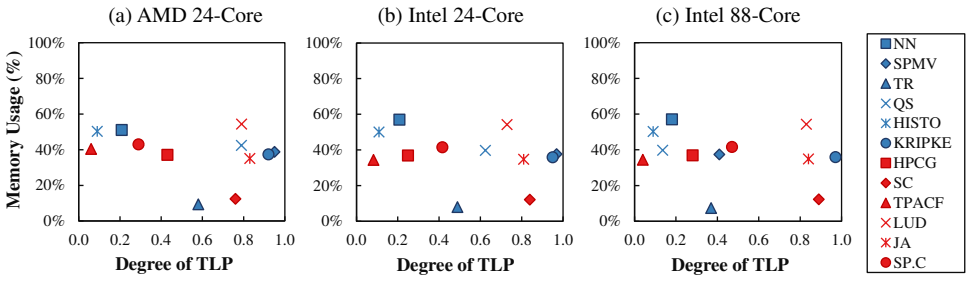
Twelve applications already parallelized with OpenMP and written in C/C++ from assorted benchmarks suites and domains were chosen. The benchmarks used were: **Three from the Parboil (STRATTON et al., 2012):** *sparse-matrix dense-vector multiplication* (SPMV); *two point angular correlation function* (TPACF); and *saturating histogram* (HISTO). **Two from the CORAL-2 Benchmarks:** Quicksilver (QS) and Kripke. **Three from the Rodinia Benchmark Suite (CHE et al., 2009):** *LU Decomposition* (LUD), *nearest neighbor* (NN), and *streamcluster* (SC); **Three from different domains:** *Turing Ring* (TR) (PAUDEL; AMARAL, 2011), *Jacobi method iteration* (JA) (QUINN, 2004), and the *high performance conjugate gradient* (HPCG) (DONGARRA; HEROUX; LUSZCZEK, 2015). **One kernel from the NAS-PB (BAILEY et al., 1991).** *scalar penta-diagonal solver* (SP). As the original version of NAS is written in FORTRAN, the OpenMP-C version developed by the authors in (SEO; JO; LEE, 2011b) were considered. The applications were executed with the standard input set defined on each benchmark suite.

As shown in Figure 7, the benchmarks cover different behaviors w.r.t. (*i*) the degree of TLP (as defined by the authors in (BLAKE et al., 2010) – the closer this value is to 1.0 normalized to the total number of available cores, the more TLP is available), which varies from TPACF (lowest TLP available), where less than 10% of the execution is performed in parallel to the SPMV benchmark (highest TLP available); and (*ii*) the memory usage, that is, the amount of time spent accessing main memory. The higher this value, the more time is spent in main memory accesses, which represents most of the time threads with high communication demands.

4.5.2 Execution Environment

The experiments were performed on three multicore architectures, as shown in Table 2. All the architectures used Linux kernel v. 4.16. The CPU frequency was

Figure 7 – Characteristics of each benchmark on each processor



Source – The author

Table 1 – Boosting frequency levels on different machines

AMD 24-Core	Threads	1-2	3-4	5-6	7-8	9-24	Base			
	Frequency [GHz]	4.6	4.5	4.4	4.3	4.2	3.8			
Intel 24-Core	Threads	1-4	5-8	9-24	Base					
	Frequency [GHz]	2.8	2.7	2.6	2.3					
Intel 88-Core	Threads	1-4	5-6	7-8	9-10	11-12	13-14	15-16	17-88	Base
	Frequency [GHz]	3.6	3.4	3.3	3.2	3.1	3.0	2.9	2.8	2.2

Source – The author

configured to adjust according to the workload application, using *ondemand* as DVFS governor, which is the standard one used in most Linux versions. The frequency range that each group of cores can scale when the boosting mode is activated is shown in Table 1. Each applications were compiled with GCC/G++ 9.2, using the *-O3* optimization flag, and the OpenMP 5.0. All the executions consider the thread affinity set to *close*, where threads are allocated near to each other (CHAPMAN; JOST; PAS, 2007). The results presented in the next section are an average of twenty executions with a standard deviation lower than 0.5%.

When executing parallel applications, the usual configuration applied is with the maximum number of threads (i.e., number of threads equals the number of cores) and boosting disabled. For this reason, we consider this configuration as our **Baseline**. We also consider other three scenarios as follows: **Boost-Only**: the same configuration as the baseline, but with boosting *on*; **DCT-Boost-Off**: the learning algorithm employed by PFG and PCG to find an ideal degree of TLP for each parallel region, but with boosting *off* during the whole execution; **DCT-Boost-OS**: the same as the previous one, but with boosting *on*, i.e, boosting controlled by the Linux OS; **Exhaustive Search Solution**: The execution of each application with the optimal number of threads and boosting mode, without the learning cost. The best static solution was obtained by executing each application with all possible combinations of the number of threads and boosting mode. We also compare our strategies to two state-of-the-art approaches: (*i*) **Varuna-PM**, by faithfully implementing its programming model (as defined in (SRIDHARAN; GUPTA; SOHI, 2014)) and applying it to our benchmarks; (*ii*) and **Core Packing**: which

Table 2 – Main characteristics of each multicore architecture

Configs.	AMD Ryzen 9 3900X	Intel Xeon E5-2630	Intel Xeon E5-2699 v4
Microarch.	Zen 2	Sandy Bridge	Broadwell
#Packages	1	2	2
#Phys. Cores	12	12 (6+6)	44 (22+22)
#HW Threads	24	24	88
L3 Cache (total)	64MB	30MB	110MB
Main memory	32GB	32GB	256GB

Source – The author

continuously monitors the TLP behavior of each application and adjusts the boosting frequency accordingly (RICHINS et al., 2018).

5 EXPERIMENTAL RESULTS

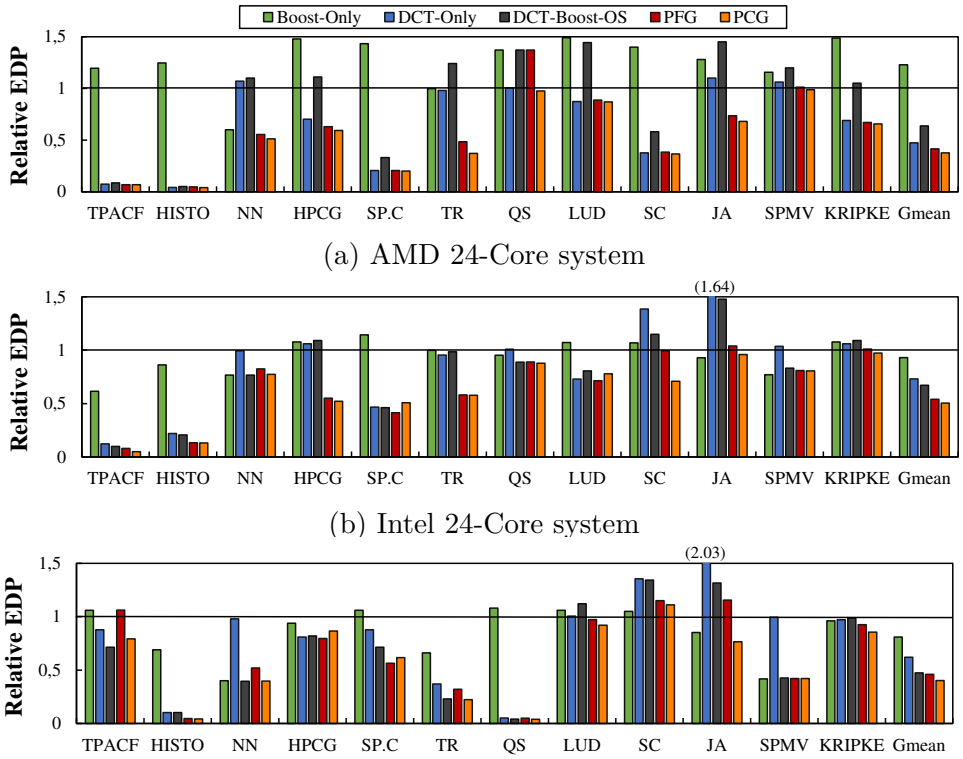
In this chapter, we first compare each of the scenarios described in Section 4.5 against the PFG and PCG, considering EDP in Section 5.1. Then, in Section 5.2 we deeply explain how our algorithms converge to the best solution of number of threads and boosting mode. Finally, in Sections 5.3 and 5.4 we compare our solutions against an exhaustive search to understand their learning costs and provide guidelines in how to better use both of them.

5.1 Energy-Delay Product Evaluation

Figure 8 presents the results for the entire benchmark set and the strategies (***Baseline***, ***Boost-Only***, ***DCT-Boost-Off***, ***DCT-Boost-OS***, ***Poseidon Fine-Grain***, and ***Poseidon Coarse-Grain***) along with their geometric mean (*Gmean*) considering the three multicore systems. Results are normalized to the baseline (represented by the black line), so values below 1.0 mean that a given strategy presents better results than the baseline.

PFG and PCG vs. Baseline. Both strategies show EDP improvements in most cases. The best results are for the *TPACF* and *HISTO* benchmarks with more

Figure 8 – EDP results of each search strategy normalized to the baseline configuration. Values lower than 1.0 mean that the strategies are better.



(a) AMD 24-Core system

(b) Intel 24-Core system

(c) Intel 88-Core system

Source – The author

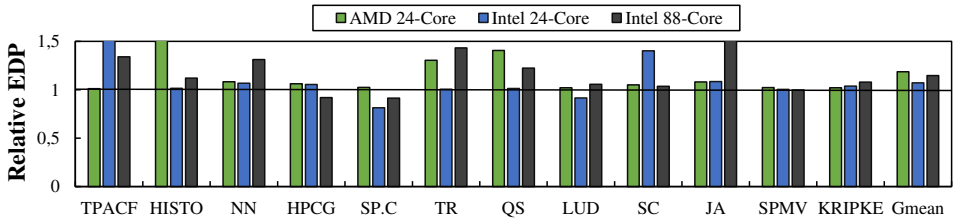
than 90% of EDP gains for PFG and PCG. These benchmarks have high synchronization levels, thus executing with the maximum number of threads (i.e., baseline configuration) severely impacts the EDP. On the other hand, in very specific scenarios where the ideal degree of TLP is with the maximum number of threads or a number near to it, both strategies are similar to the baseline. An example is for the SC benchmark on the Intel 88-core system, where PFG and PCG achieved, respectively, 15% and 11% higher EDP. That behavior can be explained because the DCT algorithm employed by PFG and PCG applies a bottom-up strategy, which means that it starts with a lower number of threads and increases until it reaches the optimal result (i.e., near the baseline), imposing an extra overhead in this type of application. If one considers the overall geometric mean, i.e., the entire benchmark set and all the multicore systems, PFG and PCG provided 53% and 57% better EDP results, respectively.

PFG and PCG vs. Boost-Only and DCT-Only. The execution of boosting *on* can provide better results for Intel processors than the baseline, however, is not capable of beating PFG and PCG results. This is caused because *Boost-Only* just optimizes one knob of the problem. Furthermore, the AMD processor does not benefit from boosting frequencies as the Intel processors due to the small range of frequencies the processor can operate (as discussed in Section 2.2). This is emphasized for the SP.C benchmark on the AMD 24-core system, where *Boost-Only* negatively impacted the EDP in 43%, while both PFG and PCG improved it by 80%.

On the other hand, *DCT-Only* solves another knob of the optimization problem and shows similar behavior. That is, this configuration brings improvements when compared with the baseline, but does reach the same level of improvements of our strategies. Hence, by analyzing the scalability of each region and choosing the number of optimal threads, and activating boosting in specific phases, our strategies could provide better optimization. In the overall geometric mean, PFG and PCG improve the EDP by 51.9% and 56.4% to the *Boost-Only*, respectively. When comparing *DCT-Only*, the EDP gains are 21.5% and 29%.

PFG and PCG vs. DCT-Boost-OS. Because PFG and PCG can tune both the number of threads and boosting mode according to the characteristics of each application's region at run-time, in most cases, it presents better results than only applying DCT and leaving the boosting control to the Linux OS. As we are using the ondemand governor, our strategies are expected to bring better results on the AMD 24-Core than *DCT-Boost-OS*. This happens because this DVFS governor is not capable of efficiently manage the AMD Turbo Core to provide the best EDP due to the low quantity of available frequencies (COSTA et al., 2020). When considering the overall geometric mean of the entire benchmark set and processors, PFG and PCG can provide, respectively, 20.2% and 27.8% better EDP when compared with *DCT-Boost-OS*. This result reinforces our state in this work that unbalanced parallel applications can benefit from a fine or coarse

Figure 9 – PFG vs. PCG: EDP normalized to the PCG results, so values lower than 1.0 mean that PFG is better.



Source – The author

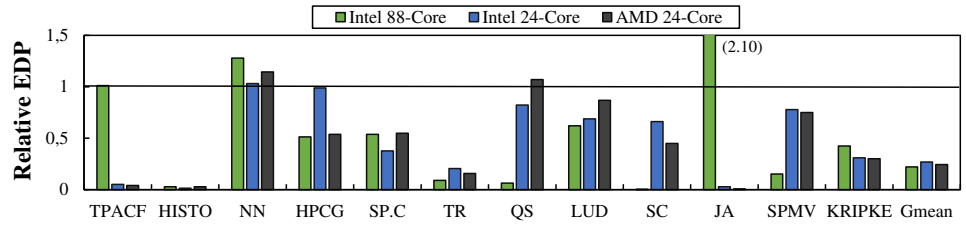
tune of the boosting techniques and the number of threads for each of its regions.

PFG vs. PCG. In Figure 9 we show the comparison between the two strategies developed in this paper. It presents the EDP of all benchmark sets along with the geometric mean normalized to the PCG strategy, which means that values below 1.0 are better for PFG. Changing the state of boosting at run-time can induce overheads since it is controlled by using system calls. Because the PFG strategy searches for an ideal configuration at fine-grain, i.e., controls the boosting status for each region, it achieved better EDP in applications with a low degree of TLP and some degree of variability in the workload between parallel and sequential regions (SP.C and LUD for the Intel 24-core and HPCG and SP.C for the Intel 88-core system). The best result for PFG is for the SP.C benchmark execution on the Intel 24-core system where it shows 18.5% lower EDP than PCG. On the other hand, the worst case of PFG is for the TPACF benchmark on the Intel 24-core system, where the EDP is 1.65 times higher than the PCG strategy. This is because the application presents sequential regions that are too short and then, the cost for controlling the boosting mode at fine-grain strongly impacts the application EDP. In general, the PFG strategy pays off in specific applications. When considering the geometric mean of the entire benchmark set, PCG was capable of achieving 18.6%, 7.1%, and 14.6% better EDP results on the AMD 24-core, Intel 24-core, Intel 88-core, respectively.

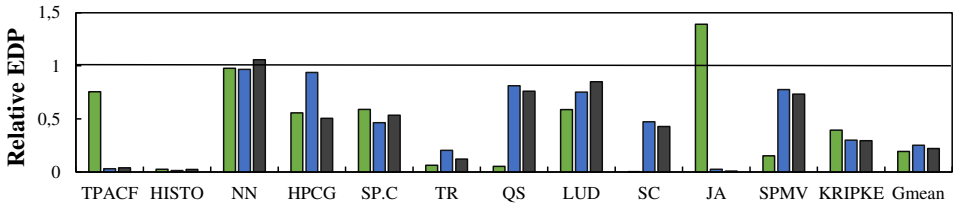
PFG and PCG vs. Varuna-PM. In Figure 10 we show the comparison between PFG (Figure 10a) and PCG (Figure 10b) against *Varuna-PM*. The results are normalized by Varuna and values lower than 1.0 mean that our strategies are better. In general, both strategies outperform *Varuna-PM* in most of the cases, achieving overall EDP improvements of 75.6% on PFG and 77.9% on PCG. In specific cases, where the application has high TLP levels (i.e., close to the maximum) and space of exploration is higher (e.g., JA in Intel 88-Core), both strategies increased the total EDP.

The main reasons for both strategies provide better results, in general, is how *Varuna-PM* performs its optimization. It creates as many threads as possible (we executed each application with 1566 threads – numbers taken from (SRIDHARAN; GUPTA; SOHI, 2014)). Hence, given that most of the applications present a limited degree of TLP (Figure

Figure 10 – EDP normalized to Varuna-PM. Values lower than 1.0 mean that our strategies are better.



(a) PFG vs. Varuna-PM



(b) PCG vs. Varuna-PM

Source – The author

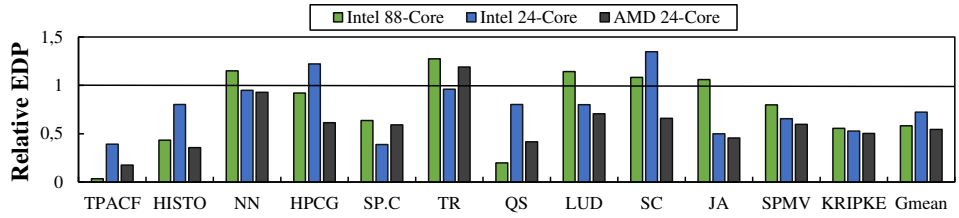
7), executing them with a higher number of threads will only rise the energy consumption and degrade performance. It is important to emphasize that these results do not consider the improvements provided by the analytical engine and the manager system of Varuna. However, even if it could improve performance by 15% and reduce energy by 31% (values taken from (SRIDHARAN; GUPTA; SOHI, 2014)), it would not be enough to provide the same EDP levels as our strategies.

PFG and PCG vs. Core Packing. Figure 11 shows the comparison between PFG (Figure 11a) and PCG (Figure 11b) against *Core Packing*. The results are normalized by Core Packing and values lower than 1.0 mean that our strategies are better. The results are similar to the comparison against *Varuna-PM*. That is in most cases PFG and PCG provide better EDP results. This happens because *Core Packing* decreases the number of active threads to increase the boosting frequency levels when the application presents a low degree of TLP to speed up the parallel regions. Therefore, it does not consider the workload behavior (CPU or Memory), nor the sequential regions leading to a non-optimal solution in most cases, i.e., it increases energy with no performance gains. In the geometric mean scenario, PFG and PCG provide 38.8% and 44.7% better EDP, respectively.

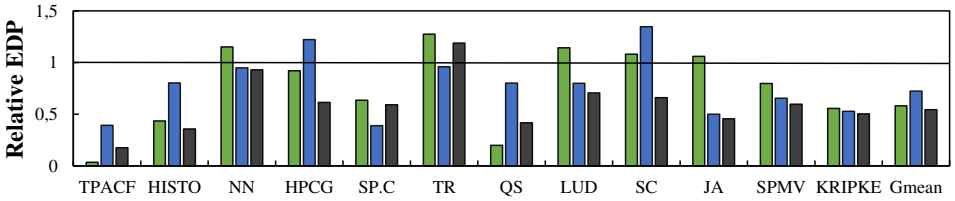
5.2 Convergence of the Algorithms

In this section, we discuss how the learning algorithms implemented in PFG and PCG converge to the ideal configuration for each region of a given application. To evaluate the quality of this convergence, we compare the results found by each one of the strategies to the results of the exhaustive search, considering all benchmarks. As discussed next,

Figure 11 – EDP normalized to Core Packing. Values lower than 1.0 mean that our strategies are better.



(a) Poseidon Fine-grained vs. Core Packing



(b) Poseidon Coarse-grained vs. Core Packing

Source – The author

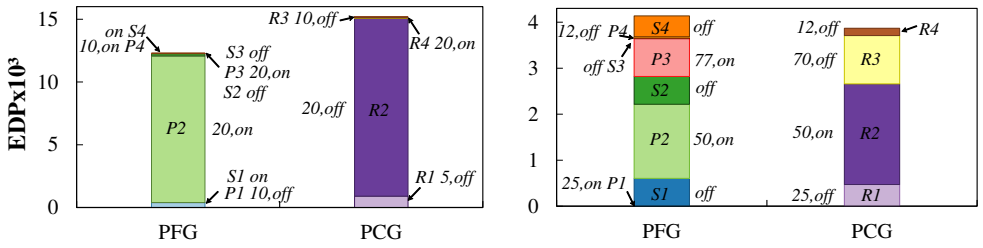
the two strategies proposed in this work can either find the ideal degree of TLP and the correct boosting mode to improve EDP or get close to it most times. To better understand how PFG and PCG conversion, two representative benchmarks were selected: HPCG and Kripke (Figure 12)

Each plot shows the EDP of each strategy on the Intel Xeon 88-core system and is organized as follows: for the PFG strategy, each part of the bar represents parallel (P1, P2, P3, and P4) and sequential (S1, S2, S3, and S4) regions; on the other hand, for the PCG strategy, each part of the bar represents the regions of interest (R1, R2, R3, and R4), as discussed in Section 4.2. Furthermore, we show in the side of each bar the result found by each strategy as follows: <#threads, boosting status> in parallel regions and regions of interest; and < boosting status > in sequential regions.

We start by discussing the behavior for the HPCG benchmark (Fig. 12a), where the PFG strategy achieved better EDP than the PCG. In this scenario, the most significant cases happen for the regions P2 and S2, which correspond to the R2 region from PCG. As P2 has a CPU-intensive behavior, the boosting feature is activated to further the EDP. Differently, S2 is memory-intensive, so boosting is set to off. Therefore, as P2 and S2 have distinct CPU workloads, the fine-grain strategy brings EDP improvements to the entire application. On the other hand, PCG is not able to understand the behavior of these individual regions. Therefore it tries to find the best boosting status that provides the best EDP for the combination of both regions, which penalizes the execution of the parallel region by executing it with boosting set to off.

Now let us discuss a scenario where the PCG strategy can deliver better EDP. It happens for the applications where there is no significant difference in the CPU workload

Figure 12 – Convergence of PFG and PCG on the Intel 88-core.



(a) HPCG benchmark (b) Kripke benchmark

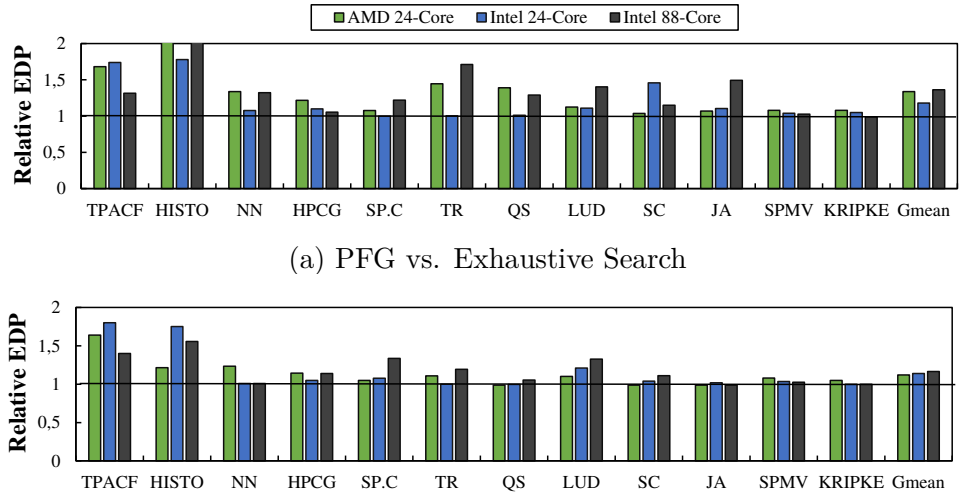
Source – The author

between parallel and sequential regions. We show this scenario Figure 12b, where the most significant EDP improvements occur in region 2 (R2) that correspond to P2 and S2 from PFG. In this region, PCG activates boosting to further accelerate the execution, as it is CPU-intensive. The same behavior does not happen for the PFG strategy because it has converged to a solution with boosting set to off for the S2 region. There are also scenarios where the two optimization strategies presented similar results: TPACF, SP.C, LUD, SPMV, and KRIPKE for the AMD system; HISTO, TR, QS, and SPMV for the Intel 24-core system; and SPMV for the Intel 88-core system. This behavior happens because such applications present similar CPU workloads between the parallel and sequential regions.

5.3 Learning Costs

Because PFG and PCG learn the configuration of TLP degree and boosting frequency at run-time, there is an overhead imposed by the learning algorithm. It can be

Figure 13 – PFG and PCG vs. best result found by the exhaust. search. Lower is better for our strategies.



(a) PFG vs. Exhaustive Search

(b) PCG vs. Exhaustive Search

Source – The author

measured by the difference between Poseidon and the best result found by the exhaustive search (explained in Section 4.5.2). The overhead caused by both strategies is originated from two cases: (*i*) the execution of a given region with a configuration that is not ideal, while searching for the ideal solution; (*ii*) the learning algorithm itself (including its system calls).

We show in Figure 13 the efficiency of our algorithm. As one can observe, in most cases, the learning overhead for both strategies is almost negligible. The highest learning overhead of the strategies is achieved in application in the following scenarios: (*i*) the degree of TLP is lower and the application scalability is limited by the amount of data-synchronization, e.g., TPACF and HISTO. Therefore, running the region with a non-optimal number of threads while converging to the ideal result increases the total EDP (SULEMAN; QURESHI; PATT, 2008; Lorenzon et al., 2019); and (*ii*) the best result is achieved with the highest degree of TLP or a number near it, e.g., LUD. In those cases, the strategies have the learning overhead to converge to a configuration that is already used by the baseline.

5.4 Guidelines

Based on the experiments shown in this chapter, to provide guidelines to the software designers in how to better use our strategies, we now emphasize the main insights of our results: (*i*) PFG and PCG present better EDP results than all the approaches compared in this work, and they present an acceptable learning overhead since they perform at run-time. (*ii*) The obtained results show that optimizing only one part of the problem can not be optimal, which reinforces the need for applying both DCT and CPU boosting frequency optimization combined. (*iii*) For applications with a high variation in the CPU workload between parallel and sequential regions, PFG strategy (fine-grain) is more suitable. (*iv*) In applications where the CPU workload is very similar between the regions (parallel and sequential), applying the PCG optimization (coarse-grain) delivers better EDP results.

6 CONCLUSION AND FUTURE WORK

In this work, two strategies to automatically and effortlessly improve OpenMP applications execution through TLP and boosting techniques optimization were proposed and validated considering well-known techniques and scenarios.

To achieve the main goal of this work, we first characterized the OpenMP applications concerning memory access behavior and thread-level parallelism. Then we studied how the different modern boosting techniques work in the Linux environment. Based on that, we developed our strategies that rely on two different design choices: **PFG**, optimizes each region of a given application parallel application, individually; and **PCG** focus on optimizing each combination of regions (i.e., parallel plus sequential). By just setting an environment variable, the user can effortlessly use both strategies. To provide maximum transparency and binary compatibility, we integrated both strategies into the OpenMP libgomp library. Since it is dynamically linked into OpenMP code, there is no need for code recompilation, OS installation, or any kind of special annotation.

The strategies were validated by executing twelve well-known benchmarks on three different multicore machines from AMD and Intel. Using the DVFS set to *ondemand*, we compared our strategies against the following scenarios: **Baseline**: maximum number of threads and boosting techniques disabled; **Boost-Only**: same configuration of baseline, but with boosting enabled; **DCT-Boost-Off**: our learning algorithm to find the number of threads, but with boosting disabled during the entire execution; **DCT-Boost-OS**: the same as the previous one, but with boosting enabled. We show that PFG improves the EDP in 53%, 51.9%, 21.5%, and 20.2% against the Baseline, Boost-Only, DCT-Only, and DCT-Boost-OS, respectively. On the other hand, in the same scenario PCG provides, respectively, 57%, 56.4%, 29%, and 27.8% EDP improvements.

By comparing the two techniques against each other, we find that applying DCT together with boosting management at a fine grain (PFG) provides better results in applications with low-TLP and high variation in the CPU workload between different regions. Differently, the coarse grain strategy (PCG) is more suitable for applications with low variation in the CPU workload. We also compared our strategy with two state-of-art techniques **Varuna-PM** and **Core Packing**. In these scenarios, we showed gains of PFG and PGC improved the EDP on average in, respectively, 75.6% and 77.9% against Varuna-PM and 38.8%, and 44.7% when compared with Core Packing.

To measure the cost of our search algorithms to find the optimal or near-optimal configuration, we compared them to an **Exhaustive Search**. In this comparison, the overall average cost to optimize EDP on PFG is 28% while in PGC it is 14%.

6.1 Future Work

This section describes some promising future works related to the extension of our strategies.

Nowadays, DVFS is a commonly used approach to optimize energy consumption for a given application. By using DVFS the supply voltage of the CPU is dynamically decreased through frequency control, hence improving the energy consumption. However, this only applies to the processor's cores. Recent Intel microarchitectures dissociated the cores frequency (computation units and L1/L2 caches) from the uncore units (last-level cache, memory-controllers, and QuickPath Interconnect), which allows decoupled frequency management (ANDRÉ et al., 2020).

The Uncore Frequency Scaling (UFS) is a technique that permits the processor to dynamically control the frequency of uncore units independently of the core frequency. This technique can be used similar to DVFS, i.e., reducing the operating frequency of the memory and as consequence the energy consumption. However, determining the ideal frequency and voltage for a given phase of a parallel application is not a straightforward task. Applying a severe cut in the uncore frequency levels on a memory-bound phase of the application will greatly impact the latency for accessing the memory. On the other hand, a naive increase in the memory frequencies on a CPU-intensive phase will increase the energy consumption with no improvements in the performance. To tackle this problem, as future work, we intend to include the optimization for the UFS in combination with the strategies proposed in this work.

6.2 List of Publications

Design space exploration of boosting techniques and TLP
Conference Papers:

- Avaliando o Impacto do AMD Turbo Core no Consumo de Energia e Desempenho de Aplicações Paralelas. In: Escola Regional de Alto Desempenho da Região Sul (ERAD-RS) - (MARQUES et al., 2019a)

- The Impact of Turbo Frequency on the Energy, Performance, and Aging of Parallel Applications. In: IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC) - (MARQUES et al., 2019b)

- Optimizing Parallel Applications via Dynamic Concurrency Throttling and Turbo Boosting. In: 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2021) - (MARQUES et al., 2021a)

- Synergically Rebalancing Parallel Execution via DCT and Turbo Boosting. In: Design Automation Conference (DAC 2021) - (MARQUES et al., 2021b)

Joint Cooperation:
Conference Papers:

- Comparação de Desempenho e Consumo de Energia de Aplicações Paralelizadas com OpenMP taskloop e parallel for. In: Escola Regional de Alto Desempenho da Região Sul (ERAD-RS) - (PEREIRA et al., 2020)

- PampaFreq: Otimizando o EDP de Aplicações Paralelas em Processadores AMD. In: Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD) - (COSTA et al., 2020)

- EDP Optimization of Parallel Applications via CPU Frequency Scaling on AMD Processors. In: IEEE Latin American Symposium on Circuits and Systems (LASCAS) - (COSTA et al., 2021)

BIBLIOGRAPHY

ALESSI, F. et al. Application-level energy awareness for openmp. In: SPRINGER. **Int. Workshop on OpenMP**. [S.l.], 2015. p. 219–232. Cited in page 33.

ALJUHNI, A. et al. Towards understanding application performance and system behavior with the full dynticks feature. In: IEEE. **2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)**. [S.l.], 2018. p. 394–401. Cited in page 29.

ANDRÉ, E. et al. Duf: Dynamic uncore frequency scaling to reduce power consumption. 2020. Cited in page 54.

BAILEY, D. H. et al. The nas parallel benchmarks—summary and preliminary results. In: **ACM/IEEE Conf. on Supercomputing**. NY, USA: [s.n.], 1991. p. 158–165. ISBN 0-89791-459-7. Cited 2 times in the pages 26 and 42.

BECKINGSALE, D. A. et al. Raja: Portable performance for large-scale scientific applications. In: IEEE. **2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)**. [S.l.], 2019. p. 71–81. Cited in page 27.

BIENIA, C. et al. The parsec benchmark suite: Characterization and architectural implications. In: **Proceedings of the 17th international conference on Parallel architectures and compilation techniques**. [S.l.: s.n.], 2008. p. 72–81. Cited in page 27.

BLAKE, G. et al. Evolution of thread-level parallelism in desktop applications. **SIGARCH Comput. Archit. News**, ACM, NY, USA, v. 38, n. 3, p. 302–313, 2010. ISSN 0163-5964. Cited in page 42.

CARDOSO, J.; COUTINHO, J.; DINIZ, P. Chapter 2—high-performance embedded computing. **Embedded Computing for High Performance; Cardoso, JM, Coutinho, JGF, Diniz, PC, Eds**, p. 17–56. Cited in page 28.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP: Portable Shared Memory Parallel Programming**. [S.l.]: The MIT Press, 2007. ISBN 0262533022, 9780262533027. Cited 2 times in the pages 27 and 43.

CHARLES, J. et al. Evaluation of the intel® core™ i7 turbo boost feature. In: IEEE. **IEEE ISWC**. [S.l.], 2009. p. 188–197. Cited 2 times in the pages 21 and 28.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: IEEE. **2009 IEEE international symposium on workload characterization (IISWC)**. [S.l.], 2009. p. 44–54. Cited 2 times in the pages 27 and 42.

CIORBA, F. M. et al. Sph-exa: Optimizing smoothed particle hydrodynamics for exascale computing. In: THE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING (ISC). **Project Poster at the 34th International Conference on High Performance Computing (ISC)**. [S.l.], 2019. Cited in page 26.

COSTA, M. et al. Pampafreq: Otimizando o edp de aplicações paralelas em processadores amd. In: SBC. **Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho**. [S.l.], 2020. p. 49–60. Cited in page 46.

CURTIS-MAURY, M. et al. Prediction-based power-performance adaptation of multithreaded scientific codes. **IEEE TPDS**, IEEE, n. 10, p. 1396–1410, 2008. Cited in page 32.

DONGARRA, J.; HEROUX, M. A.; LUSZCZEK, P. Hpcg benchmark: A new metric for ranking high performance computing systems. **Knoxville, Tennessee**, 2015. Cited in page 42.

HäHNEL, M. et al. Measuring energy consumption for short code paths using rapl. **SIGMETRICS Perform. Eval. Rev.**, ACM, NY, USA, v. 40, n. 3, p. 13–17, 2012. ISSN 0163-5999. Cited in page 37.

HAJ-YAHYA, J. et al. Power management of modern processors. In: _____. **Energy Efficient High Performance Processors: Recent Approaches for Designing Green High Performance Computing**. Singapore: Springer Singapore, 2018. p. 1–55. Cited in page 29.

JU, T. et al. Thread count prediction model: Dynamically adjusting threads for heterogeneous many-core systems. In: IEEE. **IEEE ICPADS**. [S.l.], 2015. p. 456–464. Cited in page 33.

JUAN, D.-C. et al. Learning the optimal operating point for many-core systems with extended range voltage/frequency scaling. In: IEEE PRESS. **CODES+ ISSS**. [S.l.], 2013. p. 8. Cited in page 31.

KANNADASAN, R. et al. High performance parallel computing with cloud technologies. **Procedia computer science**, Elsevier, v. 132, p. 518–524, 2018. Cited in page 26.

KHDR, H.; AMROUCH, H.; HENKEL, J. Aging-aware boosting. **IEEE Transactions on Computers**, IEEE, v. 67, n. 9, p. 1217–1230, 2018. Cited in page 32.

KHOKHRIAKOV, S.; MANUMACHU, R. R.; LASTOVETSKY, A. Multicore processor computing is not energy proportional: An opportunity for bi-objective optimization for energy and performance. **Applied Energy**, Elsevier, v. 268, p. 114957, 2020. Cited in page 33.

KONDGULI, S.; HUANG, M. A case for a more effective, power-efficient turbo boosting. **ACM TACO**, ACM New York, NY, USA, v. 15, n. 1, p. 1–22, 2018. Cited in page 32.

LO, D.; KOZYRAKIS, C. Dynamic management of turbomode in modern multi-core chips. In: IEEE. **IEEE HPCA**. [S.l.], 2014. p. 603–613. Cited in page 31.

Lorenzon, A. F. et al. Aurora: Seamless optimization of openmp applications. **IEEE Transactions on Parallel and Distributed Systems**, v. 30, n. 5, p. 1007–1021, 2019. Cited 3 times in the pages 21, 33, and 51.

Lorenzon, A. F.; Dellagostin Souza, J.; Schneider Beck, A. C. Laant: A library to automatically optimize edp for openmp applications. In: **Design, Automation Test in Europe Conference Exhibition (DATE), 2017**. [S.l.: s.n.], 2017. p. 1229–1232. Cited in page 33.

MAITI, S.; KAPADIA, N.; PASRICHA, S. Process variation aware dynamic power management in multicore systems with extended range voltage/frequency scaling. In: IEEE. **IEEE MWSCAS**. [S.l.], 2015. p. 1–4. Cited in page 31.

MARIANI, G. et al. Classification of thread profiles for scaling application behavior. **Parallel Computing**, Elsevier, v. 66, p. 1–21, 2017. Cited in page 33.

MICHAUD, P. Exploiting thermal transients with deterministic turbo clock frequency. **IEEE Computer Architecture Letters**, IEEE, v. 19, n. 1, p. 43–46, 2020. Cited in page 32.

PAGANI, S. et al. seboost: Selective boosting for heterogeneous manycores. In: IEEE. **CODES+ ISSS**. [S.l.], 2015. p. 104–113. Cited 2 times in the pages 34 and 35.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design ARM Edition: The Hardware Software Interface**. [S.l.]: Morgan kaufmann, 2016. Cited 2 times in the pages 25 and 26.

PAUDEL, J.; AMARAL, J. N. Using the cowichan problems to investigate the programmability of x10 programming system. In: **ACM SIGPLAN X10 Workshop**. New York, NY, USA: ACM, 2011. (X10 '11). ISBN 9781450307703. Cited in page 42.

PUSUKURI, K. K.; GUPTA, R.; BHUYAN, L. N. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In: IEEE. **IEEE ISWC**. [S.l.], 2011. p. 116–125. Cited in page 32.

QUINN, M. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Higher Education, 2004. ISBN 9780072822564. Cited in page 42.

RAGHAVAN, A. et al. Computational sprinting. In: IEEE. **IEEE HPCA**. [S.l.], 2012. p. 1–12. Cited 2 times in the pages 34 and 35.

RICHINS, D. et al. Amdahl's law in big data analytics: Alive and kicking in tpcx-bb (bigbench). In: IEEE. **IEEE HPCA**. [S.l.], 2018. p. 630–642. Cited 3 times in the pages 34, 35, and 44.

RUGHETTI, D. et al. Machine learning-based thread-parallelism regulation in software transactional memory. **Journal of Parallel and Distributed Computing**, Elsevier, v. 109, p. 208–229, 2017. Cited in page 33.

SEO, S.; JO, G.; LEE, J. Performance characterization of the nas parallel benchmarks in opencl. In: IEEE. **IEEE IISWC**. [S.l.], 2011. p. 137–148. Cited in page 27.

SEO, S.; JO, G.; LEE, J. Performance characterization of the nas parallel benchmarks in opencl. In: **IEEE ISWC**. [S.l.: s.n.], 2011. p. 137–148. Cited in page 42.

SRIDHARAN, S.; GUPTA, G.; SOHI, G. S. Holistic run-time parallelism management for time and energy efficiency. In: **Int. ACM Conf. on Supercomputing**. [S.l.: s.n.], 2013. p. 337–348. Cited in page 32.

SRIDHARAN, S.; GUPTA, G.; SOHI, G. S. Adaptive, efficient, parallel execution of parallel programs. In: **ACM SIGPLAN Conf. on Programming Language Design and Implementation**. [S.l.: s.n.], 2014. p. 169–180. Cited 4 times in the pages 33, 43, 47, and 48.

STRATTON, J. A. et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. **Center for Reliable and High-Performance Computing**, v. 127, 2012. Cited 2 times in the pages 27 and 42.

SUEUR, E. L.; HEISER, G. Dynamic voltage and frequency scaling: The laws of diminishing returns. In: **ICPACS**. [S.l.: s.n.], 2010. p. 1–8. Cited 2 times in the pages 21 and 29.

SULEMAN, M. A.; QURESHI, M. K.; PATT, Y. N. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmpsf. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 43, n. 3, p. 277–286, 2008. Cited 3 times in the pages 21, 32, and 51.

Verner, U.; Mendelson, A.; Schuster, A. Extending amdahl's law for multicores with turbo boost. **IEEE Computer Architecture Letters**, v. 16, n. 1, p. 30–33, 2017. Cited in page 31.

WAMHOFF, J.-T.; DIESTELHORST, S.; FETZER, C. The turbo diaries: Application-controlled frequency scaling explained. In: . [S.l.: s.n.]. Cited 3 times in the pages 21, 29, and 31.

INDEX