AtTune: A Heuristic based Framework for Parallel Applications Autotuning

Hiago Mayk G. de A. Rocha[1], Janaina Schwarzrock[1], Monica M. Pereira[2],
Lucas M. Schnorr[1], Philippe Navaux[1], Arthur F. Lorenzon[3], Antonio Carlos S. Beck[1]
[1]Institute of Informatics – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
[2]Department of Informatics and Applied Mathematics – Federal University of Rio Grande do Norte – Natal, Brazil
[3]Laboratory of Optimization Systems – Federal University of Pampa – Alegrete, Brazil
Email: {hmgarocha, jschwarzrock, schnorr, navaux, caco}@inf.ufrgs.br,
monicapereira@dimap.ufrn.br, aflorenzon@unipampa.edu.br

***Abstract*—Several aspects limit the scalability of parallel applications, e.g., off-chip bus saturation and data synchronization. Moreover, the high cost of cooling HPC systems, which can outweigh the cost of developing the system itself, has pushed the parallel application's execution to another level of requirements, in terms of performance and energy. In this work, we propose AtTune: a heuristic-based framework for tuning the number of processes/threads and CPU frequency to optimize the parallel applications' execution. AtTune is transparent for the user, independent of the input size, and it optimizes for different parallel programming models. We evaluated our proposed solution considering five well-known kernels implemented in MPI and OpenMP. Experimental results on two real multi-core systems showed that AtTune improves up to 36%, 11%, and 32% the energy efficiency, performance, and Energy-Delay Product, respectively.**

***Index Terms*—Automatic Tuning, Transparent Optimization, Thread Throttling, DVFS, Thread-Level Parallelism, Energy-efficiency**

I. Introduction

The increasing number of processing resources in High-Performance Computing (HPC) systems and the ability to parallelize costly computations have made possible advancements in diverse areas, such as medicine, bioinformatics, artificial intelligence, and weather and stock exchange forecasts. However, the parallel environments and applications growing complexity have been pushing their executions to another level of performance and energy requirements [1], [2].

Several aspects can limit the scalability of parallel execution, such as off-chip bus saturation and data synchronization. It means that not always using the maximum number of processes or threads (processes/threads) will deliver the best performance [3]. In the same way that HPC applications require performance, they also need to save energy in their executions. Many times, the costs related to energy supply and cooling may outweigh the costs related to the hardware in HPC/Cloud systems [2]. Hence, an approach to achieve even higher performance over the constraint of power dissipation and energy consumption is required [4].

A natural thought is giving to the users the option to adjust these parallel applications' non-functional requirements accords on their needs, e.g., optimizing for energy, performance, or the trade-off between both, given by the energy-delay product (EDP). This optimization is possible by adjusting the number of processes/threads and applying the Dynamic Voltage and Frequency Scaling (DVFS) technique to tune the CPU frequency levels [4]. Adapting the number of processes/threads may minimize energy consumption by reducing the number of hardware components in the computation (e.g., functional units, cores, and cache memories). On the other hand, the DVFS technique leverages time intervals of process inactivity, commonly caused by I/O operations or memory requests. Since the input voltage has quadratic influence in dynamic power, a system supporting the DVFS technique can take advantage of the CPU idleness (e.g., data-synchronization or communication among processes/threads) to achieve cubic power reduction [5].

In this work, we propose AtTune: a new framework for tuning the number of processes/threads and CPU frequency levels of parallel applications implemented with different parallel programming models that run over HPC systems environments. AtTune uses a three-phase heuristic algorithm to reduce the search space by identifying the applications' scalability and finding an optimized solution by applying a binary search-based strategy. We evaluate our framework with five kernels from the NAS Parallel Benchmark Suite [6]. Results on two real multi-core systems show that AtTune achieves significant energy saving (up to 36%) and improves up to 11% and 32% the performance and EDP when it optimizes for each of them.

II. Related Work

Several works have already proposed to optimize either the number of processes/threads [3] or CPU frequency levels [7]. As AtTune optimizes both knobs, we focus on the works that simultaneously optimize both processes/threads and CPU frequency levels.

Li and Martinez [8] propose a heuristic to optimize OpenMP applications. Their solution monitors the application executions and adjusts the number of threads and frequency levels by running different optimization mechanisms.

An algorithm to optimize hybrid MPI+OpenMP applications is proposed by Li et al. [9]. According to this approach, the user instruments the source code with a system call for each parallel region and MPI operation. The proposal uses a prediction model to predict the best configuration at runtime.

LIMO is a dynamic system that monitors the application execution and adjusts the TLP degree and CPU frequency levels

[10]. The approach monitors the application at runtime and reduces the power consumption by disabling cores whenever threads are not making progress and adjusting the frequency levels of active cores.

Alessi et al. [11] propose *OpenMPE*, an OpenMP extension designed for energy management. Using the *OpenMPE*, the programmer instruments the source code inserting directives, indicating parts of the code that can save energy.

In [12], Marathe et al. present *Conductor*, a system to optimize the number of threads and frequency levels for hybrid MPI+OpenMP applications. *Conductor* applies a local search algorithm to find the best solution to reduce power consumption with minimal execution time degradation.

Oliveira et al. [5] propose a framework that uses a Genetic Algorithm to optimize the EDP. This framework modifies the OpenMP runtime library. Also, the approach requires to run the optimization process whenever the application's input size changes.

A. Our contributions

AtTune has the following advantages over the above-presented works: *(i) No programmer influence:* our approach does not require any modification on both the application's source code or runtime libraries. Only the work of [10] provides that; *(ii) Binary compatibility:* AtTune does not require recompiling the application, which makes our solution to be applicable for legacy applications as well. Only the work of [5] provides that; *(iii) Generalization:* our approach is independent of the applications' input size. Moreover, different from some previous work that focuses only on OpenMP or Pthreads, excluding the works of [9] and [12], AtTune can be applied to different parallel programming models and programming languages. Besides, it *does not require any hardware modification*.

III. AtTune Framework

In this section, we describe how AtTune works and explain the proposed heuristic that supports it. We also analyze the time complexity of the heuristic.

A. Overview

AtTune framework consists of an offline method for finding an optimized solution for the degree of Thread (also Process) Level Parallelism (TLP) and CPU frequency level for a given parallel application. It allows the optimization for different metrics, e.g., performance, energy consumption, or EDP (we refer to this as the objective function). AtTune works as follow: (*i*) the user first provides the inputs (a binary file, the hardware configurations, a set of representative input sizes, and the objective function). Note that the user can specify a specific part of the entire target system that AtTune will perform its optimization. Otherwise, AtTune automatically considers all system's cores and CPU frequency levels. (*ii*) Then, the AtTune framework applies a search for the best configuration of the TLP degree and CPU frequency level to optimize the objective function (explained in Section III-B); and (*iii*) the output of the AtTune framework is the best configuration found for the given input set.

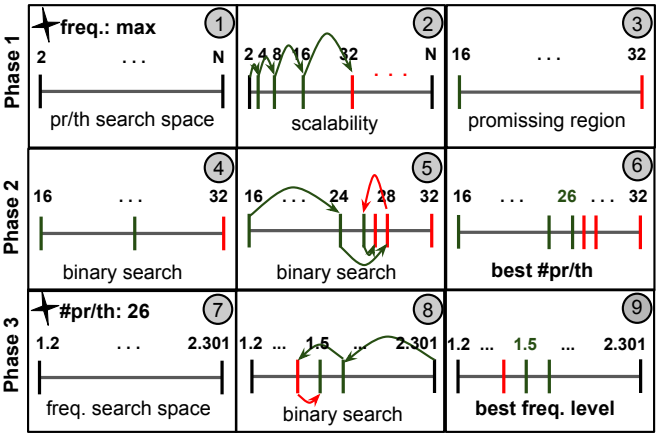


Fig. 1. Heuristic's general idea.

For the AtTune's search part, we proposed a heuristic algorithm. However, this search part can be easily modified to support any other search strategies, e.g., Local Search, Genetic Algorithms, or other heuristics. The proposed heuristic performs a three-phase search (see section III-B). It searches for a given application's best solution by considering its execution with each input from the input set. AtTune performs the heuristic once, but each search's phase requires executing the application a different number of times, depending on the space exploration (i.e., number of available hardware threads, CPU frequency levels, and the size of the input set), as discussed in Section III-C. Next, we explain this main part of our framework, which is our proposed heuristic.

B. Heuristic

We present the algorithmic idea of our proposed heuristic in Figure 1. It consists of three phases:

- ***Phase1*** works in the processes/threads (pr/th) search space considering the maximum CPU frequency level (1). This phase tests the parallel application's scalability (2) and identifies a promising region (3), i.e., a part of the search space that has more potential to deliver optimized solutions;
- ***Phase2*** takes the promising region (4), yet considering the maximum CPU frequency level, and applies a binary search based algorithm (5) to find the best number of processes/threads (6);
- ***Phase3*** works in the frequency search space (7). This phase takes the best number of threads found in the previous one and applies another binary search variant (8) to find the best CPU frequency level (9).

In the example of Fig. 1, the optimization search found that the best solution is the execution with 26 processes/threads and a frequency level of 1.5GHz. We present a detailed view of how the heuristic works in Algorithms 1, 2, 3, and 4.

Algorithm 1 presents the **Main flow** of our heuristic. It takes as input the metric to be optimized m, the application's executable e, a list containing the applications' input set P (a representative input set of the application to be optimized), the considered number of system's cores C, and the frequency levels F. The heuristic performs the three phases in sequence.

Algorithm 1 *Heuristic*

Input: $m \leftarrow$ Optimization metric,
$e \leftarrow$ Application's executable,
$P \leftarrow$ InputSet$\{I_1, I_2, ..., I_k\}$,
$C \leftarrow$ Number of cores in the system,
$F \leftarrow$ Freq$\{F_{min}, ..., F_{max}\}$.
Output: The best values of $t_{best} \in [2, |P|]$ and $f_{best} \in F$.
1: $l_{inf}, l_{sup}, m_{best} \leftarrow Phase1(m, e, P, C, F[MAX])$
2: $t_{best} \leftarrow Phase2(m, e, P, C, F[MAX], l_{inf}, l_{sup}, m_{best})$
3: $f_{best} \leftarrow Phase3(m, e, P, C, F, t_{best})$
4: **return** t_{best},f_{best}

Phase 1 and *Phase 2* are called considering the maximum frequency value $F[MAX]$. These phases optimize the number of processes/threads, but they ensure the highest performance potential by considering the highest frequency level. When optimized for energy, a variant that considers the minimum frequency level can be applied. For *Phase 3*, it optimizes the CPU operating frequency level by executing the application with the best number of processes/threads found by the previous phases. As a result, the algorithm returns the best solution regarding the objective function m.

Phase 1 is described in Algorithm 2. Besides the inputs m, e, P, and C, it also receives the maximum frequency level of f. At the beginning, it is initialized the previous t_{prev} and current t_{curr} number of threads variables, and their respective objective function values ($t_{prev}.m$ and $t_{curr}.m$). While the number of evaluated threads is lower than the considered number of cores, i.e., physical and logical cores, $t_{curr} < |C|$ and there is improvement in the objective function $t_{prev}.m > t_{curr}.m$, the following steps are performed: the number of processes/threads t_{curr} increases to the next power of 2 regarding its previous value t_{prev}; the program is executed through all the input set elements $p \in P$ i.e., each of them with different sizes, and the objective function values e.g., energy, time, or EDP, are collected m_{acc}; the collected objective function values are averaged, which represents the objective function value of current evaluated number of processes/threads $t_{curr}.m$. In the end, *Phase 1* returns a promising region with the minimum and maximum number of processes/threads indicated by t_{prev} and t_{curr}, respectively. Also, it results in the best objective function value found m_{best}.

Phase 2 is presented in Algorithm 3. Besides the inputs m, e, P, C, and f, it also receives the promising region with the minimum and the maximum number of processes/threads to be considered l_{inf} and l_{sup}, and the best objective function value found so far m_{best}. The procedure first checks if $l_{sup}.m = m_{best}$. If it is the case, the algorithm stalls because the application is scalable, then the maximal number of processes/threads will deliver the best objective function value. Otherwise, while there exists elements not evaluated in the promising region $l_{sup} - l_{inf} > 1$, the following steps are performed: it is identified the middle element in the region $(l_{sup} + l_{inf})/2$; the program is executed through all the input set elements $p \in P$ and the objective function values are collected m_{acc} and averaged m_{ave}, indicating the objective function value of current evaluated number of processes/threads; the m_{ave} is compared with the best objective function value found m_{best}, and if m_{ave} is lower than that, it means we should consider the superior part of the searching region $[med, l_{sup}]$. Otherwise, we should consider the inferior part $[l_{inf}, med]$. The procedure returns the number of threads med that results in the best objective function value being optimized m.

Phase 3 is detailed in Algorithm 4. Besides the inputs m, e, P, C, and F it also has the best number of processes/threads t_{best}. Initially, it is declared the minimum min and maximum max available frequency levels of the target system. The med variable is assigned as the maximal frequency because

Algorithm 2 *Phase1*

Input: $m \leftarrow$ Optimization metric
$e \leftarrow$ Application's executable,
$P \leftarrow$ InputSet$\{I_1, I_2, ..., I_k\}$,
$C \leftarrow$ Number of cores in the system,
$f \leftarrow$ Freq$\{F_{max}\}$.
Output: Range of promising proc/thread values [l_{inf}, l_{sup}] and the best objective function value found so far m_{best}.
1: t_{prev}, $t_{curr} \leftarrow 1$
2: $t_{prev}.m \leftarrow infinity$
3: $t_{curr}.m \leftarrow 0$
4: **while** ($t_{curr} < |C|$ and $t_{prev}.m > t_{curr}.m$) **do**
5: $t_{prev} \leftarrow t_{curr}$
6: $t_{curr} \leftarrow Power2(t_{prev})$
7: $m_{acc} \leftarrow 0$
8: **for** $p \in P$ **do**
9: $m_{acc} \leftarrow m_{acc} + Program(m, e, p, t_{curr}, f)$
10: **end for**
11: $t_{prev}.m \leftarrow t_{curr}.m$
12: $t_{curr}.m \leftarrow m_{acc}/|P|$
13: **end while**
14: $m_{best} \leftarrow min(t_{prev}.m, t_{curr}.m)$
15: **return** t_{prev}, t_{curr}, m_{best}

Algorithm 3 *Phase2*

Input: $m \leftarrow$ Optimization metric,
$e \leftarrow$ Application's executable,
$P \leftarrow$ InputSet$\{I_1, I_2, ..., I_k\}$,
$C \leftarrow$ Number of cores in the system,
$f \leftarrow$ Freq$\{F_{max}\}$,
$l_{inf} \leftarrow$ Minimum number of threads,
$l_{sup} \leftarrow$ Maximum number of threads,
$m_{best} \leftarrow$ Best objective function value found so far.
Output: The best number of procs/threads med.
1: **if** ($l_{sup}.m = m_{best}$) **then**
2: **return** l_{sup}
3: **end if**
4: $med \leftarrow l_{sup}$
5: **while** ($l_{sup} - l_{inf} > 1$) **do**
6: $med \leftarrow (l_{sup} + l_{inf})/2$
7: $m_{acc} \leftarrow 0$
8: **for** $p \in P$ **do**
9: $m_{acc} \leftarrow m_{acc} + Program(m, e, p, med, f)$
10: **end for**
11: $m_{ave} \leftarrow m_{acc}/|P|$
12: **if** ($m_{ave} < m_{best}$) **then**
13: $m_{best} \leftarrow m_{ave}$
14: $l_{inf} \leftarrow med$
15: **else**
16: $l_{sup} \leftarrow med$
17: **end if**
18: **end while**
19: **return** med

Algorithm 4 *Phase3*

Input: $m \leftarrow$ Optimization metric,
$e \leftarrow$ Application's executable,
$P \leftarrow$ InputSet$\{I_1, I_2, ..., I_k\}$,
$C \leftarrow$ Number of cores in the system,
$F \leftarrow \{F_{min}, ..., F_{max}\}$,
$t_{best} \leftarrow$ Best number of procs/threads.
Output: The best frequency level f_{best}.

1: $min \leftarrow F[MIN]$
2: $max \leftarrow F[MAX]$
3: $med \leftarrow F[MAX]$
4: **while** $(max - min > 1)$ **do**
5: $med \leftarrow (max + min)/2$
6: $m_{acc} \leftarrow 0$
7: **for** $p \in P$ **do**
8: $m_{acc} \leftarrow m_{acc} + Program(m, e, p, t_{best}, med)$
9: **end for**
10: $m_{ave} \leftarrow m_{acc}/|P|$
11: **if** $(m_{ave} < t_{best}.m)$ **then**
12: $t_{best}.m \leftarrow m_{ave}$
13: $min \leftarrow med$
14: **else**
15: $max \leftarrow med$
16: **end if**
17: **end while**
18: **return** med

if the system has only one frequency level, no optimization is needed, i.e., in that case, the algorithm indicates the maximum frequency as the best. The program is executed through all the input set elements $p \in P$. The objective function values are collected m_{acc} and averaged m_{ave}. The heuristic compares m_{ave} with the objective function value of the best number of processes/threads found considering the maximal frequency level $t_{best}.m$. If m_{ave} is lower than that, it means we should consider the superior part of the searching region $[med, max]$. Otherwise, we should consider the inferior part $[min, med]$. The procedure returns the frequency level of med, resulting in the best objective function value m.

C. Time Complexity Analysis

Considering the tackled problem with the P application's inputs, the C system's cores, and F CPU frequency levels, an exhaustive search requires a cubic asymptotic time over the inputs sets, i.e., $O(|P| \times |C| \times |F|)$. The time complexity of our proposed heuristic depends on each phase: *Phase 1* is $O(log(|C|) \times P)$; *Phase 2* is $O(log(l_{sup} - l_{inf}) \times P)$; and *Phase 3* is $O(log(|F|) \times P)$. Therefore, putting it all together, the time complexity of our heuristic is given by: $O[(log(|C|) + log(l_{sup} - l_{inf})) \times P + log(|F|) \times P]$.

IV. Methodology

In this section, we describe the methodology used in our experiments. Specifically, we present the used benchmarks, system configurations, and framework implementation details.

A. Benchmarks

We used five kernels from the NAS Parallel Benchmark Suite version 3.4.1 implemented in both MPI and OpenMP [6]: Integer Sort (IS), Embarrassingly Parallel (EP), Conjugate Gradient (CG), Multi-Grid on a sequence of meshes (MG), and Discrete 3D fast Fourier Transform (FT). Most of them are in FORTRAN (only the IS is in C). We compile them with *GNU Fortran 8.3.0* and *GCC 8.3.0*, using the -O3 optimization flag. For the MPI implementations, we used the *OpenMPI 3.1.3* version. For evaluation, we considered the kernels' inputs of A, B, and C sizes.

TABLE I
SYSTEMS CHARACTERISTICS.

Processor	Intel Xeon CPU E5-2640 v2	AMD Ryzen 9 3900X
# Cores	2×16	12
# Threads	32	24
CPU Freq. Levels	1.2 - 2.001 GHz (10 levels)	2.2 - 3.8 GHz (3 Levels)
L1 cache	16×32KB	12x32KB
L2 cache	16×256KB	12x512KB
L3 cache	20MB	16MB
RAM	64 GB	32GB

B. Execution environment

We summarize in Table I the characteristics of the target systems. We consider two multicore architectures: An *Intel* system that can concurrently execute up to 32 threads and has 10 distinct CPU frequency levels, and an *AMD* system capable to run concurrently up to 24 threads and has 3 distinct CPU frequency levels. The systems used the Ubuntu Operating System with kernel v. 4.19 and v. 5.3 for *Intel* and *AMD* systems, respectively.

For the experiments, we disabled the NUMA balance effects and the turbo boost frequency. We also bound the processes/threads in each core (to avoid threads migration) and mapped them in a round-robin fashion. For that we used the *–map-by socket* and *–bind-to core* parameters of the *mpirun* and *OMP_PROC_BIND=TRUE* and *GOMP_CPU_AFFINITY="0-31"* values to these OpenMP environment variables.

For comparison, we consider as **baseline** the parallel application running with the *maximum number of processes/threads at the maximum frequency level*. We also compare our proposed solution with the parallel application running with the *maximum number of processes/threads and using the ondemand DVFS governor* that dynamically adjust frequency levels. We executed each experiment five times, optimizing for each metric: energy, time, and EDP. We averaged the results (standard deviation of ∼1%).

C. AtTune Implementation

We implemented our framework using C++, compiled with *g++ v. 8.3.0*. AtTune's heuristic performs the execution of the parallel applications by systems calls. To optimize the number of MPI processes, AtTune uses the *-np* parameter of the *mpirun*. For OpenMP applications, the AtTune framework uses the *OMP_NUM_THREADS* environment variable. To optimizing the CPU frequency, AtTune identifies the available frequency levels and sets it according to our heuristic' steps.

To collect the execution time and energy consumption of each kernel, we embedded in the AtTune implementation the *std::chrono::high_resolution_clock* C++ object; the Intel Running Average Power Limit (RAPL) on *Intel* system and the

Application Power Management (APM) on *AMD* system. For the energy, we consider the sum of the energy consumed by the DRAM and core domains, i.e., CPU and cache memories.

V. Results

In this section, we describe the results obtained by the AtTune framework. Figures 2 and 3 present the values of energy consumption, execution time, and EDP for all applications, in both implementations (MPI and OpenMP) running on *Intel* and *AMD* systems. The results are normalized by the baseline, i.e., the maximum number of processes/threads and maximum frequency level. In Table II we summarize the solutions of the best number of processes/threads (**#P** and **#T**) and CPU frequency level (**Freq.**), in GHz, obtained for the five executions of our framework.

On average, MPI implementations provided results of energy, time, and EDP worse than OpenMP in 32%, 57%, and 138% on *Intel* system, and 36%, 38%, and 114% on *AMD*. However, MPI versions take more advantage of adjusting the number of processes and CPU frequency levels (see solutions in Table II).

By running the selected MPI kernels on a single node makes them using the shared memory to communicate processes. The MPI uses the vader Byte Transfer Layer (BTL) mechanism to perform the communication [13] (as we are using the *OpenMPI* 3.1.3 version), which provides small message latency and optimized support for zero-copy transfers. Despite that, communication among MPI processes requires explicit buffers declaration and explicit send/receive calls, which is more costly than a simple memory accesses (load and store) done by OpenMP threads.

When comparing the optimization performed on *Intel* (Fig.2) and *AMD* (Fig.3) systems, the optimization on *AMD* achieved more improvements over the baseline (and also over the kernels' executions using the ondemand governor). Several factors imply these results, mainly the architectural difference between the systems, i.e., number of cores, CPU frequency levels, cache size, and main memory size. One of them is the number of CPU frequency levels and the difference between each level. In the *Intel* system, changing the frequency level implies increasing or reducing at least 1MHz the current frequency. On the other hand, for the *AMD* system, changing the frequency level implies increasing or reducing at least 1GHz the current frequency. This difference in CPU frequency levels makes our proposal more effective in the *AMD* system, mainly when our strategy optimizes frequencies, as we can see in Table II.

TABLE II
Solutions found by our proposed heuristic.

		Intel				AMD			
		MPI		OpenMP		MPI		OpenMP	
Opt. for	**Kernel**	**#P**	**Freq.**	**#T**	**Freq.**	**#P**	**Freq.**	**#T**	**Freq.**
Energy	**IS**	32	1.8 1.6	32	2.0 2.001	8	2.8	24	2.8
	EP	32	1.9 2.0	32	1.9 2.0	24	2.8	24	2.8
	CG	16	1.5 1.6	16	2.001	2 4	2.8	4	2.8
	MG	8	1.6	8	1.6	2	2.8	2	2.8
	FT	16	2.001	32	2.001	4	2.8	8 6	2.8
Time	**IS**	32	2.0 2.001	32	2.0 2.001	16 18 20	3.8	24	3.8
	EP	32	2.0 2.001	32	2.0 2.001	24	3.8	24	3.8
	CG	16	2.0 2.001	32	2.0 2.001	4 5 6 7	3.8	10	3.8
	MG	8	2.0	16	1.9 2.0 2.001	8	3.8	2	3.8
	FT	16	2.0 2.001	32	2.0 2.001	8 9 10	3.8	10	3.8
EDP	**IS**	32	2.0 2.001	32	2.0 2.001	8	3.8	24	2.8
	EP	32	2.0	32	2.0	24	3.8	24	3.8
	CG	16	2.0 2.001	32 16	2.0 2.001	4	2.8	4	2.8
	MG	8	2.0 2.001	8	2.0 2.001	2	2.8	2	2.8
	FT	16	2.0 2.001	32	2.001	4	3.8	8	2.8

CG, MG, and FT present more potential for optimization by adjusting the number of processes/threads and CPU frequency levels. These kernels are memory-bound, then due to the OpenMP implementations perform less memory access than MPI ones, they present more scalability than the MPI implementations. We can see that by observing in Table II, our heuristic tends to find solutions with more OpenMP threads than MPI processes. From these three kernels, the MG is the one that takes more advantages of that optimization. Notice that the MG kernel presents a lower number of processes/threads, whatever the optimized metric, and it presents lower frequency levels when optimized for energy. In MG, our proposed approach optimizes by up to 49%, 33%, and %64% the energy (Fig. 3d), execution time (Fig. 2b), and EDP (Fig. 2c), respectively.

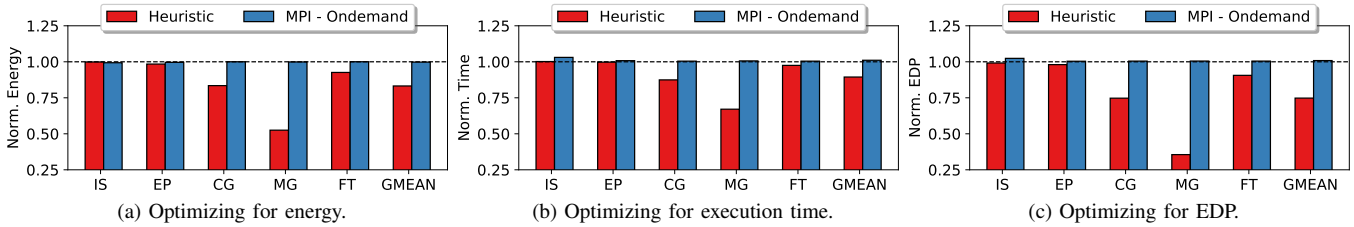
IS and EP are CPU-bound and more scalable kernels than the others. Therefore, as observed in Table II, they take more advantage of more number of processes/threads. Due to the fact that IS execution performs more memory accesses than EP, when our heuristic optimizes the MPI implementation by energy, it reduces the CPU frequency to intermediate values, i.e., 1.6 - 1.8 GHz and 2.8 GHz on *Intel* and *AMD* systems (see IS optimized for energy (Opt. for) in Table II). The adjust in the IS's frequency level incurs in reducing memory accesses intensity, which impacts the amount of communication performed by MPI processes by a time slice. Our heuristic allows us to optimize the IS kernel in up to 25%, 2%, 12% the energy, execution time, and EDP on the *AMD* system. Note that, on *Intel* system, the IS behaves roughly the same as the baseline, being IS better only in 1% when we optimize it for EDP (Fig. 2c).

In general, averaging all applications' results (with geometrical mean), for the *Intel* system, AtTune reduces energy in up to 17% (Fig. 2a) and improves performance in up to 11% (Fig. 2b). On *AMD* system, AtTune reduces energy in up to 36% (Fig. 3a) and improves performance in up to 6% (Fig. 3b). Regarding the EDP results, our approach reduces in up to 25% and 32% on *Intel* and *AMD* systems, respectively.

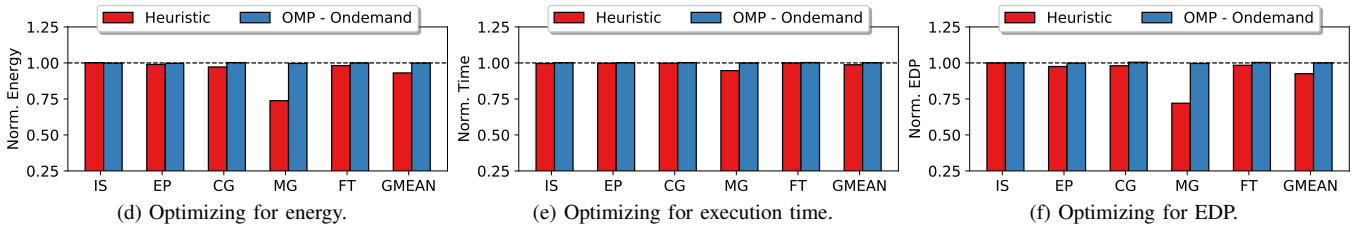
As a summary, over the considered baseline, AtTune framework provides significant energy-saving, performance improvement and significant balancing between them, as shown in the EDP results (Figs. 2c,2f and Figs. 3c,3f). Besides, although AtTune benefits both MPI and OpenMP implementations, the results are more expressive when we optimize the MPI ones.

VI. Conclusion

In this work, we proposed AtTune: a heuristic-based framework to optimize the parallel application execution by tuning the number of processes/threads and the CPU frequency levels. Experimental results on two real multi-core systems showed

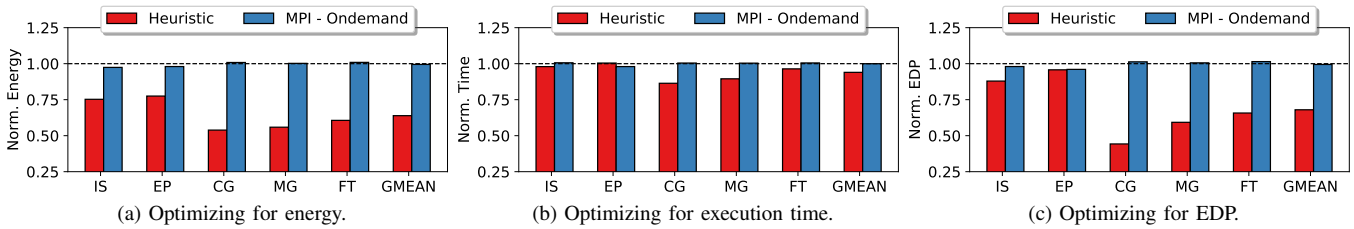


(a) Optimizing for energy. (b) Optimizing for execution time. (c) Optimizing for EDP.

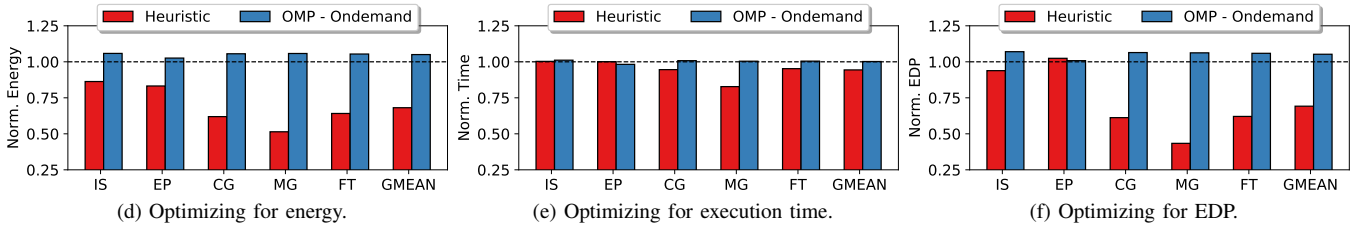


(d) Optimizing for energy. (e) Optimizing for execution time. (f) Optimizing for EDP.

Fig. 2. Result of Energy consumption, Execution time, and Energy-Delay Product (EDP) on *Intel* system. The results were normalized by the **baseline**.



(a) Optimizing for energy. (b) Optimizing for execution time. (c) Optimizing for EDP.



(d) Optimizing for energy. (e) Optimizing for execution time. (f) Optimizing for EDP.

Fig. 3. Result of Energy consumption, Execution time, and Energy-Delay Product (EDP) on *AMD* system. The results were normalized by the **baseline**.

that AtTune improves up to 36%, 11%, and 32% the energy efficiency, performance, and Energy-Delay Product, respectively. As future work, we intend to extend the AtTune framework to optimize hybrid MPI+OpenMP applications.

VII. Acknowledgment

This study was financed in part by the CAPES - Finance Code 001, FAPERGS and CNPq. Some experiments in this work used the PCAD infrastructure, http://gppd-hpc.inf.ufrgs.br, at INF/UFRGS.

References

[1] A. F. Lorenzon, M. C. Cera, and A. C. S. Beck, "On the influence of static power consumption in multicore embedded systems," in *ISCAS*. IEEE, 2015, pp. 1374–1377.

[2] P.-F. Dutot, Y. Georgiou, D. Glesser, L. Lefevre, M. Poquet, and I. Rais, "Towards energy budget control in hpc," in *CCGRID*. IEEE, 2017, pp. 381–390.

[3] A. F. Lorenzon, C. C. De Oliveira, J. D. Souza, and A. C. S. Beck, "Aurora: Seamless optimization of openmp applications," *TPDS*, vol. 30, no. 5, pp. 1007–1021, 2018.

[4] A. F. Lorenzon and A. C. S. Beck Filho, *Parallel Computing Hits the Power Wall: Principles, Challenges, and a Survey of Solutions*. Springer Nature, 2019.

[5] C. C. De Oliveira, A. F. Lorenzon, and A. C. S. Beck, "Automatic tuning tlp and dvfs for edp with a non-intrusive genetic algorithm framework," in *SBESC*. IEEE, 2018, pp. 146–153.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks," *IJSA*, vol. 5, no. 3, pp. 63–73, 1991.

[7] E. L. Padoin, M. Diener, P. O. Navaux, and J.-F. Méhaut, "Managing power demand and load imbalance to save energy on systems with heterogeneous cpu speeds," in *SBAC-PAD*. IEEE, 2019, pp. 72–79.

[8] J. Li and J. F. Martinez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *HPCA, 2006.* IEEE, 2006, pp. 77–87.

[9] D. Li, B. R. de Supinski, M. Schulz, K. Cameron, and D. S. Nikolopoulos, "Hybrid mpi/openmp power-aware computing," in *IPDPS*. IEEE, 2010, pp. 1–12.

[10] G. Chadha, S. Mahlke, and S. Narayanasamy, "When less is more (limo): controlled parallelism for improved efficiency," in *CASES*, 2012, pp. 141–150.

[11] F. Alessi, P. Thoman, G. Georgakoudis, T. Fahringer, and D. S. Nikolopoulos, "Application-level energy awareness for openmp," in *IWOMP*. Springer, 2015, pp. 219–232.

[12] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "A run-time system for power-constrained hpc applications," in *HiPC*. Springer, 2015, pp. 394–408.

[13] S. K. Gutierrez, N. T. Hjelm, M. G. Venkata, and R. L. Graham, "Performance evaluation of open mpi on cray xe/xk systems," in *HOTI*. IEEE, 2012, pp. 40–47.