Aurora: Seamless Optimization of OpenMP Applications

Arthur F. Lorenzon, Charles C. de Oliveira, Jeckson D. Souza, and Antonio Carlos S. Beck

Abstract—Efficiently exploiting thread-level parallelism has been challenging for software developers. As many parallel applications do not scale with the number of cores, the task of rightly choosing the ideal amount of threads to produce the best results in performance or energy is not straightforward. Moreover, many variables may change according to the system at hand (e.g., application, input set, microarchitecture, number of cores) and even during execution. Existing solutions lack transparency (demand changes in the original code) or adaptability (do not automatically adjust to applications at run-time). In this scenario, we propose Aurora, an OpenMP framework that is completely transparent to both the designer and end-user. Without any code transformation or recompilation, it is capable of automatically finding, at run-time and with minimum overhead, the optimal number of threads for each parallel loop region and re-adapt in cases the behavior of a region changes during execution. When executing fifteen well-known benchmarks on four multi-core processors, Aurora improves the Energy-Delay Product by up to 98%, 86% and 91% over the standard OpenMP execution, the OpenMP feature that dynamically adjusts the number of threads, and the Feedback-Driven Threading, respectively.

Index Terms—Thread-level parallelism exploitation, OpenMP, optimization, runtime environments

1 Introduction

THREAD-LEVEL parallelism (TLP) exploitation is being widely used to make the best use of hardware resources and improve performance. However, as the power consumption of high-performance computing systems is expected to significantly grow (up to 100 MW) in the next years [1], energy has become an important issue. Therefore, the objective when designing parallel applications is not to simply improve performance but to do so with a minimal impact on energy consumption.

However, neither performance nor energy improvements resulting from TLP exploitation are linear, and sometimes they do not scale as the number of threads increases. This means that in many cases the maximum number of threads will not deliver the best results. One of the main issues of existing solutions that handle such a problem is that they usually involve some kind of transformation, which can be manual or automatic (by using special languages or toolchains). These transformations modify the source or binary code, or force the programmer to use an API that is not conventional or widespread used. However, as Intel and ARM have been showing with their families of ISAs (Instruction Set Architectures), binary compatibility is mandatory, so it is possible to reuse legacy code and to maintain traditional programming paradigms and libraries.

On top of that, there are several hardware related reasons (that will be further discussed in Section 3)) that make this problem extremely complex: Instruction issue width saturation, off-chip bus saturation, data synchronization, concurrent shared memory accesses [2] [3] [4] [5] [6]. Many of them will vary according to different aspects of the application and system at hand, which can only be defined at run-time, such as:

Input set: Figure 1a shows that there are different levels of performance improvements for the LULESH benchmark [7] (also used in the next examples) over its single-threaded version, as the number of threads changes (x-axis). However, these levels vary according to the input set (small or medium). While the best number of threads is 12 for the medium input set, this number is 11 for the small set.

Metric evaluated: Figure 1b shows that the best performance is reached with 12 threads, while 6 threads bring the lowest energy consumption, and 9 presents the best trade-off between both metrics (EDP - energy-delay product).

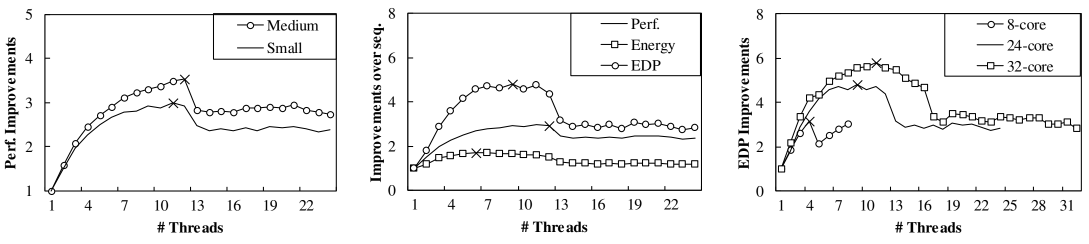
Processor architecture: Figure 1c shows that the best EDP improvements of the parallel application on a 32-core system are observed when it executes with 11 threads. However, the best choice for a 24-core system is 9 threads.

Parallel regions: many applications are divided into several parallel regions, in which each of these regions may have a distinct ideal number of threads, since their behavior may vary as the application executes.

Considering this scenario, we present Aurora. It automatically finds, at run-time and according to a given metric defined a priori by the user, the ideal number of threads for each parallel region of any OpenMP application. Moreover, it can also re-adapt according to a change in the behavior of a particular parallel region during program execution. Because of its dynamic adaptability, Aurora covers all cases discussed before. While it deals with the intrinsic characteristics of the application as well as the microarchitecture on which it will execute; it also takes into account the current input set and application changes at run-time, resulting in significant performance and energy improvements.

Aurora was built on top of the original OpenMP library and is completely transparent to both designer and end-user. Given an OpenMP application binary, Aurora runs on

• *A. F. Lorenzon is with the Federal University of Pampa, Brazil. C. C. de Oliveira, J. D. Souza, and A. C. S. Beck are with the Institute of Informatics, Federal University of Rio Grande do Sul, Brazil*
E-mail: aflorenzon@unipampa.edu.br, {ccoliveira, jdsouza, caco}@inf.ufrgs.br



(a) Different input sets (b) Different metrics evaluated (c) Different multicore processors

Fig. 1: Ideal number of threads (x-axis) to execute the LULESH 2.0 wrt the improvements over sequential version (y-axis)

it without any code changes. Therefore, existent OpenMP applications do not need to be annotated, recompiled or pass through any code transformation. Such transparency is achieved by redirecting the calls originally targeted for the dynamically linked OpenMP library to Aurora. This retargeting is configured by simply setting an environment variable in the Operating System.

Aurora is validated by executing fifteen well-known benchmarks on four distinct multicore processors. With DVFS (Dynamic Voltage and Frequency Scale) set to *ondemand*, we compare Aurora to 1) the baseline execution with the maximum possible number of threads available; 2) to the built in feature of OpenMP that dynamically changes the number of threads; and 3) to the well know feedback-driven threading (FDT) framework [2]. We show gains of 79%, 89%, 98% (performance, energy, and EDP, respectively) over the baseline; 62%, 61%, and 86% over the OpenMP dynamic; and 81%, 72%, and 91% over FDT. We also make a high-level comparison to Varuna [8]. To measure the cost of the learning curve for converging to the ideal number of threads brought by its dynamic adaptation, we also compare Aurora to an Oracle solution. The Oracle executes each parallel region with its predefined optimal number of threads, without the learning overheads. From this, we found that the average cost of Aurora to optimize performance, energy, and EDP for the whole benchmark set and processors is of only 1.8%, 1.6%, and 2.9%, respectively.

The remainder of the paper is organized as follows. Aurora is compared to the related work in Section 2. Section 3 discusses the main factors that affect the scalability of parallel applications. Aurora is presented in Section 4. The methodology is described in Section 5, while Section 6 discuss the results and the potential for improvements of Aurora. Finally, Section 7 draws the final considerations.

2 Related Work

We discuss related work considering the (1) adaptability with respect to the number of threads (i.e., when adaptation happens and whether it is continuous or not) and (2) transparency, which involves the need for special tools or compilers, programmer influence and/or changes in the source or binary codes.

Thread Reinforcer (TR) [9] presents a certain level of transparency to the user, but cannot adjust the number of threads dynamically, at run-time. It consists of a framework that comprises two steps: (*i*) the application binary is executed multiple times with a different number of threads for a short period, while TR searches for the appropriate configuration. (*ii*) Once this configuration is found, the application is fully re-executed with the number of threads defined in the first step. By executing the application binary without modifications, TR is a particular case that keeps binary compatibility. However, it works well only for applications that have a short initialization period, since it introduces a small overhead. Moreover, it considers that the behavior of the parallel regions will not change during execution.

The approaches proposed in [10] and [2] already present some adaptability, since they are capable of defining the number of threads at runtime. Jung et al. [10] present performance models for generating adaptive parallel code for SMT architectures. In this work, an analysis is applied during compile time to filter parallel loops in OpenMP in which the overhead from the thread management (creation/termination, workload distribution, and synchronization) is higher than its own workload. Then, at run-time, the master thread uses the compilation time analysis to dynamically estimate whether it should use the SMT feature of the processor or not. This approach is dependent on a compiler and only considers SMT processors.

Suleman et al. propose the FDT framework [2]. It can adapt the number of threads considering contention for locks and memory bandwidth. The framework consists of a specific compiler that samples a portion of parallel regions of an application implemented with OpenMP. Then, it inserts instructions at the entry and exit of the critical section, and executes it sequentially to analyze synchronization and communication points. FDT uses this analysis to estimate the optimal number of threads for the given parallel region. However, once the number of threads is defined, it cannot re-adapt at run-time. FDT also considers that all threads are homogeneous and ignores fundamental hardware characteristics that are highly correlated to the parallel application behavior: FDT assumes that bandwidth requirement increases linearly with the number of threads, ignoring cache contention and data-sharing between the threads. Morever, it does not consider the effects of the SMT. Nonetheless, FDT is a very popular framework and widely used for comparisons with new approaches (we also compare Aurora with FDT in Section 6).

More adaptive solutions, which consider run-time and continuous adaptation, include [11], [12], [13], [14], [15], [16], [17], and [18]. However, these solutions either rely on HW/OS support, or special compiler and need for recompilation or a previous off-line analysis, as discussed next.

In [11], Curtis-Maury et al. propose a framework for on-line adaptation of multithreaded code for low-power and high-performance execution. It has an off-line phase in which data from hardware counters are collected, and profiles of parallel execution are analyzed. Then, at runtime, the framework uses the information obtained in the off-line phase to adapt the number of threads. ACTOR [12] is a system that dynamically changes the number of threads to improve energy efficiency. ACTOR is divided into three steps: (i) artificial neural networks (ANNs) are trained off-line to model the relationship between performance counter events and the resulting performance with a different number of threads; (ii) at runtime, the derived ANN models are used to predict the performance of parallel regions that were previously identified by the programmer with special function calls from the ACTOR library; (iii) the parallel regions are executed with the predicted number of threads. Although the number of threads is predicted at runtime in [11] and [12], an off-line phase is required before the execution of each application. Therefore, if either the input set or processor is changed, the off-line analysis must be re-executed, which significantly increases the total execution time of the entire framework.

LIMO [13] is a dynamic system that monitors the application at run-time, being able to adapt the execution accordingly. However, this solution requires hardware modifications to determine the working set size of a thread, as well as additional OS support for detecting threads that block due to busy-wait (spin loop). Consequently, it cannot be applied to any existent commercial microarchitecture. Also, LIMO relies on compiler support to insert special system calls and to modify loop bodies (so applications need to be recompiled to take advantage of LIMO).

Parcae [14] is a framework that comprises a compiler and run-time system to optimize performance. The compiler identifies parallelizable regions in a sequential program and applies multiple parallelizing transforms to them. When the application is executing, the run-time system monitors the program performance and system events to determine the best configuration for the parallel application. However, Parcae relies on system support (compiler, monitor, and executor) to modify sequential applications at compilation and execution time. Therefore, if there are any changes in the environment (input set or microarchitecture), the application needs to be recompiled.

Thread Tailor [15] dynamically adjusts the number of threads to optimize some specific parts of the system, such as cache and memory footprint. The approach works as follows: (*i*) programmers create a parallel application that uses a high number of threads; (*ii*) the binary created is profiled off-line to collect statistics regarding the number of threads, communication, and synchronization to form a communication graph; (*iii*) at runtime, a dynamic compiler takes a quick snapshot of the system state to determine how many free resources are available and to decide the optimum number of threads; (*iv*) based on that information, the dynamic compiler generates code for the new threads, intercepts future calls to thread creations, and redirects them to the new threads. However, as Thread Tailor works for PThreads and MPI, it requires huge effort from the programmer to develop a parallel application that is able of using a high number of threads/processes. In this case, the developer must explicitly implement thread/process management (creation/termination, workload distribution, synchronization and communication).

Porterfield et al., [16] propose an adaptive run-time system that automatically adjusts the number of threads based on on-line measurements of system resource usage. The approach extends Qthreads (a parallel library) to be used with MAESTRO, a dynamic runtime library for power and concurrency adaptation of parallel applications [19]. However, it is dependent on a source-to-source compiler to obtain OpenMP directives and map the functions and data structures to the Qthreads library. OpenMPE [17] is an extension designed for energy management of OpenMP applications, in which the programmers expose energy saving opportunities through the insertion of directives in OpenMP codes. However, it works only for the compiler and runtime system from the Insieme Project [17].

More transparent approaches, which do not need support from special compilers, include [18], [20], [21], [22], [8], and [23]. Li et al. [18] propose a library to save energy with no performance loss for hybrid MPI/OpenMP applications. The library has an off-line phase to train a model that will be used at runtime to determine the ideal configuration (number of threads and CPU frequency) for each OpenMP region. To use this library, the user has to instrument the applications with functions calls around each OpenMP region and selected MPI operations (collectives and MPI_Waitall). Although this library does not require special compiler/tools, the user needs to modify and recompile the source code, in addition to having prior knowledge of MPI functions. Also, if the user wants to use another processor, it is necessary to rerun the entire training set.

Shafik et al. [20] propose an adaptive and scalable energy minimization model for OpenMP programs, which comprises two steps: (*i*) code notations are inserted by the programmer in the sequential and parallel parts of the code to enable energy minimization with specified performance requirements; (*ii*) the runtime system read these performance requirements and uses this information to guide the energy minimization. The same method, but aiming to improve lifetime reliability through balanced thermal controls while meeting a given power budget, was presented in [21]. All of these works need code recompilation.

In [22], Sridharan et al. propose ParallelismDial (PD), a model that automatically tunes a program's performance to the underlying system. It monitors the system efficiency, regulates the degree of parallelism, and continuously adapts the execution through a heuristic to an optimum point of operation. The heuristic used to find the best degree of parallelism is based on the hill-climbing search algorithm, which works as follows: (*i*) the parallel region runs with only one thread to establish a sequential measure; (*ii*) the same region is executed with three degrees of parallelism (low, medium, and high); (*iii*) the search is refined to the best interval and continues until the optimum point be reached.

In [8], PD was extended to Varuna system. It comprises two components: (*i*) an analytical engine which continuously monitors changes in the system using hardware performance counters, models the execution behavior, and determines the optimum degree of parallelism; and (*ii*) a manager that automatically regulates the execution to

TABLE 1: Comparison of Aurora with the related work

Proposal	Adaptability		Transparency			APIs	Diverse Metrics
	Run-time	Continuous	No special compilers/tools	No programmer Influence	Binary Compatibility		
Thread Reinforcer [9]				x	x	OpenMP, PThreads	
Jung et al. [10]	x			x		OpenMP-FORTRAN	
FDT [2]	x			x		OpenMP	x
Curtis-Maury et al. [11]	x	x		x		OpenMP-FORTRAN	x
ACTOR [12]	x	x				OpenMP	x
LIMO [13]	x	x		x		OpenMP, PThreads	x
Parcae [14]	x	x		x		Sequential Code	
Porterfield et al. [16]	x	x		x		OpenMP	
Thread Tailor [15]	x	x				PThreads, MPI	
OpenMPE [17]	x	x				OpenMP	x
Li et al. [18]	x	x	x			MPI + OpenMP	
Shafik et al. [20] [21]	x	x	x			OpenMP	
ParallelismDial [22]	x	x	x			TBB, Prometheus	
Varuna [8]	x	x	x			PThreads, TBB	x
LAANT [23]	x	x	x			OpenMP	x
Aurora	**x**	**x**	**x**	**x**	**x**	**OpenMP**	**x**

match the degree of parallelism determined by the analytical engine. PD and Varuna comprise a monitor system that intercepts thread and task creation from PThreads, TBB, and Prometheus libraries, and create a pool of tasks to optimize their degree of parallelism. However, to do so efficiently, PD and Varuna create a large number of fine-grained tasks. Consequently, it requires more effort from the programmer, who is required to create as many threads as possible, each one with the lowest possible workload. Because of this intrinsic characteristic, PD and Varuna focus on recursive applications that are mostly concentrated on big-data. Besides that, they cannot optimize OpenMP applications due to limitations of the system (virtual tasks) used to control parallelism [8]. Finally, LAANT [23] is a library that automatically adjusts the number of threads of OpenMP applications. In this approach, code must be modified by the programmer to include additional function calls in each parallel region of interest in the application.

2.1 Our Contributions

Table 1 compares Aurora to previous works. The column *run-time adaptation* indicates the approaches able to select the ideal number of threads as the parallel application executes. However, once the technique converged to a given number of threads, this number will not change anymore. On the other hand, *continuous adaptation* refers to those works that can readjust the number of threads during application execution according to variations in its workload or environment system. The column *no special compiler/tools* presents the approaches that do not need any specific compiler or tool to generate special parallel code. They use different tools from the traditional programming framework that usually involves a C/C++ compiler and a parallel API. The column *no programmer influence* contains the approaches that do not demand any changes in the source code by the software developer. *Binary compatibility* refers to the techniques that can be used without any need for code recompilation at all: the existent binary code as is can take advantage of the approach. The column *API* shows the parallel libraries supported by each referred work. Finally, *Diverse metrics* refers to works that can optimize more than one metric. As depicted in Table 1, no approach discussed covers all the needed characteristics so it could be considered completely transparent and adaptive. On the other hand, Aurora can find the ideal number of threads as the application executes, and continuously adjusts the number of threads if there are changes in the workload. It works with any C/C++ compiler and OpenMP and can target different metrics. More important, the software developer does not need to make any changes in the source code or even recompile it. Any existent binary code that uses OpenMP can take advantage from Aurora. For that, the programmer only has to enable Aurora and its optimization metric through the use of one environment variable in the Linux OS. However, because this high level of transparency, Aurora is limited to OpenMP applications only (more details in Section 6.5).

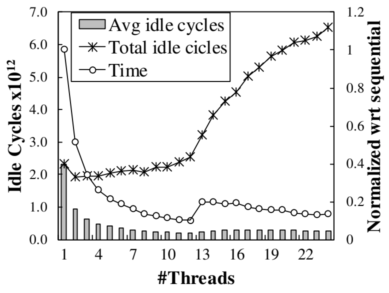
3 Scalability of Parallel Applications

Many works have associated the fact that selecting the maximum number of available threads (the common choice for most software developers [15]) will not necessarily lead to the best possible performance. The causes are related to hardware or software: saturation of functional units in SMT processors [5] [6], off-chip bus saturation [2] [3], overhead of data synchronization among threads [2] [15] [4], and number of shared memory accesses [3].

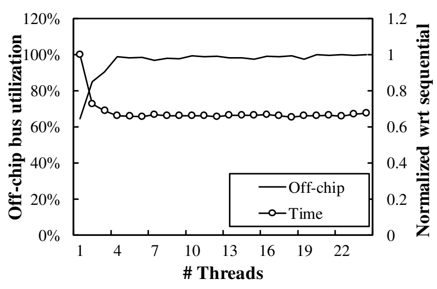
To measure (through correlation) their real influence, we have executed four benchmarks from our set (and used them as examples in the next subsections) on a 12-core machine with SMT support. Each one of them has one limiting characteristic that stands out , as shown in Table 3. The benchmark Hotspot (HS) saturates the issue-width; fast Fourier transform (FFT), the off-chip bus; MG, the shared memory accesses; and N-body (NB) saturates data-synchronization. To analyze each of the scalability issues, we considered the Pearson Correlation [24]. It takes a range of values from +1 to -1: the stronger the "r" linear association between two variables, the closer the value will be to either +1 or -1. $r \geq 0.9$ or $r \leq -0.9$ mean a very strong correlation (association is directly or inversely proportional). We discuss these bottlenecks next and revisit them in the results to show how Aurora handles them by its dynamic adaptation.

3.1 Issue-width Saturation

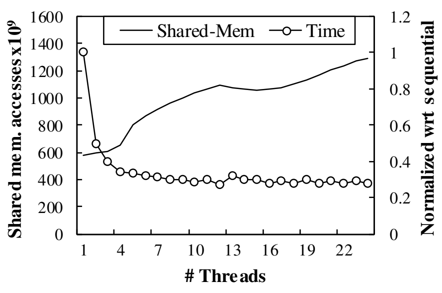
SMT allows many threads to run simultaneously on a core. It increases the probability of having more independent



(a) Issue-width saturation



(b) Off-chip bus saturation



(c) Shared memory accesses

Fig. 2: Scalability behavior of parallel applications

instructions to fill the function units (FUs). Although it may work well for applications with low ILP, it can lead to the opposite behavior if an individual thread presents enough ILP to issue instructions to all or most of the core's FUs. Then, SMT may lead to resource competition and functional unit contention, resulting in extra idle cycles. Figure 2a shows the performance speedup relative to the sequential version and the number of idle cycles (average, represented by the bars, and total) as we increase the number of threads for the HS application. As we start executing with 13 threads, two will be mapped to the same physical core, activating SMT. From this point on, as the number of threads grows, the average number of idle cycles increases by a small amount or stays constant. However, the total number of idle cycles significantly increases. Because this application has high ILP, there is not enough resources to execute both threads concurrently as if each one was executed on a single core. They become the new critical path of that parallel region, as both threads will delay the execution of the entire parallel region (threads can only synchronize when all have reached the barrier). Therefore, performance drops and is almost recovered only with the maximum number of threads executing. In the end, extra resources are being used without improving performance and potentially increasing energy consumption, decreasing resource efficiency.

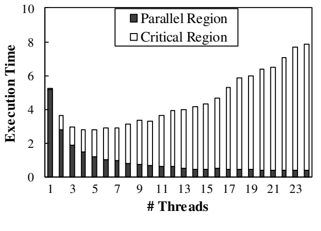
3.2 Off-chip Bus Saturation

Many parallel applications operate on huge amounts of data that are private to each thread and have to be constantly fetched from the main memory. In this scenario, the off-chip bus that connects memory and processor plays a decisive role in thread scalability: as each thread computes on different data blocks, the demand for off-chip bus increases linearly with the number of threads. However, the bus bandwidth is limited by the number of I/O pins, which does not increase according to the number of cores [25]. Therefore, when the off-chip bus saturates, no further improvements are achieved by increasing the number of threads [2].

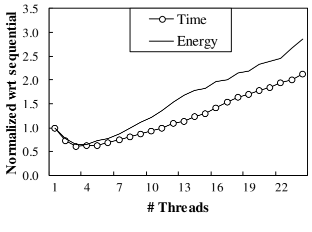
Figure 2b shows the FFT execution as an example. As the number of threads increases, execution time and energy consumption reduce until the off-chip bus becomes completely saturated (100% of utilization). From this point on (4 threads), increasing the number of threads does not improve performance, as the bus cannot deliver all the requested data. There might be an increase in energy consumption as well since many hardware components will stay active while the cores are not being properly fed with data.

3.3 Shared Memory Accesses

Threads communicate by accessing data that are located in shared memory regions, which are usually more distant from the processor (e.g., L3 cache and main memory), so they can also become a bottleneck. Figure 2c presents the number of accesses to the L3 cache (the only cache level shared among the cores) in the primary y-axis and the execution time normalized to the sequential execution in the secondary y-axis for the MG benchmark. When the application executes with more than four threads, the performance is highly influenced by the increased number of accesses to L3. Other factors may also influence L3 performance: thread scheduling, data affinity or the intrinsic characteristics of the application. For instance, an application with a high rate of private accesses to L1 and L2 may also lead to an increase in the L3 accesses. Moreover, part of the communication may be hidden from the L3 when SMT is enabled: two threads that communicate and are executing on the same SMT core may not need to share data outside it.



(a) Critical section behavior



(b) Perf./Energy degradation

Fig. 3: Data-Synchronization

3.4 Data-Synchronization

Synchronization operations ensure data integrity during the execution of a parallel application. In this case, critical sections are implemented to guarantee that only one thread will execute a given region of code at once, and therefore data will correctly synchronize. In this way, all code inside a critical section must be executed sequentially. Therefore, when the number of threads increases, more threads must be serialized inside the critical sections. It also increases the synchronization time (Figure 3a), potentially affecting the execution time and energy consumption of the whole application. Figure 3b shows this behavior for the n-body benchmark. While it executes with 4 threads or less, the performance gains within the parallel region reduces the execution time and energy consumption, even if the time spent in the critical region increases (Figure 3a). However, from this point on, the time the threads spend synchronizing overcomes the speedup achieved in the parallel region.

4 Aurora

4.1 Integration to OpenMP

OpenMP is a parallel programming interface for shared memory in C/C++ and FORTRAN. It consists of a set of compiler directives, library functions, and environment variables that eases the burden managing threads in the code. Therefore, extracting parallelism using OpenMP usually requires less effort when compared to other APIs (e.g., PThreads, MPI, TBB), making it more appealing to software developers [26]. That is why it is widely used and there are many OpenMP parallel benchmarks: NAS Parallel Benchmark [27], SPEComp [28], Parboil [29], Rodinia [30], PARSEC [31], GROMACS [32], LULESH [7], etc.

Parallelism is exploited through the insertion of directives in the code that inform the compiler how and which parts of the application should be executed in parallel [33]. OpenMP provides three ways for exploiting parallelism: parallel loops, sections, and tasks. Sections are used when the programmer must distribute the workload between threads in a similar way as PThreads. Tasks are employed when the application uses recursion (i.e., sort algorithms). They are only used in very particular cases. Parallel loops are used to parallelize applications that work on multi-dimensional data structures (i.e., array, grid, etc.), so the loop iterations (*for*) can be split into multithread executions. Therefore, parallel loops are by far the most used approach. All the benchmark sets discussed in this paper are implemented in this way. For now, Aurora only works with parallel loops and does not influence in any way OpenMP applications that are parallelized using sections or tasks.

All functionalities provided by OpenMP are implemented into the *libgomp*, a GNU Offloading and Multi-Processing Run-time Library. This library is dynamically linked to applications that use OpenMP, so any modifications in its code are completely transparent to user applications. Aurora was incorporated into this library. To use Aurora, the user simply has to replace the original OpenMP *libgomp* with Aurora's *libgomp*. This new library includes all original OpenMP functionalities plus the new functions of Aurora. When the environment variable *OMP_AURORA* is set in the Linux OS, the thread management system of Aurora is used instead of the original OpenMP functions. This environment variable can be configured with the following arguments: performance, energy, or EDP. If the variable is not set, application executes with the original OpenMP functions, without any influence of Aurora. Therefore, there is no need to make any modifications in the OS (package installation, kernel recompilation, super user permission, etc.). In this way, any existing binary code can benefit from Aurora without any modifications or need for recompilation. Furthermore, Aurora does not influence the user-defined affinity policy (e.g., GOMP_CPU_AFFINITY and KMP_AFFINITY), so it has the very same behavior as the unmodified original OpenMP library in this matter. The reason is that Aurora does not affect or change the original OpenMP thread creation function, which is responsible for binding each thread to a specific core according to the affinity policy defined by the user.

To better understand how Aurora works, let us first consider the regular way for parallelizing an iterative application with parallel loops [33] and the respective main functions implemented by *libgomp*. When the program starts executing, the *initialize_env*() function is called. It is responsible for initializing all the environment variables used by OpenMP during the execution. When the program reaches the directive #*pragma omp parallel* (used to indicate a parallel region), functions to create and define the number of threads (*gomp_resolve_num_threads*()) are called. At the end of the parallel region, the *gomp_parallel_end*() joins the threads and finalizes the parallel region environment. Finally, when the application ends, *team_destructor()* is called. Aurora features were split into four functions (discussed in details next), and incorporated into the *libgomp* aforementioned functions.

***initAurora*()** is responsible for recognizing the Aurora optimization target defined by the environment variable (OMP_AURORA). It also initializes the necessary data structures, libraries, and variables used to control the search algorithm (described in Section 4.2). This function was implemented inside the original *initialize_env*().

***auroraResolveNumThreads*()** sets the number of threads that execute each parallel region based on the current state of the search algorithm. Moreover, it initializes the counters for collecting data from the execution environment of the current parallel region. It was implemented inside the *gomp_parallel_start*(), and replaces the original *gomp_resolve_-num_threads*() function when Aurora is active.

***auroraEndParallelRegion*()** is executed after the parallel region to collect its execution time, energy, and EDP. Execution time is extracted by the *omp_get_wtime()* function, provided by OpenMP, while energy is obtained directly from the hardware counters present in modern processors. In the case of Intel processors, the Running Average Power Limit (RAPL) library is used [34], while the Application Power Management library could be used for AMD processors [35]. *auroraEndParallelRegion*() performs one step of the search algorithm (which is explained in the next sub-section) to define the number of threads that will be used for the next iteration of the current parallel region. *auroraEndParallelRegion*() is implemented inside *gomp_parallel_end*() function.

***auroraDestructEnv*()** concludes and destroys Aurora environment at the end of application execution. It was implemented inside *team_destructor*() OpenMP function.

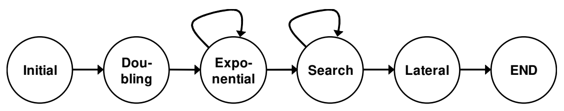


Fig. 4: States and transitions of the search algorithm

4.2 Search Algorithm

The implemented heuristic is divided into two phases. The first one evaluates the scalability of the parallel region and reduces the size of the space exploration. It exponentially increases the number of threads (i.e., 2, 4, 8, 16, ...) while there are potential improvements (states *Initial*, *Doubling*, and *Exponential* in Algorithm 1 and Figure 4). The second phase (states *Exponential*, *Search*, and *Lateral*) performs a hill-climbing based algorithm with lateral movements in the interval of threads defined by the first phase. Intuitively, finding the optimal number of threads to execute any parallel region is a convex optimization problem. In this specific problem, it means that there will be only one specific number of threads that delivers the best result for a given metric and parallel region. Hill Climbing algorithms are very suitable for such problems and are also known for having low complexity and being fast, which is essential to reduce the technique overhead (since it is executed at runtime). Other authors have already shown that when hill-climbing is used along with another approach to guide the search, in most cases such algorithms will reach a near-ideal solution, escaping from the local minima and plateaus [36], [37]. As the search algorithm implemented by Aurora learns towards the best number of threads during application execution, all the computation done in the search phase is not wasted (i.e., it is used by the application), reducing the overhead of Aurora.

The search starts in the *Initial* state (*line 5*), in which the initial number of threads (*threadStartSearch*) and the current number of threads (*currentNT*) are defined. Then, the parallel region is executed (and the target metric is measured) with the initial number of threads (e.g., 2 threads). Then, the state changes to *Doubling*. In this state, the best metric measurement (*bestMetricMsmt*) and current number of threads are set with initial values obtained in *Initial*. After that, the current number of threads is doubled, and state changes to *Exponential*. In *Exponential* (*line 13*), the measured metric (time, energy, or EDP) is evaluated, and the number of threads continues to double while the measured metric keeps improving and the maximum number of available hardware threads is not reached (*lines 15-30*). Then, the state changes to *Search*. Once in there, Aurora knows the interval of potential candidates for the ideal number of threads and starts the second phase.

To better understand the second phase, let us consider as an example that the interval of potential candidates lies in the range of 8 and 16 threads. Given that, it will start executing with 12 threads (the value in the middle between 8 and 16) to decide which is the next range to evaluate (8-12 or 12-16). This process is repeated until the best number of threads is found (state *Search*). After that, state *Lateral* starts, in which lateral movement (*line 47*) is performed to avoid minimal locals and plateaus. This movement is performed by testing a neighboring configuration (number of threads) at another point in the search space that has not yet been tested. When Aurora converges to the best number of threads for a particular parallel region, it begins to monitor the behavior of such region. If there is any change in the workload, which in this work we consider a variation of 30%, the search algorithm starts its execution again.

TABLE 2: States of the search algorithm

State	Operation
Initial	Execution with the initial number of threads.
Doubling	Double the number of threads.
Exponential	Compare the results achieved in Initial and Doubling, and exponentially increases the number of threads while either there are improvements or when the max number of hardware threads is met. Then, state changes to Search.
Search	Search the ideal number of threads in the interval of candidates defined in Exponential. When there are only two candidates, state changes to Lateral.
Lateral	Define the best number of threads and performs lateral movement.
END	Aurora begins to monitor the behavior of the parallel region.

5 Methodology

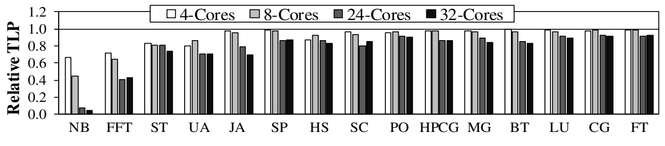
5.1 Benchmarks

Fifteen applications written in C/C++ and parallelized with OpenMP from assorted benchmarks suites were chosen according to the scalability issues discussed in Section 3:

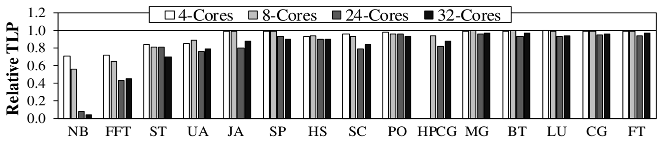
Seven kernels from the NAS Parallel Benchmark [38]: *block tri-diagonal solver* (BT), *conjugate gradient* (CG), *discrete*

Algorithm 1 Search algorithm implemented by Aurora

```
1: function SEARCHALGORITHM()
2:     if $state$ != $END$ then
3:         $metricMsmt \leftarrow$ get time, energy, or EDP according to the target metric
4:         switch $state$ do
5:             case $Initial$:
6:                 $lastNT \leftarrow currentNT \leftarrow threadStartSearch$;
7:                 $state \leftarrow Doubling$;
8:             case $Doubling$:
9:                 $bestMetricMsmt \leftarrow metricMsmt$;
10:                $bestNT \leftarrow currentNT$;
11:                $currentNT \leftarrow currentNT \times 2$;
12:                $state \leftarrow Exponential$;
13:            case $Exponential$:
14:                $step \leftarrow \frac{lastNT}{2}$;
15:                if $metricMsmt \leq bestMetricMsmt$ then
16:                    $bestMetricMsmt \leftarrow metricMsmt$;
17:                    $bestNT \leftarrow currentNT$;
18:                    if $currentNT \times 2 \leq numCores$ then
19:                        $lt \leftarrow currentNT$;
20:                        $currentNT \leftarrow bestNT \times 2$;
21:                    else
22:                        $currentNT$ -= $step$;
23:                        $state \leftarrow Search$;
24:                    end if
25:                else
26:                    if $bestNT == \frac{numCores}{2}$ then
27:                        $currentNT$ -= $step$;
28:                    else
29:                        $currentNT$ += $step$;
30:                    end if
31:                    $state \leftarrow Search$;
32:                end if
33:            case $Search$:
34:                if $metricMsmt \leq bestMetricMsmt$ then
35:                    $bestNT \leftarrow currentNT$;
36:                    $bestMetricMsmt \leftarrow metricMsmt$;
37:                end if
38:                $step \leftarrow \frac{step}{2}$;
39:                $currentNT$ += $step$;
40:                if $step == 1$ then
41:                    $state \leftarrow Lateral$;
42:                end if
43:            case $Lateral$:
44:                if $metricMsmt \leq bestMetricMsmt$ then
45:                    $bestNT \leftarrow currentNT$;
46:                end if
47:                Performs lateral movement to avoid minimum locals
48:                $state \leftarrow END$;
49:    else
50:        if $workloadVariation == true$ then
51:            run Aurora search algorithm again
52:        end if
53:    end if
54: end function
```



(a) Small input set



(b) Medium input set

Fig. 5: TLP Available for each benchmark - normalized wrt the maximum number of threads in each processor

TABLE 3: Pearson correlation between the scalability issues and each benchmark

		NB	FFT	ST	UA	JA	SP	HS	SC	PO	HPCG	MG	BT	LU	CG	FT
Small Input	*Issue-width sat.*	-0.82	-0.71	-0.56	**-0.92**	-0.80	-0.80	**-0.91**	-0.80	-0.84	-0.65	-0.81	-0.75	-0.87	**-0.91**	**-0.90**
	Off-chip bus sat.	0.46	**-0.98**	**-0.90**	-0.84	-0.57	-0.71	-0.51	-0.82	-0.56	**-0.94**	-0.76	-0.79	-0.80	-0.82	-0.68
	Shared mem. acc.	0.80	-0.43	-0.71	-0.78	0.52	-0.83	-0.52	**-0.91**	0.71	-0.86	**-0.90**	**-0.91**	**-0.96**	-0.85	-0.78
	Data-synchr.	**0.97**	-0.50	-0.61	-0.49	**0.92**	**0.95**	-0.54	-0.54	**0.94**	-0.24	-0.59	-0.64	-0.61	-0.61	-0.82
Medium Input	*Issue-width sat.*	-0.78	-0.71	-0.63	-0.73	-0.69	-0.82	**-0.92**	-0.76	-0.83	-0.74	-0.79	-0.73	**-0.90**	**-0.94**	**-0.91**
	Off-chip bus sat.	0.39	**-0.97**	**-0.95**	-0.85	**-0.90**	-0.62	-0.52	-0.86	-0.46	**-0.94**	-0.88	-0.79	-0.65	-0.82	-0.76
	Shared mem. acc.	0.81	-0.75	-0.73	**-0.94**	0.82	**-0.90**	-0.54	**-0.96**	**-0.94**	-0.86	**-0.96**	**-0.92**	0.09	0.70	-0.86
	Data-synchr.	**0.96**	-0.53	-0.38	-0.74	-0.48	-0.11	-0.64	-0.68	-0.67	-0.70	-0.78	-0.61	-0.18	-0.64	-0.77

3D fast Fourier transform (FT), *lower-upper gauss-seidel solver* (LU), *multi-grid on a sequence of meshes* (MG), *scalar penta-diagonal solver* (SP), and *unstructured adaptive mesh* (UA). As the original version of NAS is written in FORTRAN, we consider the OpenMP-C version developed in [27].

Two applications from the Rodinia Benchmark Suite [30]: *hotspot* (HS) and *streamcluster* (SC).

Six applications from different domains: *n-body* (NB) - computes a simulation of a dynamical system of particles [39]; *fast Fourier transform* (FFT) - calculates the discrete Fourrier transform of a given sequence [40]; *STREAM* (ST) - measures sustainable memory bandwidth [41]; *Jacobi* (JA) method iteration - computes the solutions of a diagonally dominant system of linear equations [42]. *Poisson* (PO) - computes an approximate solution to the Poisson equation in a rectangular region [42]; and the *high performance conjugate gradient* benchmark (HPCG), a stand-alone code that measures the performance of basic operations [43].

Two different input sets for each benchmark were considered: small and medium. Table 3 depicts the Pearson correlation between each scalability issue (discussed in Section 2) and the application. As can be observed, the chosen applications do not scale for different reasons, according to Section 3. All the data used for the scalability analysis was obtained directly from hardware using Intel Performance Counter Monitor (PCM) [44], Intel Parallel Studio, and Performance Application Programming Interface (PAPI) [45]. As one can note in Figure 5, the chosen benchmarks also cover a wide range of different TLP behaviors. We measured TLP as defined by the authors in [46]. The closer this value is to 1.0 (normalized to the total number of cores available), the more TLP is available. NB has the lowest TLP available, where only 10% of the execution is performed in parallel when the 32-core system is considered, while the the FT benchmark presents the highest TLP, in which more than 95% of the application is executed in parallel.

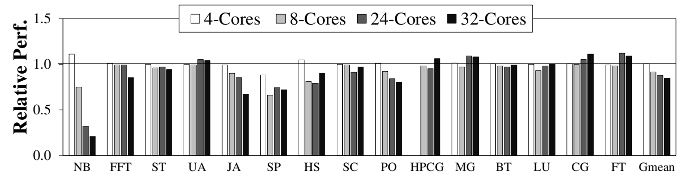
5.2 Execution Environment

The experiments were performed on four different multicore processors (Table 4). We used the Ubuntu Operating System with Kernel v. 4.4.0 in all the machines. The CPU frequency was configured to adjust according to the workload application, using ondemand as DVFS governor, which is the standard governor used in most Linux versions. We compiled the applications with gcc/g++ 6.3, using the optimization flag -O3, and the OpenMP distribution version 4.0. The results presented in the next session are the average of ten executions with a standard deviation lower than 0.5%.

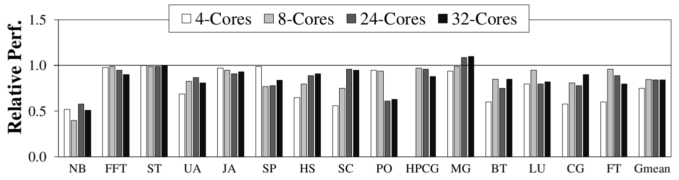
We evaluated Aurora in five different scenarios: **Baseline**: the application executes with the maximum number of threads available in the system; **OMP_Dynamic**: a built-in feature of OpenMP that dynamically adjusts the number of threads of each parallel region, aiming to make the best use of system resources, such as memory and processor. OMP_Dynamic is generally used to avoid oversubscription, in the way that the number of threads is defined based on the load average utilization of processes on the system [47]. This feature is enabled by using the environment variable OMP_DYNAMIC or through the insertion of the *omp_set_dynamic*() in the source code [33]; **Feedback-Driven Threading (FDT)**: the number of threads is defined based on the contention for locks and memory bandwidth (as discussed in Section 2). We have faithfully implemented the FDT mechanism in C language and inserted their functions into the OpenMP codes, as defined by Suleman et al. [2]. **Oracle solution**: the execution of each parallel region with the optimal number of threads for each metric, without the cost of the learning curve. The optimal number of threads was obtained through an exhaustive execution of each parallel region of each application with 1 to n threads, where n is the maximum number supported by hardware. We also performed a high-level comparison with **Varuna**, by faithfully implementing its programming model (as defined in [8]) and applying it to our benchmarks.

TABLE 4: Main characteristics of each processor

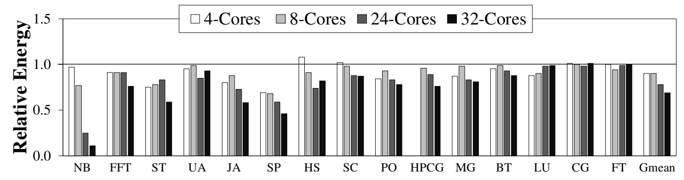
	Intel Core		**Intel Xeon**	
Model	i5-4460	i7-6700	E5-2630	E5-2640
Microarch.	Haswell	Skylake	Sandy Bridge	Ivy Bridge
# Cores - #Threads	4 - 4	4 - 8	(2x6) - 24	(2x8) - 32
CPU Freq	3.2 GHz	3.4 GHz	2.3 GHz	2.0 GHz
L1 Cache	4x32 KB	4x32 KB	12x32 KB	16x32 KB
L2 Cache	4x256 KB	4x256 KB	12x256 KB	16x256 KB
L3 Cache	6 MB	8 MB	30 MB	40 MB
RAM	16 GB	32 GB	32 GB	64 GB



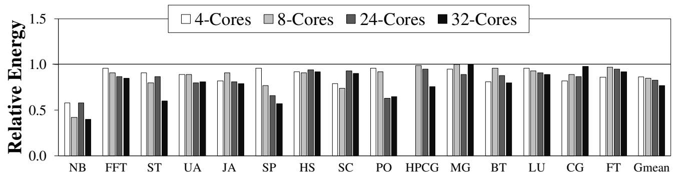
(a) Performance - Baseline



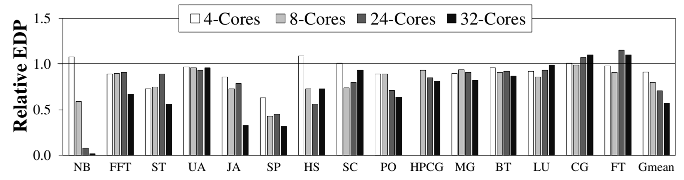
(b) Performance - OMP_Dynamic



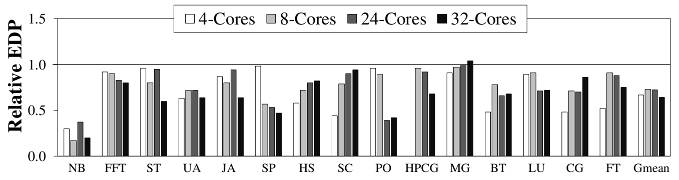
(c) Energy Consumption - Baseline



(d) Energy Consumption - OMP_Dynamic



(e) EDP - Baseline



(f) EDP - OMP_Dynamic

Fig. 6: Aurora vs Baseline and OMP_Dynamic (medium input): lower than 1.0 means that Aurora is better

6 Results

6.1 Performance, Energy, and EDP

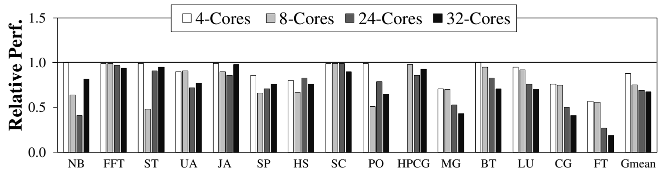
Table 7 depicts the number of threads found by Aurora that offers the best result in performance, energy, and EDP to execute the main parallel regions of each application. As an example, let us consider the LU application executing with the medium input on the 8-core system and targeting the EDP. It has two main parallel regions: the ideal number of threads for the first is four, while for the second the number is three. Moreover, depending on the input set, the ideal number of threads for each parallel region may vary. This is the case of the CG application running on the 32-core system. When changing the input set from small to medium, the workload of the second parallel region changes, increasing its TLP. Now, the best EDP for this region is achieved with 32 threads instead of 16. The ideal number of threads also varies when the target optimization metric changes. The ST executing on the 32-core system is an example: 12 threads is the best choice for performance, 4 threads for energy consumption and 6 threads for EDP.

Figures 6 and 7 present the results for the entire benchmark set when running the medium input set, along with their geometric mean (Gmean) considering the four multicore systems. Figure 6 compares Aurora to the baseline and OMP_Dynamic (represented by the black line), while Figure 7 compares Aurora to the FDT framework (also represented by a black line). Results are normalized according to the setup to be compared (baseline, OMP_Dynamic, or FDT), so values below 1 mean that Aurora is better. They are presented regarding performance, energy consumption, and EDP, depending on the optimization metric used by Aurora. As an example, Figure 6c shows the energy savings achieved by Aurora over the baseline when set to reduce the energy consumption. Table 5 summarizes the results for the small input set considering the geometric mean for the four multicore systems.

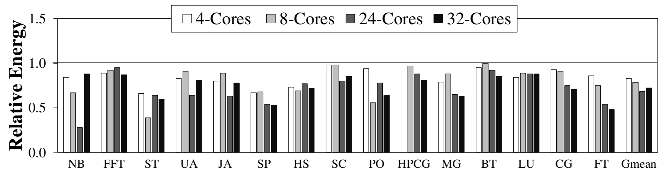
Aurora versus Baseline: as observed in Figure 6 and Table 5, in most cases Aurora shows improvements regarding any metric. If one considers the geometric mean (Gmean bars in each Figure) in any scenario, Aurora is most of times better. In very specific scenarios where the design space exploration is limited, it presents similar results as the baseline. Considering the best case, execution time was reduced by 16% with the medium input set executing on the 32-core system. The best scenario for energy consumption and EDP is with the small input set and the 32-core system: energy is reduced by 34%, and EDP is improved by 47%. When considering the overall geometric mean (entire benchmark set and all processors), Aurora provided 10% of performance improvements, 20% of energy reductions, and 28% of EDP improvements.

Aurora versus OMP_Dynamic: This specific implementation of OMP_Dynamic considers the last 15 minutes of execution to define the number of threads [33]. It does not use any search algorithm nor considers each parallel region in particular. For this reason, it is worse than the OpenMP Baseline in many cases. The advantage of potentially decreasing the overhead, since it is not often called, does not compensate the fact that it is not able to get near to the optimal number of threads. Considering the best case for each metric (Gmean) in Figure 6 and Table 5 Aurora reduced the execution time by 26% (medium input and the 4-core machine), energy consumption by 24% (medium input and the 32-core system) and EDP by 38% (small input and the 4-core system). In the overall (Gmean), Aurora was 11% faster, saved 17% of energy, and improved EDP by 32%.

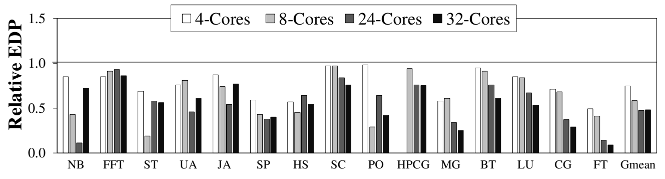
Aurora versus FDT: as observed in Figure 7 and Table 5, Aurora outperforms FDT in all scenarios. In the best case (small input set and the 24-core machine), Aurora improved



(a) Performance



(b) Energy



(c) EDP

Fig. 7: Aurora vs FDT (medium input): lower than 1.0 means that Aurora is better than the FDT

(Gmean) the execution time by 34%, energy consumption by 34%, and EDP by 56%. In the overall, the improvements were of 26%, 25%, and 45%, respectively. In very particular cases, results of FDT are similar as Aurora's when performance is considered. These are with applications that are in the group of scalability issues that FDT handles, such as FFT (off-chip bus saturation) and JA (synchronization). However, as already discussed, FDT ignores many fundamental hardware characteristics, converging to a non-optimal number of threads in many times. Moreover, the training phase of FDT executes each parallel region in single threaded mode until the standard deviation of the observed metric (memory bandwidth usage or synchronization time) is stable. It leads to a higher overhead for applications that present medium or high TLP. Because of this, in many cases FDT is worse than the baseline and OMP_Dynamic.

Varuna-PM: we selected one representative application from each benchmark class (NB, SC, ST, and FT) and implemented them using the programming model employed by Varuna. We executed them with two different amounts of threads (1566 and 10k threads, taken from [8]) on the 32-core machine. Table 6 shows that these versions are slower than the OpenMP baseline. In particular, for the NB benchmark, which has its scalability limited by data-synchronization, the greater the number of threads, the greater the time the threads spend synchronizing. This increases the execution time and energy consumption (as discussed in Section 3). It is important to emphasize that these results do not consider the improvements provided by the analytic engine and the manager system of Varuna. However, even if the analytic engine could improve performance by 15% and reduce energy consumption by 31% (values taken from [8]), it would not be enough to provide the same levels of performance and energy as the OpenMP baseline, in most cases. The main reason for these results is that Varuna was developed to be

TABLE 5: Summary of the results for the small input wrt the geometric mean: lower than 1.0 means that Aurora is better

	Performance			**Energy**			**EDP**		
	Baseline	***OMP***	***FDT***	***Baseline***	***OMP***	***FDT***	***Baseline***	***OMP***	***FDT***
4-Core	0.98	0.75	0.84	0.91	0.85	0.83	0.90	0.63	0.72
8-Core	0.91	0.83	0.78	0.89	0.85	0.80	0.80	0.71	0.62
24-Core	0.85	0.81	0.67	0.76	0.83	0.67	0.68	0.70	0.44
32-Core	0.88	0.84	0.69	0.66	0.78	0.73	0.54	0.62	0.48

TABLE 6: Times that Varuna-PM is slower than Baseline

		Small				**Medium**			
#Threads	**Metric**	***FT***	***SC***	***ST***	***NB***	***FT***	***SC***	***ST***	***NB***
1566	*Performance*	1.8	1.6	1.3	164.4	1.8	2.0	1.2	161.7
	Energy	1.4	1.1	1.0	33.0	1.4	1.6	1.1	45.0
	EDP	2.5	1.8	1.4	5421.6	2.6	3.1	1.3	7274.8
10k	*Performance*	3.1	6.6	3.6	1020.8	2.1	3.7	2.2	1026.9
	Energy	1.9	4.5	2.4	204.8	1.6	2.7	1.5	1026.0
	EDP	5.9	29.6	8.6	209072	3.3	10.1	3.3	1053593

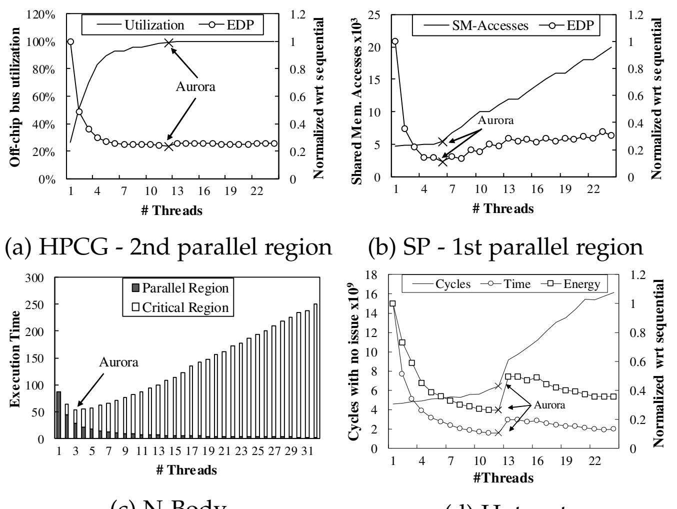
used in different kinds of applications (e.g., big data and ones that are recursively implemented), since it creates as many threads as possible. Therefore, Aurora and Varuna can be seen as two orthogonal approaches.

6.2 Handling Scalability

As a result of its run-time analysis, the search algorithm used by Aurora can detect the point in which the number of threads saturates any metric. As a first example, let us consider the off-chip bus saturation (as discussed in Section 3) and the execution of HPCG with medium input set on the 24-core system. This benchmark has two main parallel regions that are better executed with a different number of threads (Table 7) each. Figure 8a shows that when the second region is executed with more than 12 threads, the off-chip bus saturates (100% of utilization), and no further EDP improvements are achieved. By using its continuous monitoring and avoiding this saturation, Aurora was able to reduce the EDP of the whole application by 15% (Figure 6e). The very same behavior can be observed in FFT and ST (regardless the input set) and JA for the medium input set (Table 3), but at different improvement ratios.

In applications with high communication demands, there is an optimal point in which the overhead imposed by the shared memory accesses does not overcome the gains achieved by the parallelism exploitation, as discussed in Section 3. Aurora detected this point for all benchmarks in this class: SC, MG, and BT; LU (with small input); PO, UA, and SP (with medium input) (Table 3). For instance, let us consider the SP benchmark running on the 24-core system. This application has nine main parallel regions, in which each one is better executed with a different number of threads. Figure 8b shows that when the number of shared memory accesses from all threads in the first parallel region starts to increase (after six threads - primary y-axis), no further improvements in the EDP are achieved (secondary y-axis). As shown in the same Figure and the Table 7, Aurora found the best number of threads to execute this parallel region, providing EDP gains of 58% (Figure 6e).

Aurora similarly detects the point where the synchronization time overlaps the gains provided by TLP exploitation. This behavior can be observed in some benchmarks,



(a) HPCG - 2nd parallel region (b) SP - 1st parallel region

(c) N-Body (d) Hotspot

Fig. 8: Scalability behavior

such as n-body (NB) with small or medium input, or Jacobi(JA) and SP with small input set (Table 3). In these benchmarks, the higher the number of threads, the greater the time spent synchronizing, which can worsen the results, as already discussed in Section 3. The n-body benchmark with the medium input set executing on the 32-core system (Figure 8c) can be discussed as an example. When increasing the number of threads from 1 to 3, performance improves. However, from this point on, the time that the threads spend synchronizing overcomes the gains achieved by the parallelism exploitation (Figure 8c), increasing the energy consumption and EDP of the whole application. As demonstrated in Table 7, by avoiding this extra overhead in the synchronization time and setting the right number of threads, Aurora reduced the execution time by 79%, energy by 89%, and EDP by 98%.

Aurora also converges to the best number of threads for applications that are negatively influenced by the issue-width saturation. Some examples are: Hotspot (HS), FT, and CG with any input set; and UA and PO with the small input (Table 3). Let us consider the Hotspot benchmark with the medium input set executing on the 24-core system. In this case, the optimal number of threads for EDP is 12 (see Table 7). As Figure 8d shows, when increasing the number of threads from 12 to 13, the number of cycles that the threads spend without issuing any instruction abruptly increases. Therefore, performance decreases and energy consumption increases (Figure 8d). Once more, by avoiding the excessive increment in the number of threads, Aurora improved performance by 21% and reduced EDP and energy by 44% and 25%, respectively (Figure 6).

Finally, as discussed in Sections 1 and 3, it is important to note that there are cases in which the characteristic that influences the thread scalability changes according to the input set (Table 3). As a specific example, let us consider JA application. When it is executed with the small input set, the time that the threads spend synchronizing limits the application scalability. When executed with the medium input set, the off-chip bus becomes the main limiting factor because of the larger amount of data available.

6.3 Costs of the Learning Curve

Table 8 depicts (in percentage) how the results obtained by Aurora differs from the **Oracle** solution. We consider the geometric mean (gmean) of the entire benchmark set for each processor and metric. The difference between ours and the optimal solution reflects the overhead of our technique, so we can measure the cost of the learning curve. As one can observe, these overheads are not very significant when compared to the best possible solution. The overhead is originated from two situations: the execution of the search algorithm itself; and the execution of a given parallel region with a number of threads that is not the ideal, while the search algorithm is trying different possibilities to converge to the ideal number. Aurora showed higher overheads in the following situations: (*i*) The best result is achieved with either the maximum number of threads or a number close to it, which is the case of the FT and CG benchmarks executing on the 24 and 32-core systems. (*ii*) The parallel region has a relatively small number of interactions but executes for a significant time, such as HPCG. (*iii*) Applications that have short execution time (i.e., less than 10 seconds), such as the MG. Its Oracle version takes only 1.45 seconds to execute on the 32-core system with the small input set. (*iv*) Applications with many parallel regions, in which most of them have a low workload, as in the UA benchmark. UA has 54 parallel regions, and 44 of them take less than 0.5 seconds to execute regardless the target processor.

Moreover, the higher the number of hardware threads available in the system, the greater the space exploration that must be covered. However, even though the overhead of the search algorithm increases, it does so in small rates, as can be observed when one compares the averages of the 24- and 32-core systems to the 4- and 8-core ones.

We also measured the execution time of the hill climbing algorithm alone (considering only the specific calls to the respective function of the search algorithm). We consider the 32-core machine, which is the one that has the largest design space to be explored. Our experiments show that it presents an overhead of only 0.020% w.r.t. to the total execution time (geometric mean considering all benchmarks and inputs). In the worst case (MG benchmark, small input set), the search algorithm adds only 0.267% to the total execution time.

6.4 Distinct Approaches, Similar Convergence

We have implemented a Genetic Algorithm (GA) to demonstrate how Aurora's hill-climbing fares against such classes of heuristics. GA is a search metaheuristic based on natural selection and genetics. It uses a concept of a population, which is a set of individual solutions (chromosomes), that can evolve to an optimum solution through generations. As GA requires minimum previous information on the problem at hand, it is widely used in many different situations. For our experiments, we started with a random population with a fixed size of 30 to 40 individuals (depending on the application). We modeled the chromosome to represent the global solution (i.e. the number of threads for each parallel region). Thus, we had to run the entire application for each new chromosome. Our population evolved by randomly selecting new chromosomes, giving higher chances for those with the best results in EDP. While applying the crossover

TABLE 7: Number of threads found by Aurora

		Performance				Energy				EDP			
		4-cores	*8-cores*	*24-cores*	*32-cores*	*4-cores*	*8-cores*	*24-cores*	*32-cores*	*4-cores*	*8-cores*	*24-cores*	*32-cores*
NB	S	4	4	4	3	3	3	4	3	3	3	4	3
	M	4	4	4	3	3	3	3	3	4	4	4	3
FFT	S	4	2	4	6	1	1	2	4	2	1	2	4
	M	4	2	6	14	1	1	2	4	2	2	2	4
ST	S	4	2	4	12	1	1	4	4	2	1	4	6
	M	4	2	4	12	1	1	4	4	2	1	4	6
UA	S	4,4,4,4,4 4,4,4,4,4	8,3,4,8,8 8,4,2,2,2	24,6,12,12,24 12,12,6,6,6	32,11,14,32,32 32,10,12,26,10	4,1,2,4,4 4,1,1,1,1	8,1,2,4,8 8,2,1,1,1	24,6,4,11,24 12,6,4,6,4	32,6,6,32,32 32,6,6,6,6	4,2,2,4,4, 4,1,1,1,1	8,1,2,8,8, 8,2,1,2,2	24,6,6,24,12, 12,6,6,6,6	32,10,10,32,32, 32,6,6,6,8
	M	4,4,4,4,4 4,4,4,4,4	8,2,4,8,8 8,5,2,5,2	24,4,8,24,24 24,6,4,11,8	32,11,14,16,32 32,6,13,11,12	4,1,2,4,4 4,1,1,1,1	8,1,2,4,8 4,1,1,1,1	24,2,4,11,24 12,4,4,2,2	32,4,6,13,32 14,4,4,4,4	4,2,2,4,4, 4,2,2,2,2	8,1,2,8,8, 8,5,1,1,1	24,4,6,24,24, 12,6,4,4,4	32,4,6,32,32, 32,6,4,4,4
JA	S	3	3	12	28	2	2	6	14	2	2	6	15
	M	3	2	10	12	2	2	4	6	2	2	4	6
SP	S	2,4,4,2,3 2,3,3,4	2,2,4,2,4 2,4,2,2	6,6,12,6,12 6,12,6,6	12,26,15,32,15 32,15,6,8	2,1,3,1,3 1,3,1,1	2,1,3,1,3 1,3,1,1	6,4,10,4,10 4,10,4,2	6,4,13,4,13 4,10,4,4	2,2,3,2,3 2,3,2,2	2,1,4,1,4 1,3,1,1	6,4,12,6,12 6,10,4,4	6,4,15,4,15 4,15,4,4
	M	3,4,2,2,2 2,2,4,4	2,5,2,2,2 5,2,5,2	6,6,6,20,6 20,6,6,24	12,10,10,32,10 14,8,10,24	2,1,2,1,2 1,2,1,1	1,1,2,1,2 1,2,1,1	4,4,6,4,6 4,6,4,4	6,4,8,4,8 4,8,4,4	2,2,2,2,2 2,2,2,2	2,1,2,2,2 2,2,5,1	6,6,6,6,6 6,6,4,4	6,6,8,4,8 4,8,6,4
HS	S	4	4	12	16	4	4	10	16	4	4	12	16
	M	4	4	12	16	4	4	12	16	4	4	12	16
SC	S	4	8	23	9	4	8	7	9	4	8	23	9
	M	4	7	9	15	4	7	7	9	4	7	7	15
PO	S	4	8	12	16	4	8	12	16	4	8	12	16
	M	4	7	12	16	4	3	12	16	4	3	12	16
HPCG	S	4,4	2,5	10,10	8,30	1,2	1,2	4,6	6,6	2,3	2,2	6,6	8,8
	M	–	2,4	6,12	8,32	–	1,3	4,8	4,8	–	2,3	4,12	6,8
MG	S	2,4,3,3	2,3,3,2	6,8,8,10	8,10,10,12	2,2,2,1	2,2,2,1	6,6,6,4	6,8,8,4	2,3,2,2	2,3,3,2	6,8,8,6	8,8,8,6
	M	4,4,4,2	3,4,3,2	12,10,8,10	16,16,12,6	2,2,2,1	2,2,2,1	4,6,6,4	6,8,6,6	2,3,3,2	2,3,3,2	6,6,6,4	6,8,8,6
BT	S	2,4,4,4	2,8,8,8	6,23,22,23	10,28,31,30	2,4,4,4	2,8,4,8	6,22,22,22	6,28,28,29	2,4,4,4	2,8,8,8	6,22,22,22	8,28,28,29
	M	3,4,4,4	2,8,8,8	8,24,24,24	10,32,32,32	2,4,4,4	1,8,8,8	4,23,23,23	6,32,32,32	2,4,4,4	2,8,8,8	4,24,23,23	8,32,32,32
LU	S	4,4	8,8	22,22	27,31	4,3	4,8	10,22	10,15	4,4	8,8	22,22	14,26
	M	4,4	8,4	12,24	14,32	4,2	4,2	10,24	10,32	4,3	4,3	12,24	14,32
CG	S	4,4	8,8	24,24	32,32	4,4	8,8	24,12	16,16	4,4	8,8	24,24	32,16
	M	4,4	8,8	24,24	31,32	4,4	8,8	24,24	32,14	4,4	8,8	24,24	32,32
FT	S	4,4,4,4,4	8,4,4,4,5	24,24,24,24,8	32,32,32,32,14	4,4,4,4,2	8,4,4,4,2	24,24,12,12,6	32,32,16,16,6	4,4,4,4,3	8,4,4,4,2	24,24,12,24,8	32,32,32,30,8
	M	4,4,4,4,4	8,7,4,8,5	24,24,24,24,12	32,32,32,32,32	4,4,4,4,2	8,4,4,4,2	24,24,22,22,6	32,32,32,32,6	4,4,4,4,3	8,7,4,4,2	24,24,24,22,12	32,32,32,32,8

TABLE 8: Learning overheads (%) for Aurora wrt the geometric mean for all the benchmarks

	Performance				Energy				EDP			
	4-Core	*8-Core*	*24-Core*	*32-Core*	*4-Core*	*8-Core*	*24-Core*	*32-Core*	*4-Core*	*8-Core*	*24-Core*	*32-Core*
S	0.7	0.9	2.9	9.9	0.9	1.4	0.9	4.1	1.8	2.1	2.6	6.6
M	1.0	0.7	2.4	3.1	0.9	2.0	1.6	3.0	2.8	1.9	2.5	5.4

guarantees the propagation of the best individuals characteristics, the mutation ensures the whole solution space can be searched. The probability for the crossover and the mutation to happen is of 0.9 and 0.001, respectively.

While the GA performs a global search, trying for different combinations for each parallel region, Aurora splits the problem into local searches (one for each region). The GA does find local optimums and escape them through the generations. However, it tends to perform worse when the space exploration is too large, represented by applications with many parallel regions. GMEAN_GA in Fig 9 shows the EDP (y-axis) given by the geometric mean of our benchmark set through the generations (x-axis). We also include the geometric mean of Aurora's execution (GMEAN_AURORA). All the results are normalized by the Oracle execution (represented as the constant line in 1). GMEAN_GA and GMEAN_AURORA lines clearly show that the GA converges to a similar result as Aurora over the generations (Figure 9 truncates at generation 26), and when one considers the geometric mean, Aurora performs slightly better. We can also see the worst (MAX_GA and MAX_AURORA) and the best (MIN_GA and MIN_AURORA) cases from the executed applications for the GA and Aurora, respectively. As observed, the GA can find better solutions than Aurora in its best case, but cannot achieve Aurora's result in the worst

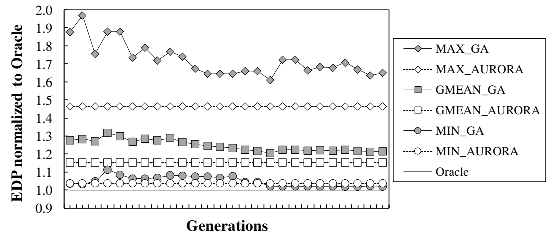


Fig. 9: Genetic Algorithm convergence and Aurora

case (an application with many parallel regions) because of its coarse tuning characteristics. Furthermore, Figure 9 omits the GA training time, which can easily exceed several hours and must be re-executed when anything in the system changes, such as the input size or microarchitecture. Aurora, on the other hand, can quickly adapt at run-time.

6.5 Limitations of Aurora

As already extensively discussed throughout this paper, Aurora works only with OpenMP and, more specifically, with the *OpenMP parallel directive*. However, it is important to remember that, as discussed in Section 4.1, *sections* and *tasks* are seldom used. In cases where there are parallel regions implemented in a different API or using unsupported OpenMP directives, Aurora will still work to find the ideal number of threads for each *OpenMP parallel directive* region. Therefore, it will not influence the execution of those other parallel regions. Moreover, there are other scenarios where Aurora will also present some limitations: (*i*) The application is being parallelized to run on distributed systems, using some hybrid approach, in which the iterations of the outer loop are distributed to different nodes using a message passing library, and the inner loops are parallelized with OpenMP. For such hybrid approaches, Aurora will work for the OpenMP regions. (*ii*) Multiple parallel loops are embedded inside an OpenMP parallel directive, using the clause *collapse*, which specifies how many loops in a nested loop should be collapsed into one large iteration space. In this scenario, Aurora will work to optimize the number of threads of the nested parallel loop. (*iii*) The programmer wants to distribute the thread across the available sockets in an SMP architecture (e.g., match the number of threads of the outer loop with the number of sockets). Such applications will not benefit from Aurora's search algorithm because the number of threads for each parallel region is defined a priori (statically) by the programmer. This is usually done by very experienced programmers and is not significantly used nowadays.

Another requirement for Aurora is an interface to access performance counters for power dissipation and execution time. Current Intel platforms (all models manufactured after the Sandy Bridge microarchitecture, including i3, i5, i7, i9, Atom, Xeon, and Xeon Phi processors), AMD (some models from the Bobcat and Bulldozer family, and all models from the Zen family), and IBM Power9 Family provide all the hardware counters for execution time and power dissipation that Aurora needs. Some architectures (such as ARM) currently provide only the former, which prevent automatic optimization for Energy and EDP. An alternative would be to estimate power based on the available performance counters, although this could lead to potentially wrong decisions by the search algorithm.

As observed in the experiments, Aurora presented its worst results executing applications with high TLP. In such cases, executing with the highest possible number of threads, as the Baseline does, is already the best solution. Therefore, Aurora will waste time (*i*) with its learning algorithm and (*ii*) executing the parallel regions with non-optimal number of threads during this process. In conclusion, when most parallel regions of an application have high TLP, Aurora may bring some small overhead compared to the baseline. Large input sets tend to alleviate this overhead, since each parallel region will proportionally execute more times using the ideal number of threads and less time in the process of learning.

7 Conclusion

We have presented Aurora, an approach capable of automatically finding, at run-time, the optimal number of threads for each parallel region. It is completely transparent to both designer and end user: given an OpenMP application binary, Aurora optimizes it without any code changes, transformation or recompilation, by simply setting an environment variable in Linux OS. We have shown that Aurora can optimize distinct OpenMP applications targeting different metrics with an almost negligible overhead. As future work, we will enhance Aurora to minimize the training overhead observed in applications with TLP degrees. We will also assess the technique in a wider diversity of platforms (e.g., AMD and ARM processors), considering the availability of the required performance counters.

References

[1] P.-F. Dutot, Y. Georgiou, D. Glesser, L. Lefevre, M. Poquet, and I. Rais, "Towards energy budget control in hpc," in *IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*. NJ, USA: IEEE, 2017, pp. 381–390.

[2] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps," *SIGARCH Comput. Archit. News*, vol. 36, no. 1, pp. 277–286, 2008.

[3] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "Mise: Providing performance predictability and improving fairness in shared main memory systems," in *IEEE HPCA*, 2013, pp. 639–650.

[4] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. NY, USA: ACM, 2012, pp. 223–234.

[5] S. E. Raasch and S. K. Reinhardt, "The impact of resource partitioning on smt processors," in *Int. Conf. on Parallel Architectures and Compilation Techniques*, 2003, pp. 15–25.

[6] H. M. Levy, J. L. Lo, J. S. Emer, R. L. Stamm, S. J. Eggers, and D. M. Tullsen, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *ISCA*, 1996, pp. 191–191.

[7] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, 2013.

[8] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *ACM SIGPLAN PLDI*. NY, USA: ACM, 2014, pp. 169–180.

[9] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via os level monitoring," in *IEEE Int. Symp. on Workload Characterization*. DC, USA: IEEE Computer Society, 2011, pp. 116–125.

[10] C. Jung, D. Lim, J. Lee, and S. Han, "Adaptive execution techniques for smt multiprocessor architectures," in *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. NY, USA: ACM, 2005, pp. 236–246.

[11] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online power-performance adaptation of multithreaded programs using hardware event-based prediction," in *Int. Conf. on Supercomputing*, 2006, pp. 157–166.

[12] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos, "Prediction-based power-performance adaptation of multithreaded scientific codes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 10, pp. 1396–1410, 2008.

[13] G. Chadha, S. Mahlke, and S. Narayanasamy, "When less is more (limo):controlled parallelism for improved efficiency," in *Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*. NY, USA: ACM, 2012, pp. 141–150.

[14] A. Raman, A. Zaks, J. W. Lee, and D. I. August, "Parcae: A system for flexible parallel execution," in *ACM SIGPLAN PLDI*. NY, USA: ACM, 2012, pp. 133–144.

[15] J. Lee, H. Wu, M. Ravichandran, and N. Clark, "Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 270–279, 2010.

[16] A. K. Porterfield, S. L. Olivier, S. Bhalachandra, and J. F. Prins, "Power measurement and concurrency throttling for energy reduction in openmp programs," in *IEEE IPDPS*, 2013, pp. 884–891.

[17] F. Alessi, P. Thoman, G. Georgakoudis, T. Fahringer, and D. S. Nikolopoulos, *Application-Level Energy Awareness for OpenMP*. Cham: Springer, 2015, pp. 219–232.

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

[19] A. Porterfield, R. Fowler, and M. Neyer, "Maestro: Dynamic runtime power and concurrency adaptation," in *Workshop on Managed Many-Core Systems*, 2008.

[20] R. A. Shafik, A. Das, S. Yang, G. Merrett, and B. M. Al-Hashimi, "Adaptive energy minimization of openmp parallel applications on many-core systems," in *Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures*. NY, USA: ACM, 2015, pp. 19–24.

[21] R. A. Shafik, A. K. Das, S. Yang, G. V. Merrett, and B. Al-Hashimi, "Thermal-aware adaptive energy minimization of openmp parallel applications," 2015.

[22] S. Sridharan, G. Gupta, and G. S. Sohi, "Holistic run-time parallelism management for time and energy efficiency," in *Int. Conf. on Supercomputing*. NY, USA: ACM, 2013, pp. 337–348.

[23] A. F. Lorenzon, J. D. Souza, and A. C. S. Beck, "Laant: A library to automatically optimize edp for openmp applications," in *DATE*, March 2017, pp. 1229–1232.

[24] J. Benesty, J. Chen, Y. Huang, and I. Cohen, *Pearson Correlation Coefficient*. Berlin, Heidelberg: Springer, 2009, pp. 1–4.

[25] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated control for energy-efficient and heterogeneous memory systems," in *IEEE HPCA*, 2013, pp. 424–435.

[26] T. C. Deepack S., K. Varaganti, R. Suresh, R. Garg, and R. Ramamoorthy, "Comparison of parallel programming models for multicore architectures," in *IEEE IPDPS*, 2011, pp. 1675–1682.

[27] S. Seo, G. Jo, and J. Lee, "Performance characterization of the nas parallel benchmarks in opencl," in *IEEE Int. Symp. on Workload Characterization*, 2011, pp. 137–148.

[28] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, *SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–10.

[29] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[30] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE Int. Symp. on Workload Characterization*. DC, USA: IEEE Computer Society, 2009, pp. 44–54.

[31] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*. NY, USA: ACM, 2008, pp. 72–81.

[32] M. J. Abraham, T. Murtola, R. Schulz, S. Pall, J. C. Smith, B. Hess, and E. Lindahl, "Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 12, pp. 19 – 25, 2015.

[33] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[34] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 3, pp. 13–17, 2012.

[35] D. Hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *IEEE Int. Symp. on Performance Analysis of Systems and Software*, 2013, pp. 194–204.

[36] A. Johnson and S. Jacobson, "On the convergence of generalized hill climbing algorithms," *Discrete Applied Mathematics*, vol. 119, no. 1, pp. 37 – 57, 2002, special Issue devoted to Foundation of Heuristics in Combinatoria l Optimization.

[37] D. Taborda and L. Zdravkovic, "Application of a hill-climbing technique to the formulation of a new cyclic nonlinear elastic constitutive model," *Computers and Geotechnics*, vol. 43, pp. 80 – 91, 2012.

[38] 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, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks&mdash;summary and preliminary results," in *ACM/IEEE Conf. on Supercomputing*. NY, USA: ACM, 1991, pp. 158–165.

[39] S. Bhatt, M. Chen, C. Y. Lin, and P. Liu, "Abstractions for parallel n-body simulations," in *Scalable High Performance Computing Conf.*, Apr 1992, pp. 38–45.

[40] W. Petersen and P. Arbenz, *Introduction to Parallel Computing : A practical guide with examples in C*, ser. Oxford Texts in Applied and Engineering Mathematics. OUP Oxford, 2004.

[41] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pp. 19–25, 1995.

[42] M. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education, 2004.

[43] J. Dongarra, M. A. Heroux, and P. Luszczek, "Hpcg benchmark: A new metric for ranking high performance computing systems," *Knoxville, Tennessee*, 2015.

[44] T. Willhalm, R. Dementiev, and P. Fay, "Intel performance counter monitor - a better way to measure cpu utilization," Intel, Tech. Rep., 2017.

[45] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, 2000.

[46] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 302–313, 2010.

[47] C. Christmann, E. Hebisch, and A. Weisbecker, "Oversubscription of computational resources on multicore desktop systems," in *Int. Conf. on Multicore Software Engineering, Performance, and Tools*, ser. MSEPT'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 18–29.

Arthur Francisco Lorenzon received his B.S. in Computer Science from UNIPAMPA, Brazil, in 2012; and the M.Sc. and Dr. degrees from UFRGS, Brazil, in 2014 and 2018, respectively. Currently, he is a professor at UNIPAMPA. His areas of interest include parallelism exploitation aiming energy efficiency and the development of approaches to automate the TLP exploitation. For more information, please visit www.inf.ufrgs.br/ aflorenzon

Charles Cardoso de Oliveira received his Computer Engineering degree from UnB, Brazil, in 2016. Presently, he is an MSc student at UFRGS. His primary research interests include the TLP exploitation in multicores systems and dynamic power management techniques.

Jeckson Dellagostin Souza received his Computer Engineering and MSc degrees from UFRGS, Brazil, in 2014 and 2015, respectively. Presently, he is a PhD student at UFRGS and his primary research interests include heterogeneous multicore environments, binary compatibility, and reconfigurable architectures, particularly focusing on power reduction techniques. For more information, please visit http://www.inf.ufrgs.br/ jdsouza/

Antonio Carlos Schneider Beck Filho received his Dr. degree from UFRGS, Brazil, in 2008. Currently, he is a professor at the Applied Informatics Department at the Informatics Institute of UFRGS, in charge of Embedded Systems and Computer Organization disciplines at the undergraduate and graduate levels. His primary research interests include computer architectures and embedded systems design, focusing on power consumption. For more information, please visit: www.inf.ufrgs.br/caco.