Chapter 5
Case Study: DCT with Aurora

5.1 The Need for Adaptability and Transparency

Figure 5.1a shows the usual way of finding the best number of threads to run a parallel application [68, 70, 71]. First, the source code is compiled and executed n times with a different number of threads, where n is the number of available cores in the processor microarchitecture. In this phase, one also has to consider that each application may contain p parallel regions, in which each region can be better executed with a different number of threads. Therefore, the search space corresponds to the execution of n^p combinations of number of threads for each application, where p is greater or equal to 1. After the offline training period, the best configuration is selected, and the next executions will be performed with the configuration found in this step.

In order to understand the huge design space exploration concerning the selection of the ideal number of threads to run a parallel application, let us consider an application with 5 parallel regions running on a 32 multicore processor. In such a case, there will be 32^5 possible combinations, which results in *33,554,432* executions before selecting the ideal configuration. Supposing that each execution would spend 1 min (60 s), it would be necessary approximately 65 days to find the best configuration. However, if there is any change in the application behavior (e.g., input set size) or the execution environment, the executions must be performed again.

Therefore, Aurora was developed to cope with the challenge of selecting the best number of threads to execute each parallel region of an OpenMP application [73]. It automatically finds, at runtime 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 readapt according to a change in the behavior of a particular parallel region during program execution. Because of its dynamic adaptability, Aurora deals with the intrinsic characteristics of the application as well as the microarchitecture on which it will execute; it also takes into account

A. Francisco Lorenzon, A. C. S. Beck Filho, *Parallel Computing Hits the Power Wall*,
SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-030-28719-1_5

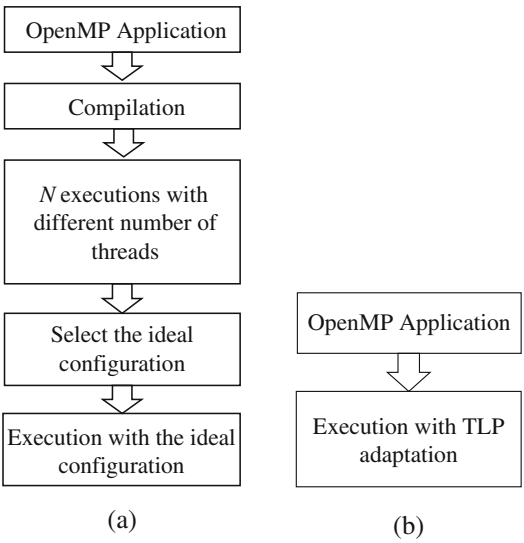

Fig. 5.1 Adaptation of OpenMP applications. (**a**) Brute force. (**b**) Aurora

the current input set and application changes at runtime, 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 it without any code changes. Therefore, existing 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.

5.2 Aurora: Seamless Optimization of OpenMP Applications

5.2.1 Integration to OpenMP

As already described in Chap. 2.1.2, parallelism in OpenMP is exploited through the insertion of directives in the sequential code that inform the compiler how and which parts of the application should be executed in parallel [22]. OpenMP provides three ways for exploiting parallelism: parallel loops, sections, and tasks. Parallel sections and tasks are only used in very particular cases: when the programmer must distribute the workload over the threads in a similar way as PThreads, and when the application uses recursion (i.e., sort algorithms), respectively. On the other hand, parallel loops are used to parallelize applications that work on multidimensional 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

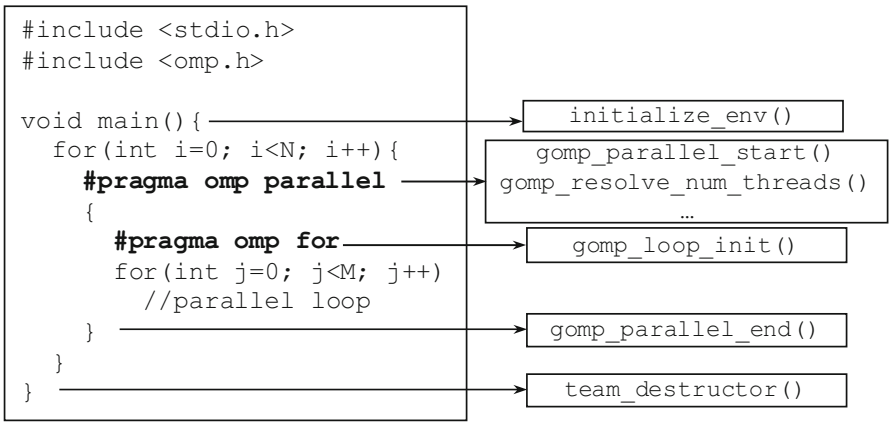

Fig. 5.2 OpenMP execution environment with the respective *libgomp* functions

(all the aforementioned benchmark sets are implemented in this way). For now, Aurora works to optimize parallel loops and does not influence in any way other 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 Runtime 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. In order to better understand how Aurora works, let us first consider Fig. 5.2, which illustrates the regular way for parallelizing an iterative application with parallel loops [22] and the respective main functions implemented by *libgomp*. When the program starts executing, the *initialize_env*() function is called, which is responsible for initializing all the environment variables used by OpenMP during the application 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. Within the parallel region, the directive *#pragma omp for* indicates the loop that must be parallelized. At the end of the parallel region, the function *gomp_parallel_end*() joins the threads and finalizes the parallel region environment. Finally, when the application ends, *team_destructor()* concludes the entire OpenMP environment.

Aurora functionalities were split into four functions (discussed in details next). They were incorporated into the *libgomp* functions previously mentioned. Algorithm 1 depicts the modifications done in the source code of each function in order to support Aurora functions. *libgomp* also has another function called *gomp_loop_init*(), which was not modified as its job is to distribute the workload between the already defined threads.

***auroraInitEnv*()** is responsible for recognizing the Aurora optimization target defined by the environment variable (OMP_AURORA) and for initializing the necessary data structures, libraries, and variables used to control the search

Algorithm 1 OpenMP functions that were modified to integrate Aurora optimization

```
1: function INITIALIZE_ENV(void)
2:     Initialization of OpenMP environment (variables, CPU affinity, wait policy, etc.)
3:     if OMP_AURORA is defined then
4:         aurora_metric ← get the value defined by the user in OMP_AURORA
5:         aurora_start_search ← get the value defined by the user in AURORA_START
6:         auroraInitEnv(aurora_metric, aurora_start_search)
7:     end if
8: end function

9: function GOMP_PARALLEL_START(*fn, *data, num_threads)
10:    ptrToRegion ← gets pointer to fn address region
11:    if Aurora is Enabled then
12:        num_threads ← auroraResolveNumThreads(ptrToRegion)
13:    else
14:        num_threads ← gomp_resolve_num_threads(num_threads, 0)
15:    end if
16:    gomp_team_start(fn, data, num_threads, 0, gomp_new_team(num_threads))
17: end function

18: function GOMP_PARALLEL_END(void)
19:    if OMP_AURORA is defined then
20:        auroraEndParallelRegion();
21:    end if
22:    finalize parallel region environment
23:    gomp_team_end()
24: end function

25: function TEAM_DESTRUCTOR(void)
26:    if OMP_AURORA is defined then
27:        auroradestructEnv();
28:    end if
29:    pthread_key_delete(gomp_thread_destructor)
30: end function
```

algorithm (described in Chap. 5.2.2). The pseudocode of this function can be seen in Algorithm 2. ***auroraInitEnv*** is called from the original *initialize_env*() only if Aurora optimization is enabled, as presented in lines 3–7 in Algorithm 1. If OMP_AURORA is not defined, the OpenMP execution follows its standard behavior.

***auroraResolveNumThreads*()** sets the number of threads that execute each parallel region based on the current state of the search algorithm. Also, it initializes the counters for collecting data from the execution environment of the current parallel region. Algorithm 3 depicts the pseudocode of this function: if the parallel region is a new region, the search algorithm will start the search from the initial state (*S0*) and with the number of threads defined either by the environment variable AURORA_START or by 2 that is the standard value used by Aurora. Also, if the search algorithm is in the *END* state, the best number of threads (*bnt*) found to

Algorithm 2 Initialization of Aurora environment

```
1: function AURORAINITENV(metric, startSearch)
2:     numCores ← get the total number of cores through sysconf
3:     threadStartSearch ← get the number of threads defined to start the search
4:     Initialize hardware counters to get the parallel region behavior
5:     for i in maxNumberOfParallelRegions do
6:         Initialize the variables used to monitor/control the parallel region i
7:         i.e., startSearch, metric, actualstate
8:     end for
9: end function
```

execute a parallel region is returned. Otherwise, the actual number of threads (*ant*) is returned. ***auroraResolveNumThreads*** is called by the *gomp_parallel_start*()[1] when Aurora is active, replacing the original *gomp_resolve_num_threads*() function, as depicted in Algorithm 1.

Algorithm 3 Setting up the number of threads

```
1: function AURORARESOLVENUMTHREADS(ptrToRegion)
2:     idR ← get the id of the parallel region from ptrToRegion
3:     if idR is a newRegion then
4:         auroraKernel[idR].state ← S0
5:     end if
6:     switch auroraKernel[idR].state do
7:         start monitoring the parallel region behavior
8:         case END
9:             return auroraKernel[idR].bnt
10:        case Default
11:            return auroraKernel[idR].ant
12: end function
```

***auroraEndParallelRegion*()** is executed after the parallel region to get its execution time, energy, or EDP, depending on the optimization metric defined by the user. 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 to get energy and power consumption of CPU-level components [40], while the APM library is used for AMD processors [39]. Such functions and libraries were incorporated to Aurora, being totally transparent to the user. That is, there is no need to make any modifications in the Operating System (package installation, kernel recompilation, etc.) to use them.

[1] *GOMP_parallel_start* is also named as *GOMP_parallel*.

Using either one of the objectives of execution time, energy, or EDP, *auroraEndParallelRegion*() performs one step of the search algorithm (which is explained in the next subsection) and, according to this algorithm, it defines the number of threads that will be used for the execution of this parallel region in the next iteration. *auroraEndParallelRegion*() is implemented inside *gomp_parallel_end*() function, and it is called when Aurora is active, as depicted in Algorithm 1.

***auroraDestructEnv*()** concludes and destroys Aurora environment at the end of application execution, when Aurora is active (Algorithm 1). It was implemented inside *team_destructor*() OpenMP function.

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 Operating System, the thread management system of Aurora is used instead of the original OpenMP functions. This environment variable can be configured to the following values (and, therefore, optimization metrics): performance, energy, or EDP. If the variable is not set, Aurora will not influence the execution of that OpenMP application (i.e., the application executes with the original OpenMP functions). In this way, any existing binary code can benefit from Aurora without any modifications or need for recompilation.

5.2.2 Search Algorithm

The heuristic used by Aurora is divided into two phases. The first one investigates the scalability of the parallel region and reduces the size of the space exploration, exponentially increasing the number of threads (i.e., 2, 4, 8, 16, ...) while there are potential improvements (states *Initial*, *Doubling*, and *Exponential* in Algorithm 4, Fig. 5.3, and Table 5.1). The second phase performs a hill-climbing based algorithm in the interval of threads defined in the first phase (states *Exponential*, *Search*, and *Lateral*). 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 thread 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 [52, 116]. 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.

Algorithm 4 Search algorithm implemented by Aurora

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

Basically, the algorithm works as follows (Algorithm 4): the search starts by the *Initial* state (*line 5*), where the initial number of threads (*threadStartSearch*) and the current number of threads (*currentNT*) are defined. Then, the parallel region is executed with the initial number of threads (e.g., 2 threads) and the state changes to

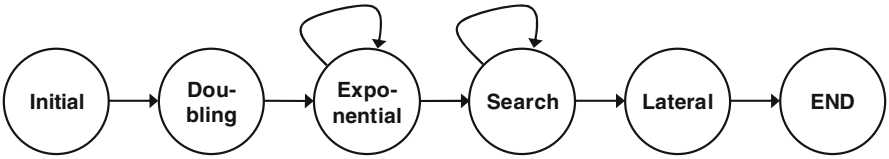

Fig. 5.3 States and transitions of the search algorithm

Table 5.1 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 S0 and S1, 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 S2. 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

Doubling. In this state (*line 8*), the best result so far (*bestMetricMsmt*) is updated with the result obtained by the execution with the number of threads defined in the *Initial* state, the number of threads is doubled, and state changes to *Exponential*. In this state (*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 hardware threads available is not reached (*lines 15–32*). Then, the state changes to *Search*. Once in there (*line 33*), Aurora knows the interval of potential candidates for the ideal number of threads, which is in the range between the last number of threads executed and the best number of threads found so far and starts the second phase.

To better understand the second phase, let us consider that the interval of potential candidates lies in the range of 8–16 threads. Then, the algorithm searches for the best number of threads in this range. It will start executing with 12 threads (the average amount between 8 and 16) and then compares to the best result so far to decide the next range (which will be between 8 and 12 or 12 and 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 a variation of 30% was considered, the search algorithm starts its execution again.

5.3 Evaluation of Aurora

5.3.1 Methodology

Fifteen applications written in C/C++ and parallelized with OpenMP from assorted benchmarks suites were chosen according to the scalability issues discussed in Sect. 1.2:

- **Seven kernels** from the NAS Parallel Benchmark [4]: *block tri-diagonal solver* (BT), *conjugate gradient* (CG), *discrete 3D fast Fourier transform* (FT), *lower–upper Gauss–Seidel solver* (LU), *multigrid 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, the OpenMP-C version developed in [103] is considered.
- **Two applications** from the Rodinia Benchmark Suite [23]: *hotspot* (HS) which iteratively solves a series of differential equations and *streamcluster* (SC), a dense linear algebra algorithm for data mining.
- **Six applications** from different domains: *n-body* (NB)—computes a simulation of a dynamical system of particles [10]; *fast Fourier transform* (FFT)—calculates the discrete Fourier transform of a given sequence [89]; *STREAM* (ST)—measures sustainable memory bandwidth [78]; *Jacobi* (JA) method iteration—computes the solutions of a diagonally dominant system of linear equations [94]. *Poisson* (PO)—computes an approximate solution to the Poisson equation in a rectangular region [94]; and the *high-performance conjugate gradient* benchmark (HPCG), a stand-alone code that measures the performance of basic operations [32].

Two different input sets for each benchmark were considered: small and medium. Table 5.2 depicts the Pearson correlation between each scalability issue (discussed in Chap. 2) and the application. As can be observed, the chosen applications do not scale for different reasons, according to Sect. 1.2. All the data used for the scalability analysis was obtained directly from hardware using Intel Performance Counter Monitor (PCM) [124], Intel Parallel Studio, and Performance Application Programming Interface (PAPI) [15].

As one can note in Fig. 5.4, the chosen benchmarks also cover a wide range of different TLP behaviors. The TLP was measured as defined by the authors in [12]: the average amount of concurrency exhibited by the program during its execution when at least one core is active, and it is expressed in Equation 5.1. c_i is the fraction of time that i cores are concurrently running different threads, n is the number of cores, and $1 - c_0$ is the non-idle time fraction. The closer this value is to 1.0 (normalized to the total number of cores available), the more TLP is available [12].

$$TLP = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0} \tag{5.1}$$

Table 5.2 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 saturation	−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 saturation	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 memory accesses	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-synchronization	**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 saturation	−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 saturation	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 memory accesses	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-synchronization	**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

Bold values highlight Pearson's correlation of scalability issue that affects each application

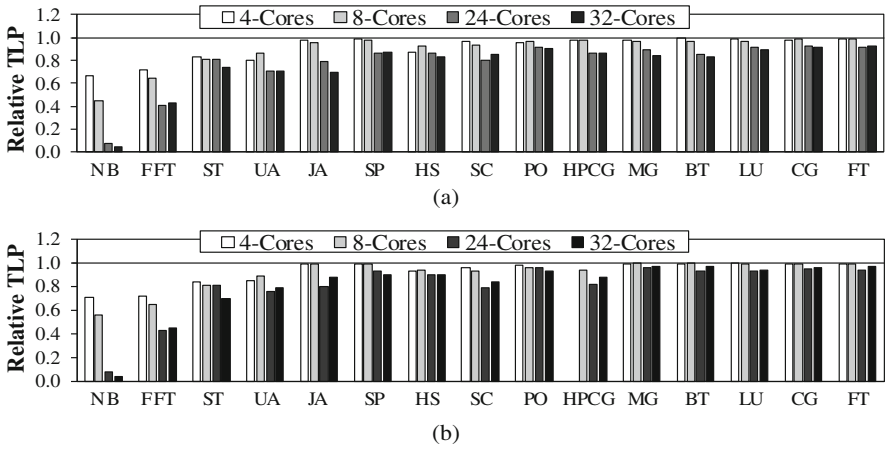

Fig. 5.4 TLP available for each benchmark—normalized w.r.t. the maximum number of threads in each processor. (**a**) Small input set. (**b**) Medium input set

Table 5.3 Main characteristics of each processor

	Intel Core		Intel Xeon	
	i5-4460	i7-6700	E5-2630	E5-2640
Microarchitecture	Haswell	Skylake	Sandy Bridge	Ivy Bridge
# cores	4	4	2 × 6	2 × 8
# threads	4	8	24	32
CPU frequency	3.2 GHz	3.4 GHz	2.3 GHz	2.0 GHz
L1 cache	4 × 32 KB	4 × 32 KB	12 × 32 KB	16 × 32 KB
L2 cache	4 × 256 KB	4 × 256 KB	12 × 256 KB	16 × 256 KB
L3 cache	6 MB	8 MB	30 MB	40 MB
RAM	16 GB	32 GB	32 GB	64 GB

The closer the TLP value is to 1.0 (normalized to the total number of cores available), the more TLP is available. As an example, 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 FT benchmark presents the highest TLP, in which more than 95% of the application is executed in parallel.

The experiments were performed on four different multicore processors (Table 5.3), each one with 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. The applications were compiled 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%.

Aurora was evaluated in the following 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 [26]. This feature is enabled by using the environment variable OMP_DYNAMIC or through the insertion of the *omp_set_dynamic*() in the source code [22];
- **State-of-the-art approaches**: Aurora was also compared to the most cited approaches in the area:
  - **Feedback-Driven Threading (FDT)**: The number of threads is defined based on the contention for locks and memory bandwidth (as discussed in Sect. 4.2.2) [115].
  - **Varuna**: A high-level comparison with it was performed, by faithfully implementing the Varuna programming model (as defined in [113]) and applying it to the benchmarks.
- **Static Approaches**: in order to measure the efficiency of the search algorithm used by Aurora, two static and offline approaches were implemented:
  - **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.
  - **Genetic Algorithm (GA)**: It was implemented to demonstrate how Aurora's hill-climbing fares against such classes of heuristics. GA is a search meta-heuristic 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.

5.3.2 Results

We start by discussing how Aurora handles the scalability issues discussed in Sect. 1.2: off-chip bus saturation, shared memory accesses, data-synchronization, and issue-width saturation. Then, we present a comparison between Aurora and the following executions in Sect. 5.3.2.2: baseline, OMP_Dynamic, FDT, and Varuna, while Sect. 5.3.2.3 compares Aurora to the results achieved by the genetic algorithm. Section 5.3.2.4 discusses the efficiency of the search algorithm implemented by Aurora through the comparison to the Oracle solution. Finally, Sect. 5.3.2.5 draws the limitations of Aurora at this time.

5.3.2.1 Handling Scalability

As a result of its runtime 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 Sect. 1.2) 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 5.4) each. Figure 5.5a 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% when compared to the baseline execution (24 threads). The very same behavior can be observed in FFT and ST (regardless of the input set) and JA for the medium input set (Table 5.2), 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 Sect. 1.2. 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 5.2). 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 5.5b 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 Table 5.4, Aurora found the best number of threads to execute this parallel region, providing EDP gains of 58% when compared to the baseline execution (24 threads).

Aurora similarly detects the point where the synchronization time overlaps the gains provided by TLP exploitation. This behavior can be observed in some benchmarks, such as n-body (NB) with small or medium input, or Jacobi(JA) and SP with small input set (Table 5.2). In these benchmarks, the higher the number of threads, the greater the time spent synchronizing, which can worsen the results, as already discussed in Sect. 1.2. The n-body benchmark with the medium input set executing on the 32-core system (Fig. 5.5c) 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 (Fig. 5.5c), increasing the energy consumption and EDP of the whole application. As demonstrated in Table 5.4, 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 5.2). Let us consider the hotspot benchmark with the medium input set

Table 5.4 Number of threads found by Aurora

		Performance				Energy				EDP			
Proc.- #cores		4	8	24	32	4	8	24	32	4	8	24	32
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	8, 3, 4	24, 6, 12	32, 11, 14	4, 1, 2	8, 1, 2	24, 6, 4	32, 6, 6	4, 2, 2	8, 1, 2	24, 6, 6	32, 10, 10
		4, 4, 4	8, 8, 8	12, 24, 12	32, 32, 32	4, 4, 4	4, 8, 8	11, 24, 12	32, 32, 32	4, 4, 4	8, 8, 8	24, 12, 12	32, 32, 32
		4, 4, 4	4, 2, 2	12, 6, 6	10, 12, 26	1, 1, 1	2, 1, 1	6, 4, 6	6, 6, 6	1, 1, 1	2, 1, 2	6, 6, 6	6, 6, 6
		4	2	6	10	1	1	4	6	1	2	6	8
	M	4, 4, 4	8, 2, 4	24, 4, 8	32, 11, 14	4, 1, 2	8, 1, 2	24, 2, 4	32, 4, 6	4, 2, 2	8, 1, 2	24, 4, 6	32, 4, 6
		4, 4, 4	8, 8, 8	24, 24, 24	16, 32, 32	4, 4, 4	4, 8, 4	11, 24, 12	13, 32, 14	4, 4, 4	8, 8, 8	24, 24, 12	32, 32, 32
		4, 4, 4	5, 2, 5	6, 4, 11	6, 13, 11	1, 1, 1	1, 1, 1	4, 4, 2	4, 4, 4	2, 2, 2	5, 1, 1	6, 4, 4	6, 4, 4
		4	2	8	12	1	1	2	4	2	1	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, 2, 4	6, 6, 12	12, 26, 15	2, 1, 3	2, 1, 3	6, 4, 10	6, 4, 13	2, 2, 3	2, 1, 4	6, 4, 12	6, 4, 15
		2, 3, 2	2, 4, 2	6, 12, 6	32, 15, 32	1, 3, 1	1, 3, 1	4, 10, 4	4, 13, 4	2, 3, 2	1, 4, 1	6, 12, 6	4, 15, 4
		3, 3, 4	4, 2, 2	12, 6, 6	15, 6, 8	3, 1, 1	3, 1, 1	10, 4, 2	10, 4, 4	3, 2, 2	3, 1, 1	10, 4, 4	15, 4, 4
	M	3, 4, 2	2, 5, 2	6, 6, 6	12, 10, 10	2, 1, 2	1, 1, 2	4, 4, 6	6, 4, 8	2, 2, 2	2, 1, 2	6, 6, 6	6, 6, 8
		2, 2, 2	2, 2, 5	20, 6, 20	32, 10, 14	1, 2, 1	1, 2, 1	4, 6, 4	4, 8, 4	2, 2, 2	2, 2, 2	6, 6, 6	4, 8, 4
		2, 4, 4	2, 5, 2	6, 6, 24	8, 10, 24	2, 1, 1	2, 1, 1	6, 4, 4	8, 4, 4	2, 2, 2	2, 5, 1	6, 4, 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

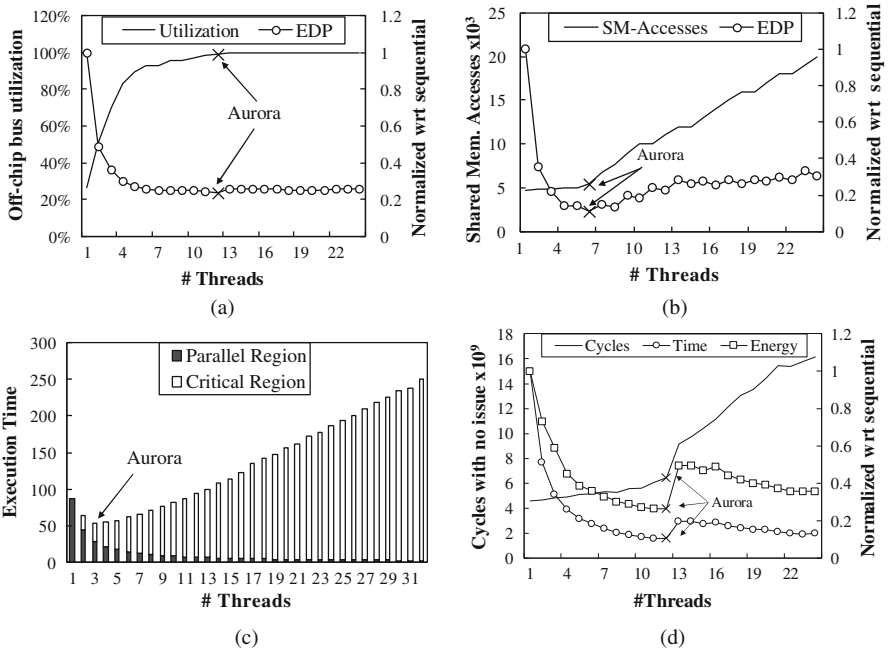

Fig. 5.5 Scalability behavior. (**a**) HPCG—2nd parallel region. (**b**) SP—1st parallel region. (**c**) *n*-body. (**d**) Hotspot

executing on the 24-core system. In this case, the optimal number of threads for EDP is 12 (see Table 5.4). As Fig. 5.5d 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 (Fig. 5.5d). 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.

Finally, as discussed in Chap. 1, 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 5.2). 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.

5.3.2.2 Performance, Energy, and EDP

Table 5.4 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 5.6, 5.7, and 5.8 present the results for the entire benchmark set when running the medium input set, along with their geometric mean (Gmean)

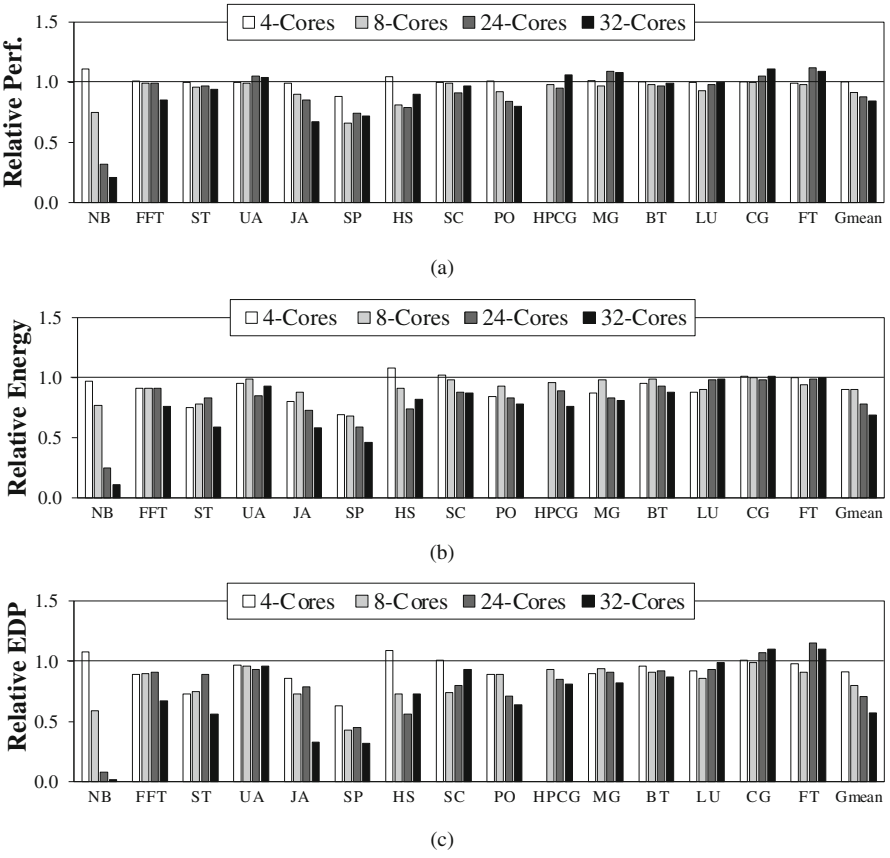

Fig. 5.6 Aurora vs Baseline (medium input): lower than 1.0 means that Aurora is better. (**a**) Performance. (**b**) Energy consumption. (**c**) Energy-delay product

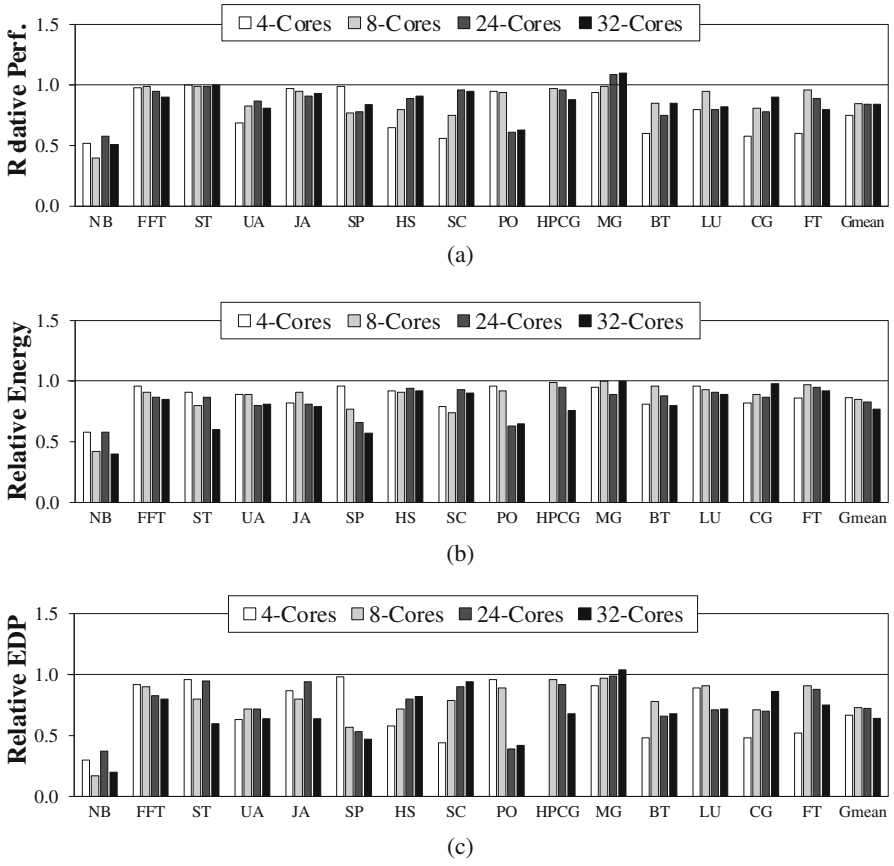

Fig. 5.7 Aurora vs OMP_Dynamic (medium input): lower than 1.0 means that Aurora is better. (**a**) Performance. (**b**) Energy consumption. (**c**) Energy-delay product

considering the four multicore systems. Figure 5.6 compares Aurora to the baseline (represented by the black line), while Figs. 5.7 and 5.8 compare Aurora to OMP_Dynamic and FDT framework (also represented by a black line), respectively. 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 for performance, energy consumption, and EDP. For each metric, we show the result for Aurora when it is set to optimize the particular metric. As an example, Fig. 5.6b shows the energy savings achieved by Aurora over the baseline when set to reduce the energy consumption. Table 5.4 summarizes the results for the small input set considering the geometric mean for the four multicore systems.

Aurora Versus Baseline As observed in Fig. 5.6 and Table 5.4, 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 the times

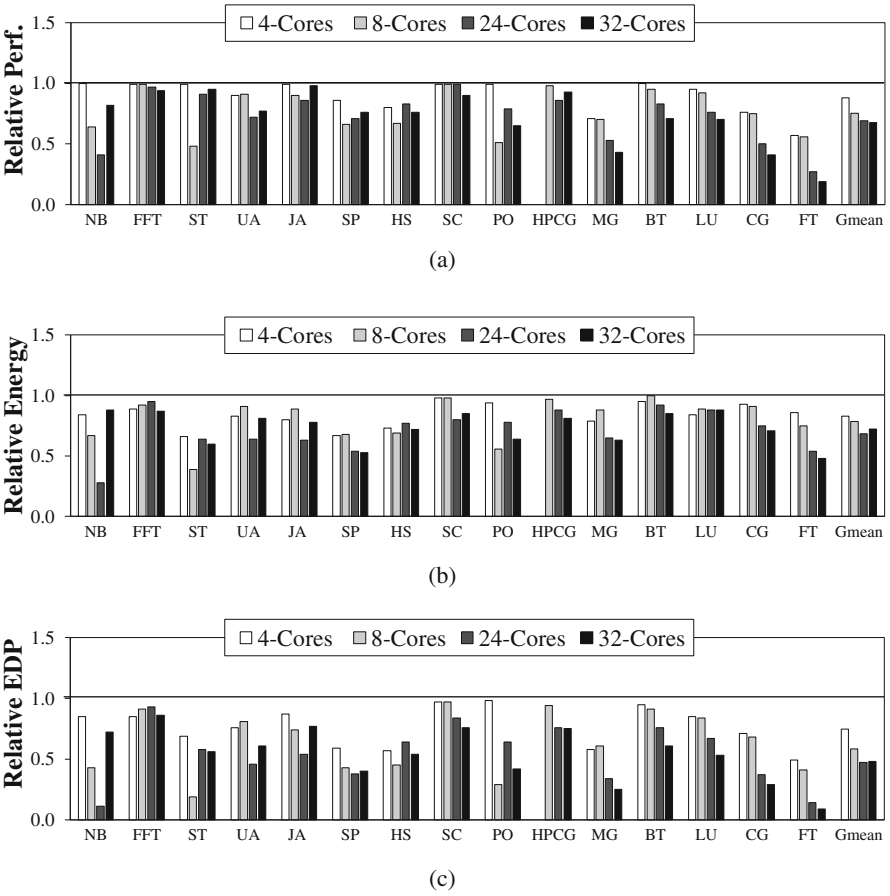

Fig. 5.8 Aurora vs FDT (medium input): lower than 1.0 means that Aurora is better. (**a**) Performance. (**b**) Energy consumption. (**c**) Energy-delay product

Table 5.4 Summary of the results for the small input w.r.t. 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

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 min of execution to define the number of threads [22]. 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 Fig. 5.6 and Table 5.4 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 Fig. 5.8 and Table 5.4, Aurora outperforms FDT in all scenarios. In the best case (small input set and the 24-core machine), Aurora improved (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 One representative application was selected from each benchmark class (NB, SC, ST, and FT) and implemented them using the programming model employed by Varuna. They were executed with two different amounts of threads (1566 and 10k threads, taken from [113]) on the 32-core machine. Table 5.5 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 Chap. 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 [113]), it would not be enough to provide the same levels of performance and energy as the OpenMP baseline, in most cases. The main

Table 5.5 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

reason for these results is that Varuna was developed to be 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.

5.3.2.3 Distinct Approaches, Similar Convergence

Genetic algorithm 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 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 5.9 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 (Fig. 5.9 truncates at generation 26), and when one considers the geometric mean, Aurora performs slightly better. We can also see the worst

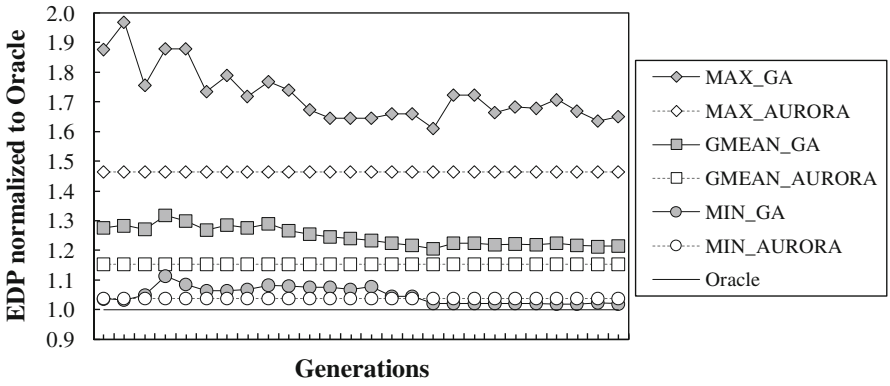

Fig. 5.9 Genetic algorithm convergence and Aurora

(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 case (an application with many parallel regions) because of its coarse tuning characteristics. Furthermore, Fig. 5.9 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 runtime.

5.3.2.4 Evaluating the Efficiency of Aurora's Search Algorithm

Table 5.6 depicts (in percentage) how the results obtained by Aurora differ 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:

1. 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.
2. The parallel region has a relatively small number of interactions but executes for a significant time, such as HPCG.

Table 5.6 Learning overheads (%) for Aurora w.r.t. 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

3. Applications that have short execution time (i.e., less than 10 s), such as the MG. Its Oracle version takes only 1.45 s to execute on the 32-core system with the small input set.
4. 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 s to execute regardless of 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.

5.3.2.5 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 previously discussed in this chapter, *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: (1) 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. (2) 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. (3) 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 (1) with its learning algorithm and (2) 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.