LAANT: A Library to Automatically Optimize EDP for OpenMP Applications

Arthur Francisco Lorenzon, Jeckson Dellagostin Souza, Antonio Carlos Schneider Beck
Institute of Informatics – Federal University of Rio Grande do Sul, Porto Alegre, Brazil
{aflorenzon, jdsouza, caco}@inf.ufrgs.br

***Abstract*— Efficiently exploiting thread level parallelism from new multicore systems has been challenging for software developers. While blindly increasing the number of threads may lead to performance gains, it can also result in disproportionate increase in energy consumption. For this reason, rightly choosing the number of threads is essential to reach the best compromise between both. However, such task is extremely difficult: besides the huge number of variables involved, many of them will change according to different aspects of the system at hand and are only possible to be defined at run-time. To address this complex scenario, we propose LAANT, a novel library to automatically find the optimal number of threads for OpenMP applications, by dynamically considering their characteristics, input set, and the processor architecture. By executing nine well-known benchmarks on three real multicore processors, LAANT improves the EDP (Energy-Delay Product) by up to 61%, compared to the standard OpenMP execution; and by 44%, when the dynamic adjustment of the number of threads of OpenMP is activated.**

Keywords— Thread-level parallelism, EDP, library, OpenMP

I. INTRODUCTION

Thread-level parallelism (TLP) exploitation is being widely used to improve performance. In this case, multiple processors simultaneously execute parts of the same program, exchanging data at runtime through shared memory regions. However, as the power consumption of high-performance computing (HPC) systems is expected to significantly grow (up to 100 MW) in the next few 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 minimal impact on energy consumption [2].

However, performance improvements resultant from TLP exploitation are not linear and sometimes do not scale as the number of threads increase, which means that in many cases the maximum possible number of threads will not offer the best results. Several are the reasons: intrinsic application characteristics, such as data synchronization and communication; or hardware related, which may involve the memory hierarchy and off-chip bus bandwidth [3][4]. For instance, applications with high communication demands may saturate cache memories and the off-chip bus, since many threads will compete for these resources. Moreover, excessive data synchronization and communication may increase energy consumption since they occur through shared memory regions, which are more distant from the processor (e.g., L3 and main memory) and have a higher delay and power consumption when compared to memories that are closer to it [5].

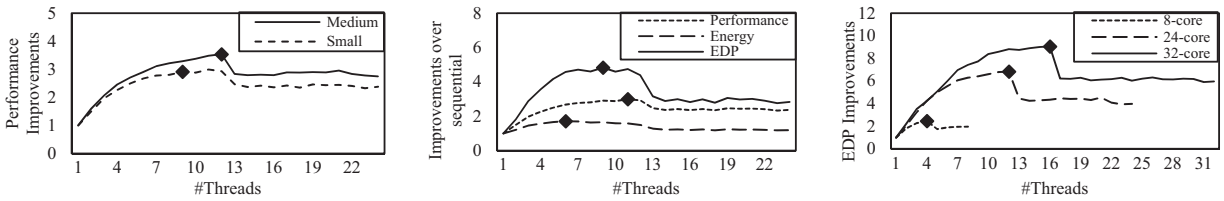
Therefore, choosing the right number of threads to a given application will offer opportunities to improve performance and save energy. However, such task is extremely difficult: besides the huge number of variables involved, many of them will change according to different aspects of the system at hand and are only possible to be defined at run-time, such as:

- *Input set:* the degree of TLP of many applications is highly dependent on their inputs. As shown in Fig. 1.a, different levels of performance improvements for the LULESH[6] benchmark over its sequential version (also used in the next examples) are reached with different number of threads (x-axis). However, these levels vary according to the input set. While the best number of threads is 12 for the medium input set, 9 threads is the ideal number for the small set.
- *Metric evaluated:* the ideal number of threads also changes accordingly to the metric evaluated. As Fig. 1.b shows, the best performance is reached with 12 threads, while 6 threads will bring the lowest energy consumption, and 10 will present the best tradeoff between both metrics (represented by the energy-delay product – EDP).
- *Processor microarchitecture:* if the same application is executed on different systems, each one may present different behaviors. Fig. 1.c shows that the best EDP improvements of the parallel application on a 32-core machine is when it executes with 16 threads. However, on a 24-core system, the best number of threads is 12.
- *Parallel regions*: many applications are divided into several parallel regions, in which each of these regions may have a distinct ideal number of threads [3].

Works with distinct approaches to optimize the number of threads have already been proposed. Some of them include only a dynamic pre-compilation or pre-execution analysis [3][4][7], while others can also include a runtime adaptation technique to readjust the number of threads accordingly workload changes or distinct parallel regions [8][9][10]. However, these works require special compilers, libraries or have some limitations with the running system. Furthermore, all previous works use only performance as the optimization metric (which may or may not indirectly bring energy gains).

Considering this challenging scenario that involves this huge number of variables, we propose LAANT, a library to automatically adjust the number of threads for optimizing EDP of OpenMP (Open-Multi Processing) applications. It is capable of finding at run-time the ideal number of threads for each parallel region of the application, learning the best number of threads as the application executes, and resulting in significant improvements in EDP with an almost negligible overhead. LAANT covers all cases discussed before and shown in Fig. 1: it considers the intrinsic characteristics of the application at

978-3-9815370-8-6/17/$31.00 ©2017 IEEE



a) Different Input sets b) Different Metrics Evaluated c) Different Multicore Processors

Fig. 1 Appropriate number of threads (♦) to execute the LULESH 2.0 Benchmark considering the improvements over sequential version.

hand, and adapts to the current input set and processor microarchitecture.

The proposed process is completely automatic to the developer. It can be applied to any parallel application developed with the OpenMP interface and compiled with GCC or G++ compilers, by simply annotating code in the parallel regions, which were already identified by OpenMP directives. These notations are automatically inserted in the source code, being transparent to the programmer. LAANT can optimize the parallel regions for different metrics, such as EDP (the focus of this work), performance, energy consumption, among others.

LAANT is validated using a set of nine well-known benchmarks, which cover a wide range of different TLP behaviors. They were executed on three different multicore processors that support a different number of threads. We show that LAANT improves the EDP of OpenMP applications at different rates regardless the processor used. In our most significant case, LAANT has EDP gains of 61%, when compared to the standard way that OpenMP applications are executed (that uses the maximum number of threads along all the execution), and 82%, when compared to the OpenMP dynamic feature (that dynamically changes the number of threads as application executes).

The remainder of the paper is organized as follows. LAANT is presented in Section II. The methodology is discussed in Section III. Section IV discuss the results. Finally, Section V draws the final considerations.

II. Design of LAANT

LAANT works with OpenMP, which is a parallel programming interface for shared memory in C/C++ and FORTRAN. OpenMP consists of a set of compiler directives, library functions, and environment variables [11]. The library we propose uses a heuristic based on a hill-climbing algorithm to find the optimal number of threads for parallel regions of OpenMP applications. Each one of these regions can be optimized for different metrics, such as performance, energy, EDP, among others. In this work, LAANT was set to find the best number of threads that optimize EDP.

```
1.  int main ( ){
2.      initLaant();
3.      for(int iter=0; i<100; iter++){
4.          /* Sequential region */
5.          startKernel();
6.          #pragma omp parallel
7.          {
8.              /* Parallel region */
9.          }
10.         endKernel();
11.     }
12. }
```

Fig. 2 Using LAANT on OpenMP applications

The EDP is calculated by multiplying the execution time with the energy consumption. To obtain the execution time of each parallel region, LAANT uses the *omp_get_wtime*() function, provided by the OpenMP. Energy is obtained directly from hardware counters present in modern processors. In the case of Intel processors, the Running Average Power Limit (RAPL) [12] library is used to get energy and power consumption of the CPU-level components. As for AMD processors, another library could be used: Application Power Management [13]. LAANT is divided into two parts: the first one contains the functions provided to the developer, and the second is the heuristic to find the ideal number of threads for parallel regions.

A. Using LAANT on OpenMP Applications

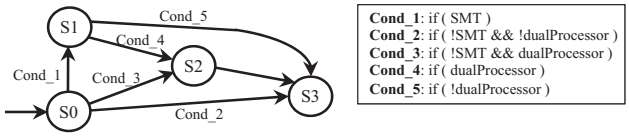
LAANT consists of three main functions: *initLaant*, *startKernel,* and *endKernel*. These are inserted by a script (provided by LAANT) into any OpenMP application, whose the parallel regions were already identified by #pragmas, as shown in line 6 of Fig 2. The *initLaant* function initializes the structures and variables used to control the hill-climbing algorithm, and the libraries used to collect information from parallel regions behavior. Thus, it is inserted at the beginning of the *main* function. The *startKernel* function sets the number of threads that executes each parallel region based on the current state of the hill-climbing algorithm. Also, *startKernel* initializes the counters for execution time, energy, and EDP of the parallel region. This function is inserted before a parallel region, as shown in line 5 from Fig. 2.

Finally, the *endKernel* function is inserted after the parallel region to get its execution time, energy, and the EDP. With this information, it performs one step of the hill-climbing algorithm to find the number of threads that will execute this parallel region in the next iteration. The functions *startKernel* and *endKernel* run until the ideal number of threads is found. Once found, they run periodically to verify changes in the behavior of parallel region, or when there are variations in its workload.

B. Automatically Adjusting the Number of Threads

LAANT uses a simple finite state machine (FSM) to implement a heuristic based on a hill-climbing algorithm. The heuristic is divided into two phases. The main states of the FSM are described in Fig. 3:

- The S0 state comprises the execution of the parallel region with the number of threads equal to the number of cores of the platform. If the platform has SMT technology or it is a dual-processor machine, then the next state will be S1 or S2. Otherwise, the hill-climbing algorithm will be performed in the S3 state, starting at the number of threads executed in S0.



S0: Execution with #threads = #total physical cores.
S1: Execution with #threads = #hardware threads available.
S2: Execution with #threads = #physical cores in one CPU socket.
S3: Hill-climbing algorithm in the interval of threads that offers the best EDP so far.

Fig. 3 Main states of the FSM

- If the platform has SMT technology, then in the S1 state, the parallel region run with the number of hardware threads.
- If the machine is a dual-processor, then in the S2 state, the parallel region is executed with the number of threads that matches the number of physical cores of one CPU socket.

These decisions are made based on previous knowledge: several executions and experiments have been done on different platforms, with distinct number of physical cores, CPU sockets and maximum number of supported threads [14]. This information is part of the LAANT and used only internally, and does not need to be updated by the designer or user. In the current work, the hill-climbing algorithm is configured to minimize the EDP. To avoid minimal locals and plateaus during the search, and so wrongly converging to an incorrect number of threads, the hill-climbing algorithm uses lateral movement. Once the number of threads that offers the best EDP is found, the parallel region will execute with this number until the periodical executions to verify changes in the behavior of parallel region.

III. Methodology

Nine applications already parallelized with OpenMP written in C/C++ from assorted benchmark suites were chosen. They are divided into four categories regarding the behavior characteristics of the parallel regions present on each application, as shown in Table I. An important statement is that the SP application has more than one parallel region. However, as all the parallel regions have the same behavior, we are considering SP as an application identified with one big parallel region that includes all the others.

The experiments were performed on three different multicore processors, each one able to execute a different number of threads. Each benchmark was executed with two different input sets: B and C for the NAS applications; and small and medium for the other benchmarks. The applications were compiled with gcc and g++ 5.3, using the optimization flag -O2. The OpenMP distribution used was version 4.0. The results presented in the next session are an average of ten executions with a standard deviation lower than 0.5%. When designing parallel applications, the usual is to execute with the number of hardware threads available in the system [7]. Therefore, we use this scenario as the comparison **Baseline**. We also ran the same benchmarks with an OpenMP dynamic feature (**OMP_Dynamic**), which dynamically adjusts the number of threads as the parallel region is being executed [15], aiming to make the best use of system resources. LAANT, on the other hand, executes its adaptation algorithm after the parallel region and targets EDP as optimization metric.

Table I. Main characteristics of the benchmarks

Characteristics	Benchmarks
Changes in the workload of the parallel region at runtime	Fast Fourier Transform (FFT)
More than one parallel region, each one with a different behavior.	Block tri-diagonal solver (BT), lower-upper gauss-seidel solver (LU), Poisson equation (PO), and LULESH 2.0
One of more parallel regions with the same behavior, and with no changes in the workload at runtime	Hotspot and Scalar penta diagonal solver (SP)
High degree of TLP exploitation	Conjugate Gradient (CG) and Discrete 3D Fast Fourier Transform (FT)

IV. Results

Figs. 4 and 5 present EDP results for LAANT and OMP_Dynamic for each application, along with their geometric mean (gmean) for the entire benchmark set on the three multicore systems. EDP is normalized considering the baseline explained in the previous section (represented by the black line). In all cases, LAANT found the best number of threads to execute each parallel region. This was validated by comparing the numbers found by LAANT with an exhaustive search using data from the execution of each parallel region with all possible numbers of threads (from 1 to the maximum number of cores). The results are discussed in the next sub-sections, considering the Baseline and the OMP_Dynamic separately.

A. LAANT versus Baseline

Comparing to the Baseline and considering the geometric mean (gmean) of the entire benchmark set, LAANT can reduce the EDP in up to 29% on the 24-core system running the small input set (Fig. 4). The smallest gains are achieved in the 32-core system running the medium input set (Fig. 5) –, and, even in this case, LAANT presents 15% of EDP reductions.

LAANT automatically detect changes in the workload of the parallel region and correctly finds the appropriate number of threads to run this region. For instance, when the workload of the FFT application changes at runtime, the number of threads that offers the best EDP also changes. Let us consider the FFT execution with a medium input set on the 24-core system. On its first parallel region execution, the best EDP is initially achieved by executing 20 threads. However, as the workload of this parallel region increases, the synchronization and communication between threads also increase, saturating the shared resources [3]. Therefore, the ideal number of threads end up reducing to 12, and then 6, when it finally stabilizes. LAANT detects these workload changes and adjusts the number of threads, which provides 31% of EDP reductions.

In applications with more than one parallel region, LAANT finds the ideal number of threads to execute each one of them. For instance, let us consider the execution of LULESH with the small input set on the 32-core system. LULESH has three parallel regions, in which the first region is better executed with 14 threads. However, the regions 2 and 3 present the best EDP with 10 and 16 threads, respectively. By finding these numbers at runtime, LAANT reduced the EDP by 49%. The same behavior happens with BT, Poisson, and LU applications, but at different EDP improvement ratios.

The lowest overhead, i.e., the time that LAANT spent in the search for the best number of threads relative to its total time and energy, occurs in applications that possess only one parallel

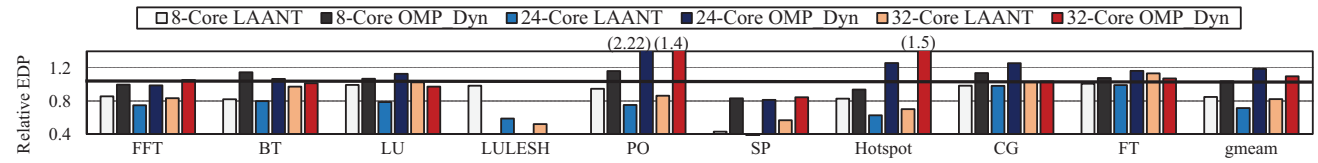


Fig. 4 Relative EDP of LAANT and OMP_Dynamic when executing with the **small input** set compared to the baseline (black line)

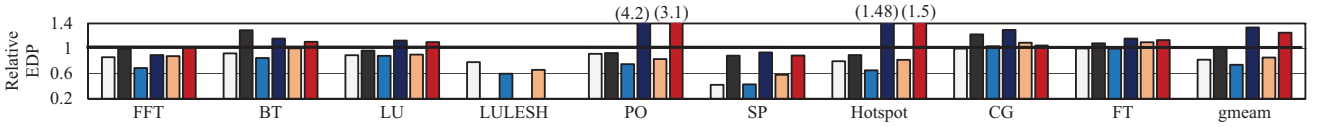


Fig. 5 Relative EDP of LAANT and OMP_Dynamic when executing with the **medium input** set compared to the baseline (black line)

region, such as SP and Hotspot. For instance, the overhead for the Hotspot application on the 24-core system was of only 0.01% of the total time and energy. This happens because the search algorithm used to find the best number of threads (12) can do so at only four steps: by running with 12, 24, 6 and 10 threads. Besides the low overhead, LAANT also provided huge EDP reductions, such as 61% when executing the SP with the small input set on the 24-core system.

The highest overhead presented by LAANT was in applications which the best EDP is achieved with either the maximum number of threads or a number near it, such as FT and CG. For the FT application with the medium input set (C) on the 32-core system, LAANT increased the execution time by 8.12% and energy by 1.62%. This happens because the execution of each parallel region with a lower number of threads during the search results in increased execution time and energy consumption for the whole application. Although this is the worst case for LAANT, this unnecessary overhead can be reduced by enhancing the search algorithm to minimize the training overhead on such applications.

B. *LAANT versus OMP_Dynamic*

LAANT outperforms the OpenMP dynamic feature in the great majority of cases (Figs. 4 and 5). On average of the entire benchmark set with the medium input size, LAANT has 21% of EDP gains on the 8-core system; 44% on the 24-core system; and 32% on the 32-core system. While LAANT has EDP improvements for the applications SP, Hotspot, and Poisson, in which the parallel regions are executed many times, the OpenMP dynamic feature has its worst results. For instance, LAANT has EDP gains of up to 82% when compared to the OMP_Dynamic on the execution of Poisson on the 24-core system (Fig. 5). This happens because when the number of threads is changed by the OMP_Dynamic, the workload must be redistributed to all threads again since it is performed as the parallel region executes.

A significant limitation of the OMP_Dynamic was observed when running the LULESH application. In such application, it was unable to correctly terminate the execution because the variables inside the parallel region are allocated based on the number of threads that initialize those regions. Therefore, as the OpenMP dynamic feature changes the number of threads during the execution of the parallel region, it will allow a thread to access an address that was not allocated, causing an error in the execution. Besides correctly finishing the LULESH execution, LAANT reached huge EDP gains (up to 49%) over the baseline regardless the processor. Finally, the only cases when OMP_Dynamic had similar behavior as LAANT are with applications (FT, CG) where the ideal number of threads is near to the maximum. This is because the overhead of the OMP_Dynamic is reduced since the behavior of the parallel regions of such applications have negligible changes at runtime.

V. Conclusions

In this work, we have presented LAANT, a library capable of automatically adjust the number of threads to optimize the application EDP. LAANT applies a hill climbing algorithm for training while the application is running, which results in an almost negligible overhead for most of the applications. We show that our approach can reduce the EDP in 29%, on average, when compared to the standard way that OpenMP applications are executed; and reduces the EDP in up to 82% when compared to the dynamic feature of OpenMP.

References

[1] K. Yelick, "Ten Ways to Waste a Parallel Computer," in *ISCA*, 2009, p. 1.

[2] A. C. S. Beck, C. A. L. Lisba, and L. Carro, *Adaptable Embedded Systems*. Springer Publishing Company, Incorporated, 2012.

[3] 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. Arch. News*, vol. 36, no. 1, pp. 277–286, 2008.

[4] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring," in *ISWC*, 2011.

[5] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," *Parallel Archit. Compil. Tech. - Conf. Proceedings, PACT*, pp. 261–270, 2009.

[6] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," techreport, 2013.

[7] J. Lee, H. Wu, M. Ravichandran, and N. Clark, "Thread tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 270, Jun. 2010.

[8] G. Chadha, S. Mahlke, and S. Narayanasamy, "When less is more (LIMO): controlled parallelism forimproved efficiency," *ICCASES*, pp. 141–150, 2012.

[9] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," *Pldi'14*, pp. 169–180, 2014.

[10] S. Sridharan, G. Gupta, and G. S. Sohi, "Holistic run-time parallelism management for time and energy efficiency," in *ACM ICS*, p. 337, 2013.

[11] J. G. V. der P. G. Chapman B., *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, Cambridge, MA, 2008.

[12] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using RAPL," *ACM SIGMETRICS*, v.40, n.3, p.13, 2012.

[13] D. Hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," *IEEE Int. Symp. Perform. Anal. Syst. Softw.*, pp. 194–204, 2013.

[14] A. F. Lorenzon, M. C. Cera, and A. C. S. Beck, "Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy," in *JPDC*, vol. 95, pp.107–123, 2016.

[15] A. R. B. OpenMP, "OpenMP application program interface version 4.0.", 2013.