GLTO: On the Adequacy of Lightweight Thread Approaches for OpenMP Implementations

Adrián Castelló
Rafael Mayo
Enrique S. Quintana-Ortí
Universitat Jaume I de Castelló
Castelló de la Plana, Spain
{adcastel,mayo,quintana}@uji.es

Sangmin Seo
Pavan Balaji
Argonne National Laboratory
Lemont, Illinois, USA
{sseo,balaji}@anl.gov

Antonio J. Peña
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
antonio.pena@bsc.es

***Abstract*—OpenMP is the *de facto* standard application programming interface (API) for on-node parallelism. The most popular OpenMP runtimes rely on POSIX threads (pthreads) implementations that offer an excellent performance for coarse-grained parallelism and match perfectly with the current hardware. However, a recent trend in runtimes/applications points in the direction of leveraging massive on-node parallelism in conjunction with fine-grained and dynamic scheduling paradigms. It has been demonstrated that lightweight thread (LWT) solutions are more appropriate for these new parallel paradigms. We have developed GLTO, an OpenMP implementation over the recently-emerged Generic Lightweight Threads (GLT) API. GLT exports a common API for LWT libraries that offers the possibility of running the same application over different native LWT solutions. In this paper we use GLTO to analyze different scenarios where OpenMP implementations may benefit from the use of either LWT or pthreads. Our study reveals that none of the threading approaches obtains the best performance in all the scenarios, but that there are important gaps among them.**

***Index Terms*—GLT, Lightweight Threads, OpenMP, POSIX Threads, Programming Models**

I. Introduction

In the last years, the number of cores per processor has been increasing periodically, reaching impressive counts such as the 260 cores per socket that are present in the Sunway TaihuLight supercomputer [1], which was ranked #1 in the June 2016 TOP500 List [2].

The trend followed in that list indicates that future exascale systems will support massive on-node parallelism, deploying thousands or millions of cores per socket. However, extracting the computational power of those machines will require efficient libraries and programming models (PMs). One of the most popular approaches to obtain acceptable on-node performance is via the POSIX threads (pthreads) application programming interface (API) [3], or through directive-based PMs such as OpenMP [4] or OmpSs [5].

Those PMs are implemented on top of the pthreads API, which matches perfectly with the current hardware and coarse-grained codes, but fails to accommodate new software paradigms that target dynamically-scheduled, fine-grained parallelism due to the high cost of management.

Several lightweight thread (LWT) libraries have been implemented in the last years to tackle the fine-grained and dynamic software requirements [6]. Each LWT solution features its own PM and target environment. Some are implemented for a specific Operating System (OS), such as Windows Fibers [7] and Solaris Threads [8]. Compared with those, ConverseThreads [9] and Nanos++ [10] support a specific high-level PM; Charm++ [11] and OmpSs [5], respectively. There are also general-purpose solutions such as MassiveThreads [12], Qthreads [13], and Argobots [14]. The Generic Lightweight Threads (GLT) API [15] is an effort to unify these LWT solutions under a unique PM to foster productivity and portability with a negligible overhead that is demonstrated in [16]. This lightweight API offers the common functionality of LWT solutions and is currently implemented on top of MassiveThreads, Qthreads, and Argobots. As a result, a runtime/application based on GLT requires no changes to be executed on top of any of these three LWT solutions.

In this paper we present GLTO: our design and implementation of an OpenMP runtime on top of the GLT API. Our OpenMP implementation is based on the open-source BOLT project [17], which is in turn based on LLVM [18] (the LLVM OpenMP runtime shares the code developed in the Intel OpenMP [19] solution). We test our OpenMP implementation with the OpenUH OpenMP Validation Suite 3.1 [20].

Based on GLTO, we analyze the most common OpenMP patterns and discuss how LWTs deal with them, in comparison with traditional pthread-based approaches. We evaluate our OpenMP implementation and compare its performance with those obtained when using the GNU and Intel OpenMP runtimes[1] in four different scenarios: basic parallel code, *for* loop based code, nested parallelism, and task parallelism. Our study reveals that none of the solutions obtains the best performance in all the scenarios, but that there are important gaps among them.

In summary, the main contribution of this paper is a thorough analysis of the OpenMP patterns that may benefit from LWT or Pthreads based implementations.

The rest of the paper is organized as follows. Section II reviews related work. Section III provides some background

[1]Since BOLT is still under development, a performance comparison with GLTO is premature.

information about OpenMP and GLT. Section IV details the GLTO implementation. Section V validates our OpenMP solution. Section VI provides an in-depth performance analysis of the distinct scenarios. Section VII discusses the lessons learned from the analysis of our experiments. Section VIII contains our conclusions.

II. Related Work

The OpenMP standard is currently supported by an important number of compilers, including both open source and vendor solutions. Although the current OpenMP specification corresponds to version 4.5 [21], some compilers may not support the complete set of directives. For example, the LLVM project compiler (`clang 3.9`) supports all non-offloading features of OpenMP 4.5. In contrast, Intel's `icc` compiler 16.0 supports the complete OpenMP 4.0 specification, and the newest `icc 17.0` and the `gcc 6.1` compiler from GNU adhere to the complete OpenMP 4.5 specification. There are other compilers that are one or more steps behind those solutions. For example `pgcc` [22], from the Portland Group, and OpenUH [23], support version 3.1 of the OpenMP specification.

Supporting an OpenMP specification means that each solution must have its own OpenMP runtime with its own features because they may target specific hardware or code. However, the most prominent runtimes are those offered by GNU and Intel—namely `libgomp` and the Intel OpenMP runtime. In some cases, the same runtime code is shared among compilers, as it occurs for the implementation of the Intel solution, which can be linked with code built by the `clang` compiler.

In the field of LWT libraries, the works in [6], [9], [12], [13], [14] introduce distinct LWT definitions, discuss implementation details, and analyze performance. The work in [24] conducts an analysis of different LWT solutions from the semantic point of view and evaluates their performance.

The relationship between LWTs and the OpenMP runtime has also been explored in the past. In [25] and [26], nested parallelism is analyzed and resolved by means of LWT solutions. Moreover, the effect of OpenMP implementations when executed in NUMA architectures and scheduling for task parallelism have also been studied in [27] and [28], respectively.

To the best of our knowledge, this is the first paper analyzing the general appropriateness of LWTs for the implementation of OpenMP runtimes.

III. Background

In this section we review the OpenMP PM and describe the GLT implementation and its interaction with the underlying LWT libraries.

A. OpenMP

The OpenMP API supports multi-platform shared-memory multiprocessing programming, and current implementations cover most architectures and operating systems. OpenMP offers a directive-based PM to accelerate a code by means of "pragmas". Intel and GNU offer two common OpenMP implementations that rely on top of Pthreads in order to exploit concurrency.

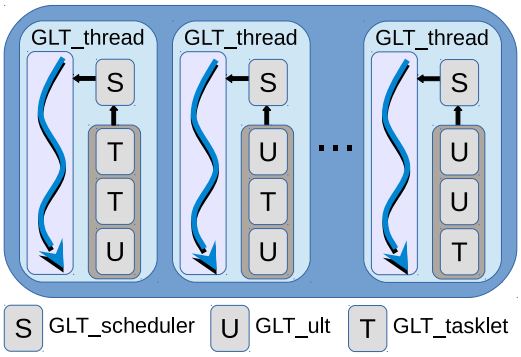

Fig. 1: PM offered by the GLT library.

OpenMP runtimes are commonly composed of two main parts: the work-sharing constructs and task parallelism. In contrast with work-sharing constructs, where all the OpenMP implementations follow a similar policy, distinct OpenMP implementations leverage different mechanisms for task management. In particular, while the GNU version implements a single shared task queue for all the threads, the Intel implementation incorporates one task queue for each thread and integrates work-stealing for load balance control. In both solutions, the task management is separated from the work-sharing implementations because task directives were added in the OpenMP 3.0 specification.

B. Generic Lightweight Threads

GLT is a common API that was designed with the aim of unifying, under the same PM, a variety of LWT libraries. It is currently implemented on top of three general-purpose LWT solutions: MassiveThreads, Qthreads, and Argobots. Despite this API does not support the complete set of the native LWT semantics, the selected group is sufficient to implement our OpenMP runtime.

The GLT API abstracts the semantics of each component under the same terminology. Figure 1 illustrates the PM offered by this API. Specifically, `GLT_thread` refers to the OS thread itself, while `GLT_ult` represents the user-level threads (ULTs). In addition, `GLT_tasklet`, a lighter work-unit that does not own a stack (preventing migration or yield operations), is offered as part of the common API. While tasklets are natively supported only by Argobots, these are implemented on top of ULTs for Qthreads and MassiveThreads. `GLT_scheduler` acts differently depending on the underlying library and it may change the performance of the PM but not the final result of the execution.

Although adding an extra software layer between the OpenMP runtime and the underlying libraries may affect performance, GLT does not add any appreciable overhead because it offers a header-only version that allows the compilers to

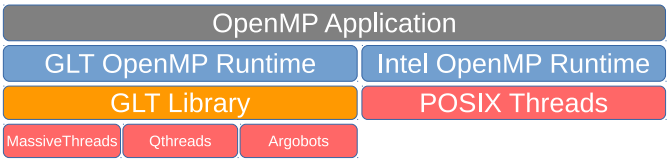

Fig. 2: Software stack choices of an OpenMP code.

avoid the extra calls by embedding the LWT code by means of `static inline` declarations [29].

The use of this intermediate software level allows the programmer to test and leverage different LWT solutions under just a single code version. This feature provides portability, enabling the adaptation to the underlying hardware/software combination.

IV. OpenMP over GLT

In this section we justify the design decisions that we made in order to adapt the LLVM OpenMP runtime to the use of LWTs.

As argued in Section I, our implementation is based on the BOLT project that is, in turn, based on LLVM. We selected this starting point because both the runtime and the `clang` compiler [30] are open source. In addition, this runtime can be linked from code generated with the Intel compiler.

A. GLTO Interactions

GLTO offers a complete implementation of OpenMP 4.0 for C, C++, and Fortran codes. GLTO can be linked with code generated by the `clang` or `icc` compilers. Figure 2 shows that an OpenMP code compiled with these tools can be linked to the original Intel OpenMP runtime and executed using pthreads, or linked with the GLTO runtime and executed over the desired LWT solution. The flexibility added by GLTO helps developers in two ways: if a LWT solution implements the GLT API, an OpenMP code can be executed on top of that LWT solution; in case a code benefits from a certain mechanism, the user can change the underlying library without modifying the OpenMP code.

B. GLTO Implementation Details

LWT libraries use two threading levels. The lowest level is composed by a number of OS threads. Those threads are scheduled by the OS (like the pthreads) and ULTs run on top of them. These ULTs are created, scheduled, and executed inside the user space so their handling overhead is lighter than that of their OS counterparts.

Complying with the OpenMP Specifications [21], our GLTO implementation responds to the `OMP_NUM_THREADS` environment variable to create as many `GLT_threads` as OpenMP threads are requested by the user. As depicted in Figure 3, `GLT_threads` are bound to CPU cores and are created when the library is loaded. They will be the responsible to execute the `GLT_ults` created at runtime. Standard-compliant dynamic adjustment of threads via the `num_threads` clause and the `omp_set_num_threads` library routine is also possible.

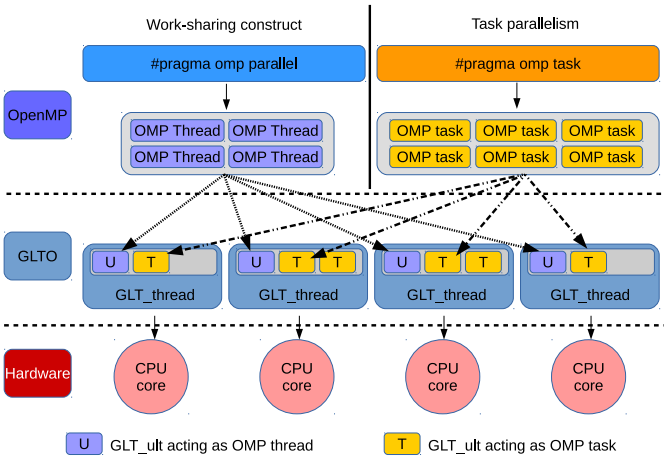

Fig. 3: Relationship between OpenMP code and the GLTO implementation.

`GLT_ults` act as Pthreads do inside the POSIX-based OpenMP solutions when work-sharing constructs are invoked. The left-hand side plot of Figure 3 shows that each OMP Thread is converted into a `GLT_ult` in that scenario.

For task parallelism (right-hand side of Figure 3), each OMP task is also converted to a `GLT_ult`. However, due to the different data structures used by the OpenMP runtime for OMP Thread and OMP task, inside the GLTO implementation, the behavior of the `GLT_ult` differs when acting as an OMP Thread or an OMP task.

In the next subsections we discuss in more detail how GLTO acts in each scenario.

C. Work-sharing Construct

For work-sharing constructions, our OpenMP solution mimics the mechanism that the GNU and Intel runtimes implement. The master thread assigns the function pointer to each thread in the runtime and then, once the work is done, the master thread joins the others. When the merge is completed, the master thread finalizes the parallel construct and continues executing the sequential code until a new parallel region is detected.

In GLTO, the work is assigned by creating a `GLT_ult` with the function pointer for each `GLT_thread`, and the master thread waits for work completion using a join function. As in the pthread solutions, the master thread continues with the execution of the sequential code.

D. Task Parallelism

In contrast with work-sharing structures, the task parallelism implementation may differ depending on the specific OpenMP solution. The main reason is that task directives were introduced in the OpenMP 3.0 specification, and the runtimes added the required functions with the primary goal of maintaining the performance attained by the work-sharing implementations.

As demonstrated later in our experimentation, it is in these scenarios where LWTs can deliver higher performance, especially in the case of leveraging fine-grained tasks. GLTO contemplates two possible scenarios when tasks are used. In case the code enters a master or single region, it indicates that a single `GLT_thread` creates all the tasks and the remaining `GLT_threads` execute them. If our runtime detects this scenario, it uses a round-robin dispatch so that it can schedule the tasks to any of the `GLT_threads`. In contrast, if the code is not inside such region, each `GLT_thread` creates its own tasks and then executes them.

E. Nested Parallelism

Although nested parallel codes are rare, this type of parallelism may appear implicitly. For example, a code can present an OpenMP parallel `for` loop and, inside the loop, it can call an external library that is also accelerated via OpenMP directives. That code features nested parallelism and current pthread-based OpenMP solutions tend to offer low performance.

GLTO deals with nested parallelism applying the following policy. For the outer parallel level, the runtime divides the work as in the work-sharing case. If a nested level is found, each `GLT_thread` generates and executes the `GLT_ults` for the nested code. This mechanism avoids the oversubscription that impairs performance when the pthread-based OpenMP solutions are used.

F. Load Imbalance

The GLTO default configuration may be affected because of irregular parallel codes. More concretely, if there is a load imbalance among OpenMP tasks or in nested parallelism, the performance of our GLTO design suffers. However, in these cases, we take advantage of GLT to modify the number of work-unit queues inside the GLT PM. In some detail, if we use the `GLT_SHARED_QUEUES` environment variable, all the `GLT_threads` share the same queue, and the load imbalance is neutralized by enforcing a work-sharing behavior.

G. Specific Implementation Issues

Although GLT offers a common API for LWT libraries, the specific scheduling and management mechanisms depend on the underlying native LWT library. Therefore, these features may affect the performance behavior of the entire implementation. This aspect may not be noticeable when the GLT library is used directly. However, OpenMP relies on a master thread that handles all the thread structures and executes the serial code. Therefore, the primary `GLT_thread` cannot be changed. In LWT implementations it is common that the main execution becomes a schedulable item, so that it can be stolen (if the library allows work-stealing) by a non-primary `GLT_thread`. If this situation occurs, the master thread in OpenMP will not be the primary `GLT_thread` any longer.

This feature forced us to slightly modify the OpenMP runtime when MassiveThreads is used as the library under GLT because MassiveThreads allows that a thread steals the main execution. This modification does not allow the main thread to yield and, as a consequence, the potential performance improvement cannot be fairly measured.

TABLE I: Results of the OpenUH OpenMP Validation Suite 3.1 for the OpenMP runtimes.

	GNU	Intel	GLTO
OpenMP constructs	62	62	62
Used tests	123	123	123
Successful test	118	118	121/122
Failed Test	5	5	2/1

V. GLTO Validation

In this section, we show the results for the validation tests and compare them with those obtained with the Intel and GNU OpenMP runtimes.

The OpenUH OpenMP Validation Suite 3.1 is leveraged to test the OpenMP 3.1 specification. It consists of 123 benchmark tests that analyze 62 OpenMP constructs, including task parallelism. The suite employs an automatic approach to run different types of tests in *normal*, *cross*, and *orphan* modes [20].

Table I displays the results of executing the Validation Suite with the GNU, Intel, and GLTO runtimes. `gcc 6.1` was used for the GNU runtime and `icc 16.0.1` for Intel and GLTO. GLTO is combined with the `icc` compiler because it can accommodate a larger number of tests than the `clang` tool does. Those results expose that, while the Intel and GNU runtimes pass 118 tests, our OpenMP implementation succeeds in 121 or 122, depending respectively on the use of Argobots/Qthreads or MassiveThreads as the underlying library. The failed tests for GLTO over Argobots/Qthreads are the `omp_taskyield` and the `omp_task_untied`. The reason is that, once a task is bound to a `GLT_thread`, there is no work stealing, so the task is resumed in the same `GLT_thread` and the test counts the number of tasks that have been created and started by one `GLT_thread` and resumed by another `GLT_thread`. If we use MassiveThreads, which allows work stealing, only `omp_taskyield` fails, because there are not enough tasks that change from one `GLT_thread` to another. The same tests fail for the GNU and the Intel runtimes, which indicates that they do not integrate any mechanism for migrating tasks from one OMP thread to another once the tasks have been created. With these runtimes, the tests fail in the *normal* and *orphan* modes. The other failed test is the `omp_task_final`, because the task marked as *final* is not directly executed.

The general test failure when task parallelism is used indicates that the solutions adopted are not as solid as in the case of work-sharing constructs. This agrees with the fact that task management was added as a separate mechanism.

VI. Performance Evaluation

In this section we first describe the hardware and software employed for our experimental evaluation. Then we present the results for the different experimental scenarios.

A. Hardware and Software

The results were obtained on a 36-core (72-hardware thread) machine equipped with two 18-core Intel Xeon E5-2699 v3 (2.30 GHz) CPUs and 128 GB of RAM. The libraries are Intel OpenMP Runtime 20160808, GOMP 6.1, GLT 01-2017, Argobots 01-2017, Qthreads version 1.10, and MassiveThreads version 0.95. GLT, GOMP, and LWT libraries were compiled with `gcc 6.1`. The Intel OpenMP implementation and GLTO were compiled with `icc 16.0`.

The OpenMP environment variables were set to the values that show higher performance for each scenario. OMP_NESTED and OMP_BIND_PROC were set to `true` for all tests. The former was asserted in order to measure the actual nested management, because otherwise the OpenMP runtime treats nested parallelism as one level of parallelism and sequential code. The latter was asserted in order to prevent thread migration among cores. Moreover, for the POSIX-based OpenMP implementations, the environment variable `OMP_WAIT_POLICY` was initialized to `active` for work-sharing codes and to `default` for task parallelism. In the work-sharing codes, keeping active the OMP threads improves the time of work completion. In the task parallelism cases, conversely, the `active` mode increments the overhead caused by contention in the work-stealing mechanism.

B. OpenMP as Environment Creator

One way to use OpenMP is by adding just a `#pragma omp parallel` embracing all the application code. The OpenMP threads can be handled as simple Pthreads, distinguished by their thread ID. In this manner, the user only benefits from OpenMP in the thread creation and joining operations while, inside the code, it is the user's responsibility to divide the work among them.

In this experiment we use the UTS Benchmark [31], a code parallelized with OpenMP that measures the performance attained when executing an exhaustive search on an unbalanced tree. The tree is built at execution time by using a divisible random number generator that splits the structure, enabling a parallel processing while still generating a deterministic tree.

Figure 4 shows the performance when UTS is executed on top of the OpenMP implementations using the problem size T1XXL, which corresponds to the larger problem instance that fits into memory. The results are the average of 50 executions and the represented error bars reveal reduced execution time variations. The bars labeled as GCC and ICC correspond to the GNU and Intel solutions, respectively; GLTO(ABT), GLTO(QTH), and GLTO(MTH) correspond to the GLTO implementations over Argobots, Qthreads, and MassiveThreads, respectively. The results expose close performances among almost all the OpenMP solutions. The reason for this behavior is that the code only employs OpenMP for the environment set up, and the interactions among threads are then managed by the programmer's code. The performance difference between GCC and the other solutions is caused by the different compiler outputs.

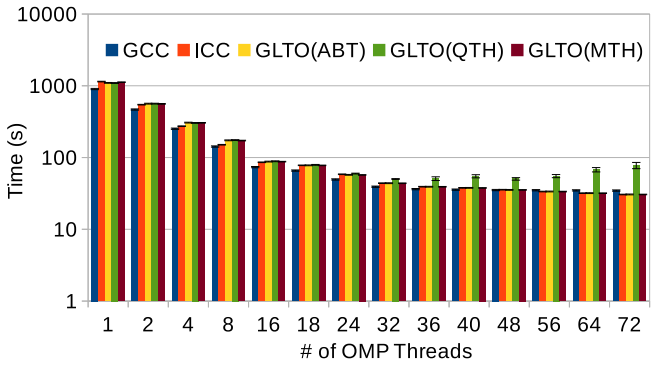

Fig. 4: Execution time for the UTS benchmark (T1XXL size) on top of OpenMP runtimes increasing the number of OpenMP threads.

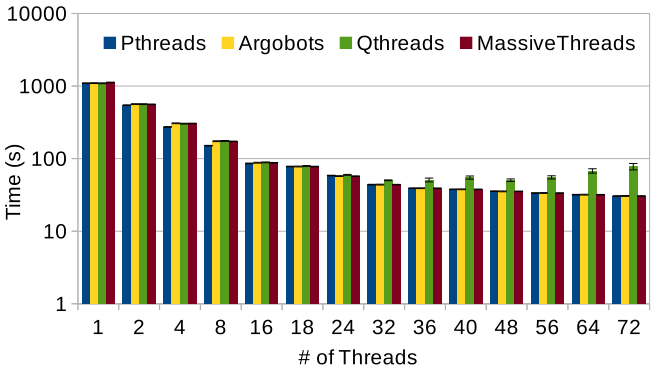

Fig. 5: Execution time for the UTS benchmark (T1XXL size) on top of pthreads and native LWT solutions increasing the number of threads.

There is a noticeable performance loss when GLTO is used on top of Qthreads. In order to discard a bad design in the GLTO runtime we translated UTS from the pthreads version [32] to the native LWT APIs. The results are reported in Figure 5 (as in the previous case, they are the average of 50 executions), and reveal that the increase of time is because of the LWT implementation itself. The main reason is that two threads are bound to the same CPU and the Qthreads implementation protects all the memory words with mutex regions, adding a noticeable contention when we increase the number of OS threads.

In summary, the choice among OpenMP implementations in this scenario is not critical for the application performance.

C. OpenMP in a Compute-Bound Code

This case study reflects the most frequent target for OpenMP. It mainly consists of an iterative code that is executed a certain number of times. This code configuration is highly favorable for OpenMP, and it is where the runtimes may exploit a substantial fraction of the hardware parallelism. In order to study this scenario, we have chosen the CloverLeaf mini-app [33], which solves the compressible Euler equations

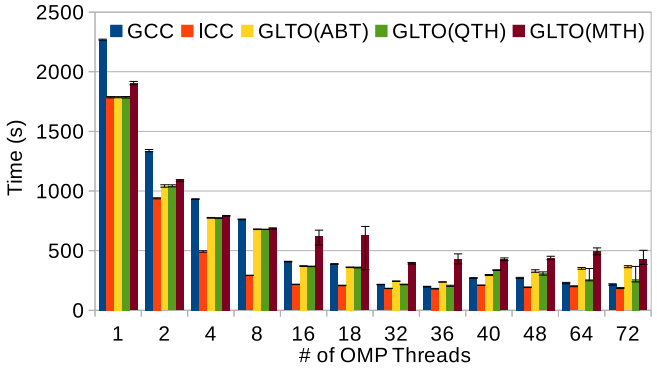

Fig. 6: Execution time for the CloverLeaf mini-app (clover_bm4.in size) on top of OpenMP runtimes increasing the number of OpenMP threads.

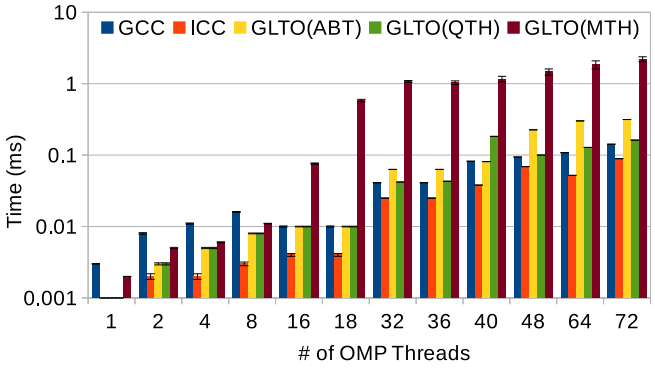

Fig. 7: Execution time for the work assignment mechanism in OpenMP runtimes increasing the number of OpenMP threads.

on a Cartesian grid, using an explicit second-order accurate method. Each cell stores three values: energy, density, and pressure, and a velocity vector is stored at each cell corner. This organization of the data, with some values at cell centers and others at cell corners, is known as a staggered grid. This code is written in Fortran.

The main part of the mini-app is a `for` loop that is executed 2,955 times. The loop is divided into several kernels, each calculating a value of the cells using `#pragma omp parallel for` directives. Concretely, 114 parallel `for` loops are executed 2,955 times, resulting in a total of 336,870 executions. Figure 6 depicts the average of 50 executions of the application for each of the OpenMP solutions using the clover_bm4.in problem instance. In this scenario the time variation is slightly larger for MassiveThreads because of the internal work-stealing mechanism. In addition, the mechanism implemented by the GNU and Intel runtimes (labeled as GCC and ICC, respectively) in the work-sharing constructs performs better. As argued earlier, Intel and GNU just pass the function pointer to be executed to the threads, while the GLTO implementation creates as many ULTs as `GLT_threads`.

In order to analyze this time gap we have measured the time spent in the work assignment step inside the OpenMP runtime. Figure 7 shows the difference among OpenMP implementations, demonstrating that the non-LWT solutions deploy the most efficient mechanism. Although the single time difference among implementations is barely noticeable, repeating this operation the over 336,000 times of the entire execution yields a nonnegligible total time difference.

In contrast with the scenario presented in Section VI-B, the well-developed work-sharing construct mechanism benefits from using the pthread-based OpenMP solutions.

D. OpenMP with Nested Parallelism

Nested parallelism is not a common OpenMP pattern, but it may appear hidden to the user. Moreover, an increasing number of cores may allow programmers to introduce several levels of parallelism in order to extract all the computational power of future hardware.

```
1  #pragma omp parallel for
2  for (int i = 0; i < N; i++) {
3   #pragma omp parallel for firstprivate(i)
4   for (int j = 0; j < N; j++){
5     null_code(i,j);
6   }
7  }
```

Listing 1: OpenMP nested parallelism code example.

Due to the defective design of the nested parallelism mechanism in current OpenMP implementations, it is extremely difficult to find an application that exploits this parallel paradigm. In order to study this behavior, we have thus implemented a microbenchmark that measures the overhead of managing nested parallel codes inside the OpenMP runtimes. This test is composed of two `for` loops accelerated via `#pragma omp parallel for` directives without any code in order to measure the management time as showed in Listing 1.

Figure 8 reveals the performance difference among the OpenMP implementations when the outer and inner loop comprise 100 iterations, and Figure 9 does the same with 1,000 iterations for each loop. These results are the average of 1,000 repetitions. The execution time of the pthread-based implementation is, at least, one order of magnitude higher than

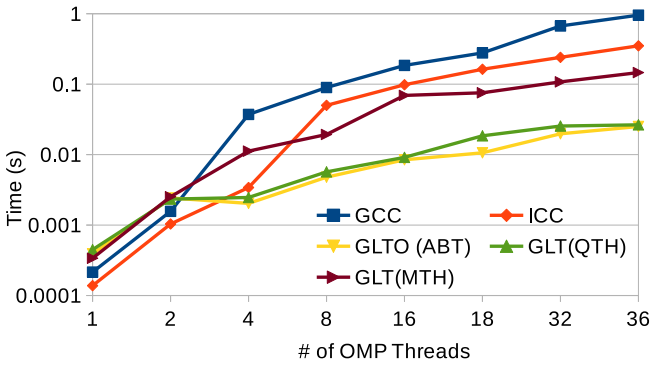

Fig. 8: Execution time for the nested parallel code on top of OpenMP runtimes with 100 iterations in the outer loop increasing the number of OpenMP threads.

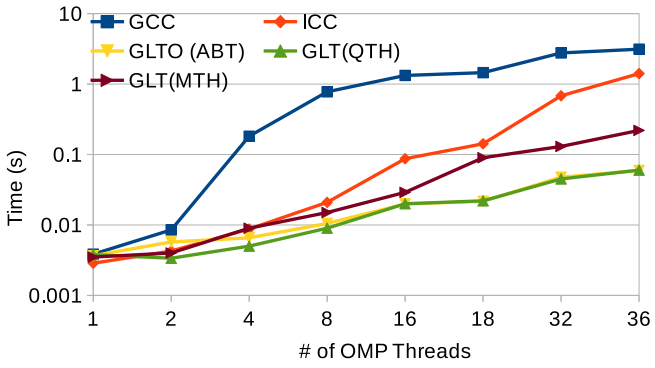

Fig. 9: Execution time for the nested parallel code on top of OpenMP runtimes with 1,000 iterations in the outer loop increasing the number of OpenMP threads.

that of GLTO over Argobots and Qthreads. The performance of GLTO over MassiveThreads is affected by the design issue discussed in Section IV-G. In this case, the action of the master thread has a strong impact in the overall execution time because it needs to yield in order to execute the inner loop code. As GLTO over MassiveThreads does not allow this, the work of the master thread needs to be stolen by the remaining threads.

The problem with the pthread-based OpenMP implementations is due to CPU core oversubscription. On the one hand, the GNU solution creates a number of threads for the outer loop, and for each of the iterations of the outer loop a new team of threads is created for the inner loop. This approach does not reuse idle threads to save the context of each outer loop thread. On the other hand, the Intel implementation acts like GNU's for the outer loop, but Intel solution reuses the idle threads. Nevertheless, Intel still creates new teams for the inner loop. GLTO only creates `GLT_ults` and, as a result, the system is not affected by oversubscription, yielding a reduced performance loss.

Table II summarizes the actual number of used threads when the environment variable OMP_NUM_THREADS is set to 36 in the scenario with 100 iterations for each loop. The total number of Pthreads exceeds the 72 CPU cores of the machine, and the management overhead impairs performance. In our test, with OMP_NUM_THREADS set to 36, setting 100 iterations in the outer loop implies that GNU generates 100 x 35 threads in order to complete the teams for the nested parallel constructs that are composed by 1 thread of the outer loop and 35 new threads for the inner loop. If we include the 36 threads of the main team, we have the 3,536 threads. Intel just creates 36 teams of 36 threads (1,296 threads) once, and then reuses the idle ones. Although GLTO creates 3,500 `GLT_ults`, those are lighter to handle than the OS threads and, in addition, they do not incur in oversubscription because just 36 `GLT_threads` are created. For the larger scenario, the total number of threads is multiplied by a factor of 10 (corresponding to the number of iterations of the outer loop).

TABLE II: Number of created and reused threads for each OpenMP implementation in nested parallel constructs with 100 iterations for each loop.

OpenMP Implementation	Created Threads	Reused Threads	Created `GLT_ults`
GCC	3,536	0	—
Intel	1,296	2,240	—
GLTO	36	0	3,500

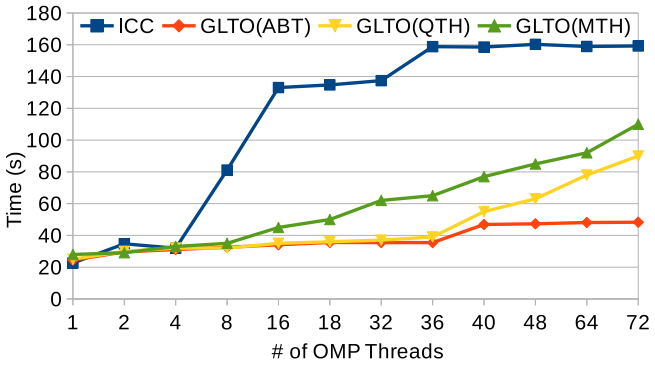

Fig. 10: Execution time of CG with a granularity 10 on top of OpenMP runtimes increasing the number of OpenMP threads.

In summary, for nested parallelism the use of the LWT implementations provides a performance improvement against the Pthread solutions.

E. OpenMP in Task Parallelism

To study the performance in this scenario, we selected a conjugate gradient (CG) benchmark. In mathematics, the CG method is an algorithm for the numerical solution of symmetric positive definite systems of linear equations. We have converted the OpenMP `#pragma omp parallel for` directives in the implementation of CG [34] into `#pragma omp task` directives. In our implementation, a single thread acts as a producer while the remaining threads perform the consumer actions. The input matrix is the `bmwcra_1` with a total number of 14,878 rows. The code transformation allows us to adjust the task granularity and the number of tasks. Here we show the result for granularities of 10, 20, 50, and 100 rows per task, which result in 1,488, 744, 298, and 149 tasks, respectively. We study the effect of three parameters on performance: number of threads, task granularity, and number of tasks.

In contrast with the previous scenarios, we have not included the GNU OpenMP implementation because of two reasons. First, the original CG implementation uses the Intel Math Kernel Library [35] and, therefore, the comparison between this library and other GNU-available solutions would not be fair. Second, the mechanism used in GOMP in order to deal with task parallelism is totally different from that implemented by the Intel OpenMP runtime.

Figures 10 to 13 display the results for granularities of 10, 20, 50, and 100 rows per task. Those results reflect the

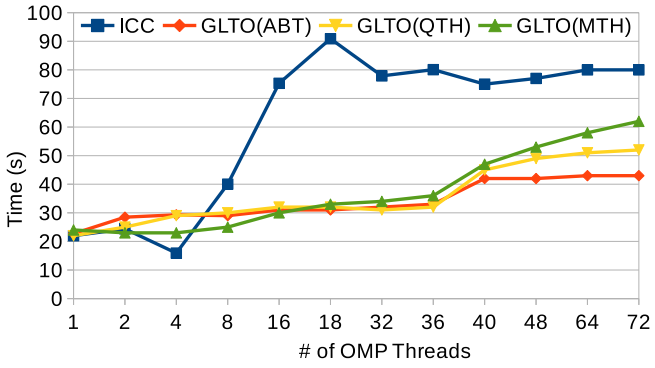

Fig. 11: Execution time of CG with a granularity 20 on top of OpenMP runtimes increasing the number of OpenMP threads.

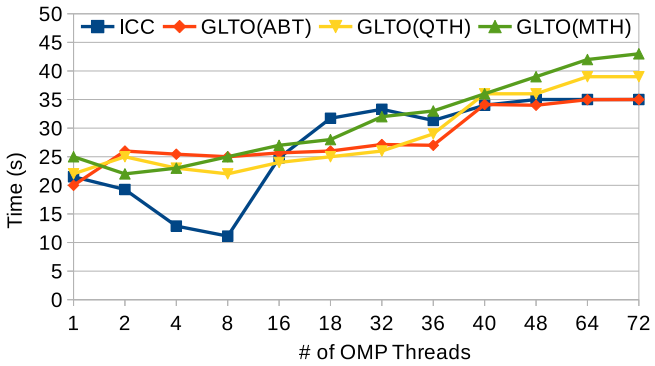

Fig. 12: Execution time of CG with a granularity 50 on top of OpenMP runtimes increasing the number of OpenMP threads.

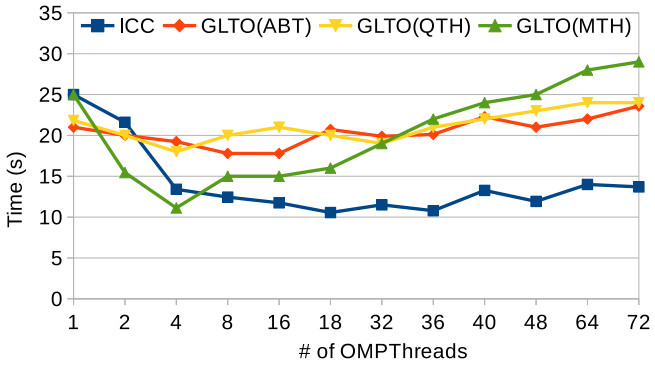

Fig. 13: Execution time of CG with a granularity 100 on top of OpenMP runtimes increasing the number of OpenMP threads.

average time of 1,000 executions. Since a smaller number of tasks implies less runtime overhead, it makes sense that the execution time decreases when moving from fine-grained to coarse-grained tasks. However, the execution time of the GLTO solutions is much lower than that of the Intel OpenMP runtime for granularities of 10 and 20 (Figures 10 and 11, respectively). For this benchmark, only GLTO on top of Argobots maintains an acceptable performance for a granularity of 50 (Figure 12). If we compare the GLTO options among them, we observe the effect of different implementation details of the underlying libraries. On the one hand, GLTO(ABT) exhibits almost flat performance lines for the 4 scenarios, which means that the interaction between the `GLT_threads` is almost non-existent, while GLTO(MTH) and GLTO(QTH) suffer from contention (the execution time increases as the number of threads does). The former because of work-stealing between `GLT_threads` and the latter because of the mutex-protected access to each word in memory.

In the Intel OpenMP runtime, the execution time gap between fine-grained and coarse-grained tasks is critical. However, this solution shows good performance up to 4 threads in the finest-grained scenario (Figure 10) and up to 8 for granularities of 20 (Figure 11) and 50 (Figure 12) rows per task. Once this number of threads is reached, the performance of Intel OpenMP drops. This loss is caused by a combination of two causes: 1) the contention introduced by the work-stealing mechanism; and 2) an internal cut-off mechanism implemented in the runtime. In this scenario, the producer thread creates the tasks into its own task queue while the consumers try to gain access to that queue, in order to steal a task each time. Moreover, the cut-off mechanism is triggered once a certain number of tasks are queued—256 in the case of the Intel OpenMP runtime—and then the new tasks are executed directly as a sequential code. It is important to remark that a task that is directly executed is less expensive than a queued task. This is because the latter needs to be handled by the runtime scheduler and thus has to wait to be executed. If task creation is faster than task consumption, the cut-off mechanism is triggered and the performance is maintained. Conversely, if task creation is slower than task consumption, the size of the task queue never reaches the limit to trigger the mechanism, and all tasks must pass through the internal OpenMP task mechanism, decreasing performance.

We have analyzed those reasons in detail by measuring both the number of queued tasks and the cut-off mechanism separately. Table III summarizes the percentage of the number of queued tasks for each granularity size. Here it is remarkable that a reduced number of non-queued tasks benefits the overall performance. That suggests that the OpenMP task management needs more development effort.

Additionally, we have implemented a test where a single thread creates 4,000 tasks. We have executed that test with three different cut-off values: the default (256), a number where all the tasks are queued (4,096), and a configuration where most of the tasks are executed directly (16). Figure 14 shows the effect of incrementing the number of OpenMP

TABLE III: Percentage of queued tasks for each task granularity configuration.

# OpenMP Threads	Task Granularity			
	10	20	50	100
1	100	100	100	100
2	80	93	84	100
4	88	81	63	100
8	90	97	39	100
16	94	100	100	100
18	94	100	100	100
32	95	100	100	100
36	100	100	100	100
40	100	100	100	100
48	100	100	100	100
64	100	100	100	100
72	100	100	100	100

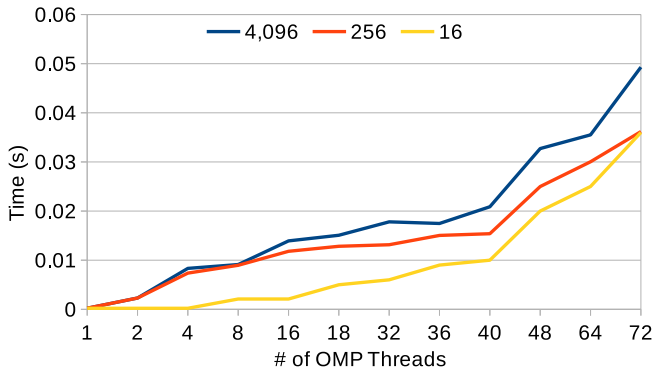

Fig. 14: Execution time of executing 4,000 tasks with three different cut-off values in the OpenMP Intel runtime increasing the number of OpenMP threads.

threads. The largest size (labeled as 4,096) exposes the contention time because, in this case, all the tasks pass through the runtime mechanism so that, adding more threads, increments the total cost of accessing the task queue. In contrast, the smaller value offers an acceptable performance though, when more than 16 threads are used, the producer is not fast enough to create tasks and to complete the queue. At that point, contention appears. Up to 8 threads, the execution time is almost the same as that obtained with the sequential code. Although this cut-off value should be better than the default number, 16 is a small number of queued tasks.

In contrast with other scenarios, the Intel OpenMP runtime outperforms the GLTO implementations for the coarse-grained problem (Figure 13). Although all the tasks are queued and scheduled, the time spent in the task execution stage prevents that the threads ask immediately for more work reducing contention. In this case, the behavior of the Intel OpenMP runtime is close to that observed in the `for` loop case. Also, the work dispatch in GLTO does not help because work stealing is not leveraged. As an exception, GLTO over MassiveThreads (GLTO(MTH)) outperforms the other alternatives up to 4 threads because this library does work stealing by default.

Summarizing, the results in the Intel OpenMP implementation indicate that, compared with LWT-based solutions, it cannot deal successfully with the fine-grained parallel paradigm. In that case, a LWT-based approach should be selected.

VII. Selection of the Appropriate OpenMP Solution

Reviewing the performance results presented in this paper, the choices for two of the scenarios are well defined. Concretely, compute-bound applications composed by `for` loops benefit from the pthread-based implementation thanks to the accurate work distribution mechanism that is, at present, more efficiently implemented than the LWT work dispatch by means of ULTs. In contrast, nested parallel applications benefit from the use of a LWT-based solution due to the reduced management cost of the ULTs, compared with the expensive context-switch mechanism of the Pthreads.

When OpenMP is employed as an environment creator, the difference between LWT and pthread-based approaches does not show any clear winner. In this scenario, it is the application code which determines performance.

For applications leveraging task parallelism, there are more aspects to be taken into account. While fine-grained tasks tend to benefit LWT-based solutions, coarse-grained codes favor the use of pthread-based runtimes. The performance of a code with intermediate granularity depends on two factors. The first is the number of threads. A reduced number of these (compared with the total number of cores) performs better for the Intel solution because of the reduced contention and the cut-off mechanism. However, increasing the number of threads avoids this mechanism and, at the same time, introduces contention due to thread interaction. The second factor is the time gap between task creation and execution, which is critical in order to trigger the cut-off mechanism. LWT-based solutions do not experience a high performance drop. Instead, they are just affected by the interaction among OS threads. In this case, the best choice for the GLTO implementation employs Argobots as the underlying library thanks to the close to null interaction between `GLT_threads`.

Although scientific applications are usually implemented following a single pattern, it may well occur that more than one pattern is used inside the application code. In that case, the best choice will depend on the pattern dominating the performance of the application.

VIII. Conclusions

We have presented a new OpenMP implementation on top of the GLT API named GLTO [36]. This library presents a common API for LWT solutions and it is currently implemented on top of Argobots, MassiveThreads, and Qthreads. The GLTO runtime allows us to compare OpenMP codes with different underlying LWT solutions without modifying the code.

We discussed the design decisions taken during the implementation of GLTO, and we exposed how the runtime behaves in different OpenMP scenarios. In addition, we have tested our runtime implementation with the OpenUH OpenMP Validation Suite 3.1, attaining better results than the reference runtimes. Moreover, we have presented a fair comparison between the

well-known GNU and Intel OpenMP runtimes and our new approach in different OpenMP scenarios: environment creator, compute bound `for` loop-based codes, nested parallelism, and task parallelism. For each case, we have showed the performance difference and analyzed the reasons (if any) of the disparity of results.

Our results revealed that no OpenMP implementation is a clear winner. While pthread-based solutions outperform the LWT-based implementations in simple work-sharing constructs, the latter attain better results in nested and fine-grained task parallelism.

ACKNOWLEDGMENT

The Researchers from the Universitat Jaume I de Castelló were supported by project TIN2014-53495-R of the MINECO and FEDER, the Generalitat Valenciana fellowship programme Vali+d 2015. Antonio J. Peña is cofinancied by the Spanish Ministry of Economy and Competitiveness under Juan de la Cierva fellowship number IJCI-2015-23266. This work was partially supported by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

REFERENCES

[1] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang, "The sunway taihulight supercomputer: system and applications," *Science China Information Sciences*, vol. 59, no. 7, p. 072001, 2016.

[2] "TOP500 Supercomputer Sites," www.top500.org/, June 2016.

[3] "Pthreads API," computing.llnl.gov/tutorials/pthreads/.

[4] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[5] BSC, "The OmpSs programming model," http://pm.bsc.es/ompss/.

[6] D. Stein and D. Shah, "Implementing lightweight threads." in *USENIX Summer*, 1992.

[7] Microsoft MSDN Library, "Fibers."

[8] "Programming with Solaris Threads," docs.oracle.com/cd/E19455-01/806-5257/6je9h033n/index.html.

[9] L. V. Kalé, M. A. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, "Converse: An interoperable framework for parallel programming," in *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, April 1996, pp. 212–217.

[10] BSC, "Nanos++," pm.bsc.es/projects/nanox/.

[11] L. V. Kale and S. Krishnan, *CHARM++: A portable concurrent object oriented system based on C++*. ACM, 1993, vol. 28, no. 10.

[12] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," in *Concurrent Objects and Beyond*, ser. Lecture Notes in Computer Science, 2014, vol. 8665, pp. 222–238.

[13] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Proceedings of Workshop on Multithreaded Architectures and Applications*, April 2008.

[14] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, and P. Beckman, "Argobots: A lightweight threading/tasking framework," 2016, https://collab.cels.anl.gov/display/ARGOBOTS/.

[15] "Generic Lightweight Threads API," github.com/adcastel/GLT.

[16] A. Castelló, S. Seo, R. Mayo, P. Balaji, E. S. Quintana-Ortí, and A. J. Peña, "GLT: A unified API for lightweight thread libraries," in *Proceedings of the IEEE International European Conference on Parallel and Distributed Computing*, Santiago de Compostela, Spain, August 2017.

[17] "BOLT: A Lightning-Fast OpenMP Implementation," bolt-omp.org/.

[18] "LLVM project," http://openmp.llvm.org/.

[19] "Intel OpenMP Runtime Library," https://www.openmprtl.org/.

[20] C. Wang, S. Chandrasekaran, and B. Chapman, "An openmp 3.1 validation testsuite," in *Int. Workshop on OpenMP*, 2012, pp. 237–249.

[21] OpenMP Architecture Review Board, *OpenMP Application Programming Interface Version 4.5*, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, Nov. 2015.

[22] "PGI Compilers & Tools," http://www.pgroup.com/.

[23] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "Openuh: an optimizing, portable openmp compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, 2007.

[24] A. Castelló, A. J. Peña, S. Seo, R. Mayo, P. Balaji, and E. S. Quintana-Ortí, "A review of lightweight thread approaches for high performance computing," in *Proceedings of the IEEE International Conference on Cluster Computing*, Taipei, Taiwan, September 2016.

[25] P. E. Hadjidoukas and V. V. Dimakopoulos, "Nested parallelism in the ompi openmp/c compiler," in *European Conference on Parallel Processing*. Springer, 2007, pp. 662–671.

[26] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa, "Performance evaluation of openmp applications with nested parallelism," in *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. Springer, 2000, pp. 100–112.

[27] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "Forestgomp: an efficient openmp environment for numa architectures," *International Journal of Parallel Programming*, vol. 38, no. 5, pp. 418–439, 2010.

[28] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2011, pp. 49–56.

[29] P. Chang and W. Hwu, "Inline function expansion for compiling c programs," in *Procedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. ACM, 1989, pp. 246–257.

[30] "Clang project," http://clang.llvm.org/.

[31] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C. Tseng, "UTS: An unbalanced tree search benchmark," in *Languages and Compilers for Parallel Computing*. Springer, 2006, pp. 235–250.

[32] "The Unbalanced Tree Search (UTS) benchmark," https://sourceforge.net/projects/uts-benchmark/.

[33] "CloverLeaf miniapp," http://uk-mac.github.io/CloverLeaf/.

[34] J. I. Aliaga, H. Anzt, M. Castillo, J. C. Fernández, G. León, J. Pérez, and E. S. Quintana-Ortí, "Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors," *Conc. and Comp.: Practice and Experience*, vol. 27, no. 4, pp. 885–904, 2015.

[35] "Intel Math Kernel Library," https://software.intel.com/en-us/intel-mkl.

[36] "GLTO: Generic Lightweight Thread OpenMP," github.com/adcastel/glto-runtime.