On the Adequacy of Lightweight Thread Approaches for High-Level Parallel Programming Models

Adrián Castelló*[1], Rafael Mayo[1], Kevin Sala[2], Vicenç Beltran[2], Pavan Balaji[3], Antonio J. Peña[2]

Abstract

High-level parallel programming models (PMs) are becoming crucial in order to extract the computational power of current on-node multi-threaded parallelism. The most popular PMs, such as OpenMP or OmpSs, are directive-based: the complexity of the hardware is hidden by the underlying runtime system, improving coding productivity. The implementations of OpenMP usually rely on POSIX threads (pthreads), offering excellent performance for coarse-grained parallelism and a perfect match with the current hardware. OmpSs is a task oriented PM based on an ad hoc runtime solution called Nanos++; it is the precursor of the tasking parallelism in the OpenMP tasking specification. A recent trend in runtimes and applications points to leveraging massive on-node parallelism in conjunction with fine-grained and dynamic scheduling paradigms. In this paper we analyze the behavior of the OpenMP and OmpSs PMs on top of the recently emerged Generic Lightweight Threads (GLT) API. GLT exposses a common API for lightweight thread (LWT) libraries that offers the possibility of running the same application over different native LWT solutions. We describe the design details of those high-level PMs implemented on top of GLT and analyze different scenarios in order to assess where the use of LWTs may

*Corresponding author
[1]Universitat Jaume I de Castelló, 12071 Castelló de la Plana, Spain.
Email: {adcastel,mayo}@uji.es
[2]Barcelona Supercomputing Center (BSC).
Email: {ksala,vbeltran,antonio.pena}@bsc.es
[3]Argonne National Laboratory, Lemont, IL, USA.
Email: balaji@anl.gov

benefit application performance. Our work reveals those scenarios where LWTs overperform pthread-based solutions and compares the performance between an ad hoc solution and a generic implementation.

Keywords: Lightweight Threads, OpenMP, OmpSs, GLT, POSIX Threads, Programming Models

1. Introduction

In the past few years, the number of cores per processor has increased steadily, reaching impressive counts such as the 260 cores per socket in the Sunway TaihuLight supercomputer [1], which was ranked #1 for first time 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 of cores per socket. Extracting the computational power of those machines will thus require efficient libraries and programming models (PMs). The most popular approaches to obtain acceptable on-node performance rely on POSIX threads (pthreads) application programming interface (API) [3] or directive-based PMs such as OpenMP [4] or OmpSs [5].

Directive-based PMs are usually implemented on top of the pthreads API, which matches perfectly the current hardware and coarse-grained parallelism. Because of the high cost of management, however, it fails to accommodate new software paradigms that target dynamically scheduled, fine-grained parallelism.

Several lightweight thread (LWT) libraries have been implemented in the last years to tackle fine-grained and dynamic software requirements [6]. Each LWT solution features its own PM and target environment. Some of these solutions 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], [16] is an effort to unify these LWT solutions under a unique PM in order to foster productivity and portability with negligible overhead. This lightweight layer offers the common functionality of LWT solutions and is currently implemented on top of MassiveThreads, Qthreads, and Argobots. As a result, a runtime or application based on GLT requires no changes in order to be executed on top of any of these three LWT solutions.

In this paper we analyze common OpenMP and OmpSs parallel patterns and discuss how LWTs deal with them, in comparison with traditional approaches. While OpenMP is the most widely-adopted directive-based PM, OmpSs is the precursor of task-parallelism and features a runtime which leverages a custom LWT implementation. We evaluate our implementations and compare their performances with those obtained when using the original runtimes.

In order to perform the comparison, we have implemented the OpenMP and OmpSs runtimes on top of the GLT API, called Generic Lightweight Thread OpenMP (GLTO) and Generic Lightweight Thread OmpSs (GOmpSs), respectively. 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. Our OmpSs version is basedon the Nanos++ library [10] from the Barcelona Supercomputing Center (BSC).

Our study reveals that the use of LWTs instead of pthread-based approaches in the OpenMP PM may yield performance benefits, depending on the application nature. In addition, our results expose that the performance with the OmpSs runtime implemented on top of GLT is close to that obtained with an ad-hoc implementation, improving the task management in fine-grained code tasks.

In summary, the main contributions of this paper are as follows: (1) design of OpenMP and OmpSs runtimes on top of a generic LWT API; (2) analytical study of the relationship between high-level PMs and LWT solutions; and (3) the experimental performance evaluation of that relationship in different OpenMP and OmpSs scenarios.

The rest of the paper is organized as follows. Section 2 provides some background information about OpenMP, OmpSs, and GLT. Section 3 reviews a few related works. Section 4 details the GLTO implementation and Section 5 describes the GOmpSs implementation. Section 6 provides an in-depth performance analysis of the distinct scenarios. Finally, Section 7 contains our conclusions.

2. Background

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

2.1. OpenMP

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

The OpenMP runtimes are commonly composed of two main parts: the work-sharing constructs and task parallelism. In contrast to 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 task queue shared by all the threads, the Intel implementation incorporates one task queue for each thread and integrates workstealing 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.

2.2. OmpSs

OmpSs [20], developed at BSC, aims to provide an efficient programming model for heterogeneous and multicore architectures. It embraces a task-oriented execution model similar to the OpenMP tasking features.

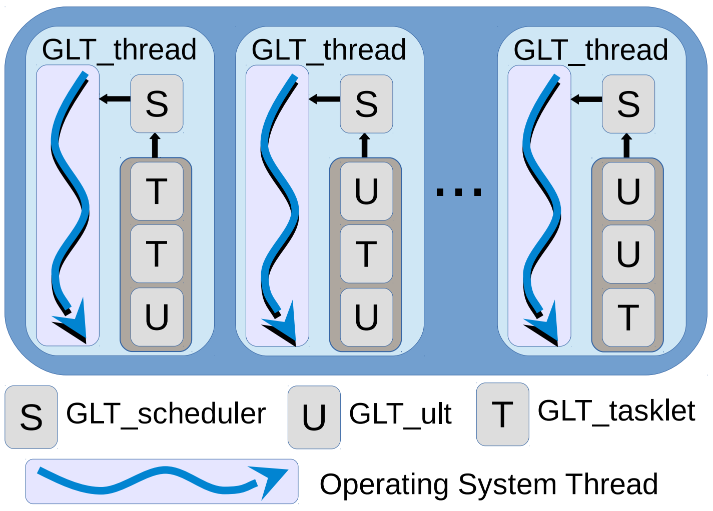


Figure 1: PM offered by the GLT library.

OmpSs detects data dependencies between tasks at execution time, with the help of directionality clauses embedded in the code, and leverages this information to generate a task graph during the execution. This graph is then employed by the runtime to exploit the implicit task-parallelism, via a dynamic out-of-order, dependency-aware schedule. This mechanism provides a means to enforce the task execution order without the need for explicit synchronization. This PM is task-oriented and, therefore, it does not support work-sharing constructs.

2.3. 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 defined and implemented for three general-purpose LWT solutions: MassiveThreads, Qthreads, and Argobots.

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 by Argobots only, these are implemented on top of ULTs for Qthreads and MassiveThreads. `GLT_scheduler` acts differently depending on the underlying library and it may affect the performance of the PM but not the final result of the execution.

In principle adding an extra software layer between the user application and the underlying libraries may impact performance; however, GLT does not add any significant overhead because it offers a header-only version that allows the compilers to avoid the extra calls by embedding the LWT code by means of `static inline` declarations [21].

Despite some LWT solutions offer an API of more than 300 functions, GLT offers just 52 functions grouped in 7 modules: Setup, Work Unit, Mutex, Barrier, Condition, Util, and Key. It has been demonstrated that the reduced set of instructions that forms the GLT API are sufficient for implementing any programming pattern [16], and high-level PM [22] on top of the LWT solutions.

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.

3. Related Work

The OpenMP standard is currently supported by a significant number of compilers, including both open source and vendor solutions. Although the current OpenMP specification corresponds to version 4.5 [23], 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. Other compilers are one or more steps behind those solutions. For example `pgcc` [24], from the Portland Group, and OpenUH [25] support version 3.1 of the OpenMP specification.

Supporting an OpenMP specification implies 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 occurs in the Intel implementation, which can be linked with code built by the `clang` compiler.

OmpSs is a task-oriented PM which was the precursor of the tasking parallelism in OpenMP. Its development focuses on different tasking features such as automatic detection of task dependencies. At this time, this PM is only supported by the Mercurium compiler [26] and the Nanos++ runtime.

In the field of LWT libraries, the work in [6, 9, 12, 13, 14] introduces distinct LWT definitions, discuss implementation details, and analyze performances. The work in [27] 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 been explored in the past. In [28] and [29], nested parallelism is analyzed and resolved by means of LWT solutions. Moreover, the effect of OpenMP implementations when executed in NUMA architectures is depicted in [30] and scheduling for task-parallelism has been studied in [31].

In a previous work, we analyzed the behavior of the OpenMP PM over LWTs [22]; in this work we expand our previous work to the analysis of the OmpSs PM and a completely different runtime system, in order to generalize our conclusions.

Although OmpSs relies on top of a custom LWT solution (Nanos++), there is no other released implementation that makes use of standard LWT libraries. Therefore, with this work, we study the general behavior of the OmpSs PM on top of LWTs.

To the best of our knowledge, this is the first paper that analyzes OpenMP and OmpSs on top of LWT solutions discussing the general adequacy of the use of LWTs for the implementation of runtimes supporting directive-based PMs.

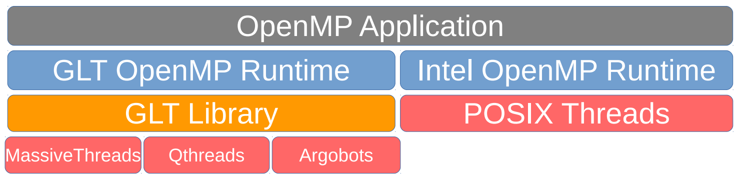


Figure 2: Software stack choices of an OpenMP code.

4. OpenMP over GLT

In this section we review the design decisions that were made in order to adapt the LLVM OpenMP runtime to the use of LWTs (GLTO).

As argued in Section 1, our implementation is based on the BOLT project which is, in turn, based on LLVM. We selected this starting point because both the runtime and the `clang` compiler [32] are open source. In addition, this runtime can be linked from code generated with the Intel compiler.

4.1. GLTO Interactions

GLTO offers a complete implementation of OpenMP 4.0 for C, C++, and Fortran codes. GLTO can be linked with code generated with the `clang` or `icc` compilers. Figure 2 shows that an OpenMP code compiled with these tools can be linked with 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.

4.2. GLTO Implementation Details

LWT libraries use two threading levels. The lowest level comprises 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 [23], our GLTO implementation responds to the definition of the `OMP_NUM_THREADS` environment variable creating as many `GLT_thread`s as OpenMP threads are requested by the user. As depicted in Figure 3, `GLT_thread`s are bound to CPU cores and are spawned when the library is loaded. They are in charge of executing `GLT_ult`s 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.

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

When exploiting task-parallelism (right-hand side of Figure 3), each OMP task is also transformed into 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 the operation modes of GLTO in each scenario.

4.3. GLTO Work-sharing Construct

For work-sharing constructs, our OpenMP solution mimics the mechanism that the GNU and Intel runtimes feature. The master thread assigns the function pointer to each thread in the runtime; 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 with the execution of the sequential code until a new parallel region is detected.

In GLTO, the work is dispatched 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.

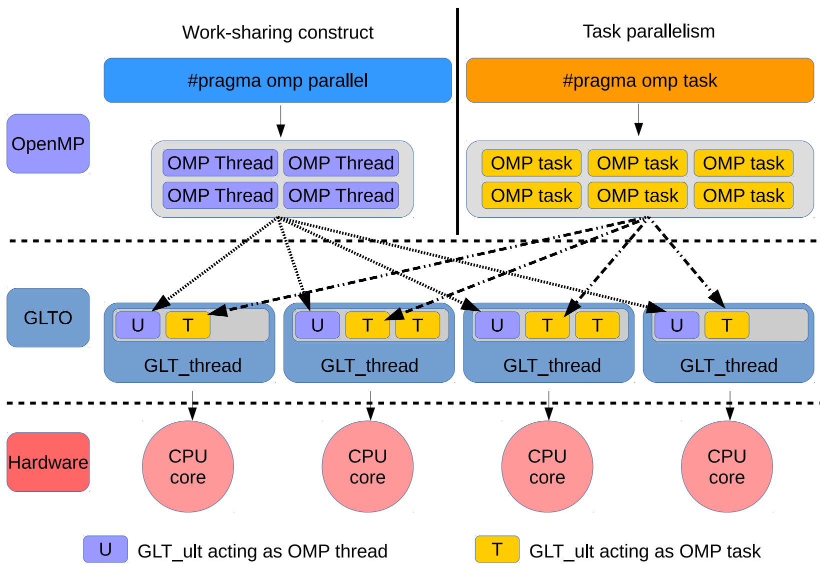


Figure 3: Relationship between OpenMP code and the GLTO implementation.

4.4. GLTO Task Parallelism

In contrast with work-sharing structures, the task-parallel 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, particulary for fine-grained tasks. GLTO contemplates two possible scenarios when tasks are used. In case the code enters a `master` or `single` region, a unique `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 a region, each `GLT_thread` creates its own tasks and executes them.

4.5. GLTO Nested Parallelism

Nested parallel codes are not common inside applications because its management is not as well designed as the parallel coarse-grained scenarios causing a performance drop. However, this type of parallelism may appear implicitly. For example, a code with an OpenMP parallel `for` loop may invoke, from inside the loop, an external library that is also parallelized via OpenMP directives. That code features nested parallelism and current pthread-based OpenMP solutions tend to offer low performance.

GLTO deals with nested parallelism by 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_ult`s for the nested code. This mechanism avoids the oversubscription that impairs performance when the pthread-based OpenMP solutions are used.

4.6. GLTO 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 implement a modified OpenMP runtime when MassiveThreads is used as the library under GLT because this LWT library allows that a thread steals the main execution task. This modification does not allow the main thread to yield and, as a consequence, the potential performance improvement cannot be fairly measured.

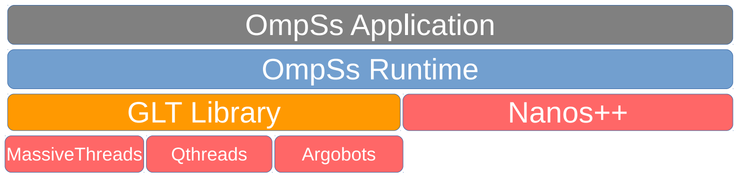


Figure 4: Software stack choices of an OmpSs code.

5. OmpSs over GLT

In this section we justify the design decisions that we made in order to adapt the OmpSs runtime to the use of LWTs (GOmpSs).

5.1. GOmpSs Interactions

GOmpSs offers a complete implementation of OmpSs version 16.06.3. OmpSs allows to select the underlying implementation by means of an environment variable thanks to its modular implementation (see Figure 4). We have maintained this feature in order to allow that the user selects the GLT or default implementations. With this work, OmpSs applications can run on top of Argobots, Qthreads, or MassiveThreads in addition to the custom Nanos++ solution. Therefore, once the OmpSs application has been built with the `mercurium` compiler, the underlying threading library can be selected by means of environment variables.

5.2. GOmpSs Implementation Details

As in the GLTO implementation, `GLT_threads` are bound to CPU cores (as depicted in Figure 5) and they are spawned when the library is loaded. In this runtime, those threads will execute all the OmpSs tasks created during the execution of the application. The number of the `GLT_threads` can be modified via the `GLT_NUM_THREADS` or the `--smp-workers` variables corresponding to the GLT or Nanos++ implementation, respectively.

5.3. GOmpSs Task Parallelism

As introduced in Section 2.2, OmpSs is a task-oriented PM and it is not designed for work-sharing constructs. For that reason, our study of both the OmpSs and GOmpSs runtimes is focused on the pragmas related to tasks for creation (`#pragma omp task`, `#pragma omp taskloop`) and synchronization (`#pragma omp taskwait`).

Figure 5 depicts how an OmpSs task is treated in the GOmpSs implementation. A `pragma task` generates an OmpSs call that creates a pending task. The runtime evaluates the task dependencies (if any), and once they are accomplished, it promotes the OmpSs task to "ready" state. Then, the runtime generates a `GLT_ult` associated with the OmpSs task that is placed in a shared queue and remains there until a `GLT_thread` executes it.

We have modified the default runtime environment of the GLT API forcing the underlying libraries to use just one shared queue. This feature is supported in the native GLT API and is enabled with environment variables. The main reason is that, once an OmpSs task has been promoted to "ready" inside the OmpSs runtime, all the dependencies have been already solved and it is ready to be executed. Therefore, there is not need of a dispatch policy or a certain scheduling. In that scenario, the use of a shared queue between the `GLT_thread`s helps with the load balance.

In contrast to with GLTO, there is no restriction on the master thread, and GOmpSs allows to change the `GLT_thread` that runs the main execution. The reason is that the main execution is also considered an OmpSs task. Therefore, it can be resumed by any of the `GLT_thread`s once a synchronization point is achieved.

6. Performance Evaluation

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

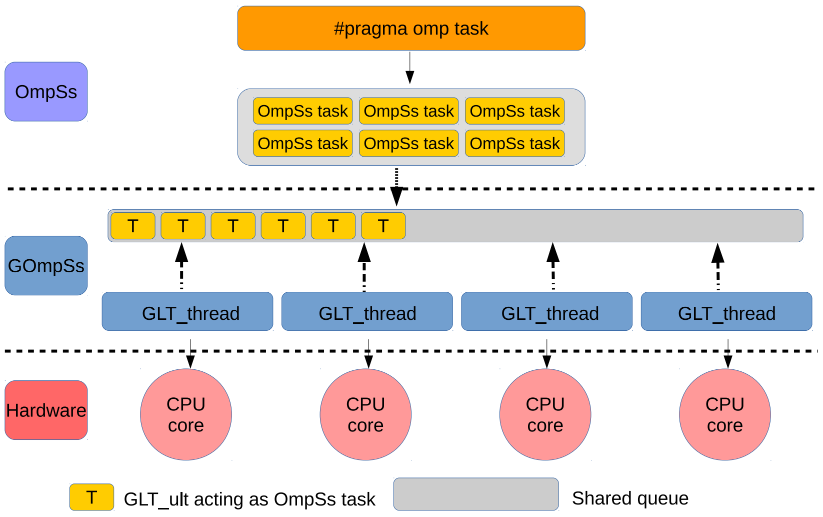


Figure 5: Relationship between OmpSs code and the GOmspSs implementation.

6.1. 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, OmpSs 16.06.3, GLT 01-2017, Argobots 01-2017, Qthreads version 1.10, and MassiveThreads version 0.95. GLT, GOMP, OmpSs, GOmpSs 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 reported 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-parallel cases, conversely, the `active` mode augments the overhead caused by contention in the work-stealing mechanism.

In the scenarios where OmpSs is used, the default environment values have been maintained and the performance-oriented OmpSs library is employed.

6.2. Work-sharing Constructs

We next present the results for work-sharing constructs. As OmpSs runtime is not designed for this kind of pragmas, we only compare OpenMP implementations in this section.

6.2.1. OpenMP in a Compute-Bound Code

Our first 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 often allows the runtimes to exploit a substantial fraction of the hardware parallelism. To study this scenario, we have chosen the CloverLeaf mini-app [33], which solves the compressible Euler equations 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 parallel loops. Figure 6a 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) for the work-sharing constructs attains up to 50% higher performance. The reason of the difference between pthreads-based OpenMP and LWT-based runtimes relies on the creation of `GLT_ult`s. As argued earlier, Intel and GNU just pass the function pointer to be executed to the threads, while the GLTO implementation creates as many `GLT_ult`s as `GLT_thread`s.

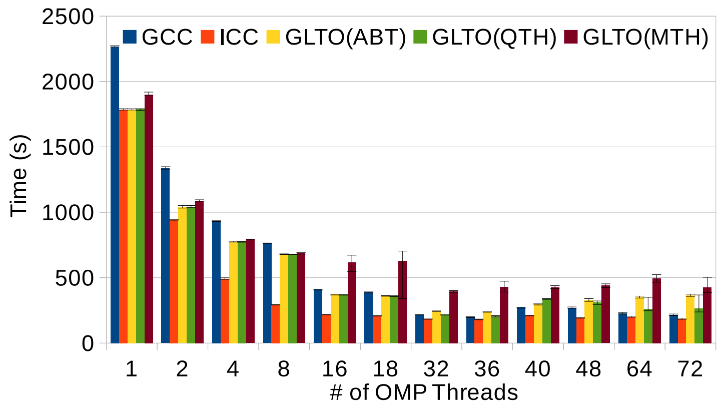
In order to analyze this time gap we have measured the time spent in the work assignment step inside the OpenMP runtime with a microbenchmark that measures the time spent distributing and joining the work. Figure 6b shows the difference among OpenMP implementations, demonstrating that the non-LWT solutions deploy the most efficient mechanism. Although the single time difference among implementations is barely noticeable, repeating this operation over 336,000 times of the entire CloverLeaf app execution yields a nonnegligible total time difference.

6.2.2. OpenMP with Nested Parallelism

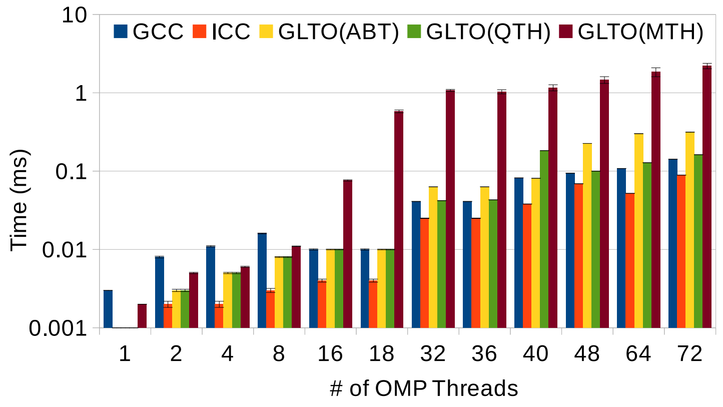
Nested parallelism is not a common OpenMP pattern, but it may appear hidden to the user. Moreover, an increasing number of cores may allow programmers to introduce several levels of parallelism in order to extract all the computational power of future hardware.

Due to the suboptimal 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` nested loops accelerated via `#pragma omp parallel for` directives with an empty code in order to measure the management time.

Figure 7a reveals the performance difference among the OpenMP implementations when the outer and inner loop comprise 100 iterations, and Figure 7b 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 that of GLTO over Argobots

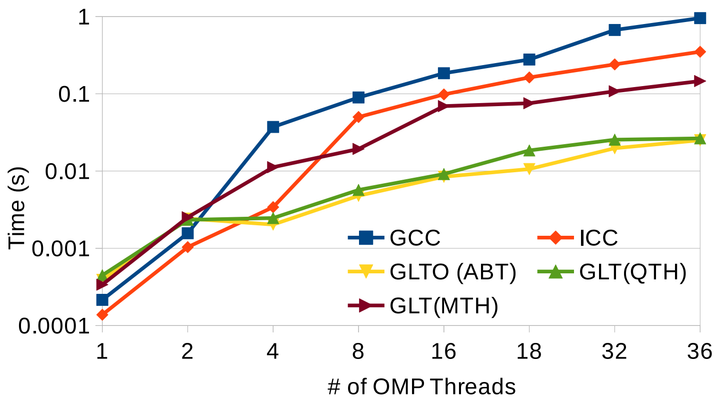


(a) CloverLeaf mini-app.

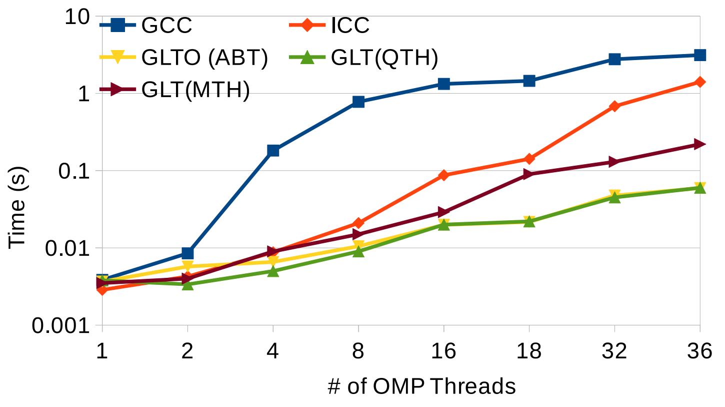


(b) Work assignment mechanism.

Figure 6: (a) Execution time for the CloverLeaf mini-app (clover_bm4.in size) on top of OpenMP runtimes increasing the number of OpenMP threads; and (b)Execution time for the work assignment mechanism in OpenMP runtimes increasing the number of OpenMP threads.



(a) 100 iterations in the outer loop.



(b) 1,000 iterations in the outer loop.

Figure 7: Execution time for the nested parallel code on top of OpenMP runtimes increasing the number of OpenMP threads.

and Qthreads. The performance of GLTO over MassiveThreads is affected by the design issue discussed in Section 4.6. In this case, the action of the master thread has a strong impact on the overall execution time because it needs 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 mimics GNU for the outer loop, but the 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, suffering a lower overhead.

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

6.3. Task-Parallelism

We next present the results from OpenMP and OmpSs with different already-existing applications. The comparison between those PMs is out of the scope of this work.

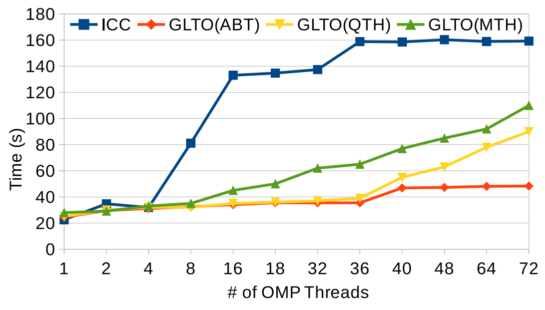
6.3.1. OpenMP in Task Parallelism

To study the performance in this scenario, we employ the 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 `bmwcra_1` from University of Florida Math Collection with a total number of 14,878 rows. The code transformation is leveraged 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.

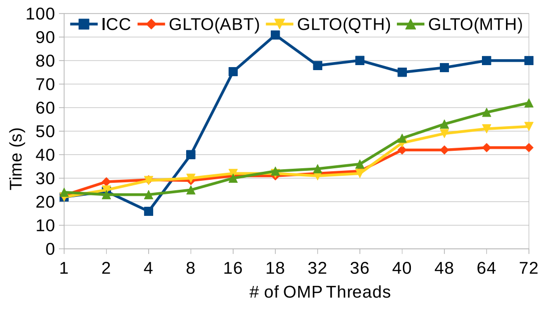
In contrast with the previous scenarios, we have not included the GNU OpenMP implementation because of the original CG implementation uses the Intel Math Kernel Library [35] and, therefore, the comparison between this library and other GNU-available solutions would not be fair.

Figure 8 displays the results for granularities of 10, 20, 50, and 100 rows per task. ICC, GLTO(ABT), GLTO(QTH), and GLTO(MTH) refer to Intel OpenMP, GLT on top of Argobots, Qthreads, and MassiveThreads respectively. Those results reflect the 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 (up to 3 times faster when using Argobots as the underlying solution) than that of the Intel OpenMP runtime for granularities of 10 and 20 (Figures 8a and 8b, respectively). For this benchmark, only GLTO on top of Argobots maintains an acceptable performance for a granularity of 50 (Figure 8c). 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 among the `GLT_threads` is almost non-existent. On the other hand, 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.

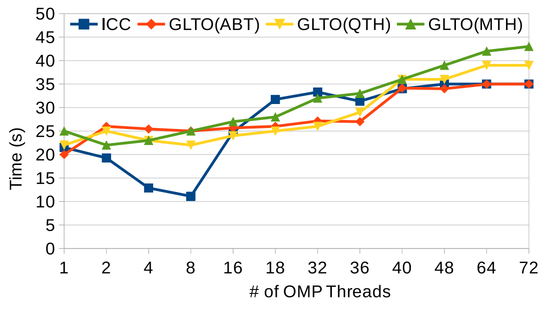
In the Intel OpenMP runtime, the execution time gap between fine-grained and coarse-grained tasks is critical. However, this solution shows good performance for up to 4 threads in the finest-grained scenario (Figure 8a) and up to 8 for granularities of 20 (Figure 8b) and 50 (Figure 8c) rows per task. Once this



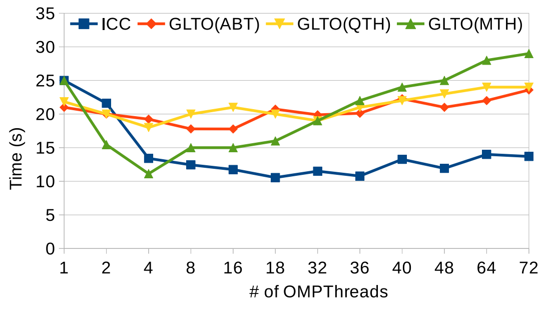
(a) Granularity 10 rows per task (1,488 tasks).



(b) Granularity 20 rows per task (744 tasks).



(c) Granularity 50 rows per task (298 tasks).



(d) Granularity 100 rows per task (149 tasks).

Figure 8: Execution time of CG with different task granularity on top of OpenMP runtimes increasing the number of OpenMP threads.

number of threads is reached, the performance of Intel OpenMP drops. This loss is caused by two combined 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 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 issues in detail by measuring both the number of queued tasks and the cut-off mechanism separately. Table 1 summarizes the percentage of the number of queued tasks for each granularity size. There it is relevant to note that a reduced number of non-queued tasks benefits the overall performance. That scenario suggests that the OpenMP task management needs additional development effort.

In contrast with the previous scenarios, the Intel OpenMP runtime outperforms the GLTO implementations for the coarse-grained problem (Figure 8d). Although all the tasks are queued and scheduled, the time spent in the task execution stage prevents that the threads immediately request 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 employ work stealing by default.

Table 1: Percentage of queued tasks for each task granularity configuration.

Task Granularity	# OMP Threads							
(rows per task)	1	2	4	8	16	18	32	36-72
10	100	80	88	90	94	94	95	100
20	100	93	81	97	100	100	100	100
50	100	84	63	39	100	100	100	100
100	100	100	100	100	100	100	100	100

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.

6.3.2. OmpSs in Task Parallelism

As discussed earlier, the PM offered by OmpSs is task-oriented and the only runtime that is currently available lies on top of the ad-hoc LWT library called Nanos++. Therefore, our main goal in this scenario does not aim to obtain a performance gain, but to analyze this PM on top of different LWT solutions and to compare the ad-hoc implementation with the generic solution. The current OmpSs runtime release uses a shared queue among all the OmpSs threads and all the created tasks are queued there waiting to be executed.

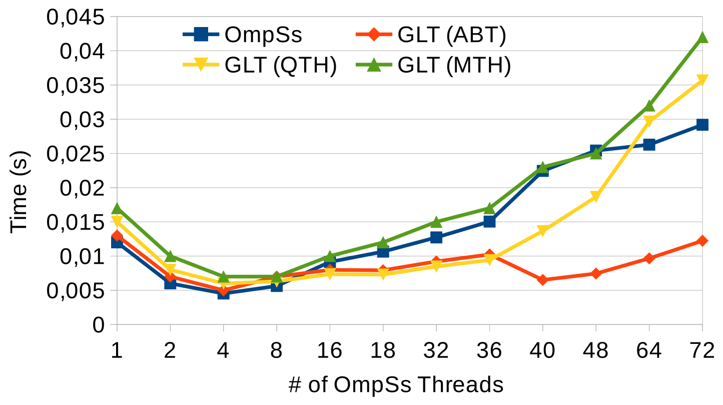
In order to study the differences between the current OmpSs and GOmpSs runtime implementations, we started by analyzing the time spent in task management. With this work, we tried to assess whether our implementation adds any overhead in this procedure. We implemented a microbenchmark that creates a certain number of tasks and then joins them. Figures 9a and 9b show the average time of 100 executions of creating and joining 1,000 and 10,000 empty tasks without dependencies, respectively. The line labeled as OmpSs refers to the OmpSs 16.06.3 version while those labeled as GLT (ABT), GLT (QTH), and GLT (MTH) correspond to our OmpSs implementation over Argobots, Qthreads, ans MassiveThreads, respectively.

Those times are negligible if a task is composed by heavy-coarsed code: however, this indicates that our implementation results are close to those obtained with the current OmpSs release with a reduced number of threads, and they improve upon the current OmpSs solution performance when more than 18 threads are used. As expected, with fine-grained tasks, using a single queue and increasing the number of consumers (OmpSs threads) produces contention. This behavior was also experimented when exploiting the task parallelism with OpenMP. In this case, GLT (MTH) delivers the worst performance because the internal work-stealing requires extra synchronization points. GLT (QTH) performs close to OmpSs and GLT (ABT) when less than 36 threads are used. The reason is that, when 2 threads share a CPU, the performance in this library drops because of the memory locks, as we saw in the OpenMP work-sharing evaluation. In the other, GLT (ABT) is the best solution in almost all the situations, overperforming (up to 2 times faster) the ad-hoc solution when more than 36 threads are used because of its independence among threads that avoids internal synchronization procedures.

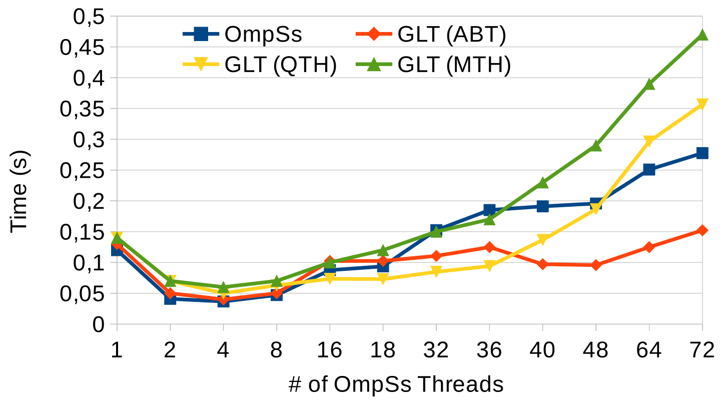
We also evaluated GOmpSs with a production application. We selected the SparseLU Decomposition application from [36]. This application performs an LU decomposition over a square sparse matrix that is allocated by blocks of contiguous memory. We used two different matrix sizes: the default size 3,200x3,200 (Figure 10a), and 12,800x12,800 (Figure 10b) in both cases with real double elements. The execution of these problems spawns 1,500 and 89,000 tasks, respectively.

Figures 10a and 10b show the average of 100 executions for the SparseLU Decomposition and reveal that the time gap among all the OmpSs implementations is almost negligible. Also, the error bars indicate the small time variability.

In summary, the results with OmpSs PM demonstrate that there is room for improvement in the management of fine-grained tasks. However, once that time becomes negligible, the selected LWT implementation does not significally affect performance.

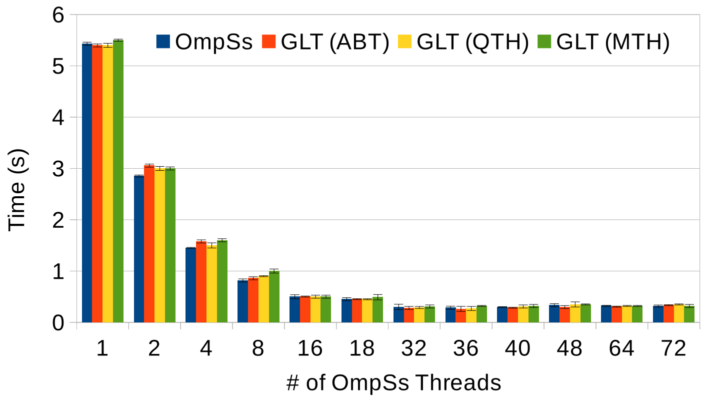


(a) 1,000 OmpSs Tasks.

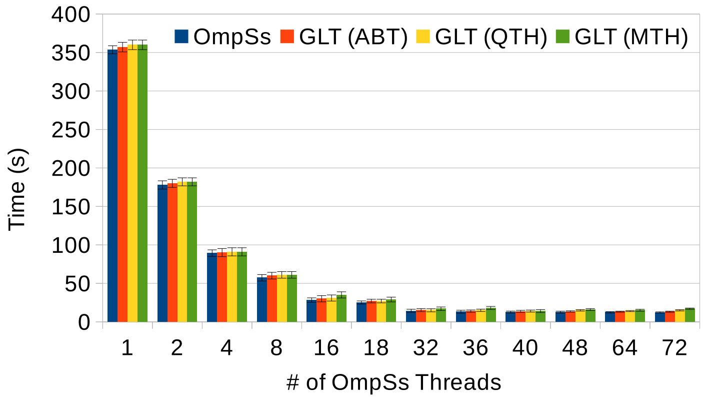


(b) 10,000 OmpSs Tasks.

Figure 9: Execution time for creating and joining OmpSs tasks on top of OmpSs runtimes increasing the number of OmpSs threads.



(a) Matrix size of 3,200 x 3,200 elements.



(b) Matrix size of 12,800 x 12,800 elements.

Figure 10: Execution time for SparseLU application on top of OmpSs runtimes increasing the number of OmpSs threads.

7. Conclusions

We have presented two directive-based PMs, OpenMP and OmpSs, implemented on top of the GLT API, named GLTO and GOmpSs, respectively. GLT presents a common API for LWT solutions and is currently implemented on top of Argobots, MassiveThreads, and Qthreads. The GLTO and GOmpSs runtimes allow us to execute codes written in OpenMP and OmpSs on top of different underlying LWT solutions without modifying the code.

We discussed the design decisions taken during the implementation of both runtimes, and we showed how they behave in different parallel scenarios. Moreover, we compared the current production releases of OpenMP (GNU and Intel implementations) and OmpSs runtimes and our approaches for those PMs in different scenarios: work-sharing constructs (compute bound `for` loop-based codes and nested parallelism), and task parallelism.

For each case, we have shown the performance difference and analyzed the reasons (if any) for the disparity of results.

In the case of work-sharing constructs, the results indicate that no OpenMP implementation is a clear winner because each implementation shows benefits for different cases: pthreads for the compute-bound scenario and LWT for the nested parallelism.

In the task parallelism scenario with OpenMP, LWTs attain better performance than do pthreads with fine-grained tasks.

In the case of task parallelism using OmpSs, our implementation performs close to the original runtime (implemented with an ad hoc solution) in the application scenario and improves the time spent in fine-grained task management when more than 18 threads are used, achieving the best performance when Argobots is used as the underlying library.

These results reinforce our findings within the OpenMP PM; in general, LWTs are highly appropiate to leverage fine-grained tasks, which may be well described by employing high-level PMs

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 cofinanced 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 Enrique S. Quintana-Ortí (Universitat Jaume I) and Sangmin Seo (Samsung Corp.)for their advice in this work and 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, G. Yang, The Sunway TaihuLight supercomputer: system and applications, Science China Information Sciences 59 (7) (2016) 072001.

[2] TOP500 Supercomputer Sites, `www.top500.org/` (June 2016).

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

[4] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE computational science and engineering 5 (1) (1998) 46–55.

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

[6] D. Stein, 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, J. Yelon, Converse: An interoperable framework for parallel programming, in: Proceedings of the 10th International Parallel Processing Symposium (IPPS), 1996, pp. 212–217.

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

[11] L. V. Kale, S. Krishnan, CHARM++: A portable concurrent object oriented system based on C++, Vol. 28, ACM, 1993.

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

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

[14] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, S. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, P. Beckman, Argobots: A lightweight low-level threading and tasking framework, IEEE Transactions on Parallel and Distributed Systems PP (99) (2017) 1–1. `doi:10.1109/TPDS.2017.2766062`.

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

[16] A. Castelló, S. Seo, R. Mayo, P. Balaji, E. S. Quintana-Ortí, 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, 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] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, OmpSs: A proposal for programming heterogeneous multi-core architectures, Parallel Processing Letters 21 (02) (2011) 173–193.

[21] P. Chang, 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.

[22] A. Castelló, S. Seo, R. Mayo, P. Balaji, E. S. Quintana-Ortí, A. J. Peña, GLTO: On the adequacy of lightweight thread approaches for OpenMP implementations, in: Proceedings of the International Conference on Parallel Processing, Bristol, UK, 2017.

[23] OpenMP Architecture Review Board, OpenMP Application Programming Interface Version 4.5 (Nov. 2015).

[24] PGI Compilers & Tools, `http://www.pgroup.com/`.

[25] C. Liao, O. Hernandez, B. Chapman, W. Chen, W. Zheng, OpenUH: an optimizing, portable OpenMP compiler, Concurrency and Computation: Practice and Experience 19 (18) (2007) 2317–2332.

[26] BSC, Mercurium, `pm.bsc.es/mcxx`.

[27] A. Castelló, A. J. Peña, S. Seo, R. Mayo, P. Balaji, 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, 2016.

[28] P. E. Hadjidoukas, V. V. Dimakopoulos, Nested parallelism in the OMPI OpenmP/C compiler, in: European Conference on Parallel Processing, Springer, 2007, pp. 662–671.

[29] Y. Tanaka, K. Taura, M. Sato, 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.

[30] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, R. Namyst, ForestGOMP: an efficient OpenMP environment for NUMA architectures, International Journal of Parallel Programming 38 (5) (2010) 418–439.

[31] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, 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.

[32] Clang project, `http://clang.llvm.org/`.

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

[34] J. I. Aliaga, H. Anzt, M. Castillo, J. C. Fernández, G. León, J. Pérez, E. S. Quintana-Ortí, Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors, Conc. and Comp.: Practice and Experience 27 (4) (2015) 885–904.

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

[36] BSC, Bsc application repository, `pm.bsc.es/projects/bar`.