A Review of Lightweight Thread Approaches for High Performance Computing

Adrián Castelló* Antonio J. Peña† Sangmin Seo‡ Rafael Mayo* Pavan Balaji‡ Enrique S. Quintana-Ortí*

* Universitat Jaume I, {adcastel,mayo,quintana}@uji.es
† Barcelona Supercomputing Center, antonio.pena@bsc.es
‡ Argonne National Laboratory, {sseo,balaji}@anl.gov

***Abstract*—High-level, directive-based solutions are becoming the programming models (PMs) of the multi/many-core architectures. Several solutions relying on operating system (OS) threads perfectly work with a moderate number of cores. However, exascale systems will spawn hundreds of thousands of threads in order to exploit their massive parallel architectures and thus conventional OS threads are too heavy for that purpose. Several lightweight thread (LWT) libraries have recently appeared offering lighter mechanisms to tackle massive concurrency. In order to examine the suitability of LWTs in high-level runtimes, we develop a set of microbenchmarks consisting of commonly-found patterns in current parallel codes. Moreover, we study the semantics offered by some LWT libraries in order to expose the similarities between different LWT application programming interfaces. This study reveals that a reduced set of LWT functions can be sufficient to cover the common parallel code patterns and that those LWT libraries perform better than OS threads-based solutions in cases where task and nested parallelism are becoming more popular with new architectures.**

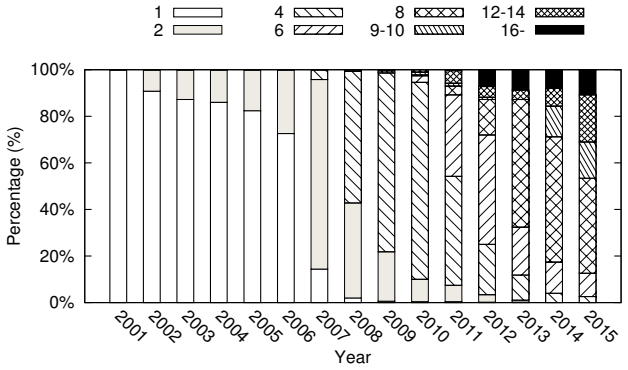

Fig. 1: Top500 supercomputers grouped by the number of cores per socket (Top500 November List).

I. Introduction

The number of cores in high-performance computing (HPC) systems has been increasing during the last years as reflected in Figure 1, which illustrates the evolution of the number of cores per socket in the supercomputers of the November Top500 list [1]. This trend indicates that exascale systems are expected to leverage hundreds of millions of cores. Therefore, future applications will have to accommodate this massive concurrency in some way. To do so, applications will need to deploy billions of threads and/or tasks in order to extract the computational power of such hardware.

Current threading approaches are based on operating system (OS) threads (e.g., `Pthreads` [2]) or high-level programming models (PMs) (e.g., OpenMP [3]) that are closely tied with OS threads. Due to their relatively expensive context switching and synchronization mechanisms, efficiently leveraging a massive degree of parallelism with these solutions may be difficult. Therefore, dynamic scheduling and user-level thread (ULT)/ tasklet models were first proposed in [4] to deal with the required levels of parallelism, offering more efficient context switching and synchronization operations.

Some of these lightweight thread (LWT) libraries are implemented for a specific OS, such as `Windows Fibers` [5] and `Solaris Threads` [6], or a specific hardware such as `TiNy-threads` [7] for the Cyclops64 cellular architecture. Other solutions emerged to support a specific higher-level PM. This is the case of `Converse Threads` [8] for Charm++ [9] and `Nanos++` LWTs [10] for task parallelism in OmpSs [11]. Moreover, there are general-purpose solutions such as `MassiveThreads` [12], `Qthreads` [13], and `Argobots` [14]; and solutions that abstract the LWT facilities such as `Cilk` [15], `Intel TBB` [16], and `Go` [17]. In addition, other solutions like `Stackless Python` [18] and `Protothreads` [19] are more focused on stackless threads. In spite of their potential performance benefits, none of these solutions has been significantly adopted to date.

To address the current scenario, in this paper we demonstrate the usability and performance gain of this type of libraries. For this purpose, we decompose several LWT solutions from a semantic point of view, identifying the strong points of each LWT solution. Moreover, we offer a detailed performance study by using OpenMP PM because of its position as the *de facto* standard parallel programming model for multi/many-core architectures. Our results reveal that the performance of most of the LWT solutions is similar each other and that they are as efficient as OS threads in some simple scenarios while outperforming them in many cases.

In summary, the contributions of this paper are: (1) a semantic analysis of the current and most used LWT solutions; (2) a performance analysis demonstrating the benefits of leveraging LWTs instead of OS threads; and (3) a study of the functionality and PM that are shared between LWT libraries.

2168-9253/16 $31.00 © 2016 IEEE
DOI 10.1109/CLUSTER.2016.12

The rest of the paper is organized as follows. Section II briefly reviews related work. Section III offers some background on our reference libraries. Section IV presents a semantic analysis of the different LWT approaches. Section V offers detailed information about the used hardware and software. Section VI provides a preliminary analysis of important parallel mechanisms. Section VII introduces the different parallel patterns that are analyzed. Section VIII discusses the microbenchmark design decisions. Section IX analyzes the performance of LWT libraries. Section X contains conclusions and future work proposals.

II. Related Work

The use of ULTs to increase concurrency while maintaining performance is not a new topic. In its evolution, as in other paradigms, new solutions aim to improve upon previous approaches. The concept of LWT was first introduced in [4], focusing on fundamentals such as scheduling, synchronization, and local storage. `Converse Threads` was later presented in [20] as a low-level LWT library. It supports not only ULTs but also stackless threads called Messages. `Qthreads` was presented in [13] and compared with the `Pthreads` library by means of a set of microbenchmarks and applications. This solution increases the number of hierarchical levels to three with an intermediate element known as Worker. `MassiveThreads` was presented in [12]. This work provides a performance comparison among `MassiveThreads`, `Qthreads`, `Nanos++`, `Cilk`, and `Intel TBB` on several benchmarks. `Argobots` was presented in [14] with microbenchmark and application evaluations versus `Qthreads` and `MassiveThreads`. This library is conceptually based on `Converse Threads` and allows the use of stackless threads called Tasklets. In addition, it features complete design flexibility and stackable, independent schedulers.

Our purpose of this work is to present an analytic comparison of the LWT libraries from the semantic and PM points of view as well as a performance evaluation demonstrating that LWTs can be a promising replacement for `Pthreads` as the base of high-level PMs.

III. Background

In this section, we review the OpenMP PM and the different LWT libraries analyzed and evaluated in this paper. While `Qthreads` and `MassiveThreads` have been selected because they are among the best-performing lightweight threading models in HPC, `Converse Threads` and `Argobots` were chosen because they are one of the first and currently used LWT library and the most flexible solution, respectively. Despite `Go` is not HPC-oriented, we have also included it as a representative of the high-level abstracted LWT implementations.

A. OpenMP

OpenMP is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming. Currently, there exist implementations for most platforms, processor architectures, and operating systems. OpenMP exposes a directive-based PM that helps users to accelerate their codes exploiting hardware parallelism. Intel and GNU OpenMPs are two commonly used OpenMP implementations that lie on top of `Pthreads` in order to exploit concurrency. These runtimes automatically create all the needed structures and distribute the work.

Since version 3.0, OpenMP has supported the concept of tasks, which constitutes different pieces of code that may be executed in parallel. In contrast with work-sharing constructs, distinct OpenMP implementations leverage different mechanisms for task management. In particular, while the GNU version implements a shared task queue for all threads, the Intel implementation incorporates one task queue for each thread and integrates a work-stealing procedure for the load balance.

B. Converse Threads

`Converse Threads` is a parallel-programming, language-integration solution designed to allow the interaction of different PMs. This library seeks portability among hardware platforms. `Converse Threads` exposes two types of work units: ULTs and Messages. The former represents a migratable, yieldable, and suspendable work unit with its own stack; the latter represents a piece of code that is executed atomically. Messages do not have their own stack and thus they cannot be migrated, yielded, or suspended. Messages are used as inter-ULT communication and synchronization mechanisms. Each thread has its own work unit queue but only messages can be inserted, before their execution, into other thread's queues.

The implementation of the Charm++ programming model is currently built on top of `Converse Threads`, and several `Converse Threads` modules (e.g., client-server) have been implemented specifically for that interaction.

C. MassiveThreads

`MassiveThreads` is a recursion-oriented LWT solution that follows the work-first scheduling policy. When a new ULT is created, it is immediately executed, and the current ULT is moved into a ready queue. This behavior can be configured differently inside the library at compile time. Using a work-first policy benefits recursive codes that need less synchronization. `MassiveThreads` uses the concept of Worker as a hardware resource (generally a CPU or core), and the number of Workers can be specified with environment variables.

Load balance is pursued with a work-stealing mechanism that allows an idle Worker to gain the access to other Worker's ready queue and to steal a ULT. This mechanism requires *mutex* protection in order to access the queue.

D. Qthreads

This library presents a hierarchical PM composed of three levels: Shepherds, Workers, and Work Units. The first two elements can be bound to several types of hardware resources (nodes, sockets, cores, or processing units). The Shepherd boundary level lies in a higher level than the Worker level.

Depending on the level of the Shepherds, they can manage one or more Workers. These configurations are determined by the programmer via environment variables and are created inside the initialization function. `Qthreads` enables creating ULTs into other Shepherds' queues, providing enhanced flexibility to the programmer. A large number of distributed structures such as queues, dictionaries, or pools are offered along with *for loop* and *reduction* functionality.

`Qthreads` allows a large number of ULTs accessing any word in memory. Associated full/empty bits are used not only for synchronization among ULTs but also to leverage *mutex* mechanisms. This free-access to memory requires hidden synchronization, which may severely impact performance.

E. Argobots

`Argobots` is the likely most flexible and recent solution. It presents a mechanism-oriented LWT library that allows programmers to create their own PMs. Like `Converse Threads`, `Argobots` presents two types of work units: ULTs and Tasklets (similar to `Converse Threads` Messages).

This library provides the programmer with absolute control of all the supported resources. Execution Streams may be dynamically created at run time instead of at the initialization point. Moreover, users can also decide the number of required work unit pools as well as which Execution Streams have access to each pool. Although a scheduler is defined for each pool, programmers may still create their own instances and apply them individually to the desired pools. Furthermore, `Argobots` allows stackable schedulers, enabling dynamic changes to the scheduling policy.

F. Go

`Go` is an object-oriented programming language focused on concurrency that is practically hidden to programmers. From the point of view of LWTs, this language supports concurrency by means of `goroutines` that are ULTs executed by the underlying threads. The number of threads may be decided by the user at execution time.

In `Go`, all threads share a global queue where `goroutines` are stored. A scheduler is responsible to assign them to idle threads. This global, unique queue needs a synchronization mechanism that may impact performance when an elevated number of threads are used. The synchronization procedure implemented by `Go` is an out-of-order communication channel that, from the point of view of performance, can obtain better results than the sequential mechanisms.

IV. Semantic Analysis of the LWT Libraries

The semantic analysis of the LWT libraries presented in this section aims to expose the flexibility offered to the programmer. All these libraries were designed to provide more flexible parallelization paradigms and with the main goal of reducing the overhead caused by conventional OS threading mechanisms. Although these solutions are executed in the user space and the thread management is done without the participation of the OS, the libraries lie on top of OS threads. However, each library has its own PM, and the functionality offered to the programmers may vary.

The most important features of the LWT libraries from the point of view of the PM are summarized in Table I. POSIX Threads (`Pthreads`) are also included for reference. The number of hierarchical levels exposed by the different threading library may vary and it depends on the number of execution units or concepts that each library exposes. While `Pthreads` only supports one level (the `Pthread` itself), the LWT solutions support at least two different levels. The former level corresponds to their own `Pthread` representation with a queue/pool of work units that are scheduled and executed. This structure is called Execution Stream in `Argobots`, Shepherd in `Qthreads`, Worker in `MassiveThreads`, Processor in `Converse Threads`, and Thread in `Go`. The number of these elements that are spawned in this level can be defined by the user at run time (Group Control row) via environment variables; but for `Argobots` the programmer can also create them at run time. In contrast, `Pthreads` only allows to create the OS thread itself, while schedulers and queues need to be created entirely by the user. The second level corresponds to work units, such as ULTs or Tasklets, that can be executed by these OS threads. `Qthreads` adds one more level, called Worker, that is positioned between the previous two, managed by a Shepherd, and responsible for executing the work units.

Different types of work units can be used in LWTs (Number of Work Unit Types row). All of them support ULTs that are independent, yieldable, migratable codes with their own private stack. `Argobots` and `Converse Threads` support an extra work unit called Tasklet (atomic work unit without a private stack). These work units are lighter than ULTs and can be used in codes that do not require blocking calls or context switches, or as a communication mechanism as `Converse Threads` does.

The manner in which work units are stored and scheduled is also important to understand the PM. On the one hand, `Argobots` and `Pthreads` can create several pool/queue configurations thanks to their flexibility. On the other hand, `Qthreads`, `MassiveThreads`, `Converse Threads`, and `Go` do not offer that feature to programmers. While the latter only uses a global shared queue, the former three assign one work unit storage structure per thread.

Another key element of the LWT PMs is the scheduler. `Go` is the weaker option because it is not oriented to resource utilization but to concurrent tasks. This implementation is less flexible (not even offering the common yield function) and only has a shared work unit queue that the internal scheduler manages. At the other extreme, `Argobots` is the most flexible solution because it offers the function `yield_to`, which avoids a call to the scheduler, giving directly the control to another ULT. Moreover, it allows the user to create its own ad-hoc, stackable schedulers that may be used by different Execution Streams. In the middle between these two sides of the spectrum, the other libraries use a predetermined scheduler for the threads. In order to balance the workload, `Qthreads` allows users to create work units from one Shepherd to another

TABLE I: Summary of the execution and scheduling functionality offered by the LWT libraries.

Concept	Pthreads	Argobots	Qthreads	MassiveThreads	Converse Threads	Go
Levels of Hierarchy	1	2	3	2	2	2
# of Work Unit Types	1	2	1	1	2	1
Thread Support	✓	✓	✓	✓	✓	✓
Tasklet Support		✓			✓	
Group Control		✓	✓	✓	✓	✓
Yield To		✓				
Global Work Unit Queue	✓	✓				✓
Private Work Unit Queue	✓	✓	✓	✓	✓	
Plug-in Scheduler	✓	✓	✓(configure)	✓	✓	
Stackable Scheduler		✓				
Group Scheduler		✓				

one's queue; `MassiveThreads` implements a random Work-Stealing mechanism; and, `Converse Threads` leverages the Messages.

V. Hardware and Software Resources

All tests have been executed in an Intel 36-core (72 hardware threads) machine consisting of two Intel Xeon E5-2699 v3 (2.30 GHz) CPUs and 128 GB of memory.

GNU's `gcc` 5.2 compiler was used to compile all the LWT libraries and OpenMP examples. Intel icc compiler 15.0.1 was used to evaluate the performance of the OpenMP implementations and linked with the OpenMP Intel Runtime 20151009 version. For LWT libraries, `Argobots`, `Converse Threads`, and `Go` libraries updated to 04-2016; `Qthreads` 1.10 and `MassiveThreads` 0.95 were evaluated.

All results presented next were calculated as the average of 500 executions. The maximum relative standard deviation (RSD) observed in the experiments was around 2%.

VI. Basic Functionality

In this section, we review the basic functionality offered by the different LWT solutions focusing on the OpenMP PM. From a parallel PM point of view, all the features discussed in Section IV have a crucial impact on performance. All these LWT solutions as well as those based on OpenMP follow the same programming approach. On the one hand, programmers are responsible for controlling the main thread that executes the sequential code. This thread is in charge of creating secondary/worker threads, assigning work units, executing their own work and, finally, joining them. This completion may be done using different mechanisms, such as barriers, messages, or thread joins. On the other hand, worker threads wait for work to be done, acting over parallel codes. The parallel code may vary depending on different aspects, such as granularity, the type of code, or the data locality, but the work unit creation and join phases are clearly critical steps. Therefore, they need to be measured when LWTs are used.

Figure 2 reports the time spent by the main thread in order to create one work unit for each thread used. Except `MassiveThreads (H)`, which maintains the performance because it creates all the work units into its own queue and waits for the work-stealing, the other libraries (including Intel and GNU OpenMP implementations labeled as `icc` and `gcc`, respectively) show a linear increase of time because the creation of the work units is done sequentially by the main thread. `Go`'s performance is affected by using just one shared queue. In that scenario the main thread is busy creating work units while the other threads are accessing the queue to obtain one work unit. This situation adds contention (mutexes) in the queue access. The `MassiveThreads (W)` result is caused by its underlying creation policy. The main thread creates the first work unit, pushes the main execution flow to its own queue, and executes the new created work unit. Another thread steals the main task and creates the second work unit, and so on. These steps add a non-negligible overhead when the number of created work units is small. `Converse Threads` and `Argobots` Tasklet use the lightest work unit available for those libraries. This type of work unit yields the best performance, thanks to its stackless structure, being up to twice faster than the `Argobots` ULT approach and three times faster than the `Qthreads` implementation. In this scenario, when OpenMP is employed all threads are created in a previous parallel section so that the overhead of the `Pthreads` creation step is not added for a fair comparison.

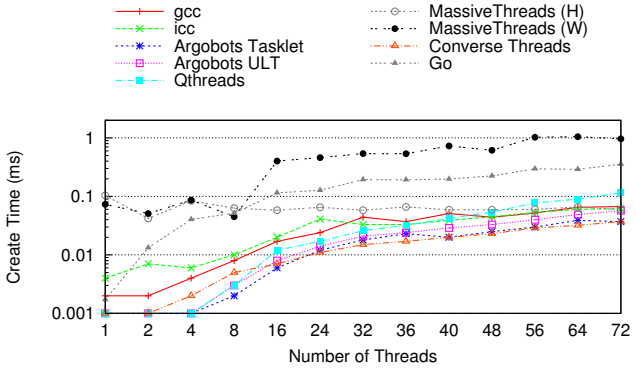

Fig. 2: Time of creating one work unit for each thread.

Figure 3 displays the time spent while the master thread is waiting for the parallel code completion. In this analysis we can distinguish different behaviors in the approaches of these libraries. Since `gcc` OpenMP and `Converse Threads` use a barrier mechanism, the join time increases linearly with the addition of more threads. In this situation `Converse Threads` does not benefit from the Tasklet utilization. The fast time

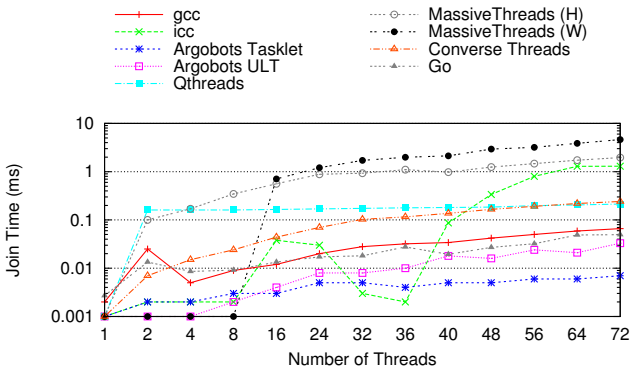

Fig. 3: Time of joining one work unit for each thread.

increment in `icc` OpenMP is caused by using more than one thread per CPU. The behavior changes when more than 8 threads are used (9 cores per socket) and when more than 36 threads are spawned (36 core machine). This runtime performs several checks that require the master thread to access to other threads' allocated memory. The other libraries use a join mechanism but, while `Go` implements the most efficient of them based on out-of-order channel communication, `Qthreads` and `Argobots` use a sequential approach that checks either a memory word value or the work unit status respectively. The unique difference between the last two implementations is that `Argobots` not only checks the status but also frees the work unit structure. Nevertheless, this additional action does not cause a performance drop and `Argobots` still obtains the best result. Conversely, scenarios using `MassiveThreads` deliver the worst performance because, since the main task can be executed by any of Workers, each time a thread is joined, a query of the current work unit queue size and several scheduling procedures occur.

VII. Parallel Code Patterns

Many scientific applications can be easily accelerated using OpenMP. The basic mechanism is to use pragmas in order to indicate the compiler which portion of code can be executed in parallel. A few code patterns are common in many scientific applications. In this section, we present and discuss some of the common parallel code patterns and then analyze how current OpenMP runtimes deal with them.

A. For Loop

The most frequently used OpenMP pragma and probably also the easiest way to express parallelism is: `#pragma omp parallel for` (see Listing 1). It can be placed right before a parallel loop that does not have any iteration dependence, and produces a code where all available threads execute their own iteration range. All the process is transparent to the programmer who is only responsible for selecting the parallelizable code portion and adding the pragma. From the point of view of current OpenMP runtimes, `gcc` and `icc` manage this scenario similarly. The master thread sets the pointer function call of the parallel code in each thread's data

Listing 1: OpenMP for loop parallelism.

```
1 #pragma omp parallel for
2 for (int i = 0; i < N; i++) {
3   code(i);
4 }
```

structure and then the master thread also calls the function. All threads wait in a barrier (unless a `nowait` clause is used) at the end of this code.

B. Task Parallelism

Task parallelism appeared in the OpenMP 3.0 specification as an alternative to parallelize unbounded loops, recursive codes, adding more flexibility to parallel codes. It follows the LWT approach in the sense that tasks are pieces of queued code waiting to be executed by an existing idle thread. This is expressed with the pragma `#pragma omp task`, but each OpenMP runtime leverages its own approach for task management. For example, the `gcc` implementation creates a shared task queue that can be accessed by all the team's threads. On the other hand, the `icc` allows each thread to allocate a private task queue where tasks are stored. Moreover, it implements a work-stealing mechanism that is triggered once a thread's task queue is empty and the thread is idle. Both implementations add a non-configurable cutoff mechanism that avoids performance loss when a large number of tasks are created. Once a certain number of tasks is reached (64×number of threads for `gcc` and 256 in each thread's queue in the `icc`), new tasks are executed sequentially instead of being pushed into the queues. The following two situations can be found depending on the structure where tasks are created:

1) Single Region: In this scenario, a single thread inside a single or master OpenMP (`#pragma omp single` or `#pragma omp master`) region is responsible for creating all the tasks, as shown in Listing 2. While this thread is creating tasks, the other threads execute them. Once the task creation code is finished, the task creator thread also participates in the task execution process. Each OpenMP runtime has its own task mechanism implementation. As the `gcc` OpenMP has only one shared queue, all the tasks are pushed into it and all the threads compete to gain access there to obtain a task. This shared queue is protected by a mutex and thus contention increases with the number of threads. In the `icc` implementation, the task creator thread pushes the new tasks into its own task queue while the other threads try to steal them. Here the performance is affected by the effectiveness of the work-stealing mechanism.

2) Parallel Region: This pattern is employed when all the threads in the team create a certain number of tasks. First, the threads create all the tasks pushing them into the task queue (if the cutoff value is not reached), and then they execute the queued tasks. In the `gcc` implementation, again, all threads compete to gain access to the shared queue, while in the `icc` approach, each thread will push the tasks into its own

Listing 2: OpenMP task parallelism inside a single region.

```
#pragma omp parallel{
  #pragma omp single{
    for (int i = 0; i < N; i++) {
      #pragma omp task{
        code(i);
      }
    }
  }
}
```

Listing 3: OpenMP nested parallelism.

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
  #pragma omp parallel for firstprivate(i)
  for (int j = 0; j < N; j++){
    code(i,j);
  }
}
```

queue and work-stealing will be reduced thanks to a better load balance.

C. Nested Parallel Constructs

When the runtime implementations find a parallel pragma in the user's code, they create a team of the specified number of threads. Hence, if the current parallel code is not nested, the main thread becomes the master thread of a thread team. If it is a nested parallel structure, however, a new team of threads is created for each thread in the main team. Therefore, the total number of created threads grows quadratically. The nested parallelism is not common in applications because the performance drops when the number of threads exceeds that of CPUs and because the code behavior is difficult to understand. There are some types of situations that the user may not be aware of. For example, a programmer may accelerate code with OpenMP pragmas, and inside this parallel code, threads may call an external library function that is parallelized using also OpenMP pragmas. Both OpenMP implementations accommodate nested parallelism. However, the way they manage the new thread teams is different. The `icc` OpenMP runtime creates a new thread team for each thread in the main team reusing idle threads or creating them. The `gcc` implementation does not reuse the idle threads. Each time an OpenMP pragma is found, a new team is created for each thread in the main team. Since the idle threads are not deleted, the total number of threads may increase exponentially. In order to simplify this situation, we have reduced this pattern to two nested for loops, each with its own `#pragma omp parallel for` directive as shown in Listing 3.

D. Nested Task Parallelism

Sometimes, a parallel code may be separated into several independent tasks, such as in divide-and-conquer algorithms. In these cases, task parallelism is commonly exploited. Therefore, this situation combines the previous two task scenarios. In the first step, a single thread creates parent tasks; then each parent task spawns children tasks. As discussed in Section VII-B, `gcc` creates all tasks into the shared queue while `icc` inserts all the parent tasks into the single thread queue. Once this is performed, the children tasks are executed.

VIII. Microbenchmark Design

In this section, we detail the microbenchmark implementations that will run on top of the LWT libraries. These microbenchmarks mimic the behavior of the parallel patterns discussed in the previous section. First, we introduce a general approach for each test. Then, we describe some specific implementations that depend on the LWT library. Lastly, we present a summary of the most frequently used functions.

A. General Development

Although each LWT library internally handles the work unit storage and execution scheduling, all of them allow the programmer to control the main thread. This thread works as a master thread, while the other threads, defined by the user at run time, wait until they obtain work. Therefore, the work division and work unit creation is similar for all of them. Moreover, all the libraries implement a joining mechanism to wait for work units to complete their execution.

1) For Loop: The main thread divides the iteration space among a number of threads and creates a work unit for each thread that contains a function pointer to be executed. An argument structure is initialized in order to store the data (the number of iterations, variables, and so on) that is necessary to execute the function.

2) Task Parallelism: When a single region is used, the main thread creates one work unit for each OpenMP task and, as in the previous case, the work unit is initialized with the function pointer and the needed data. Conversely, if it is placed inside a parallel region, the main thread first divides the work among the other threads, as in the for loop case, and then each thread creates its own work units that represent the OpenMP tasks.

3) Nested Parallel Constructs: For the outer for loop, the behavior of our implementation is the same as in the for loop microbenchmark, but each work unit that executes a range of iterations of the outer loop creates as many work units as there are threads being used to divide the inner loop iterations.

4) Nested Task Parallelism: This case is implemented analogously to the single region of the task parallelism microbenchmark. When a task is executed, it creates a certain number of child tasks.

B. Specific Implementations

Although all the approaches are similar, each LWT library adds its own characteristics to the microbenchmark implementations.

1) Converse Threads: As discussed in Section III, this library was conceived to be controlled by a higher layer. Despite it supports two types of work units (ULTs and Messages), only the latter can be pushed from the main thread execution to the other thread's queues. Therefore, all the results in next section have been obtained using Messages. For this type

TABLE II: Summary of the most used functions in microbenchmark implementations using LWT.

Function	Argobots	Qthreads	MassiveThreads	Converse Threads	Go
Initialization	ABT_init	qthread_initialize	myth_init	ConverseInit	
ULT creation	ABT_thread_create	qthread_fork	myth_create	CthCreate	go function
Tasklet creation	ABT_task_create			CmiSyncSend	
Yield	ABT_thread_yield	qthread_yield	myth_yield	CthYield	
Join	ABT_thread_free	qthread_readFF	myth_join		channel
Finalization	ABT_finalize	qthread_finalize	myth_fini	ConverseExit	

of implementations, the main thread employs a round-robin dispatch in order to push the work units directly into other thread's ready queue. Moreover, the return mode is used as it is the only mode that matches the OpenMP PM from the point of view of the master thread's behavior. Both features, the use of Messages and the Converse return mode, restrict the use of `Converse Threads` in nested parallel scenarios.

2) MassiveThreads: For this library, both available policies (i.e., Work-first and Help-first) are analyzed, though just the best for each scenario is presented in the results. The real difference between both implementations resides in the way that the new work unit is treated. In the former, the current work unit is pushed into the ready queue and the thread executes the new work unit. In the latter the new work unit is pushed into the ready queue and the thread continues with the execution of the current work unit.

3) Qthreads: With its three levels of hierarchy, `Qthreads` accommodates multiple combinations in order to achieve high performance in each of the previous described situations. We have tested a set of combinations, including one Shepherd managing all the node (it manages up to 72 Workers), one Shepherd for each socket (each one manages up to 36 Workers), and one Shepherd for each CPU (each one manages just one Worker). After a preliminary analysis, we chose two combinations: one Shepherd bound to a node or one Shepherds per CPU. The first choice is better with a reduced number of work units but increases the load imbalance. The second option is more appropriate for scenarios with a moderately high number of work units. We discarded the option with a single Shepherd for each socket because it performed much worse than the other choices for all scenarios. Moreover, we decided to test the functions `qthread_fork` and `qthread_fork_to`, which differ in the work queue where the new work unit is stored. The former puts the work unit into the current Shepherd queue and the later allows the user to put the work unit into other Shepherd's queue. If the last option is chosen, the main thread distributes the work using a round-robin dispatch. Hence, four implementations have been evaluated for each test.

4) Argobots: The flexibility offered by `Argobots` is two-fold. On the one hand, two different types of work units can be used: ULTs and Tasklets. On the other hand, the work unit pools can be private for each thread or shared among all of them. If the private pool option is selected, the main thread needs to dispatch the created work units directly to each thread's pool in a round-robin fashion. Therefore, four possible implementations have been tested. Since Tasklet does not have its own stack and is not yieldable, in those scenarios that require two steps, the first of them is performed using ULTs.

5) Go: This library only allows one implementation due to its unique shared work unit queue. All work units need to be pushed into this queue as the `gcc` OpenMP task implementation does. Therefore, only one possibility is analyzed.

Listing 4: Pseudo-code using abstracted LWT functions.

```
#define N 100

void example() {
  printf("Hello world\n");
}

int main(int argc, char * argv []) {
  initialization_function();

  for (int i=0; i<N; i++)
    ULT_creation_function(example);

  yield_function();

  for (int i=0; i<N; i++)
    join_function();

  finalize_function();
}
```

C. Function Summary

Table II summarizes the most commonly used functions in the microbenchmarks that we have developed. We postulate that the majority of codes can be implemented using this reduced set of functions. A pseudo-code illustrating this summary is shown in Listing 4.

IX. Evaluation

In this section, we analyze the performance of the parallel code patterns.

In order to avoid modifying the code for each parallel pattern, we have carefully chosen to implement a BLAS-1 function that matches perfectly the fine-grained approach of LWT and is highly parallelizable. We use the well-known Sscal function, which multiplies (and overwrites) the components of a vector by a scalar. The kernel code is shown in Listing 5. In the for loop and the nested for loop cases, the elements in the vector are divided between the current threads. In the cases where task parallelism is exploited, one task is created for each vector element. This granularity is useful to understand each LWT behavior because this kind of parallelism does not hide the thread management overhead. Concretely, if the execution

Listing 5: Sscal BLAS-1 function kernel code.

```
1 for (int i = 0; i < N; i++) {
2   v[i] = v[i] * a;
3 }
```

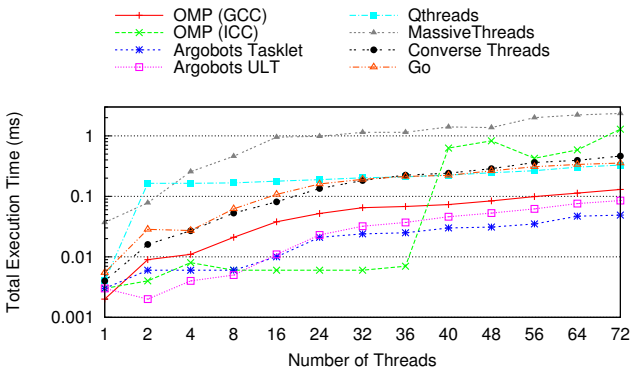

Fig. 4: Execution time of 1,000 iteration for loop.

time of a piece of code is long, this overhead is hidden and there is not any difference between using LWT or OS threads.

A. For Loop

In this scenario, a 1,000 iteration for loop is executed with each iteration calculating one vector position.

Figure 4 illustrates the results. The implementations selected for this scenario are `Argobots` with private pools, `Qthreads` with one Shepherd per CPU, and `MassiveThreads` with the Help-first policy. While `Argobots` results present the best performance thanks to their minimum creation and join times (see Figures 2 and 3), the other implementations suffer from an appreciable overhead when more threads are added to the test. `Qthreads` maintains its performance because of its constant joining time, but this behavior changes when more Shepherds than the number of cores are used. Once this number is reached, the total time is increased. `gcc` and `Converse Threads` pay the contention added by the barrier, while `Go` suffers from the shared queue. The `icc` implementation also experiences a noticeable overhead when more threads than the number of physical cores are used (as in Figure 3). `MassiveThreads`, due to its work-stealing mechanism, shows the worst performance. As the number of created work units is the same as that in the Figures 2 and 3, the behavior of this test is similar to adding those results.

B. Task Parallelism

This test creates a set of tasks. Each task is in charge of one vector element. Two sizes have been evaluated when a single thread creates all the work units: 100 tasks and 1,000 tasks. The results for the former test are similar to those in Figure 4, because the LWT libraries use the same approach and the number of tasks is small. Figure 5 exposes the results for 1,000 tasks. In this case, the implementation choices are `Argobots` using private pools, `Qthreads` with

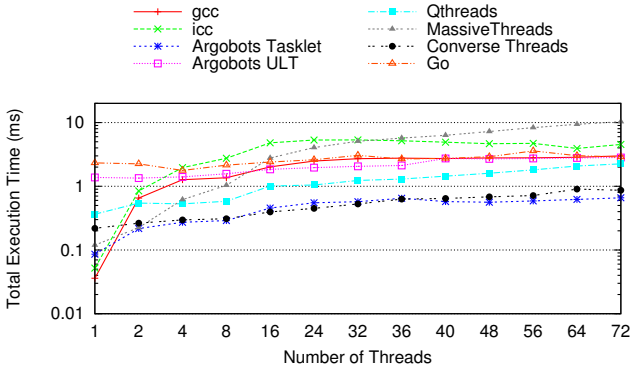

Fig. 5: Execution time of 1,000 tasks created into a single region.

one Shepherd for each Worker, and `MassiveThreads` with the Work-first policy. The default behavior in the `gcc` OpenMP Runtime has been modified by setting the OMP_WAIT_POLICY environment variable to `passive`, in order to decrease the overhead caused by the task queue contention when the number of threads is increased. Otherwise, the threads try to gain the queue access more frequently adding more contention. The `icc` implementation suffers because of the work-stealing mechanism as `MassiveThreads` does. Worker threads try to gain access to the master thread queue and steal work units, adding contention. Moreover, `Go` follows the trend of `gcc` because both of them rely on a single shared queue. Due to the high number of work units, the difference between `Argobots` ULTs and Tasklets is observable. This situation reveals that if the executed code does not need any context switch, it is beneficial to use Tasklets instead of ULTs. Furthermore, as `Argobots` Tasklets are inspired in `Converse Threads` Messages, their performance is similar, and their utilization reduces the execetion time by a factor of two compared with ULT implementations. On the other hand, `Qthreads` follows a linear trend caused by the task dispatch. In addition, the performance change in the OpenMP implementations when more than eight threads are used is due to the cut-off mechanism discussed in Section VII-B. Once this number of threads is reached, the mechanism is not triggered, and the performance comes from a real task parallel execution. In contrast with the for loop case, `Qthreads` outperforms `Argobots` becoming the best ULT choice in this scenario because the time spent in the join-and-free task mechanism implemented by the latter is longer than that in the `Qthreads` join implementation. However, `Argobots` presents a more regular behavior when increasing the number of threads because of the reduced number of interferences in the resource utilization.

The case where tasks are created inside a parallel region and each thread has to create its work units into its own queue represents the first of a two-step algorithm. In the first step, the pointer to the parallel code is assigned (like in the for loop scenario); and in the second, the tasks are created. Here, two different approaches have been evaluated with 100 and 1,000

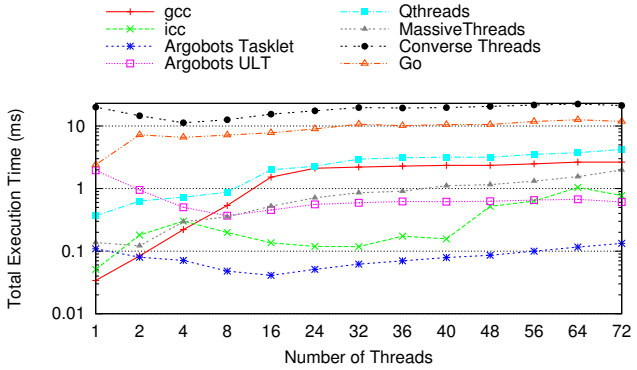

Fig. 6: Execution time of 1,000 tasks created into a parallel region.

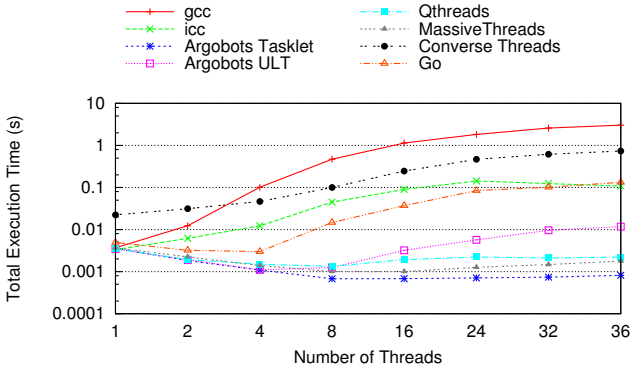

Fig. 7: Execution time of a nested parallel for structure with 1,000 iterations per loop.

tasks but only the latter is shown because the conclusions are similar. In this case, the implementation choices are the same as those in the previous test. Figure 6 displays that `Go` and `Converse Threads` are negatively affected by the two-step implementation due to the shared queue contention in the former and the synchronization (more than 70% of the total time) in the latter. `Converse Threads` needs extra yield calls due to the use of Messages in the first step. This implementation is possible because we know how many parallel code levels exist. This approach would not be possible with an automatic code generator. `MassiveThreads` is now more efficient because its implementation is designed to deal with recursive paradigms. `icc` offers better performance because now, with practically a perfect load balance, the work-stealing has disappeared. `gcc` outperforms other solutions thanks to its cut-off mechanism (up to eight threads) and to the wait policy value set as in the previous test. Again, `Qthreads` experiences a significant increment because of the time with the number of threads and performs much worse than other libraries. Most of this performance drop is because of the time spent in the join mechanism, which doubles the `Argobots` time. Although both `Argobots` implementations use ULTs (that can yield) in the first step, the difference between ULTs and Tasklets is noticeable. Again, when just computation code is executed the use of Tasklets is highly recommended.

C. Nested Parallel Structures

Two approaches have been executed and analyzed for this case. A scenario where the outer and the inner loops have 100 iterations and an alternative with 1,000 iterations each. The same conclusions can be extracted from them. `Argobots` with private pools, `Qthreads` with one Shepherd for each Worker, and `MassiveThreads` with Work-first policy are used. Figure 7 shows the results for this test (notice that the y-axis is in seconds). The OpenMP implementations show a change if we compare the performance difference with LWT libraries in this figure with that shown in the previous. This is due to the suboptimal implementation of the nested parallel structures. On the one hand, `gcc` does not reuse the idle threads in nested parallel codes, so each time an OpenMP pragma is found, a set of new threads is created. This situation causes that, with 36 threads, this implementation spawns 35,036 threads (36 for the main team, and 35 for each outer loop iteration). On the other hand, `icc` reuses the idle threads but it still creates a large number of threads (1,296: 36 for the main team and 35 for each secondary team) much more than the total number of cores (72), causing oversubscription. As in previous tests, `Go` and `Converse Threads` suffer from the two step algorithm. The former is because all the ULTs are pushed in the same shared queue, and the latter is because of the extra yield and barrier functions. However, these implementations perform close to the OpenMP solutions. Conversely, `Argobots` Tasklets, `Qthreads`, and `MassiveThreads` show the best performance because they do not create more threads, just work units. This approach avoids the oversubscription problem reducing the OS thread management and increasing the performance with respect to the Intel OpenMP approach by factors of 130, 48 and 60 for `Argobots` Tasklets, `Qthreads`, and `MassiveThreads` respectively when 36 threads are used. Again, the difference between `Qthreads` and `Argobots` ULT performance is due to the join-and-free mechanism.

D. Nested Task Parallelism

Both task parallelism tests introduced earlier are joined in the last test. First, a single thread creates the parent tasks, and then each thread that executes them creates a set of child tasks. We have tested two approaches, both of them creating 100 parent tasks, but each creating 4 or 10 child tasks. The implementations used in this scenario are `Argobots` with private pools, `Qthreads` with one Shepherd for each Worker, and `MassiveThreads` with Work-first policy. The results in Figure 8 correspond to the former, with 100 parent tasks and 400 child tasks. The two-step algorithm does not help `Converse Threads` and `Go` implementations because of the same reasons discussed in the Task Parallelism case. `Converse Threads` expends up to 75% of its execution time in performing barrier and yield operations. At the other extreme, `Argobots` achieves the best performance with its two implementations outperforming `Qthreads` by a factor of 2.8. `gcc` and `icc` performances are affected by the single

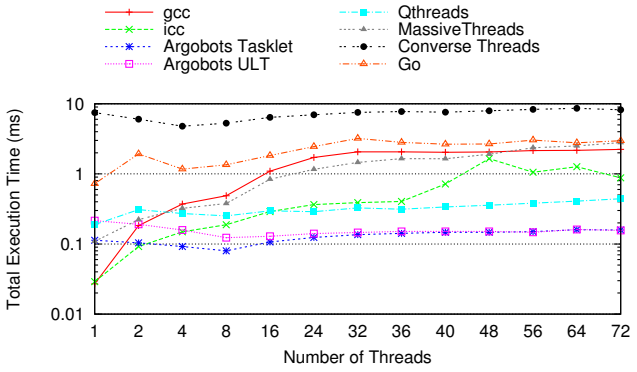

Fig. 8: Execution time of 400 nested tasks.

region step but helped by the cut-off mechanism up to eight threads (as in Figure 5). `MassiveThreads` shows a small overhead due to the combination of work-stealing and the increasing number of threads, but it performs close to `gcc` implementation.

E. Discussion

Although some libraries offer several configuration options, those we selected show a good match for almost all the evaluated tests. `Argobots` with one private queue for each Execution Stream, and `Qthreads` with one Shepherd per core and `qthread_fork_to` function were always chosen. The preferred option for `MassiveThreads` varies depending on the number of created ULTs but, for most of the tests, the Work-first policy was the selected option. These three solutions outperform the other evaluated approaches (including OpenMP runtimes) in our benchmarks. Concretely, they are clearly better than the POSIX thread-based solutions in nested and task parallelism and fine-grained codes.

X. Conclusion

We have proved that the use of LWT approaches for fine-grained parallel codes is feasible, because these libraries can deal with common parallel code patterns that are accelerated with OpenMP pragmas, offering a performance level that is, at least, as good as that reached with the OpenMP runtimes implemented by GNU and Intel. LWTs improve performance in scenarios that are becoming more popular such as task parallelism or nested parallel structures. Moreover, we have detected some implementation choices with strong impact on performance in OpenMP runtimes such as the nested parallelism treatment and the effect of the work-stealing mechanism in the Intel case.

This work has also identified a reduced set of functions that suffice to implement each parallel code pattern. In the future, we plan to design and implement a common API for the LWT libraries. This API could be placed under several high-level PMs, such as OpenMP or OmpSs, that are currently implemented on top of `Pthreads` or custom ULT solutions.

XI. Acknowledgments

The researchers from the Universitat Jaume I de Castelló were supported by project TIN2014-53495-R of the MIMECO, the Generalitat Valenciana fellowship programme Vali+d 2015, and FEDER. 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] "TOP500 Supercomputer Sites," http://www.top500.org/.
[2] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.
[3] "OpenMP 4.5 specification," www.openmp.org/.
[4] D. Stein and D. Shah, "Implementing lightweight threads." in *USENIX Summer*, 1992.
[5] Microsoft MSDN Library, "Fibers," https://msdn.microsoft.com/en-us/library/ms682661.aspx.
[6] "Programming with Solaris Threads," https://docs.oracle.com/cd/E19455-01/806-5257/6je9h033n/index.html.
[7] J. d. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "TiNy threads: A thread virtual machine for the Cyclops64 cellular architecture," in *Proceedings of the Fifth Workshop on Massively Parallel Processing*, April 2005.
[8] 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.
[9] L. V. Kale and S. Krishnan, *CHARM++: A portable concurrent object oriented system based on C++*. ACM, 1993, vol. 28, no. 10.
[10] BSC, "Nanos++," https://pm.bsc.es/projects/nanox/.
[11] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.
[12] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," in *Concurrent Objects and Beyond*. Springer Berlin Heidelberg, 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 the 2008 Workshop on Multithreaded Architectures and Applications (MTAAP)*, April 2008.
[14] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, 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] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.
[16] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.
[17] F. Schmager, N. Cameron, and J. Noble, "Gohotdraw: Evaluating the go programming language with design patterns," in *Evaluation and Usability of Programming Languages and Tools*. ACM, 2010, p. 10.
[18] "Stackless Python," http://www.stackless.com.
[19] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06, October 2006, pp. 29–42.
[20] L. V. Kalé, J. Yelon, and T. Knuff, "Threads for interoperable parallel programming," in *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '96, August 1996, pp. 534–552.