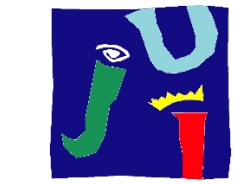
UNIVERSITAT JAUME I DE CASTELLÓ
ESCOLA DE DOCTORAT DE LA UNIVERSITAT JAUME I

Unification of Lightweight Thread Solutions and their Application in High Performance Programming Models

Castelló de la Plana, July 2018

Ph.D. Thesis

Presented by: Adrián Castelló Gimeno
Supervised by: Rafael Mayo Gual
Antonio J. Peña Monferrer



Programa de Doctorat en Informàtica

Escola de Doctorat de la Universitat Jaume I

Unification of Lightweight Thread Solutions and their Application in High Performance Programming Models

Memòria presentada per Adrián Castelló Gimeno per optar al grau de doctor/a per la Universitat Jaume I.

Adrián Castelló Gimeno **Rafael Mayo Gual i Antonio J. Peña Monferrer**

Castelló de la Plana, Juliol de 2018

Agraïments Institucionals

Esta tesi ha estat finançada pel projecte FP7 318793 "EXA2GREEN" de la Comisió Europea, i l'ajuda predoctoral FPI de la Generalitat Valenciana mijançant el programa Vali+D 2015 .

Contents

List of Figures

List of Tables

Listings

Per a tu, Aitana.

Summary

In the last decades, the number of cores per processor has increased steadily, reaching impressive counts such as the 260 cores per socket in the Sunway TaihuLight supercomputer [48], which was ranked #1 for the first time in the June 2016 TOP500 List [24]. This hardware evolution implies an additional effort ro exploit the on-node computational power via concurrent Programming Models (PMs) and applications. Moreover, this trend indicates that future exascale systems may elevate this massive on-node parallelism to thousands of cores per socket. Therefore, extracting the computational power of those machines will require efficient libraries and PMs.

Currently, one of the most popular approach to obtain acceptable on-node parallel performance is the use of Operating System (OS) threads that are exposed to the programmer via the Pthreads Application Programming Interface (API) [20]. The Pthreads API matches current hardware because spawning one Pthread per core pursues that all cores are working in parallel.

From the point of view of software, the Pthreads API also matches with coarse-grained parallelism because the high cost of the management is compensated by the computation time. However, the Pthreads API fails to accommodate new software paradigms that target dynamically-scheduled and fine-grained parallelism. In this scenario, the computation time does not hide the overhead of the management, reducing the performance due to the heavy-weight mechanism. In addition, all types of codes (just-computation, I/O, blocking calls) are managed equally by the Pthreads API avoiding possible performance improvements derived from adapting the resource to the code.

Another way to leverage Pthreads is by using directive-based PMs such as OpenMP [42]. Current runtimes are built on top of the Pthread API and are used for parallelizeing sequential codes by adding hints to the code that the compiler translates into function calls. This runtime is in charge of managing the threads and the execution of the parallel code.

In contrast with Pthreads, several Lightweight Thread (LWT) libraries have been implemented in the last years to tackle fine-grained and dynamic software requirements [64]. These libraries are based on the concept of lighter threads that are managed by OS threads in the user space. Therefore, the OS is not in charge of LWT management and management overheads, such as that context switches, are almost negligible. These inexpensive procedures permit to adapt the thread to each code, even creating lighter threads for just-computation codes.

Although LWT solutions demonstrate semantic and performance benefits over the well-known Pthreads, the variety of LWT libraries hinders portability reducing the general usage in the application/PM development for High-Performance Computing (HPC), because each LWT solution features its own PM and target environment [55], [19], [51], [29], [56], [69], [63].

In response to this situation, the general objective of this thesis is the study, design, development and analysis of a unified LWT API that boosts the use of LWTs for HPC in two manners. First, via code portability by joining the main features of LWT solutions under unified semantics. Second, via the implementation of high-level PMs on top of this common interface in order to demonstrate the viability of the proposed unified semantics. With the aim of demonstrating the benefits of these contributions, we selected a representative group of LWT libraries and extracted their common features. From the insights gained, we designed and implemented a unified API for LWT solutions. Finally, we have implemented two high-level PMs on top of this unified API. For each step, we have analyzed and compared the obtained performance when using LWTs against that obtained with current Pthread-based approaches.

Resumen

En la últimas décadas, el número de núcleos por procesador se ha incrementado, alcanzando impresionantes cifras como por ejemplo los 260 núcleos por socket en el supercomputador Sunway TaihuLight [48], que se situó por primera vez en el número 1 de la lista TOP500 [24] en junio de 2016. Esta evolución hardware implica un esfuerzo adicional para extraer todo el poder computacional a nivel de nodo via modelos de programación y aplicaciones. Además, esta tendencia indica que los futuros sistemas exaescala elevarán este paralelismo masivo a nivel de nodo a miles de núcleos por *socket*. Así, obtener el poder computacional de esas máquinas requerirá de bibliotecas y modelos de programación eficientes.

Actualmente, una de las vias más populares para obtener un aceptable rendimiento paralelo se basa en el uso de hilos del sistema operativo que son ofrecidos al programador mediante la Interfaz de Programación de Aplicaciones (IPA) de "Pthreads" [20]. La IPA de Pthreads encaja perfectamente con el hardware actual porque, creando un Pthread por núcleo asegura que todos ellos trabajan de forma concurrente.

Desde el punto de vista del software, la IPA de Pthreads también encaja con el paralelismo de grano grueso porque el alto coste de gestión de estos hilos es compensado por el tiempo utilizado en el cómputo. Sin embargo, la IPA de Pthreads falla al acomodar nuevos paradigmas software que afrontan la planificación dinámica y paralelismo de grano fino. En este escenario, el tiempo de ejecución no oculta el coste de gestión, reduciendo así el rendimiento debido al impacto de este tiempo en el tiempo total de ejecución. Además, los distintos tipos de códigos (solo cómputo, I/O, llamada bloqueante) son tratados de la misma forma por la IPA de Pthreads evitando posibles mejoras de rendimiento obtenidas al adaptar los recursos disponibles al código ejecutado.

Otra manera de utilizar Pthreads es mediante modelos de programación de alto nivel basados en directivas como OpenMP [42]. Este modelo de programación está construido sobre la IPA de Pthreads y paraleliza código secuencial añadiendo sentencias que el compilador traduce a llamadas a funciones. El sistema en tiempo de ejecución de OpenMP es el encargado de gestionar los hilos y de la ejecución del código paralelo.

En contraste con Pthreads, algunas bibliotecas de hilos ligeros han sido implementadas en los últimos años para lidiar con paralelismo de grano fino y requisitos dinámicos del software [64]. Estas bibliotecas estan basadas en el concepto de hilos más ligeros que son gestionados por los hilos del sistema operativo en el espacio de ususario. Por lo tanto, el sistema operativo no se encarga de esta gestión y como consecuencia, el sobrecoste de los mecanismos como el cambio de contexto son prácticamente despreciables. Estos reducidos sobrecostes permiten adaptar el hilo a cada código, creando incluso tareas aún más ligeras para códigos de solo cómputo.

A pesar de que las soluciones de hilos ligeros han demostrado mejoras de rendimiento sobre los Pthreads clásicos, la variedad de estas soluciones obstaculiza la portabilidad reduciendo el uso general en el desarrollo de aplicaciones y modelos de programación para Computación de Altas Prestaciones (CAP), puesto que cada solución de hilos ligeros ofrece su propio modelo de programación y su software objetivo [55], [19], [51], [29], [56], [69], [63].

Como respuesta a esta situación, el objetivo general de esta tesis es el estudio, diseño, desarrollo y análisis de una IPA unificada para hilos ligeros que aumente el uso de estas soluciones en el campo de la CAP de dos formas distintas. Primero, gracias a la portabilidad del código, uniendo las características principales de las soluciones de hilos ligeros bajo la misma semántica. Segundo, implementando modelos de programación de alto nivel sobre la IPA común con el objetivo de expandir la oferta de bibliotecas de hilos ligeros. Con el propósito de demostrar los beneficios de estas contribuciones, se ha seleccionado un grupo de bibliotecas de hilos ligeros y se han extraido sus características comunes. Después, se ha diseñado e implementado una IPA común para las bibliotecas de hilos ligeros. Finalmente, se han implementado dos modelos de programación de alto nivel sobre la IPA común. En cada paso se han analizado y comparado los resultados de rendimiento obtenidos al utilizar bibliotecas de hilos ligeros con los obtenidos con las implementaciones actuales basados en Pthreads.

Agradecimientos

Una tesis doctoral parece un reto eterno, duro y agotador, y así es. Aunque hay un dicho que dice: "Trabaja en lo que te gusta, y no volverás a trabajar". Y ese, afortunadamente, es mi caso. Sin embargo, este camino no está exento de retos, dificultades y momentos negativos en los que se necesita de apoyo externo para superarlos. Y es por ello que quiero agradecer a todas las personas que han trabajado conmigo en este proyecto y a las que me hay ayudado a nivel personal.

A mis directores, Rafael Mayo Gual y Antonio J. Peña Monferrer y a mi tutor Enrique S. Quintana Ortí. A Rafa, por creer en mí desde el primer momento, y por su incansable aportación de ideas. A Toni, por su dedicación tanto personal como profesional en mi carrera. A Enrique por su sabiduría y su experiencia.

A las personas del grupo HPC&A de la Universitat Jaume I y a las que en algún momento han estado en él, José I., José Manuel, Sergio B., Asun, Maribel, Juan Carlos, Germán L., Germán F., Merche, Gregorio, Manel, Toni, Fran, José Antonio, Maria, Rocío, Sergio I., Héctor, Sandra, Sisco, Rafa, Sonia, Andrés y Goran. Gracias por hacerme sentir como en casa y por vuestra colaboración. También quiero agradecer su ayuda a los técnicos del Departamento de Ingeniería y Ciencia de los Computadores, Gustavo y Vicente.

A los doctores Sangmin Seo y Pavan Balaji por su participación activa en el desarrollo de esta tesis así como su acogida en el grupo PMRS en Argonne National Laboratory en Chicago (EEUU).
To Dr. Sangmin Seo and Dr. Pavan Balaji for their active collaboration in the development of this thesis, as well as for the easy integration and acceptation in the PMRS research group at Argonne National Laboratory in Chicago (US).
También me gustaría dar las gracias por su hospitalidad a los compañeros del grupo Programming Models del BSC.

A mi familia, tanto a los que están como a los que nos han dejado, ya que lo que soy hoy, se lo debo a todos ellos. En especial a mis padres, José Ignacio y Mari Toni que siempre han creido en mí y me han apoyado incondicionalmente. A mi hermano Aarón, quien, sin necesidad de hablar, sabe lo que me pasa. A mi otra familia, Araceli, Pedro y Arancha, de los que he recibido ánimos en cada momento.

Y finalmente, a la persona más importante de mi vida, Araceli, la que me aguanta cuando algo no me funciona, la que me escucha cuando estoy agobiado, la que me ha seguido allá donde esta

aventura me ha llevado haciéndolo todo más fácil, la que mueve cielo y tierra para que yo esté feliz y también, la que me ha dado el mejor regalo del mundo, mi hija Aitana.

A todos:

– ¡Gracias! · Gràcies! · Thanks! · Danke! –

Castellón, julio de 2018.

CHAPTER 1

Introduction

1.1 Motivation

In the last decades, the number of cores per processor has increased steadily, reaching impressive counts such as the 260 cores per socket in the Sunway TaihuLight supercomputer [48], which was ranked #1 for the first time in the June 2016 TOP500 List [24]. This hardware evolution implies an additional effort to exploit the on-node computational power via concurrent PMs and applications. Moreover, this trend indicates that future exascale systems may elevate this massive on-node parallelism to thousands of cores per socket. Therefore, extracting the computational power of those machines will be crucial and, thus require efficient libraries and PMs.

One of the most popular approaches to obtain acceptable on-node parallel performance relies on the use of OS threads that are expossed to the programmer via the Pthreads API [20]. The Pthreads API matches current hardware because spawning one Pthread per core ensures that all cores are working concurrently.

From the point of view of software, the Pthreads API matches with coarse-grained parallelism because the high cost of the thread management is compensated by the computation time. However, the Pthreads API fails to accommodate new software paradigms that target dynamically-scheduled and fine-grained parallelism. In this scenario, the computation time does not hide the overhead of the OS thread management, reducing the performance due to the heavy-weight mechanism. In addition, all types of codes (just-computation, I/O, blocking calls) are managed equally by the Pthreads API avoiding possible performance improvements derived from adapting the resource to the code.

Another way to leverage Pthreads is by using directive-based PMs such as OpenMP [42]. OpenMP defines a set of hints that the programmer may use in the sequential code, indicating the parallelization options. Those hints are translated by the compiler in a set of thread function calls. The execution of the resulting code is performed by an OpenMP runtime in charge of managing the threads and the execution of the parallel code.

In contrast with those threads, several LWT libraries have been implemented in the last years to tackle fine-grained and dynamic software requirements [64]. These libraries are based on the concept of lighter threads that are managed by OS threads in the user space. Therefore, the OS is not in charge of their management, and then the management overheads such as that of

context-switches are almost negligible. These inexpensive procedures permit to adapt the thread concurrency to each code, even creating lighter threads for just-computation codes.

Each LWT solution features its own PM and target environment. Some of these solutions are implemented for a specific OS, such as Windows Fibers [55] and Solaris Threads [19]. Compared with those, ConverseThreads [51] and Nanos++ [29] support specific high-level PMs: Charm++ [52] and OmpSs [30], respectively. There are also general-purpose solutions such as MassiveThreads [56], Qthreads [69], and Argobots [63].

These solutions demonstrate semantic and performance benefits over the classic Pthreads. However, the variety of LWT libraries hinders portability and reduces its usage to scenarios only. Moreover, this lack of portability reduces the use of LWT implementations in HPC.

In addition, their PMs and internal strategies differ among implementations, and hence applications and runtime systems have to be redesigned to exploit the benefits of different LWT libraries by adapting them to distinct PMs and features.

In this scenario, a unified standard interface may be highly beneficial, as long as it supports most of the functionalities offered by the LWT libraries while maintaining their performance.

Moreover, the general adoption of Pthreads as a low-level API as well as a base of high-level PMs increments the effort in order to offer visibility to those alternative LWTs solution. Therefore, high-level PMs and Pthreads API implemented on top of a unified LWT API are necessary to achieve a better diffusion.

1.2 Objectives

Given the potential benefits of LWT solutions in parallel scenarios that are becoming popular, such as fine-grained and nested parallelism, the main objective of this thesis is *to study, design, develop, and analyze a unified API that joins, under unique semantics, the characteristic features of current LWT libraries.* That general objective is realized in the following specific objectives:

- Decomposition of several threading solutions from a semantic point of view, identifying the strong and weak points of each threading solution. This objective will require a detailed performance study by using the OpenMP PM as a base-line because of its position as the *de facto* standard parallel PM for multi/many-core architectures.

- Design and implementation of a unified LWT API, named Generic Lightweight Threads (GLT), that groups the functionality of general-purpose LWT solutions for HPC under the same PM. As part of this objective, an overhead analysis is needed in order to ensure that an extra software layer does not add any perceptible overhead.

- Implement a complete interaction between the Pthreads API and the new GLT API in two ways: implementing the Pthreads API on top of GLT; and implementing the GLT API with Pthreads functionality. These interactions may help in the adoption of the LWT solutions by legacy codes.

- Design and implement OpenMP and OmpSs runtimes on top of the GLT API, called Generic Lightweight Threads OpenMP (GLTO) and Generic Lightweight Threads OmpSs (GOmpSs), respectively. This point requires the analysis of common OpenMP and OmpSs parallel patterns and a discussion about how LWTs deal with them. Moreover, this requires an evaluation of our implementations and a comparison of their performance with those obtained when using the original runtimes.

1.3 Structure of the Document

This chapter presents the motivation and the objectives pursued in this thesis. The structure of the remaining parts of the document are detailed next.

Chapter 2 reviews the current state of the threading libraries including both variants OS threads and LWTs. PMs that are based on threading solutions are also presented in this Chapter.

In Chapter 3, existing threading libraries are analyzed from the semantic point of view. Then, a performance analysis is depicted via microbenchmarks that mimic common parallel patterns. The results compare LWT solutions against OpenMP implementations.

Chapter 4 presents the GLT API. In this chapter, the benefits of a unified API are demonstrated. Moreover, the PM, API design, and its semantical mapping with the underlying libraries are depicted. In addition, an overhead study is performed in order to assert that no overhead is added with this intermediate level.

In Chapter 5, the design and implementation of OpenMP and OmpSs over the GLT API is presented. For each high-level PM, the interaction between "pragmas" and runtime, implementation details and a performance evaluation are discussed.

In Chapter 6 we present the general conclusions of the thesis, the main results compiled as a list of publications, and a collection of open lines of research.

CHAPTER 2

Background

In this chapter we offer a review of software threads from different points of view. First, we introduce the thread concept. Then, we differentiate between OS and LWT threads. In addition, we decompose the PM of each thread solution. Finally we overview two high-level programming models that rely on top of thread implementations.

2.1 Introduction

Threads made an early appearance in 1967 [70, 47], however, in that context they were called "tasks". The term "thread" has been attributed to Victor A. Vyssotsky [61].

A thread is a set of programmed instructions that can be managed independently by a scheduler. This scheduler may be part of an OS or a user-level runtime defined by a PM. Although threads and processes may differ among OSs, in most cases a thread is a component of a process. Therefore, multiple threads may exist within one process, executing concurrently and sharing resources of the process, while different processes do not share these resources.

Single processor systems usually implement multithreading by time slicing: the Central Processing Unit (CPU) divides the total time among different software threads. This context switching occurs very often and fast enough that users perceive a concurrent execution of threads or tasks. On a multiprocessor or multi-core system, multiple threads may be executed in parallel, with every processor or core executing simultaneously a separate thread.

Schedulers of many modern OSs directly support both mechanisms: time-sliced and multiprocessor threading, and the OS kernel allows programmers the use of threads by exposing the required functionality through the system call interface.

There are two kind of thread implementations: the most widely used are called kernel threads or OS threads, and a lighter version of threads (LWTs) that are a specific type of thread that share the same state and information. In contrast with OS threads, these LWTs are placed in the user space and count with timers, signals, or other methods to interrupt their own execution, performing a sort of ad-hoc time slicing. Both thread versions are reviewed in upcoming sections.

2.2 Operating System Threads

OS threads are usually employed by the OSs in order to manage different actions such as system interruptions, execution of applications, and so on. These threads, also known as kernel threads, are exposed to the user via an API. In that way, the programmer is able to express concurrency and to execute several tasks in parallel.

Each thread is usually mapped to a physical/logical core. Although this approach matches perfectly with most current hardware, its management is not costless because the OS controls each event that involves a thread (e.g. creation, joining, context-switch).

Current threading approaches are based on this kind of threads (e.g., Pthreads [57]) or high-level PMs (e.g., OpenMP [15]). However, due to their relatively expensive context switching and synchronization mechanisms, efficiently leveraging a massive degree of parallelism with these solutions may be difficult. Next subsection depicts the Pthreads API.

2.2.1 POSIX Threads API

The Pthreads API [20] defines interfaces and functionality to support multiple control flows—called *threads*—within a process. This API exposes thread management and synchronization primitives to be implemented by libraries.

However, this API does not include functionality for mapping threads with each Kernel Schedule Entity (KSE). KSEs can be managed directly by the OS kernel and the PM changes depending on the threads–KSE mapping. Therefore, this mapping is needed in order to add flexibility to the PM. The GNU is Not Unix (GNU) C library [8] overcomes this limitation by including non-portable functions such as `pthread_yield` or `pthread_setaffinity_np`; however, these functions are not included in the Pthreads standard API.

As depicted in Figure 2.1, Pthreads offers three PMs that differ in how the threads are bound and who is in control. In that figure, each subfigure represents a process.

The *library–thread model* (Figure 2.1a) contains a single KSE, and several threads are scheduled and executed on top of it. This relationship is N:1 and may limit concurrency because just one thread is scheduled at a time. It is leveraged by the GNU Portable Threads library [9].

The *kernel–thread model* employs one KSE for every thread that is generated (1:1 relationship, see Figure 2.1b). This increases the overhead of the management mechanism because the OS kernel is involved in the scheduling and execution of the threads. The GNU C library [8] implements this PM. In this configuration, the thread is the KSE itself.

The *hybrid model* (Figure 2.1c) is composed of a set of KSEs, each managing several threads in an M:N relationship. Since LWT libraries follow this hybrid approach, the Pthreads API is able to accommodate the PM offered by LWTs.

2.3 Lightweight Threads

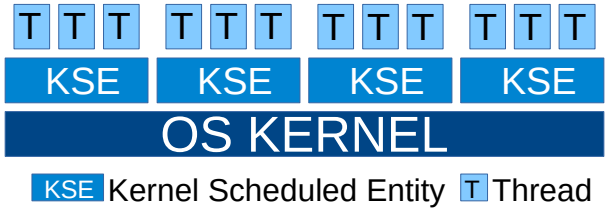
In contrast with OS threads, there are user-space threads, also known as LWTs or User-Level Threads (ULTs). LWTs are managed in the user space and the OS is not aware of them. Therefore, the operations over these threads pose less overhead.

These dynamic scheduling and ULT/tasklet models were first proposed in [64] to deal with the required levels of parallelism, offering more efficient context switching and synchronization operations.

Since then, some of these LWT libraries have been implemented for a specific OS, such as Windows Fibers [55] and Solaris Threads [19], specific hardware such as TiNy-threads [41] for the

(a) Library-thread model (N:1).

(b) Kernel-thread model (1:1).



(c) Hybrid model (M:N).

Figure 2.1: PMs offered by the Pthreads API.

Cyclops64 cellular architecture, or for network services such as Capriccio [67]. Other solutions emerged to support a specific higher-level PM. This is the case of Converse Threads [51, 53] for Charm++ [52] and Nanos++ LWTs [29] for task parallelism in OmpSs [45]. Moreover, there are general-purpose solutions such as GNU Portable Threads [9], StackThreads/PM [65], ProtoThreads [44], MPC [59], MassiveThreads [56], Qthreads [69], and Argobots [63]; and solutions that abstract the LWT facilities such as Cilk [26], Intel TBB [60], and Go [62]. In addition, other solutions like Stackless Python [21] and Protothreads [44] are more focused on stackless threads. In spite of their potential performance benefits, none of these solutions has been significantly adopted to date.

Next subsections review the LWT libraries that are employed in this thesis.

2.3.1 Converse Threads

Converse Threads [51, 53] is one of the first LWT implementations and it was developed at the University of Illinois (US) in 1996. It is a parallel-programming, language-integration solution designed to allow the interaction of different PMs. The main goal of this library is to seek portability among hardware platforms and parallel constructs generality.

Although Converse Threads was designed and developed more than 20 years ago and appeared as a general-purpose solution, nowadays Converse Threads is still one of the most used LWT solutions because it comprises the underlying layer of the Charm++ [52] PM. Since its creation, Converse Threads has been improved with several modules (e.g., client-server) that improve the basic functionality and adapt the PM to diverse application scenarios. This continuous development maintains Converse Threads as a valid solution for HPC environments.

Converse Threads offers two hierarchical levels, processes (OS threads) and work-units. The former allocates a queue where the latter are stored. The user may select the number of active processes by means of environment variables. As an innovative feature, Converse Threads exposes two types of work-units: ULTs and Messages. The former, base of the LWT solutions, 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 and they are recommended as inter-ULT communication, for short, nonblocking tasks, and synchronization mechanisms. As it was noticed before, each thread features its own work-unit queue with its own scheduler where ULTs and messages are stored waiting for their execution. However, only messages can be inserted into other thread's queues and this situation reduces the flexibility because some codes (e.g., a blocking code) cannot be encapsulated as a message.

Figure 2.2 depicts the PM offered by Converse Threads showing the interaction of Converse Threads processes via messages. In that scenario, processes are executing ULTs (Figure 2.2a). For synchronization, process 0 sends a message to process 1, which is scheduled and executed (Figure 2.2b). Once process 1 finishes the execution of the message, it communicates process 0 the work completion via another message (Figure 2.2c).

From the point of view of the PM, Converse Threads allows several execution manners, aimed to deal with different scenarios. The behavior is selected with the function `ConverseInit` that initializes the environment and wakes up the sleeping processors. On the one hand, if the normal mode is selected, threads operate like Message Passing Interface (MPI) processes and all the threads execute the overall code. The user is able to select the code portion depending on the thread identifier. On the other hand, if the return mode is chosen, Converse Threads mimics the OpenMP PM and one thread acts as master thread, controlling the workers by sending messages.

The Converse Threads scheduler is a powerful priority system and supports efficiently stackless and standard threads. This scheduler allows two strategies: First-In-First-Out (FIFO) and Last-In-First-Out (LIFO). With the aim of making Converse Threads more flexible, this library also allows user-defined schedulers that interact with threads.

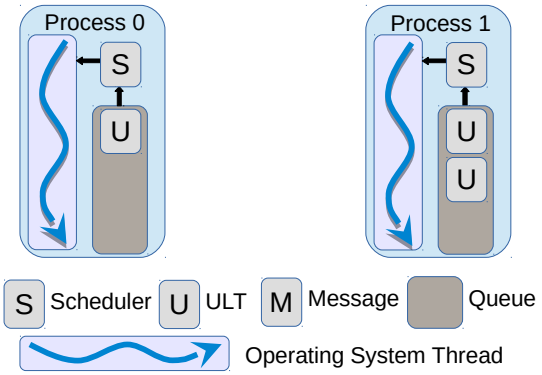
In order to complete a total concurrent environment, the Converse Threads library offers several concurrent-oriented implementations of data structures developed specifically for this PM. These data structures include queues and lists.

2.3.2 MassiveThreads

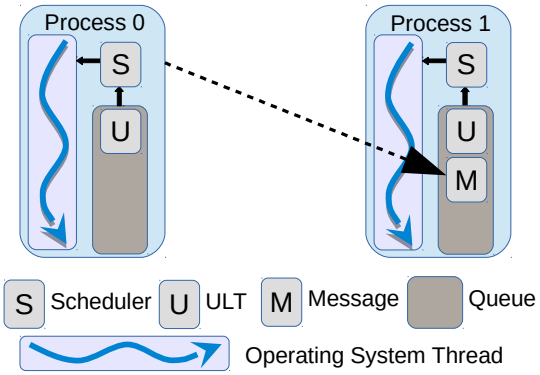
MassiveThreads [56] was presented in 2014, developed at the University of Tokyo (Japan). This LWT library is a recursion-oriented solution and it tackles the thread blocking problem when an I/O operation is executed. In addition, this solution provides an almost-perfect load balancing due to its work-stealing mechanism among threads.

Although MassiveThreads is a consolidated solution, it is in a continuous development state that allows MassiveThreads to be used in current hardware systems. In contrast with other solutions, MassiveThreads is used directly as a low-level library, only avoiding possible overheads caused by higher-level PMs.

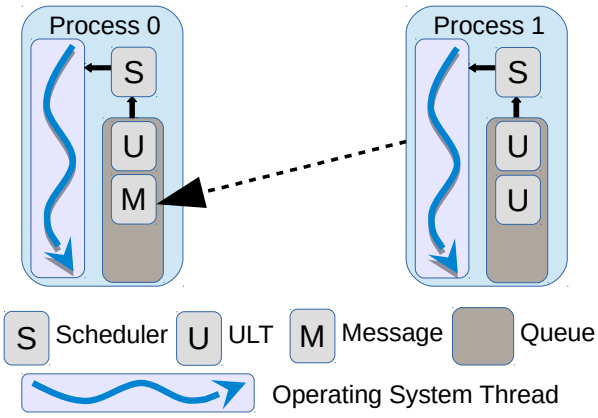
As almost all the LWT solutions do, MassiveThreads also offers two hierarchical levels: Workers (the OS thread) and ULTs. Each worker includes its own work-unit queue that is managed by a scheduler. The representation of the PM is illustrated in figures 2.3 and 2.4. The default scheduler follows the work-first scheduling policy (Figure 2.3): when a new ULT is created, it is immediately executed, and the main task/current ULT is moved into a ready queue (Figure 2.3a). Therefore,



(a) Step 1: Main thread creates OpenMP tasks into its own pool.



(b) Step 2: Idle threads in the team try to steal tasks from other queues.



(c) Step 3: A stolen task is located in the task slot before its execution.

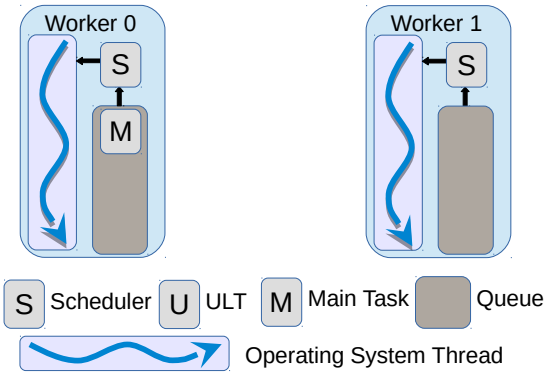
Figure 2.2: Converse Threads PM and process interaction.

the main task may be stolen by other Workers (Figure 2.3b). Then, the main task is executed by its new owner (Figure 2.3c).

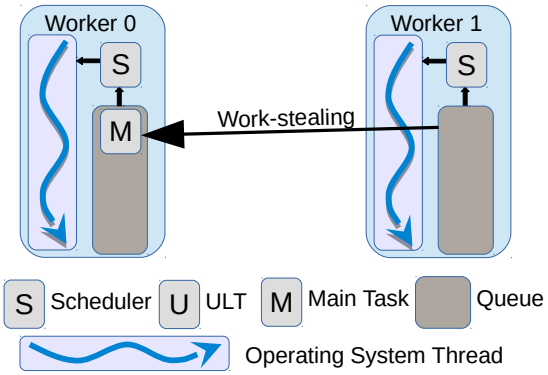
Although this policy benefits recursive codes because of the data locality, this behavior may be modified to a help-first policy (Figure 2.4) inside the library at compile time. The help-first policy does not allow a worker to execute the new ULTs unless a yield function is called. Instead, Worker 0 creates the ULTs into its own pool (Figure 2.3a). Idle threads steal those queued ULTs (Figure 2.3b). With this policy, the load imbalance is reduced (Figure 2.3c).

The number of workers that are spawned by the MassiveThreads environment is selected by the user by setting the environment variable `MYTH_NUM_WORKERS`. Once the application is launched, this number cannot be modified.

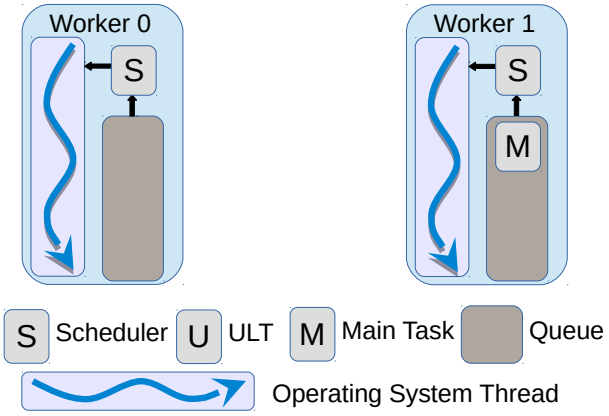
In contrast with Converse Threads, MassiveThreads does not allow introducing work-units into other Worker's queues. Therefore, all the work-units are created into the current Worker's queue and the load balance is pursued with a work-stealing mechanism that allows an idle Worker to gain the access to other Worker's ready queue and to steal a ULT. The work-stealing mechanism is also



(a) Step 1: Worker 0 creates a new ULT and the main task is pushed into its queue, executing the new ULT.



(b) Step 2: Idle threads may steal the main task.



(c) Step 3: Worker 1 continues with the execution of the main task.

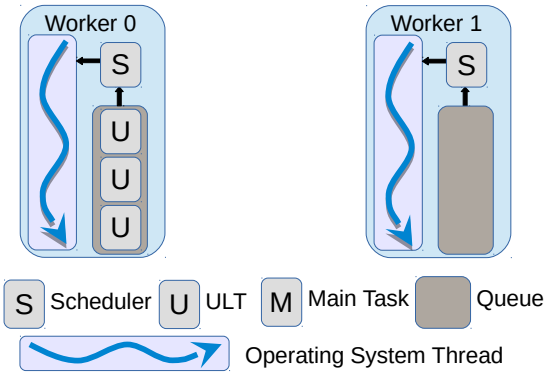
Figure 2.3: MassiveThreads PM and work-first policy.

depicted in figures 2.3 and 2.4. This mechanism requires *mutex* protection in order to access the queue.

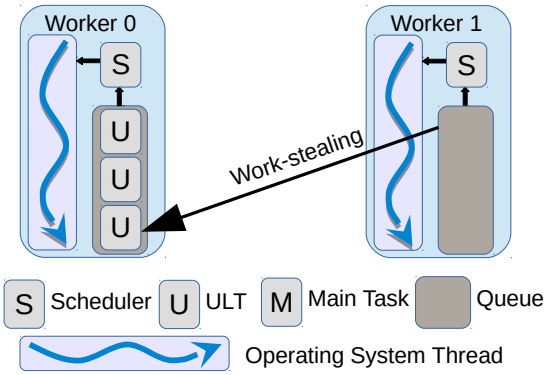
Once the work-units are placed in the queues, the execution follows the LIFO approach for each worker's work and FIFO in case of work-stealing. This algorithm was selected because this scheduling policy is known to be efficient for recursive task parallelism.

MassiveThreads includes a mechanism for I/O handling that consists of three procedures, namely registering a new file descriptor, performing the I/O call, and polling to resume blocked threads. With this procedure, MassiveThreads tackles the blocking thread problem by allowing concurrency between communication and computation.

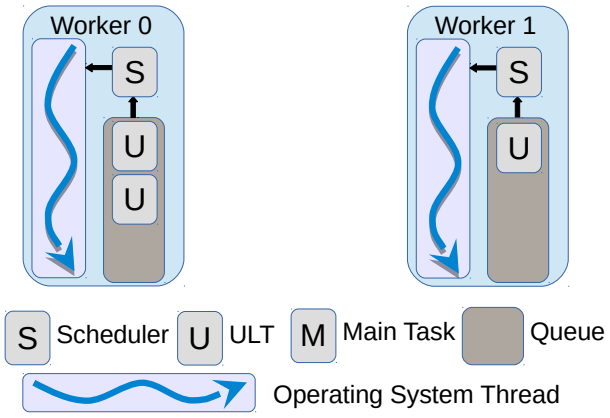
With the aim to offer a soft portability from Pthreads to MassiveThreads, this solution provides a POSIX-like API. This feature enables programmers to convert their legacy codes into MassiveThreads applications without any effort. Moreover, it allows the use of high-level PMs that are currently written on top of Pthreads, with MassiveThreads as the underlying library.



(a) Step 1: Worker 0 creates a ULT into its queue.



(b) Step 2: Idle threads may steal ULTs.



(c) Step 3: Stolen ULTs are executed by their new owners. It reduces the load imbalance.

Figure 2.4: MassiveThreads PM and help-first policy.

2.3.3 Qthreads

Qthreads was developed by Sandia National Laboratory (US) in 2008 as a general-purpose LWT implementation based on the Full-Empty Bit (FEB) design [69]. The feature that differentiates this LWT library from the others is the use of a new hierarchical level. This new level is located between the OS thread (called Shepherd) and the work-units (ULTs), and it is known as a Worker. Shepherds and Workers can be bound to several types of hardware resources (nodes, sockets, cores, or processing units) with the unique restriction that the Shepherd boundary level may lie in a higher level than the Worker.

Depending on the boundary level of the Shepherds, these can manage one or more Workers and this flexibility allows the portability of the applications developed with Qthreads among different hardware architectures. Therefore, if a Shepherd is bound to a node, it could manage up to n Workers where n is the number of logical cores. On the other end, if a Shepherd is bound to a logical core, it only manages one Worker bound to the same core. These configurations are determined by the programmer via environment variables: `QTHREAD_NUM_SHEPHERDS` and `QTHREAD_NUM_WORKERS_PER_SHEPHERD` for number of Shepherds and number of Workers per Shepherd, respectively. And `QTHREAD_SHEPHERD_BOUNDARY` and `QTHREAD_WORKER_UNIT` for their bound-

aries. As in the case of MassiveThreads, all the environment is created inside the initialization function. Figures 2.5 and 2.6 show two different environment configurations. Figure 2.5 depicts the system when one Shepherd is bound to a node and six Workers are created for each Shepherd and bound to a CPU. Figure 2.6 illustrates the environment when one Shepherd per core is created (and bound) and one Worker per Shepherd is spawned.

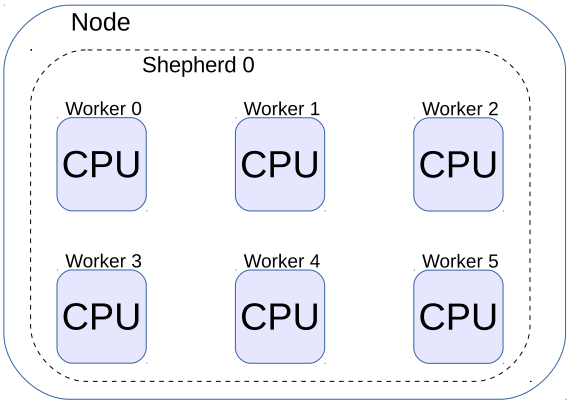


Figure 2.5: Qthreads with 1 Shepherd per node.

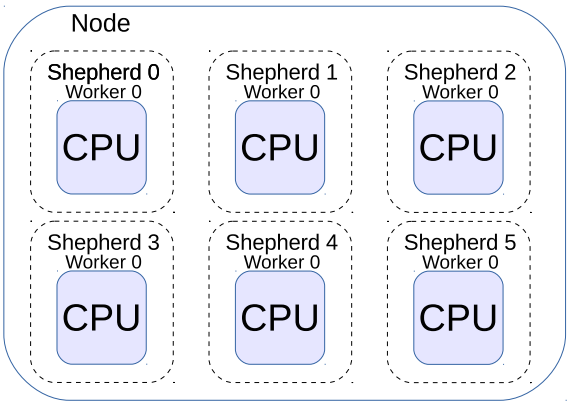
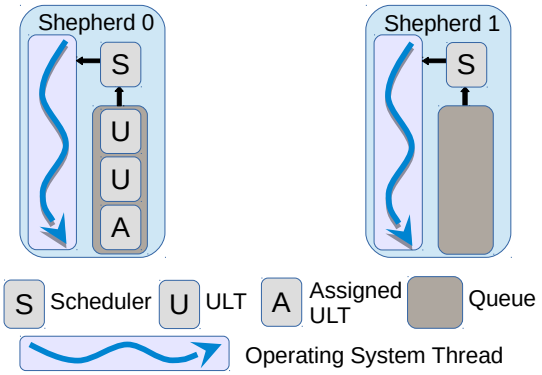
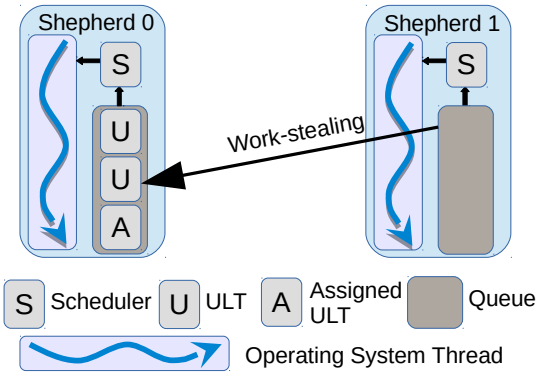


Figure 2.6: Qthreads with 1 Shepherd per CPU.

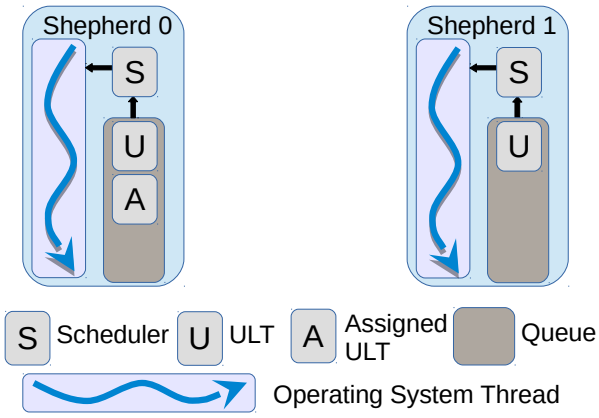
Depending on the number of Shepherds (single or multiple) the user is allowed to select different work-unit schedulers during the library configuration step. In the case of a single-shepherd environment (Figure 2.5), the user can select nemesis, LIFO, mdlifo, mutexfifo or mtsfifo. In the scenario with multiple Shepherds (Figure 2.6), the choices are sherwood, nottingham, and loxley [54]. Figure 2.7 depicts the Qthreads system when one Shepherd is bound to a core and one Worker (omitted for simplicity) per Shepherd is spawned. The scheduler configurations allow work-stealing in order to achieve a good load balance among Shepherds. However, Qthreads enables creating ULTs into a specific Shepherds' queues (assigned ULT) by means of the `qthread_fork_to` function call, and that ULT cannot be stolen by other Shepherds. In Figure 2.7, Shepherd 0 creates regular ULTs and one assigned ULT (Figure 2.7a). Shepherd 1 is not able to steal the last ULT, so it steals the previous ULT (Figure 2.7b). Then, Shepherd 1 executes the stolen ULT (Figure 2.7c).



(a) Step 1: Shepherd 0 creates two ULTs and one assigned ULT.



(b) Step 2: Idle threads can steal regular ULTs but not assigned units.



(c) Step 3: Stolen ULTs are executed by their new owners.

Figure 2.7: Qthreads PM and process interaction.

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

The Qthreads API includes distributed structures such as queues, dictionaries, or pools, which are offered along with *for loop* and *reduction* functionality. Moreover, ULT-aware system function calls are also exposed in the Qthreads API.

2.3.4 Argobots

Argobots was developed at Argonne National Laboratory (US) in 2015. This library is presented as a mechanism-oriented LWT solution that allows programmers to create their own PMs [63]. Therefore, it is likely the most flexible and recent solution among the LWT libraries.

Thanks to its development approach, this library provides the programmer with absolute control of all the supported resources. In contrast with previous LWT solutions, the OS threads (named Execution Streams (ESs)) may be dynamically created at run time by the user instead of at the initialization point with environment variables. Those ESs are independent so there is no need of any internal synchronization mechanism among them. Moreover, users can also decide the number

of required work-unit pools as well as which ESs have access to each pool. Each pool may be configured with different access patterns depending on the number of producers and consumers. For example, a queue could be accessed for a single ES in order to create ULTs while it could be accessed by several ESs for executing the work-units, and vice-versa.

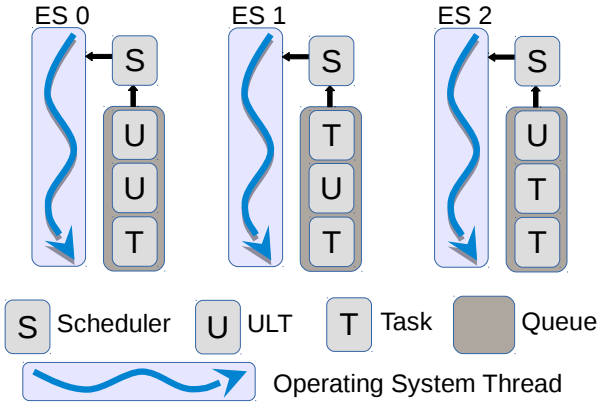


Figure 2.8: Argobots PM using one private pool for each ES.

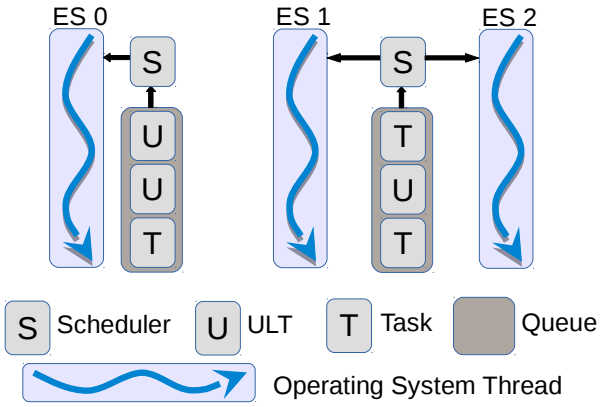


Figure 2.9: Argobots PM using one private pool for ES0 and a shared pool for ES1 and ES2.

In addition, although a default scheduler is defined for each pool, programmers may create their own instances and apply them individually to the desired pools. The default scheduler implements a LIFO policy and only allowed ESs may interact with the scheduler. Furthermore, Argobots allows stackable schedulers, enabling dynamic changes to the scheduling policy that may benefit code portions.

The Argobots flexibility is represented in Figures 2.8, 2.9 and 2.10. This feature allows the programmer to create different environments inside a unique code. For example, in Figure 2.8, each ES manages (creates and executes ULTs/Tasklets) its own pool, which are totally independent. In Figure 2.9, ES0 features its own private queue while ES1 and ES2 share a work-unit queue. In Figure 2.10, each ES owns a private queue and all ESs access to a shared pool. For the latter configuration, the programmer may create a function to describe the order of the calls to each

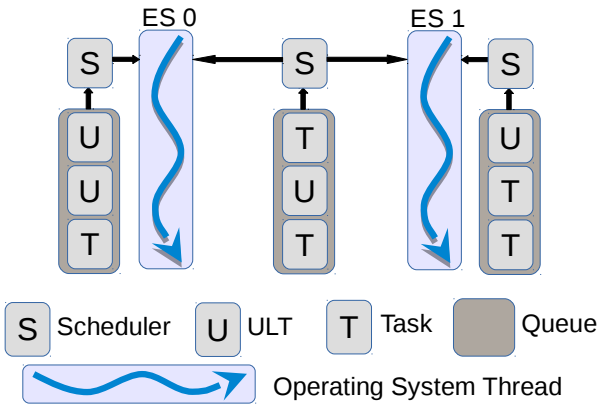


Figure 2.10: Argobots PM using one private pool for each ES and a shared pool for all ESs.

scheduler. This complete flexibility increases the programming difficulty but improves, at the same time, code adaptability.

Like Converse Threads, Argobots presents two types of work-units: ULTs and Tasklets (similar to Converse Threads Messages). However, Tasklets are closer to Argobots ULTs than to Converse Threads messages because these are treated as a ULT excluding the rescheduling functionality (yield, migrate, and pause).

Since this is a mechanism-oriented LWT solution, its low-level API offers a high variety of functionality that enables implementing other LWT solutions and low/high-level runtimes on top of Argobots.

2.3.5 Go

Go was developed by Google in 2009 [62]. It is an object-oriented programming language focused on concurrency that is practically hidden to programmers. This library abstracts the existence of LWTs from the user with the aim to increase the productivity in the web-service scenario.

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 via the environment variable `GOMAXPROCS`. In addition, the user is able to specify the number of threads for a specific code portion at runtime.

Due to the LWT abstraction, Go is the less flexible solution. The Go mechanisms offered to the programmers are: in the creation step, the `goroutine` call. And in the joining step, the creation of a communication channel. In the creation step, all threads share a global queue where `goroutines` are stored. This queue is managed by a scheduler which is responsible to assign the ULTs to idle threads. This global, unique queue needs a synchronization mechanism that may impact performance when an elevated number of threads is used. In Figure 2.11, the interactions between Go processes and the shared queue are depicted.

The synchronization procedure implemented by Go is an out-of-order communication channel that, from the point of view of performance, may obtain better results than the sequential mechanisms. However, it is the programmer the responsible to identify which `goroutine` has sent the message. This identification is usually done by returning a data structure which contains the thread information.

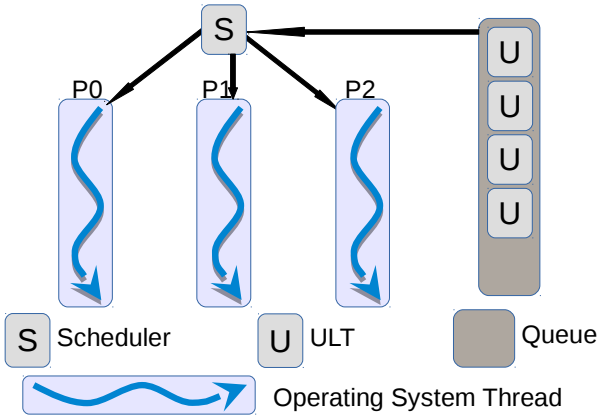


Figure 2.11: Go PM.

2.4 Thread-Based Programming Models

In this section we present two high-level, directive-based PMs whose production implementations lie on top of the Pthreads API. Those high-level PMs are productivity-oriented, hiding the threading management to the user.

2.4.1 OpenMP

OpenMP is a multiplatform shared-memory multiprocessing PM, and current implementations cover most architectures and operating systems such as ARM [1], Flang [5], GNU [10], Intel [11], Lahey [12], LLVM [13], NAG [14], Open UH [16], Oracle [17], PGI [18], or Texas Instruments [22]. OpenMP offers a directive-based PM to parallelize a code by means of "pragmas". This kind of PMs improves productivity because they abstract the thread management from users. This management is hidden thanks to the use of "pragmas" that are translated in compilation time into runtime functions that execute the parallel code. Intel and GNU offer two widely-used OpenMP implementations that rely on Pthreads in order to exploit concurrency.

The OpenMP runtimes are able to manage two types of parallel constructs: the work-sharing constructs and task parallelism. As depicted in Figure 2.12, work-sharing constructs follow the fork-join model where the master thread spawns a team of threads that execute a parallel code and, at the end of the parallel region, the master thread joins the spawned threads. All the OpenMP implementations follow a similar implementation policy.

In contrast with work-sharing constructs, 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 (Figure 2.13), the Intel implementation incorporates one task queue for each thread and integrates work-stealing for load balance control (Figure 2.14). In both solutions, the task management is separated from the work-sharing implementations because task directives were added in the OpenMP 3.0 specification.

In OpenMP 4.0, task directives introduce the task dependencies. These dependencies enable to control the order in the task execution by ensuring that a task is not executed until all the indicated predecessors are finalized.

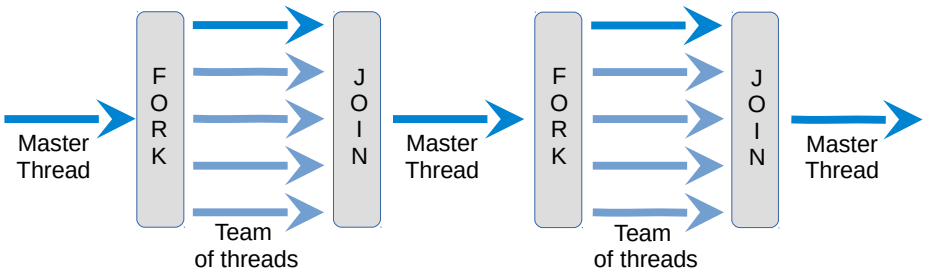


Figure 2.12: OpenMP fork-join model for work-sharing constructs.

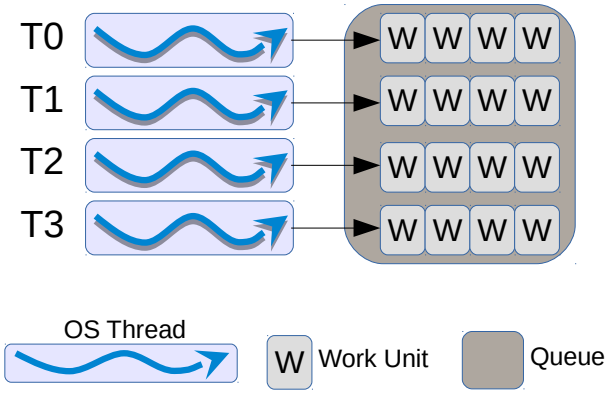


Figure 2.13: GNU OpenMP implementation for task parallelism. All threads push the tasks into the shared queue.

2.4.2 OmpSs

OmpSs, designed and developed at BSC, aims to provide an efficient PM for heterogeneous and multicore architectures [45]. Like OpenMP, OmpSs is a directive-based PM focused on task parallelism. In fact, the task adoption in OpenMP was influenced by OmpSs. Therefore, it embraces a task-oriented execution model similar to the OpenMP tasking features.

This PM is implemented on top of an ad-hoc LWT library called Nanos++. Moreover, it needs an specific compiler (`Mercurium` [28]) which is developed at BSC.

OmpSs detects data dependencies among 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.

A graph example is depicted in Figure 2.15. In that scenario, Task B and C depend on the completion of Task A. Task D depends directly on Task B. Task E depends on two instances of Task C. Finally, Task F depends on Tasks D and E. This graph is employed by the runtime to exploit the implicit task parallelism, via a dynamic out-of-order, dependency-aware scheduler. This mechanism provides a means to enforce the task execution order without the need for explicit synchronization.

Although the OpenMP and OmpSs PMs follow similar approaches in task parallelism (task queues, scheduler, dependencies, and so on), the main difference lies on the application code. While OpenMP requires that the programmer specifies the parallel region where the tasks are going to

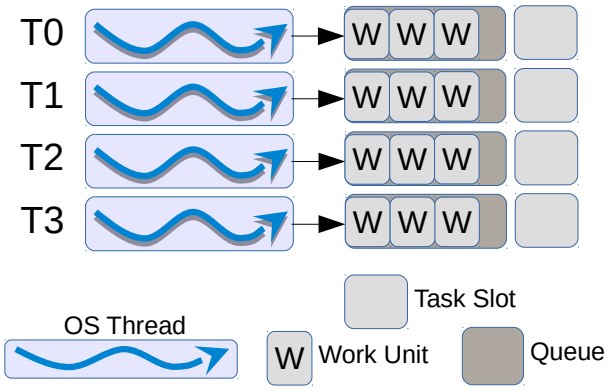


Figure 2.14: Intel OpenMP implementation for task parallelism. Each thread pushes the tasks into its own queue.

be created and executed, in OmpSs the overall application is a parallel region and the programmer just indicates where tasks are created. From the point of view of synchronization, while OpenMP ensures that all tasks created inside a parallel region are completed at the end of that region, OmpSs requires the synchronization directive `#pragma omp taskwait`.

```
1  int main(int argc, char * argv [])
2  {
3    int N = 100;
4
5    //A parallel region is created
6    #pragma omp parallel
7    {
8      //Just one thread creates the tasks
9      #pragma omp single
10     {
11       //100 tasks are created
12       for (int i = 0; i < N; i++)
13       {
14         #pragma omp task
15         {
16           code(i);
17         }
18       }
19     }
20   }   //Implicit barrier that ensures that all tasks have
       been completed
21   ...
22 }
```

Listing 2.1: OpenMP example that creates and executes 100 tasks.

Listings 2.1 and 2.2 show the code for creating and executing 100 tasks in OpenMP and OmpSs, respectively. Both codes are equivalent. While OmpSs hides the parallel section creation (lines 6–

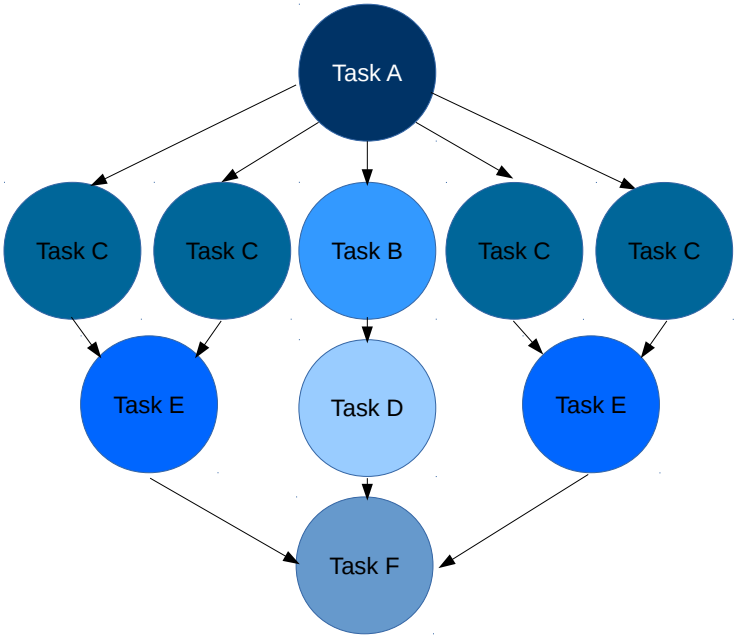


Figure 2.15: OmpSs model for task parallelism.

```
1  int main(int argc, char * argv [])
2  {
3    int N = 100;
4
5    //100 tasks are created
6    for (int i = 0; i < N; i++)
7    {
8      #pragma omp task
9      {
10       code(i);
11     }
12   }
13   //OmpSs asserts that the 100 tasks have been completed
14   #pragma omp taskwait
15   ...
16 }
```

Listing 2.2: OmpSs example that creates and executes 100 tasks.

11 of Listing 2.1) focusing on the task management, the OpenMP code (Listing 2.1) allows the programmer creating an execution environment for the tasks. For example, the programmer may indicate the number of threads to execute the created tasks by using the clause `num_threads` in the `pragma omp parallel` directive (line 6). The synchronization points are located implicitly in line 20 of Listing 2.1 for OpenMP, and explicitly in line 14 of Listing 2.2 for OmpSs.

CHAPTER 3

State of the Art

Different threading libraries are reviewed in this chapter. We demonstrate the usability and performance gain of this LWT libraries. For this purpose, we decompose several threading solutions from a semantic point of view, identifying the strong points of each threading solution. Moreover, we offer a detailed performance study by using OpenMP PM code patterns implemented with LWTs.

3.1 Semantic Analysis of the Threading Libraries

The semantic analysis of the threading libraries presented in this section aims to expose the flexibility offered to the programmer. All these libraries were designed to extract the computational power of the many/multi-core architectures. The LWT solutions provide more flexible parallelization paradigms and appeared with the main goal of reducing the overhead caused by conventional OS threading mechanisms. Although LWT solutions are executed in the user space and the thread management is performed without the participation of the OS, these libraries lie on top of OS threads. However, each library exposes its own PM, and the functionality offered to the programmers vary among them.

The most important features of the threading libraries from the point of view of the PM are summarized in Table 3.1. The number of hierarchical levels exposed by the different threading libraries varies and 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 first level corresponds to their own Pthread representation with a queue/pool of work-units that are scheduled and executed. This structure is called ES 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 initialization (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 in between the previous two, managed by a Shepherd, and responsible for executing the work units.

Concept	Pth	Arg	Qth	MTh	CTh	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		✓	✓	✓	✓	
Yield To		✓				
Global Work-Unit Queue	✓	✓				✓
Private Work-Unit Queue	✓	✓	✓	✓	✓	
Plug-in Scheduler	✓	✓	✓(configure)	✓	✓	
Stackable Scheduler		✓				
Group Scheduler		✓				

Table 3.1: Summary of the execution and scheduling functionality offered by the LWT libraries. Pth, Arg, Qth, Mth, CTh and Go identify the threading libraries: Pthreads, Argobots, Qthreads, MassiveThreads, Converse Threads, and Go, respectively.

Different types of work-units may be used in LWTs (Number of Work-Unit Types row). All of the LWT solutions 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 uses only 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 has only 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, providing 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 ESs. 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 one's queue; MassiveThreads implements a random Work-Stealing mechanism; and Converse Threads leverages the Messages. Although some Pthreads implementations allow the use of yield functions, this functionality is not included in the API standard.

3.2 Performance Analysis of the Threading Libraries

In this section we evaluate the performance of selected LWT solutions. In our comparisons, we use GNU and Intel OpenMP implementations as well as the Pthreads library as base lines.

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` 6.3 compiler was used to compile all the LWT libraries and OpenMP examples. Intel `icc` compiler 17.0.1 was used to evaluate the performance of the OpenMP implementations and linked with the OpenMP Intel Runtime 20151009 version. For LWT, Argobots, Converse Threads, and Go libraries where updated to 04-2016; Qthreads 1.10 and MassiveThreads 0.95 versions were evaluated.

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

3.2.1 Basic Functionality

In this section we review the basic functionality offered by the different threading solutions and the OpenMP PM. From a parallel PM point of view, all the features discussed in the semantic analysis section have a crucial impact on performance. All these threading 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 (mainly in fine-grained codes). Therefore, these need to be measured.

Figure 3.1 reports the time spent by the main thread in order to create one work-unit for each thread used. Except MassiveThreads (labeled as MTH), 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. Converse Threads and Argobots Tasklet, labeled as CTH and ABT(T), respectively, use the lightest work-unit available for those libraries. This type of work-unit yields the best performance, thanks to its stackless structure, being slightly better than the Argobots ULT (ABT(U)) approach and two times faster than the Qthreads (QTH) implementation. The results clearly show that, creating Pthreads (PTH) is more expensive than LWTs (excluding the Go implementation). This is because of the OS intrinsic overhead involving the former. When OpenMP is employed all threads are created in a previous parallel section so that the time spent in this mechanism is close to that of the LWT solutions.

Figure 3.2 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 GNU OpenMP and Converse Threads (labeled as GCC and CTH, respectively) use a barrier mechanism, the join time increases linearly with the addition of more threads. In this situation Converse Threads does not benefit from Tasklet utilization. The fast time increment in Intel OpenMP is caused by using more than one thread per CPU. The behavior changes when

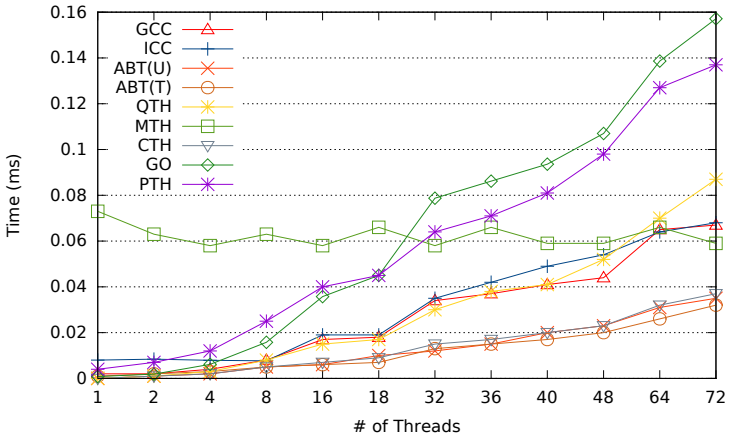


Figure 3.1: Time of creating one work-unit for each thread.

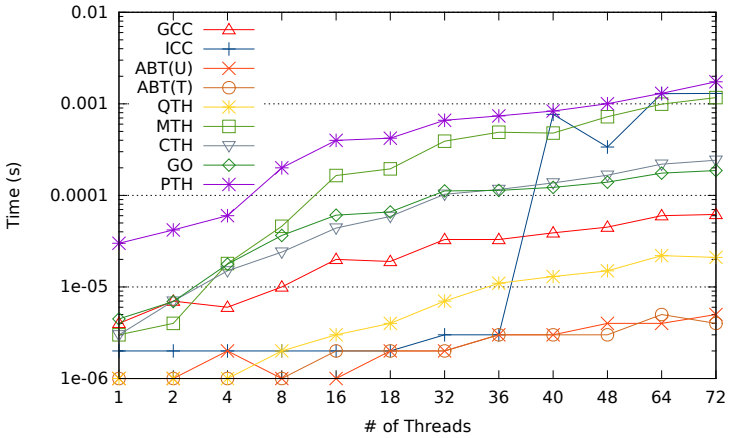


Figure 3.2: Time of joining one work-unit for each thread.

more than 36 threads are spawned (36 core machine) because this runtime performs several checks that require the master thread to access other threads' allocated memory. The other libraries use a join mechanism but, while Go implements an 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 among 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 for both tasklets and ULTs. Conversely, scenarios using MassiveThreads and Pthreads deliver the worst performance. The former because, since the main task can be executed by any Worker, each time a thread is joined a query of the current work-unit queue size and several scheduling procedures occur. The latter, because the OS itself waits until the thread has finished and frees the allocated memory.

3.2.2 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. In this section, we explain the behavior of each pattern, and translate it into LWT code. Then, we depict the different strategies that may be applied by using each threading library. We present separately the performance results for each library. Finally, we select the best implementation of each solution for an overall comparison. We have chosen the following libraries and configurations: Pthreads, Intel and GNU OpenMP implementations as representative of Pthread-based solutions; Argobots using shared and private queues as well as Tasklets and ULTs; Qthreads with one Shepherd per node and one Shepherd per core; MassiveThreads leveraging both help-first and work-first policies; and Converse Threads and Go with the default configurations.

These configurations present different strategies and environments for obtaining a better performance. We find three kinds of environment/behavior configurations: shared structures where all threads gain access and are able to execute work-units (Figure 3.3); independent structures with a round-robin work-unit dispatch (Figure 3.4); and independent structures with work-stealing mechanism (Figure 3.5) for load balance.

Argobots with one shared queue among all ESs, Qthreads with one Shepherd controlling all the CPUs in the node, Go, and Pthreads follow the mechanism depicted in Figure 3.3. In the first step (Figure 3.3a), the main thread creates the work-units into a shared structure. Each work-unit contains a certain number of iterations of the loop. Then, all the threads gain access to the queue and execute the work-units, once a time (Figure 3.3b). However, with this structure, we can not ensure which thread executes an specific work-unit.

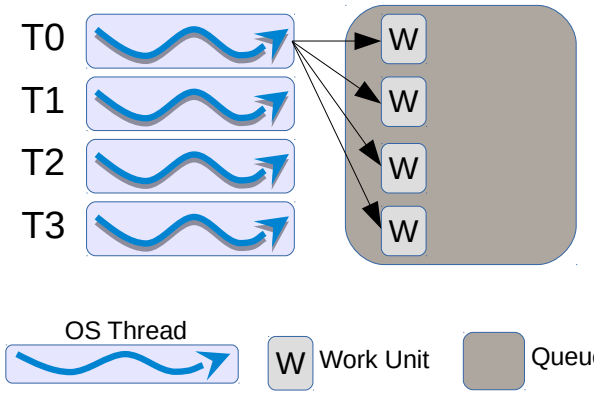
The round-robin dispatch scenario (Figure 3.4) is used by the implementations of Argobots with one private pool for each ES, Qthreads when using one Shepherd per CPU and Converse Threads. The master thread creates one work-unit for each thread (Figure 3.4a). These work-units are allocated in the other thread's queue and without work-stealing, we ensure that each thread executes its own one work-unit (Figure 3.4b).

The work-stealing case (Figure 3.5) is found in both MassiveThreads help-first and work-first policies. In this case, all the work-units are created inside the master queue (Figure 3.5a), and the other threads need to gain access to the queue in order to obtain work-units (Figure 3.5b). This access does not guarantee a successful steal so the accuracy relies on the work-stealing algorithm.

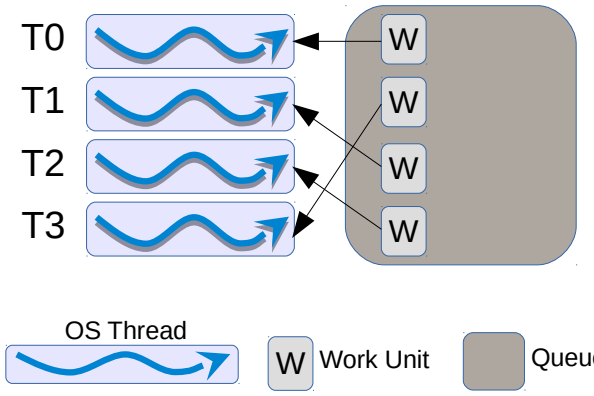
In order to avoid modifying the code for each parallel pattern, we have carefully chosen to implement a Basic Linear Algebra Subprograms (BLAS) function that matches perfectly the fine-grained approach of LWTs and is highly parallelizable. Concretely, we use the well-known Sscal function from level 1 BLAS, which multiplies (and overwrites) the components of a vector by a scalar. The kernel code is shown in Listing 3.1. In the for loop and the nested for loop cases, the elements in the vector are divided among 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 time of a piece of code is long, this overhead is hidden and there is no difference between using LWT and OS threads.

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

Listing 3.1: Sscal BLAS-1 function kernel code.



(a) Step 1: Main thread creates work-units into a shared queue. Each work-unit contains a pointer to the parallel code.



(b) Step 2: Threads gain access to the shared queue and execute work-units.

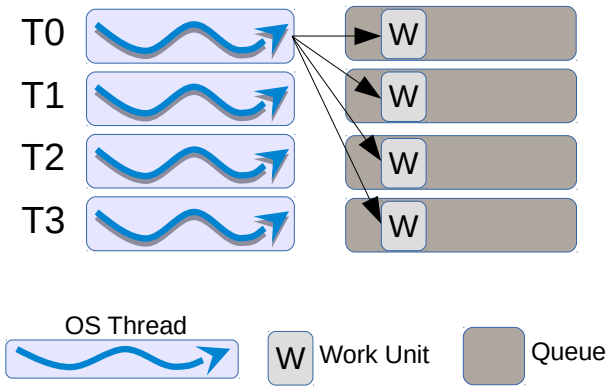
Figure 3.3: Steps for the execution of a parallel code with threading libraries when a shared queue is employed.

3.2.2.1 For Loop

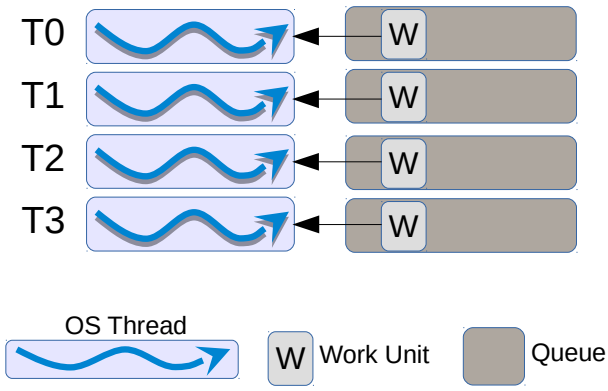
The most frequently used OpenMP directive and probably also the easiest way to express parallelism is `#pragma omp parallel for` (see Listing 3.2). It must be placed right before a parallel loop that does not feature any inter-iteration dependence, and produces a code where all available threads execute 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, both Intel and GNU implementations manage this scenario similarly. The master thread sets the pointer function call of the parallel code in each thread's data 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.

In the case of threading solutions (Listing 3.3), the main thread divides the iteration space among a number of threads (lines 24–28) and creates a work-unit for each thread that contains a



(a) Step 1: Main thread creates work-units into other threads' queues. Each work-unit contains a pointer to the parallel code.



(b) Step 2: Each thread executes the assigned work-unit.

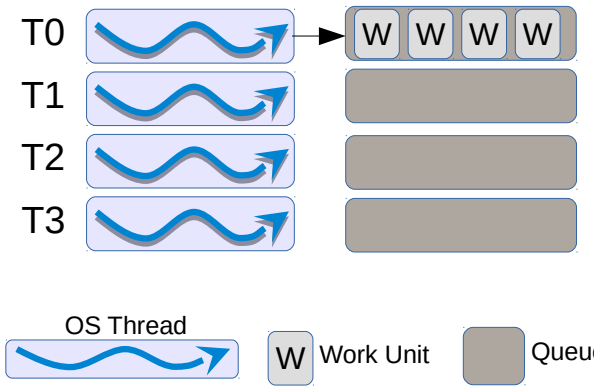
Figure 3.4: Steps for the execution of a parallel code with threading libraries when a round-robin dispatch is employed.

```
1 #pragma omp parallel for
2 for(int i = 0; i < N; i++)
3 {
4   code(i);
5 }
```

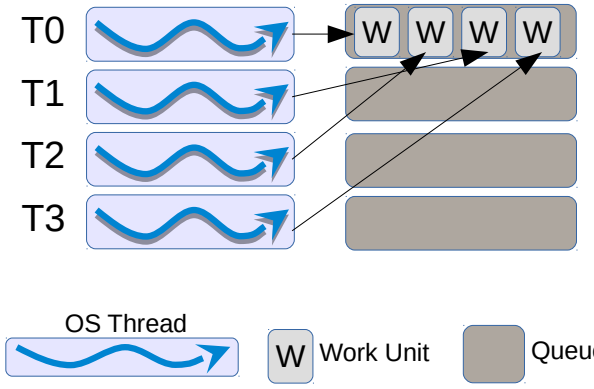
Listing 3.2: OpenMP for loop parallelism.

function pointer to be executed (line 31). 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.

Figure 3.6 exhibits the time differences between directly using Pthreads (PTH) and Intel/GNU OpenMP implementations (ICC and GCC, respectively). In this case, we advert that the time



(a) Step 1: Main thread creates work-units into its own pool. Each work-unit contains a pointer to the parallel code.



(b) Step 2: Each thread executes its own work-units. If the queue is empty, threads try to steal work-units from other queues

Figure 3.5: Steps for the execution of a parallel code with threading libraries when work-stealing is employed.

difference is because in OpenMP runtimes, the team of threads have been created in a previous parallel region and the execution time is just for the work assignment.

When using Argobots (Figure 3.7), the differences between ULTs and Tasklets are barely appreciable. Due the reduced number of ULTs/Tasklets, this time difference may be depictable in this scenario. However, this gain of time may leverage a performance difference in a more complex application when using Tasklets. Moreover, we can appreciate the contention overhead when using a shared pool for all ESs (labeled with the *SP* suffix). This contention causes drop in performance once more ESs than cores in a Non-Uniform Memory Access (NUMA) node are used.

For Qthreads (Figure 3.8), the use of just 1 Shepherd for the entire node reduces the execution time. This situation is caused because there is no synchronization among Shepherds. However, if we use one Shepherd per core, the necessity of the synchronization in all the memory word accesses reduces the performance.

```
#define NUM_ULTS 4

struct arg_for
{
  int ini;
  int fini;
}

void for_lwt(void * args)
{
  arg_for *arg = (arg_for*) args;

  for(int i = arg.ini; i < arg.fini; i++)
    code(i);
}

int main(int argc, char * argv [])
{
  //Allocate memory for structures
  ULT * lwts[NUM_ULTS];
  arg_for * args[NUM_ULTS];

  for(int i = 0; i < NUM_ULTS; i++)
  {
    //Calculate the number of iterations per LWT
    ...
    //Arguments initialization
    args[i].ini = XXX;
    args[i].fini = XXX;

    //LWT creation
    create_lwt(for_lwt,args[i],lwts[i]);
  }

  lwt_yield();

  for(int i = 0; i < NUM_ULTS; i++)
  {
    //Wait for LWT completion
    join_lwt(lwts[i]);
  }
}
```

Listing 3.3: LWT for loop parallelism implementation.

In this scenario (Figure 3.9), MassiveThreads benefits from the help-first policy because the main thread creates all the work-units without yielding the main task. This yield happens in the work-first policy where each new work-unit implies a scheduler call and a context-switch between current and new task.

Figure 3.10 illustrates the results with the best library configuration. The implementations selected for this scenario are Argobots with private pools, Qthreads with one Shepherd per node, and

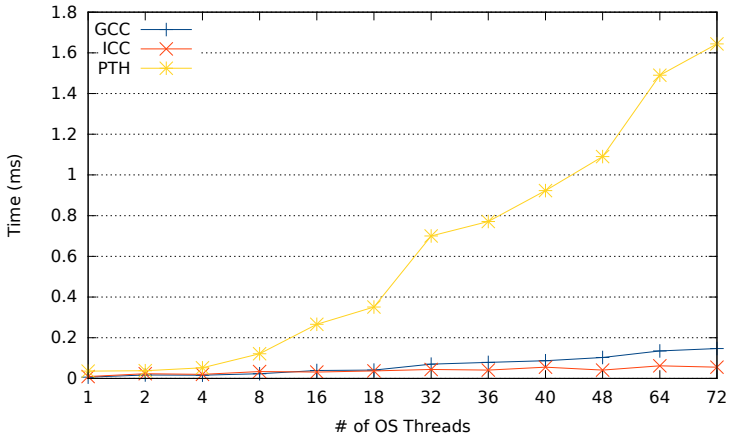


Figure 3.6: Execution time of a 1,000-iterations for loop with Pthreads-based approaches.

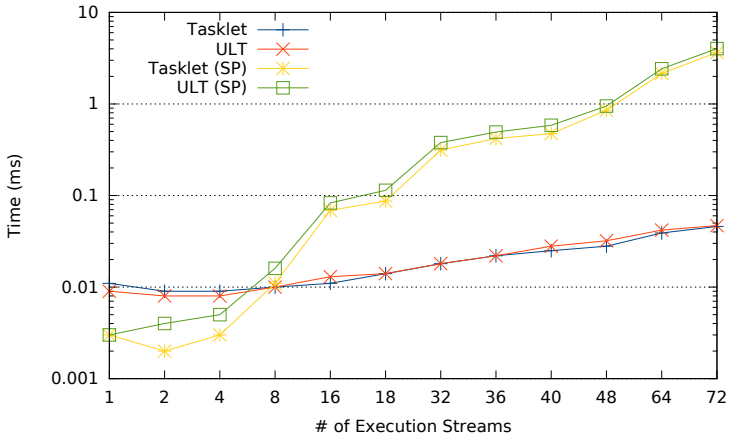


Figure 3.7: Execution time of a 1,000-iterations for loop with Argobots approaches.

MassiveThreads with the help-first policy because these attain higher performance than other configurations. While Argobots results (ABT(T) and ABT(U)) present the best performance thanks to their minimum creation and join times (see Figures 3.1 and 3.2), the other implementations suffer from an appreciable overhead when more threads are added to the test. Qthreads (QTH) maintains its performance because of its small joining time. MassiveThreads, Pthreads and Converse Threads (labeled MTH, PTH, and CTH, respectively) present results 25 times slower than Argobots or `icc`. MassiveThreads suffers from work-stealing because all tasks are created inside the main worker queue, and Pthreads because of the OS management. Moreover, when Converse Threads uses more threads than physical cores, the performance drops due to synchronization mechanisms. Although Go suffers from the contention that produces the shared queue, it maintains acceptable performance results considering that it is not a HPC-oriented solution. The OpenMP Intel and GNU implementations (labeled as ICC and GCC) perform close to Argobots and Qthreads, respectively, when all the cores are used.

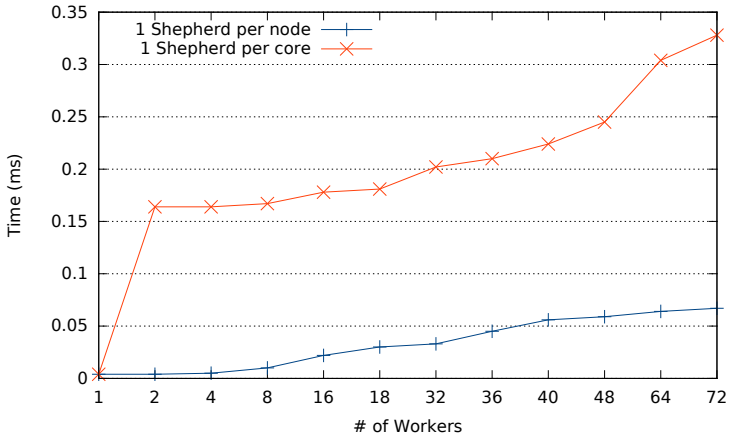


Figure 3.8: Execution time of a 1,000-iterations for loop with Qthreads approaches.

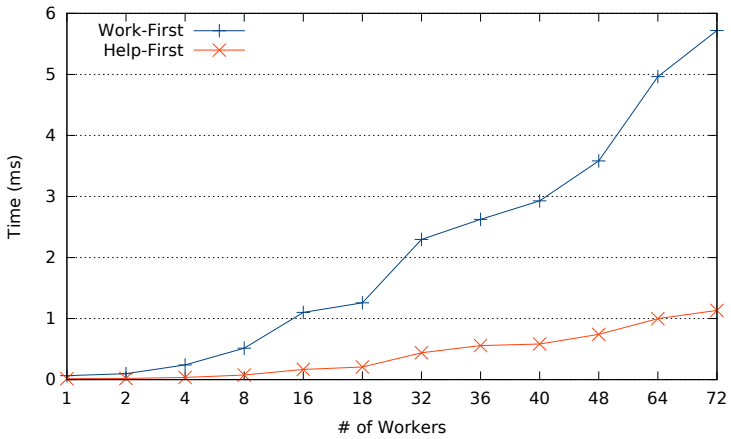


Figure 3.9: Execution time of a 1,000-iterations for loop with MassiveThreads approaches.

3.2.2.2 Task Parallelism

Task parallelism appeared in the OpenMP 3.0 Specification as an alternative to parallelize unbounded loops and 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 directive `#pragma omp task`, but each OpenMP runtime leverages its own approach for task management. For example, the GNU implementation creates a shared task queue that may be accessed by all the team's threads. On the other hand, the Intel approach 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 GNU and 256 in each thread's queue in the Intel case), new tasks are executed sequentially instead of being pushed into the queues. Two patterns may appear depending on the structure where tasks are created: single region and parallel region.

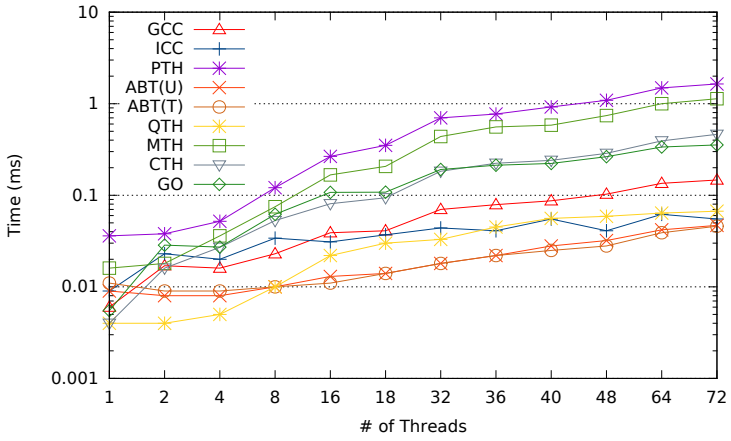


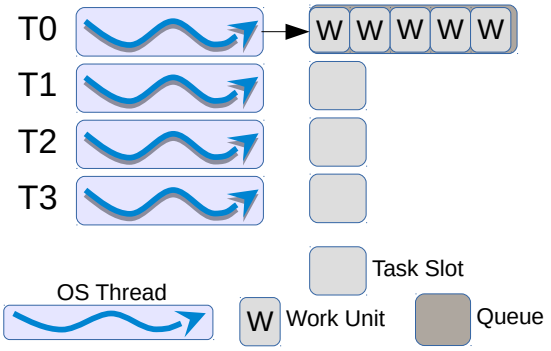
Figure 3.10: Execution time of a 1,000-iterations for loop with the best configuration for each library.

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 3.4. Each task computes one vector element. 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 features its own task mechanism. Since the GNU runtime leverages only one shared queue, all the tasks are pushed into it and all the threads compete to gain access there to obtain a task, as shown in Figure 3.3. This shared queue is protected by a *mutex* and thus contention increases with the number of threads. In the Intel implementation (Figure 3.11), the task creator thread pushes the new tasks into its own task queue (Figure 3.11a) while the other threads try to steal them (Figure 3.11b). Once there is a successful steal, the task is moved to a task slot before its execution (Figure 3.11c). Here the performance is affected by the effectiveness of the work-stealing mechanism.

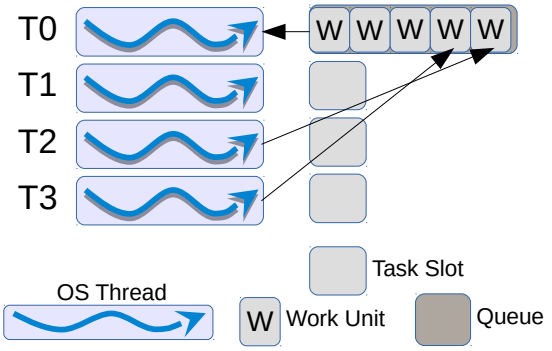
```
1  #pragma omp parallel
2  {
3    #pragma omp single
4    {
5      for (int i = 0; i < N; i++)
6      {
7        #pragma omp task
8        {
9          code(i);
10       }
11     }
12   }
13 }
```

Listing 3.4: OpenMP task parallelism inside a single region.

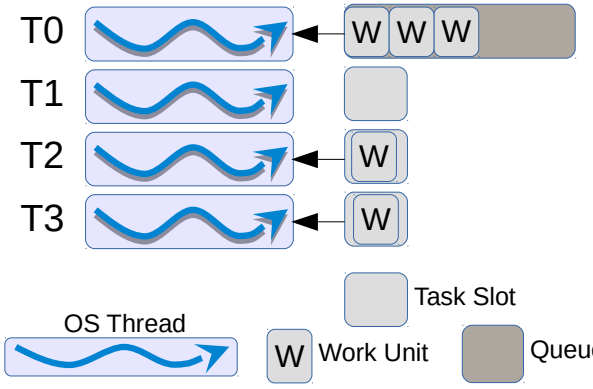
When using the threading libraries (Listing 3.5), the main thread creates one work-unit for each OpenMP task (lines 22–30) and, as in the previous case, the work-unit is initialized with the



(a) Step 1: Main thread creates OpenMP tasks into its own pool.



(b) Step 2: Idle threads in the team try to steal tasks from other queues



(c) Step 3: A stolen task is located in the task slot before its execution.

Figure 3.11: Steps of the creation and execution of OpenMP tasks inside a single region.

function pointer and the needed data. Then, the main thread waits until the tasks complete with a join function (line 37).

Figure 3.12 shows the time differences between directly using Pthreads (PTH) and Intel and GNU OpenMP implementations (ICC and GCC, respectively). In the case of Pthreads, each OpenMP task is translated into a Pthread. Moreover, we have limited the number of cores with the command `taskset`, in order to restrict the number of available resources, as we do by setting the `OMP_NUM_THREADS` environment variable. In order to decrease the overhead caused by the task queue contention when the number of threads is increased, the default behavior in the `gcc` OpenMP runtime has been modified by setting the `OMP_WAIT_POLICY` environment variable to `passive`.

In this scenario both OpenMP implementations perform similar, although the Intel solution presents more variability because of the work-stealing mechanism. This mechanism performance relies on how accurate are the work-stealing procedures.

As in the `for loop` code, using Argobots (Figure 3.13), the performance gain when using Tasklets over ULTs is noticed. Moreover, the effect of a shared pool (SP) is also beneficial for

```
#define NUM_ULTS 4
#define NUM_TASKS 100

struct arg_task
{
  arg1;
  arg2;
}

void task_lwt(void * args)
{
  arg_task *arg = (arg_task*) args;

  code(args);
}

int main(int argc, char * argv [])
{
  //Allocate memory for structures
  ULT * lwts[NUM_TASKS];
  arg_for * args[NUM_TASKS];

  for(int i = 0; i < NUM_TASKS; i++)
  {
    //Argument initialization
    args[i].arg1 = XXX;
    args[i].arg2 = XXX;

    //LWT creation with a round-robin dispatch
    create_lwt_to(for_lwt,args[i],lwts[i],i%NUM_ULTS);
  }

  lwt_yield();

  for(int i = 0; i < NUM_TASKS; i++)
  {
    //Wait for LWT completion
    join_lwt(lwts[i]);
  }
}
```

Listing 3.5: Implementation of LWT task parallelism inside a single region.

a reduced number of ESs but it affects the performance because of the contention when more resources are added.

For Qthreads (Figure 3.14), the use of just 1 Shepherd for the entire node does not benefit performance. The elevate number of tasks in a single queue (1 Shepherd per node) adds contention because Workers need to gain access to the structure. Conversely, using separated Shepherds and a round-robin dispatch reduces the interaction among Workers.

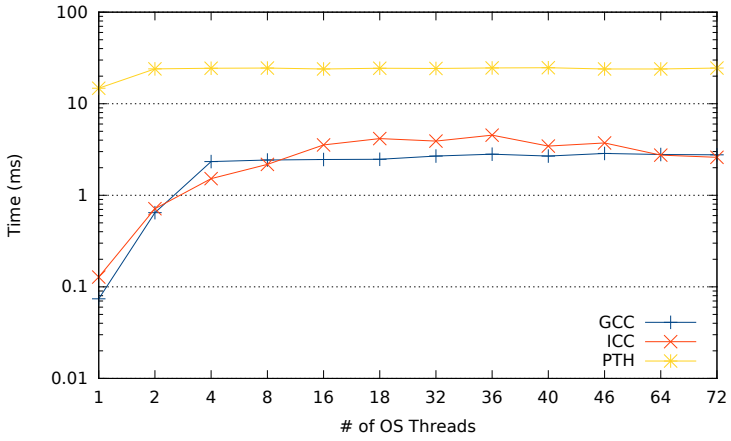


Figure 3.12: Execution time of 1,000 tasks created in a single region with Pthreads-based approaches.

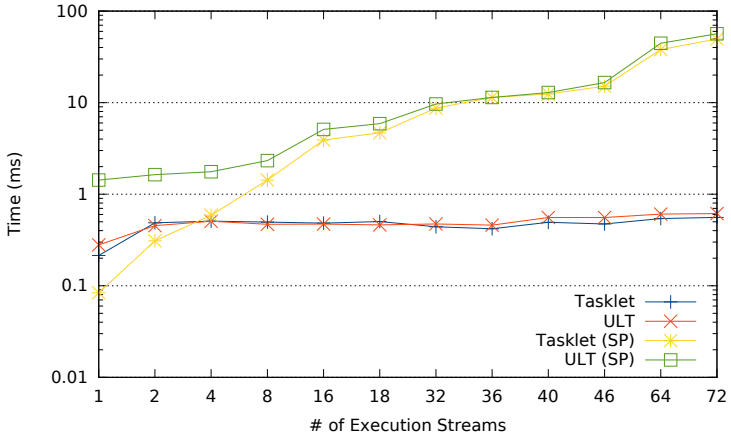


Figure 3.13: Execution time of 1,000 tasks created in a single region with Argobots approaches.

Figure 3.15 shows the execution time when MassiveThreads is employed. In this scenario, the work-first policy outperforms the help-first solution because each Worker generates one task and executes it. This situation reduces contention because just the main task will be stolen.

Figure 3.16 exposes the overall results for 1,000 tasks in a single region. In this case, the implementation choices are Argobots using private pools, Qthreads with one Shepherd for each core, and MassiveThreads with the work-first policy. In this scenario, Argobots both work-units (Tasklets and ULTs) obtains the best performance thanks to its lighter management mechanism and its ES independence allows to avoid internal synchronization procedures. Due to the 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, since Argobots Tasklets are inspired in Converse Threads Messages, their performance is similar, and their utilization reduces the execution time by a factor of two compared with other ULT implementations. Therefore, Converse Threads is among the highest performers thanks to its Messages and their management that is lighter than the ULT functions. Qthreads performs slightly worse than the previous solutions because two main reasons: the use of FEB checks in each memory word and the utilization of more Shepherds than physical cores

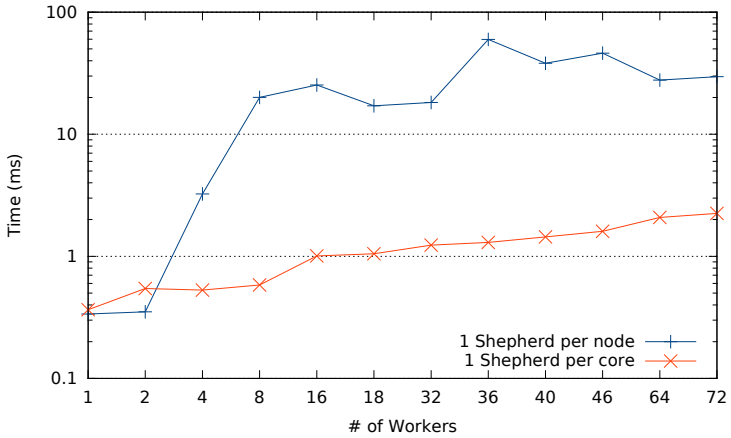


Figure 3.14: Execution time of 1,000 tasks created in a single region with Qthreads approaches.

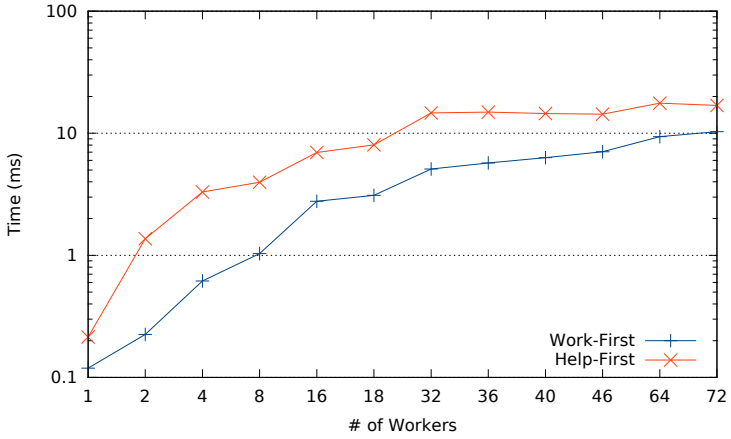


Figure 3.15: Execution time of 1,000 tasks created in a single region with MassiveThreads approaches.

which requires extra synchronization. Go, Intel and GNU are in the middle and this situation demonstrates that the use of an elevated number of tasks affects negatively the performance. The Intel implementation suffers because of the work-stealing mechanism. Worker threads try to gain access to the master thread queue and steal work-units, adding contention. Go follows the trend of GNU because both of them rely on a single shared queue. The worst performance is attained by Pthreads and MassiveThreads: the former because we are creating 1,000 OS threads that causes oversubscription. The latter because the work-first policy implies that each time a new task is created, the main task is stolen by other thread so the data locality drops the overall performance. This mechanism reduces the performance when compared with mechanisms implemented in other LWT solutions.

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, and then they execute the queued tasks. In the GNU implementation (Figure 3.17), all threads compete to gain access to the shared queue, create the tasks (Figure 3.17a) and then execute them (Figure 3.17b).

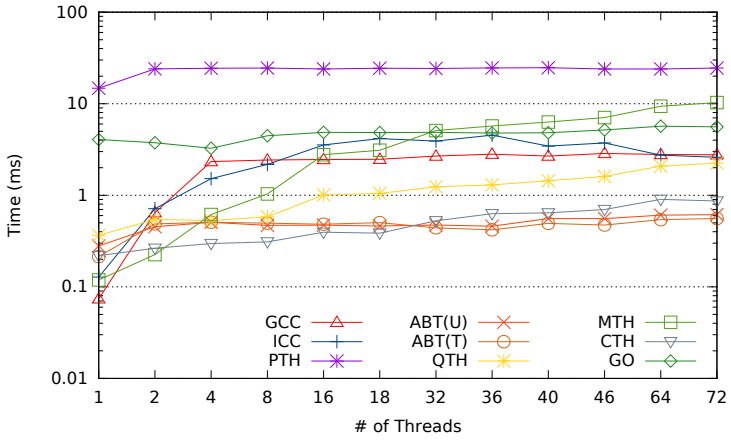


Figure 3.16: Execution time of 1,000 tasks created in a single region with the best configuration for each library.

Figure 3.18 illustrates the Intel OpenMP mechanism. Each thread will push the tasks into its own queue (Figure 3.18a) and then each thread executes its own OpenMP tasks (Figure 3.18b). In this situation, load balance is close to perfect and work-stealing is reduced.

When threading libraries are employed, tasks are created inside a parallel region and each thread has to create its work-units into its own queues. This is implemented by a two-step algorithm. In the first step, the pointer to the parallel code is assigned (like in the for loop scenario); in the second, the tasks are created.

Figure 3.19 reveals the time gap between Pthreads (PTH), Intel and GNU OpenMP implementations (ICC and GCC, receptively). In the case of Pthreads, each thread executes a chunk of iterations of the for loop. Then, each OpenMP task is translated into a Pthread. As in the previous case, we set the `OMP_WAIT_POLICY` environment variable to `passive` in GNU.

This scenario benefits Intel against GNU because in the former each thread creates its own tasks so the work-stealing is almost nonexistent.

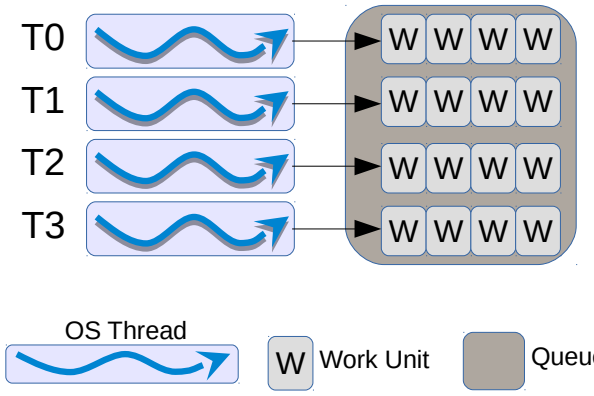
When using Argobots (Figure 3.20), as in the previous cases, the effect of a shared pool (lines labeled as Tasklet (SP) and ULT (SP)) affects the performance because of the contention when more resources are added. This feature encourages the use of separate structures for a higher performance.

Figure 3.21 represents the execution time for Qthreads. Here, the overhead caused when one single Shepherd per node deals with a high number of tasks and Workers is crucial for performance. Conversely, using a round-robin dispatch and separate Shepherds (one Shepherd per core) attains a lower execution time.

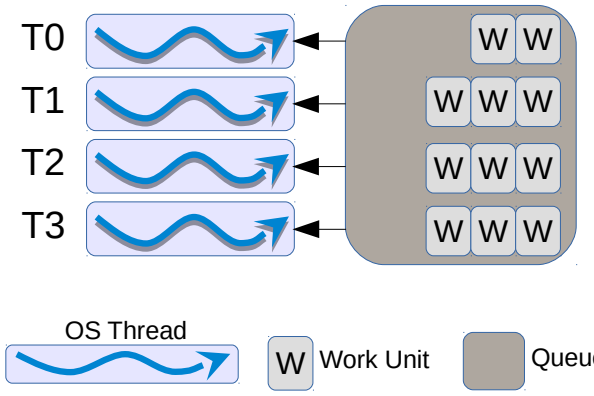
Figure 3.22 shows the MassiveThreads execution time. As in the previous code pattern, the work-first policy benefits from an elevate number of tasks, reducing the queue contention that appears in the help-first policy.

For task parallelism in a parallel region, the implementations yielding higher performance are the same as those in the previous test: Argobots using private pools, Qthreads with one Shepherd for each core, and MassiveThreads with the Work-first policy.

Figure 3.23 displays that, from the point of view of ULT libraries, 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. MassiveThreads is now more



(a) Step 1: All threads in the team create OpenMP tasks into the shared queue.

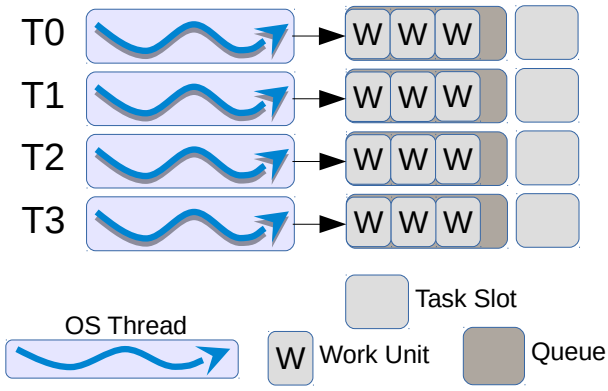


(b) Step 2: All threads in the team execute the existing OpenMP tasks.

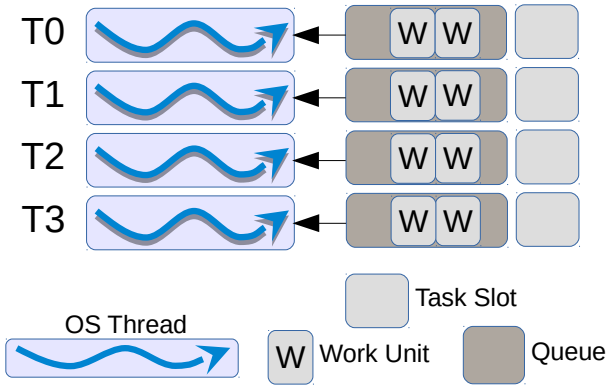
Figure 3.17: Steps the task creation and execution in a parallel region with GNU OpenMP.

efficient because its implementation is designed to deal with recursive paradigms. In addition, all the threads in MassiveThreads are busy, so the work-stealing is almost nonexistent. Qthreads experiences a significant increment because of the number of threads and performs much lower than other ULT libraries (up to 32 times slower than Argobots). Most of this performance drop is because of the time spent in the join mechanism. Although both Argobots implementations use ULTs (that can yield) in the first step, the difference between ULTs and Tasklets is practically negligible.

From the point of view of OS threads, Intel offers higher performance because now, with practically a perfect load balance, the work-stealing has disappeared. GNU outperforms other solutions thanks to its cut-off mechanism (up to eight threads) and to the wait policy value. However, it attains similar time than that of Qthreads. The lowest performance comes from the Pthreads solution due to the oversubscription caused by creating 1,000 threads.



(a) Step 1: Each thread creates its own OpenMP tasks into its queue.



(b) Step 2: Each thread executes its own OpenMP tasks.

Figure 3.18: Steps the task creation and execution in a parallel region with Intel OpenMP.

3.2.2.3 Nested Parallel Constructs

When the current OpenMP implementations find a parallel "pragma" in the user's code, these 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.

Nested parallelism is not common in applications because the performance drops when the number of threads exceeds that of CPU cores, causing oversubscription. However, 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".

Intel and GNU OpenMP implementations accommodate nested parallelism. However, the way they manage the new thread teams is different. The Intel OpenMP runtime creates a new thread team for each thread in the main team reusing or creating idle threads. The GNU implementation does not reuse the idle threads. Each time an OpenMP "pragma" is found, a new team is created

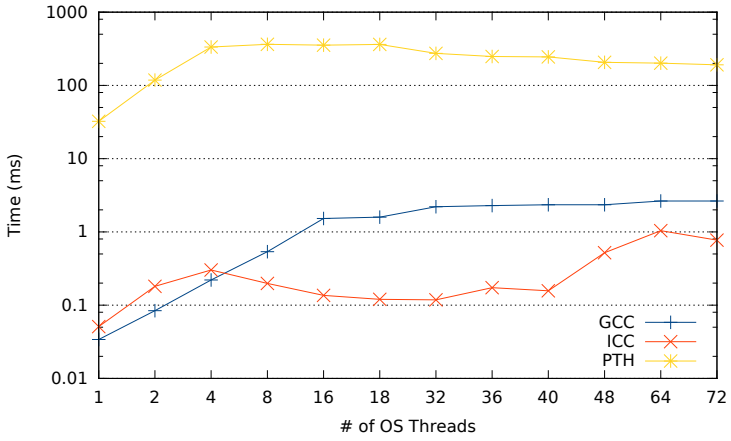


Figure 3.19: Execution time of 1,000 tasks created in a parallel region with Pthreads-based approaches.

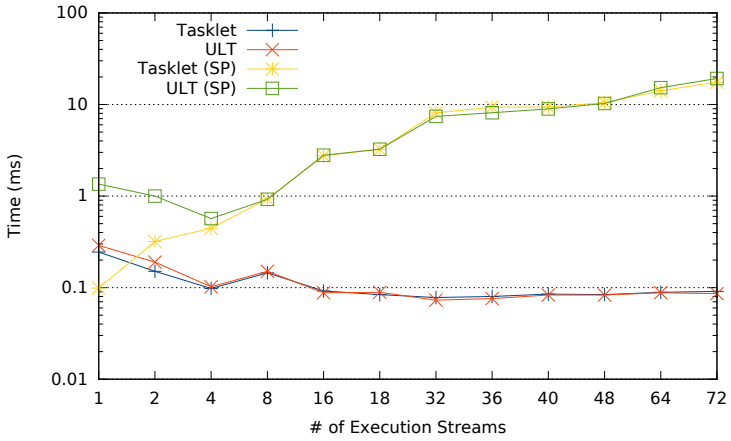


Figure 3.20: Execution time of 1,000 tasks created in a parallel region with Argobots approaches.

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 3.6.

```
#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);
  }
}
```

Listing 3.6: OpenMP nested parallelism.

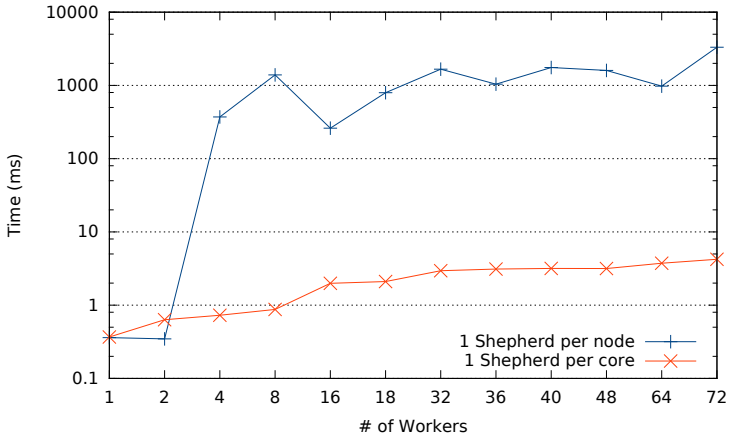


Figure 3.21: Execution time of 1,000 tasks created in a parallel region with Qthreads approaches.

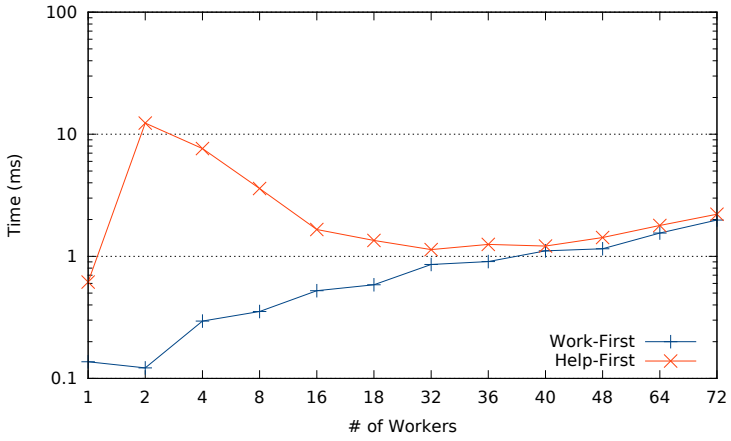


Figure 3.22: Execution time of 1,000 tasks created in a parallel region with MassiveThreads approaches.

With threading libraries (Listing 3.7), 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 threads, which are used to divide the inner loop iterations (lines 18–45).

Figure 3.24 manifests the current problem with nested parallelism. Pthreads (PTH) and GNU OpenMP (GCC) offer an execution time that is unacceptable for HPC. On the one hand, GNU 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). Our Pthreads implementation performs close to the GNU solution because it follows the same approach. On the other hand, Intel 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.

With Argobots (Figure 3.25), as in the previous cases, the effect of a shared pool affects negatively the performance. One more time, the Tasklet usage may overperform the ULT, although in this scenario the time difference is almost nonexistent.

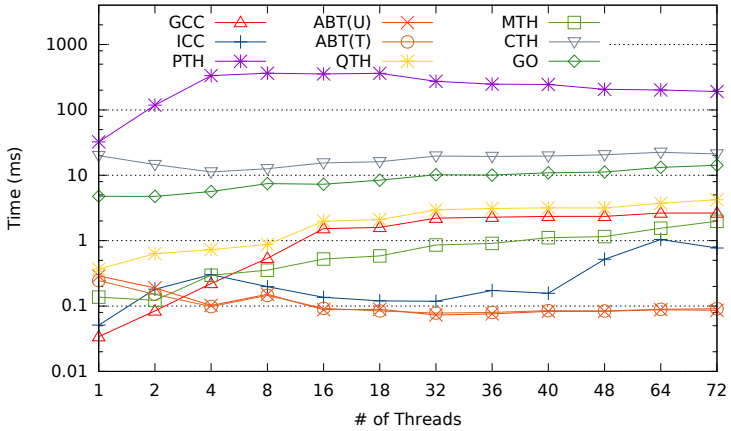


Figure 3.23: Execution time of 1,000 tasks created in a parallel region with the best configuration for each library.

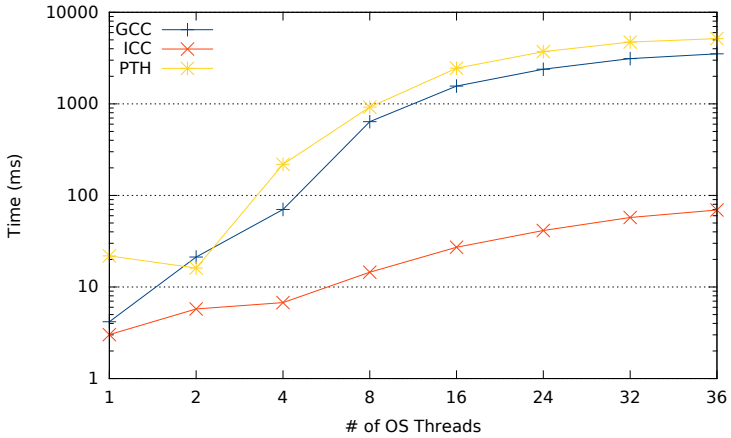


Figure 3.24: Execution time of 1,000-iterations nested for loop with Pthreads-based approaches.

Figure 3.26 depicts the execution time when using Qthreads. The use of one single Shepherd per node dealing with a high number of tasks and Workers drops the performance when more than 8 Workers are used because each processor contains 9 cores. Therefore, a Shepherd located in the CPU 0 may access to other processor core memories. Hence, using a round-robin dispatch and a separate Shepherd per core reduces the execution time.

With MassiveThreads (Figure 3.27), the policy selection depends on the number of requested resources. If there are less than 8 Workers, the help-first policy yields higher performance because the work-stealing happens in the same NUMA node. Conversely, if we select more than 8 Workers, the work-first policy obtains better performance because the work-stealing is reduced.

Figure 3.28 shows the results for this test. Argobots with private pools, Qthreads with one Shepherd for each Worker, and MassiveThreads with Work-first policy are used. Both OS-based approaches, OpenMP and Pthreads 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 in the case of OpenMP and the oversubscription in the case of Pthreads. 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,

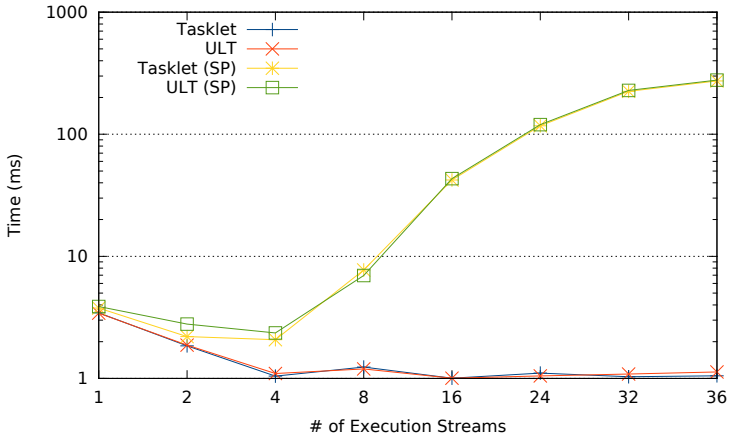


Figure 3.25: Execution time of 1,000-iterations nested for loop with Argobots approaches.

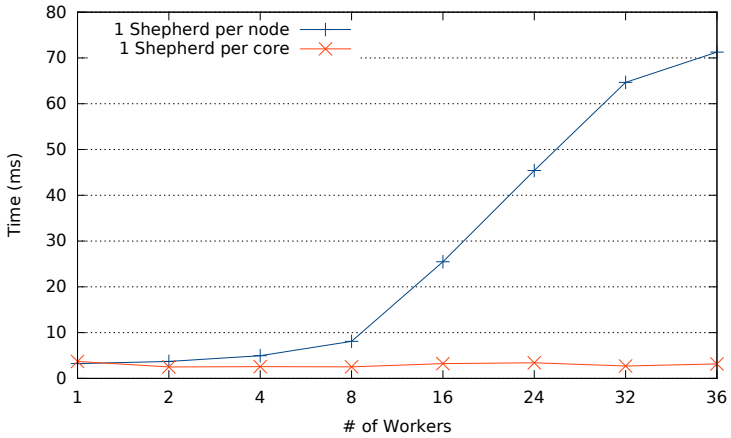


Figure 3.26: Execution time of 1,000-iterations nested for loop with Qthreads approaches.

and the latter is because of the extra yield and barrier functions. However, these implementations perform in between the OpenMP solutions. Conversely, Argobots Tasklets/ULTs, Qthreads, and MassiveThreads show the highest performance because these do not create more threads, just work-units. LWT approaches avoid the oversubscription problem reducing the OS thread management and increasing the performance with respect to the Intel OpenMP approach by factors of 62, 21 and 25 for Argobots, Qthreads, and MassiveThreads, respectively, when 36 threads are used.

3.3 Summary

In this chapter we have presented a deep analysis of a set of threading solutions including both OS threads and LWTs. More concretely, we have performed a semantic and PM decomposition of Pthreads, Converse Threads, MassiveThreads, Qthreads, Argobots, and Go. For each library, we depict their features and strong points, as well as the different (if any) PMs offered depending on the library configuration.

We have proved, by means of experimental tests, 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 normally parallelized with OpenMP "pragmas", offering a performance level that is, at least, as

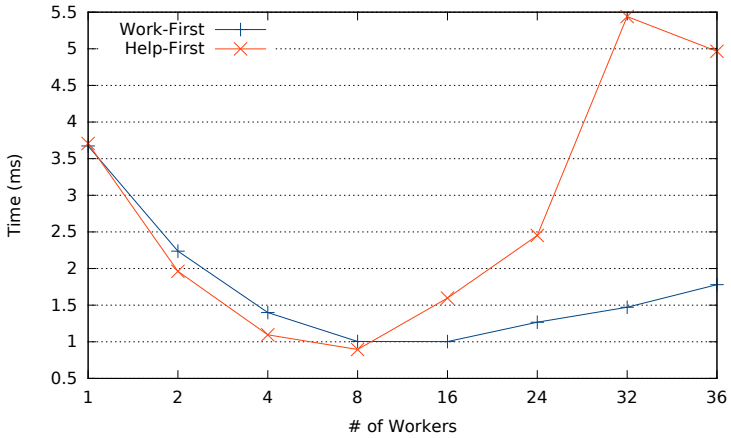


Figure 3.27: Execution time of 1,000-iterations nested for loop with MassiveThreads approaches.

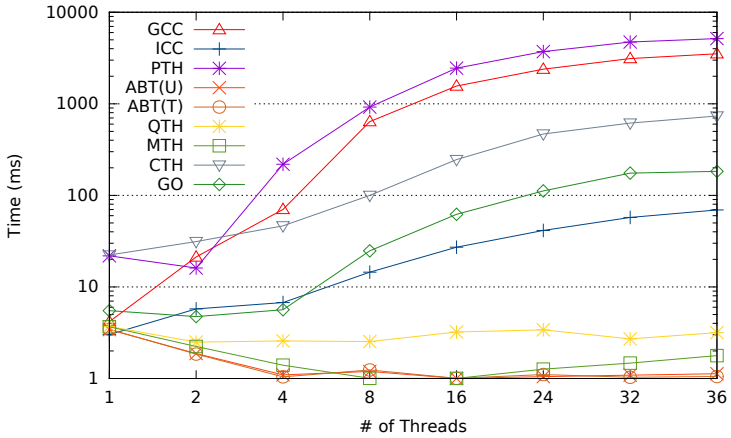


Figure 3.28: Execution time of 1,000-iterations nested for loop with the best configuration for each library.

good as that attained with Pthreads and the OpenMP runtimes, implemented by GNU and Intel. 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.

In conclusion, LWTs improve performance in scenarios that are becoming more popular, such as task parallelism or nested parallel structures. These scenarios are aimed to tackle the problem of extracting all the computational power of exascale systems.

```
//Inner for loop code
void inner_for_lwt(void * args) {
  arg_for *arg = (arg_for*) args;
  for(int i = arg.ini; i < arg.fini; i++)
    code(i);
}

//Outer for loop code
void outer_for_lwt(void * args) {
  arg_for *arg = (arg_for*) args;

  //Allocate memory for structures
  ULT * lwts[NUM_ULTS]; arg_for * args[NUM_ULTS];

  for(int i = arg.ini; i < arg.fini; i++)
  {
    //Calculate the iterations for the outer loop per LWT
    args[i].ini = XXX;
    args[i].fini = XXX;
    //LWT creation
    create_lwt(inner_for_lwt, args[i], lwts[i]);
  }

  lwt_yield();

  //Wait for LWT completion
  for(int i = 0; i < NUM_ULTS; i++)
    join_lwt(lwts[i]);
}

int main() {
  //Allocate memory for structures
  ULT * lwts[NUM_ULTS]; arg_for * args[NUM_ULTS];

  for(int i = 0; i < NUM_ULTS; i++)
  {
    //Calculate the iterations for the outer loop per LWT
    //Argument initialization
    args[i].ini = XXX;
    args[i].ini = XXX;
    //LWT creation
    create_lwt(outer_for_lwt, args[i], lwts[i]);
  }

  lwt_yield();

  //Wait for LWT completion
  for(int i = 0; i < NUM_ULTS; i++)
    join_lwt(lwts[i]);
}
```

Listing 3.7: LWT nested parallelism implementation.

CHAPTER 4

Generic Lightweight Threads (GLT)

The main objective of this chapter is to justify the necessity of a unified API for LWT and present our solution called GLT. To achieve that goal, we first show the benefits of the unified API in two manners: improving the Pthreads API, and demonstrating the code portability. Then, we present the GLT itself detailing its PM, API, modular structure, and semantical mapping. We have implemented our GLT API on top of those LWT libraries in order to expose the usability of the unified API and validate its design. Furthermore, we perform an overhead study verifying that the use of this common API does not introduce any perceptible overhead.

4.1 Limitations of the Pthreads API

In this section we demonstrate the necessity of a new unified API by indicating the limitations of the Pthread API. As it was described in Section 2.2.1, the Pthreads API offers different PMs that can be implemented by different solutions. However, from the semantic point of view, it fails in two critical points for HPC: block control and load balance. Block control is critical in order to fully control the execution flow of the application/runtime. This feature allows programmers to exactly know the system-resource status and to change the executing work-unit. Load balance functionality offers the ability of indicating exactly the relationship between work-units and the resource in charge of executing it (e.g. process–CPU, or ULT–OS thread). Specifically, the Pthreads API does not support the *yield*, *migrate*, and *work-dispatch* semantics. Although some Pthreads implementations include this functionality, it is not defined in the API specification. Therefore, these functions are named with the suffix `_np`, which means "non portable".

Yield functions allow context-switching from a thread that is being executed to another *ready* thread. This feature may benefit blocking calls as depicted in Listings 4.1 and 4.2. Listing 4.1 shows the behavior of a blocking call that stops the execution until the requested operation is completed. Conversely, Listing 4.2 illustrates a behavior that allows executing another work-unit while the current query is not fulfilled.

An improvement of this functionality is represented by `yield_to` functions to signal the next thread to be executed without the scheduler's participation.

migration and *work-dispatch* functions address load imbalance by moving threads from one *ready* queue to another queue and by creating work-units in other threads' queues, respectively.

```
1 void blockingcall(arg1, arg2, ...)
2 {
3   while(!completed()){
4     ;
5   }
6 }
```

Listing 4.1: Example of a blocking call without block control.

```
1 void blockingcall(arg1, arg2, ...)
2 {
3   while(!completed()){
4     lwt_yield();
5   }
6 }
```

Listing 4.2: Example of a blocking call with block control.

Therefore, the LWT API should accommodate that functionality, as well as the semantics offered by the Pthreads API. Hence, a LWT API could be more complete.

This advanced functionality is available in the GLT API because it exposes the KSEs to the programmer in order to enable the threads' mapping/scheduling. Therefore, LWT APIs offer a higher degree of performance overhead control than the Pthreads API, which is important in HPC runtimes and applications.

Pthreads implementations interpret KSEs differently, leading to the previously discussed mappings between KSEs and threads (N:1, 1:1, or M:N). Therefore, users do not have control over this mapping; instead, they have to follow the mapping offered by the threading implementation. In contrast, a unified LWT API like GLT would allow programmers to create the required mapping for each application regardless of the underlying LWT implementation.

4.2 GLT Programming Model

As introduced in Section 2.3, each LWT library offers its own PM. Therefore, choosing a correct default PM for the GLT API is critical. Figure 4.1 depicts the set of elements that compose the GLT PM. A *GLT_thread* is composed of the OS thread, a queue of ULTs/tasklets, and a scheduler that sets the order of the execution of these work-units. The different functionality exposed by their PMs is explained in this section.

A GLT_thread executes work-units (ULTs/Tasklets) in an OS thread. GLT_threads are conceptually equivalent to *Shepherds* in Qthreads, *ESs* in Argobots, and *Workers* in MassiveThreads. GLT_ults are conceptually equivalent to *qthreads* in Qthreads and to *threads* in Argobots and MassiveThreads. The concept of GLT_tasklet is directly borrowed from that of Argobots.

4.2.1 Resource Setup

GLT sets the environment during the initialization function. By default, one thread is created per CPU core. This number, however, can be defined by the user by means of the `GLT_NUM_THREADS`

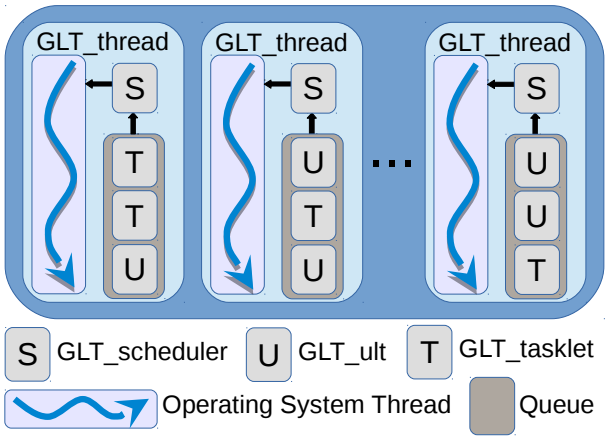


Figure 4.1: GLT PM elements abstraction.

environment variable. Each thread is bound to a specific CPU core in the system following a round-robin assignment. This behavior mimics that of Qthreads and MassiveThreads. When GLT is used on top of Argobots, the GLT initialization function creates one thread per CPU core and distributes the threads among the same number of pools.

Furthermore, nothing prevents users from changing the default initial resources for the underlying LWT library (e.g., number of pools in Argobots or number of workers per thread in Qthreads) by means of its own environment variables, which is honored by the GLT implementation.

Affinity is always enabled mapping one GLT_thread to each CPU system. No other bindings are allowed due to the GLT PMs.

4.2.2 Work-unit Types

While all our reference libraries provide ULTs, Argobots additionally supports tasklets. Tasklets are lighter than ULTs, but cannot migrate or yield because a tasklet does not own a stack. These work-units are suitable for computation codes that do not include blocking calls. All codes that can be executed by a tasklet can also be executed by a ULT. For that reason, the GLT API provides these two kinds of work-units.

If GLT is used on top of a library with no native support for tasklets, ULTs are transparently used underneath instead, yielding the expected functionality but no performance benefits.

4.2.3 Scheduling

GLT scheduling relies on the underlying library. This may be specified during the configuration step prior to building those libraries or, as in the case of Argobots, it can be changed at execution time (by means of EXTENDED functions in GLT). Nevertheless, while leveraging different schedulers may affect performance, GLT semantics are not affected.

4.3 GLT Design and Implementation Details

4.3.1 API

In this section we discuss the decisions made in the design and implementation of this API. The complete API is described in Appendix A.

4.3.1.1 Semantic Mapping

Many GLT functions are simple wrappers to those in the underlying LWT libraries, hence yielding low performance overhead. Some other GLT functions require more elaborate implementations because no direct mapping to the underlying library functionality exists. Listing 4.3 illustrates the example of the `glt_ult_create` function. While for MassiveThreads and Qthreads it is a simple wrapper, for Argobots an additional function call is required (lines 4–5). This extra function obtains the rank of the current ES.

```
1  void glt_ult_create(void(*thread_func)(void *), void *arg,
       GLT_ult *new_ult) {
2  //ARGOBOTS code
3  #ifdef ARGOBOTS
4    int rank;
5    ABT_xstream_self_rank(&rank);
6    ABT_thread_create(main_team->spools[rank], thread_func,
       arg, ABT_THREAD_ATTR_NULL, new_ult);
7  #else
8  //MASSIVETHREADS code
9  #ifdef MASSIVETHREADS
10   *new_ult = myth_create((void *) thread_func, arg);
11 #else
12 //QTHREADS code
13 #ifdef QTHREADS
14   qthread_fork((void *) thread_func, arg, new_ult);
15 #else
16   printf("Error: %s not implemented yet\n", __func__);
17 #endif
18 #endif
19 #endif
20 }
```

Listing 4.3: Example of the `glt_ult_create` GLT function implemented with Argobots, MassiveThreads and Qthreads.

A set of features is common in the management of all types of threads. We group these semantics into the GLT basic part of the API, while other features are left for the extended part, which consists of optionally implemented modules. A basic module may be fulfilled for implementing the GLT API. In contrast, the implementation of an extended module is not mandatory.

The mapping between the most important functions of the GLT API and the reference libraries is shown in Table 4.1.

Init and finalization functions configure and destroy the library environment, respectively. These functions are only used once at the beginning and end of applications. For correctness, first the `GLT_NUM_THREADS` environment variable is translated to each library environment variables. Then, the library is initialized. Listings 4.4, 4.5, and 4.6 show the `glt_init` function for Argobots, MassiveThreads, and Qthreads, respectively.

For work-unit management (`glt_ult_create(_to)`, `glt_tasklet_create(_to)`, `glt_ult_join`, and `glt_tasklet_join`), the lack of tasklet support in Qthreads and MassiveThreads is compensated with the use of the ULT functions. Listing 4.7 reproduces these function equivalences.

GLT	Argobots	Qthreads	MassiveThreads
glt_init	ABT_init	qthread_initialize	myth_init
glt_tasklet_create	ABT_task_create	qthread_fork	myth_create
glt_ult_create	ABT_thread_create	qthread_fork	myth_create
glt_tasklet_create_to	ABT_task_create	qthread_fork_to	myth_create
glt_ult_create_to	ABT_thread_create	qthread_fork_to	myth_create
glt_yield	ABT_thread_yield	qthread_yield	myth_yield
glt_tasklet_join	ABT_task_free	qthread_readFF	myth_join
glt_ult_join	ABT_thread_free	qthread_readFF	myth_join
glt_finalize	ABT_finalize	qthread_finalize	myth_fini

Table 4.1: Mapping between some GLT functions and their equivalent in the underlying libraries.

Moreover, since MassiveThreads does not allow creating ULTs in other workers' ready queue, when a `glt_tasklet/ult_creation_to` is called, the library just creates a ULT in the current worker's queue. The internal work-stealing mechanism compensates for the possible load imbalance. Despite the fact that the different implementation approaches over different underlying native LWT libraries may have performance implications, these all conform to the exposed GLT semantics (offering the same functionality to GLT users) while transparently leveraging the most efficient mechanism underneath.

In addition, the GLT API includes the required functionality for HPC (block control and load balance) that are not offered in the Pthreads API.

Basic Modules. The basic functionality supported by the GLT PM is distributed into the following 8 API modules:

- **Setup.** This module initializes and finalizes the library.
- **Work-unit.** It is composed of 20 functions that are used for work-unit management. This module includes allocation, creation, yield, join, migration, and cancellation operations for work-units as well as handler functions. It supports two types of work-units: ULTs and tasklets. In case the underlying library does not support tasklets, ULTs are leveraged to deliver analogous functionality.
- **Mutex.** This module includes 6 basic functions to create, destroy, lock, unlock, and try to lock mutexes. Qthreads supports only locking and unlocking natively because of the FEB mechanism; the remaining functions have been implemented on top of these semantics.
- **Barrier.** Three functions are provided for barrier management.
- **Condition.** Five condition management functions are supported natively by Argobots and MassiveThreads and developed for Qthreads.
- **Util.** It consists of 6 functions to measure elapsed times or to obtain a timestamp and 2 functions that return the number of threads and the rank of the current thread.
- **Key.** This module hosts 4 work-unit data management functions. Natively supported by Argobots and MassiveThreads, and implemented for Qthreads.

```
void glt_init(int argc, char * argv[])
{
  int num_threads = get_nprocs();
  main_team = (glt_team_t *) malloc(sizeof (glt_team_t));

  ABT_init(argc, argv);

  //Set the number of Execution Streams
  if(getenv("GLT_NUM_THREADS") != NULL) {
    num_threads = atoi(getenv("GLT_NUM_THREADS"));
  }
  int num_pools = num_threads;

  //Set the number of Pools
  if(getenv("GLT_NUM_POOLS") != NULL) {
    num_pools = atoi(getenv("GLT_NUM_POOLS"));
  }

  ABT_xstream_self(&main_team->master);

  //Generate the main team data structure
  main_team->num_xstreams = num_threads;
  main_team->num_pools = num_pools;
  main_team->team = (ABT_xstream *) malloc(sizeof(
   ABT_xstream) * num_threads);
  main_team->pools = (ABT_pool *) malloc(sizeof(ABT_pool) *
    num_pools);

  //Create the required pools
  for(int i = 0; i < num_pools; i++) {
    ABT_pool_create_basic(ABT_POOL_FIFO,
   ABT_POOL_ACCESS_MPMC, ABT_TRUE, &main_team->pools[i]);
  }

  //Set the pools for the main Execution Stream
  ABT_xstream_self(&main_team->team[0]);
  ABT_xstream_set_main_sched_basic(main_team->team[0],
   ABT_SCHED_DEFAULT,1, &main_team->pools[0]);

  //Create the secondary Execution Streams and bind them to
    their pools
  for(int i = 1; i < num_threads; i++) {
    ABT_xstream_create_basic(ABT_SCHED_DEFAULT, 1,&
   main_team->pools[i % main_team->num_pools],
   ABT_SCHED_CONFIG_NULL, &main_team->team[i]);
    ABT_xstream_start(main_team->team[i]);
  }
}
```

Listing 4.4: `glt_init` function implemented with Argobots.

```
void glt_init(int argc, char * argv[])
{
  int num_threads = get_nprocs();
  main_team = (glt_team_t *) malloc(sizeof (glt_team_t));
  char buf[10];

  //Set the number of Workers
  if(getenv("GLT_NUM_THREADS") != NULL) {
    num_threads = atoi(getenv("GLT_NUM_THREADS"));
    sprintf(buf, "%d", num_threads);
    setenv("MYTH_WORKER_NUM", buf, 1);
  } else if(getenv("MYTH_WORKER_NUM") != NULL) {
    num_threads = atoi(getenv("MYTH_WORKER_NUM"));
  } else {
    sprintf(buf, "%d", num_threads);
    setenv("MYTH_WORKER_NUM", buf, 1);
  }

  setenv("MYTH_BIND_WORKERS", "1", 1);
  main_team->num_workers = num_threads;

  //Start the library
  myth_init();
}
```

Listing 4.5: `glt_init` function implemented with MassiveThreads.

- **Query.** This module includes 3 functions (one for each extended module). These offer the programmer the ability to check the availability of an advanced feature.

Extended Modules. Although some LWT solutions offer a set of diverse functionalities, not all the functions are strictly necessary in order to implement a threading application/runtime. For example, Qthreads includes a module of system calls or atomic operations that are wrappers of the GNU functions. These functions are discarded in the common API because they can be added by the programmers in their codes. Other discarded functionality are the profiling and logging functions offered by MassiveThreads, as well as Eventuals that compose the Argobots API.

There is also functionality that is only usable in the low-level LWT library, such as the FEB functions offered by Qthreads. Those functions are not included in the common API.

However, we have selected three extended function modules. These modules offer an additional functionality but are not required in order to complete the GLT API.

These modules are:

- **Scheduler.** It consists of 11 functions that allow programmers to change the scheduler at runtime.
- **Event.** Two functions are provided for event management.
- **Future.** This module hosts 4 future management functions.

Those functionalities are currently only supported by Argobots and do not perform any action if called on top of Qthreads or MassiveThreads. Scheduler functions allows the GLT users to configure

```
void glt_init(int argc, char * argv[])
{
  int num_threads = get_nprocs();
  main_team = (glt_team_t *) malloc(sizeof(glt_team_t));
  char buf[10];

  int num_workers_per_thread;

  //Set the number of Shepherds
  if(getenv("GLT_NUM_THREADS") != NULL) {
    num_threads = atoi(getenv("GLT_NUM_THREADS"));
    sprintf(buf, "%d", num_threads);
    setenv("QTHREAD_NUM_SHEPHERDS", buf, 1);
  } else if(getenv("QTHREAD_NUM_SHEPHERDS") != NULL) {
    num_threads = atoi(getenv("QTHREAD_NUM_SHEPHERDS"));
  } else {
    sprintf(buf, "%d", num_threads);
    setenv("QTHREAD_NUM_SHEPHERDS", buf, 1);
  }
  //Set the number of Workers per Shepherd
  if(getenv("GLT_NUM_WORKERS_PER_THREAD") != NULL) {
    num_workers_per_thread = atoi(getenv("
   GLT_NUM_WORKERS_PER_THREAD"));
    sprintf(buf, "%d", num_workers_per_thread);
    setenv("QTHREAD_NUM_WORKERS_PER_SHEPHERD", buf, 1);
  } else if(getenv("QTHREAD_NUM_WORKERS_PER_SHEPHERD") !=
   NULL) {
    num_workers_per_thread = atoi(getenv("
   QTHREAD_NUM_WORKERS_PER_SHEPHERD"));
  } else {
    setenv("QTHREAD_NUM_WORKERS_PER_SHEPHERD", "1", 1);
    num_workers_per_thread = 1;
  }

  main_team->num_shepherds = num_threads;
  main_team->num_workers_per_shepherd =
   num_workers_per_thread;

  //Start the library
  qthread_initialize();
}
```

Listing 4.6: `glt_init` function implemented with Qthreads.

ad-hoc schedulers at runtime replacing those created by default in the library initialization. In the case of Event and Future, those mechanism are used for synchronization aspects. Both modules helps the Argobots' users to better design its environment without abandoning the GLT API.

```
1  void glt_tasklet_create(void(*thread_func)(void *), void *
       arg, GLT_tasklet *new_tasklet) {
2  //ARGOBOTS code
3  #ifdef ARGOBOTS
4    int rank;
5    ABT_xstream_self_rank(&rank);
6    ABT_task_create(main_team->spools[rank], thread_func, arg
       , new_tasklet)
7  #else
8  //MASSIVETHREADS code
9  #ifdef MASSIVETHREADS
10   *new_tasklet = myth_create((void *) thread_func, arg);
11 #else
12 //QTHREADS code
13 #ifdef QTHREADS
14   qthread_fork((void *) thread_func, arg, new_tasklet);
15 #else
16   printf("Error: %s not implemented yet\n", __func__);
17 #endif
18 #endif
19 #endif
20 }
```

Listing 4.7: Example of the `glt_tasklet_create` GLT function implemented with Argobots using tasklets, and with MassiveThreads and Qthreads using ULTs.

4.3.1.2 GLT Objects

GLT objects start with the upper-case prefix "`GLT_`". Table 4.2 shows the equivalences between the main GLT object types and those of the reference libraries. Those objects that are not supported by a LWT solution are represented as `void *`.

As in the function mapping, when a "GLT_tasklet" is invoked, MassiveThreads and Qthreads use "thread_t" and "aligned_t" objects, respectively, as they do with "GLT_ult".

4.3.2 Implementations

Our GLT implementation can be used in two ways. On the one hand, a set of *dynamic* libraries compiled on top of the different reference libraries may be generated. This eases the switch among the underlying LWT implementations by linking the application to a different library at load time. On the other hand, we have devised our GLT implementation as a *header-only* library. This second approach offers higher performance than the former because all the functions are labeled as `static inline`. Most compilers will honor these modifiers and prevent the additional function call. The performance result in most cases is analogous to that obtained if the user employs the original library directly, yielding no performance impact for those functions with a direct mapping to the underlying library. The performance of these two approaches is analyzed in Section 4.5.

GLT	Argobots	Qthreads	MassiveThreads
GLT_ult	ABT_thread	aligned_t	myth_thread_t
GLT_tasklet	ABT_task	aligned_t	myth_thread_t
GLT_thread	ABT_xstream	qthread_shepherd_id_t	myth_thread_t
GLT_mutex	ABT_mutex	aligned_t	myth_mutex_t
GLT_barrier	ABT_barrier	qt_barrier_t	myth_barrier_t
GLT_cond	ABT_cond	aligned_t	myth_cond_t
GLT_timer	ABT_timer	qtimer_t	myth_timer_t
GLT_bool	ABT_bool	<int>	<int>
GLT_thread_id	ABT_thread_id	<int>	<int>
GLT_ult_id	<int>	<int>	<int>
GLT_sched	ABT_sched	void *	void *
GLT_sched_conf	ABT_sched_config	void *	void *
GLT_event_kind	ABT_event_kind	void *	void *
GLT_event_cb_fn	ABT_event_cb_fn	void *	void *
GLT_future	ABT_eventual	void *	void *

Table 4.2: GLT object equivalences (C basic data types between "<" and ">").

4.3.3 Code Example

The code example in Listing 4.8 illustrates the use of the GLT API. In this code, the main thread generates 100 tasklets that are dispatched using a round-robin policy (lines 18–23). Then, it yields to allow the main thread to execute some tasklets (line 26). Finally the main thread waits until the completion of the tasklets (lines 28–32). Since tasklets are natively supported by Argobots only, the function calls in lines 16, 21, 31 and 33 lead to the use of their ULT equivalent automatically for the case of Qthreads and MassiveThreads only.

4.4 Benefits of a Unified LWT API

In this section we present the usability of a unified LWT API in terms of code portability.

A unified threading API implemented on top of several underlying libraries avoids having to modify the application code in order to execute it on top of different threading solutions. Different hardware platforms may leverage distinct native LWT libraries for technical or strategic reasons. If more than one is available, users may want to select the library delivering the best performance for their particular case.

To support this assertion experimentally, we have designed two simple microbenchmarks that create fine-grained ULTs. These microbenchmarks are merely created in order to demonstrate how a programmer could benefit from the unified API, selecting the desired underlying solution and attaining the best performance possible without modifying the application code.

The results in this chapter are the average of 1,000 executions 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 LWT libraries are Argobots 06-2017, Qthreads version 1.10, and MassiveThreads version 0.95. The test cases and the libraries were compiled using `gcc 6.3` version.

In the first microbenchmark, each *thread* (ESs, Shepherds, Workers, and GLT_threads for Argobots, Qthreads, MassiveThreads, and GLT, respectively) creates and executes a range of ULTs.

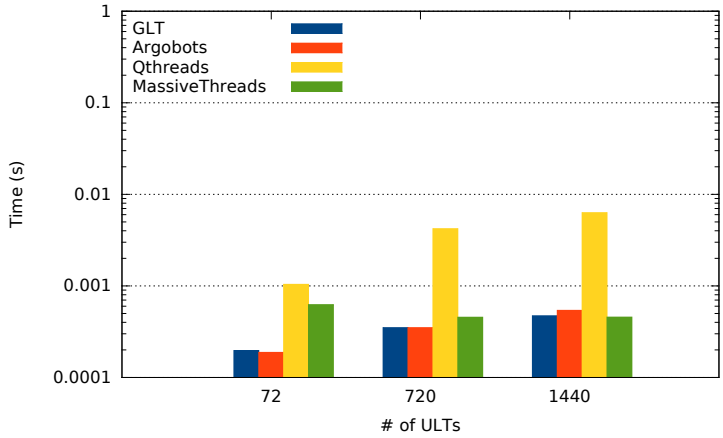
```c
#define N 100

void example()
{
  printf("Hello world, I'm being executed by Thread %d/%d",
       glt_get_thread_num(), glt_get_num_threads());
}

int main(int argc, char * argv [])
{
  // Library initialization
  glt_init(argc,argv);

  // Allocation for Tasklet handlers
  // If an underlying library does not allow tasklets, ULTs
    are used
  GLT_tasklet * tasklets_id = glt_tasklet_malloc(N);

  for(int i=0; i<N; i++)
  {
    //Work-unit creation
    glt_tasklet_create_to(example, NULL, &tasklets_id[i],
                i%glt_get_num_threads());
  }

  //Master thread starts executing pending work
  glt_yield();

  for(int i=0; i<N; i++)
  {
    //Synchronous call for work completion
    glt_tasklet_join(&tasklets_id[i]);
  }
  glt_free(tasklets_id);

  //Library environment cleaned
  glt_finalize();
}
```

Listing 4.8: Example of a LWT program using the GLT API.

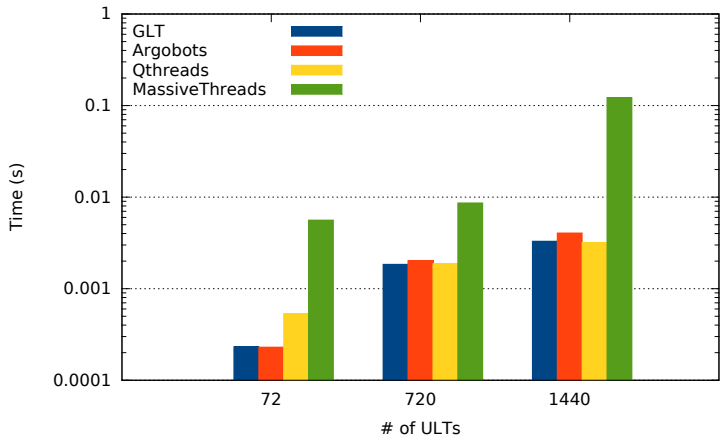
In the second microbenchmark, a single *thread* creates all the ULTs, which are executed by all the *threads*. These microbenchmarks have been implemented natively on top of each native LWT library (Argobots, Qthreads, and MassiveThreads), as well as using the static version of GLT API. Each test has been executed using 72 *threads* with 72, 720, and 1,440 ULTs.

Figure 4.2 shows the microbenchmarks' performance results. Although the GLT implementations are executed on top of the three libraries, only that offering the highest performance is shown. In Figure 4.2a this corresponds to GLT over Argobots for 72 and 720 ULTs and to GLT over MassiveThreads for the largest size. In Figure 4.2b, on the other hand, GLT over Argobots is

the best option for the smallest dataset size, while GLT over the Qthreads library offers the highest performance for the other two problem sizes.



(a) All threads create ULTs.



(b) A single thread creates all the ULTs.

Figure 4.2: Performance of the underlying LWT libraries and the best GLT implementation choice when a set of ULTs are created and executed.

These experiments demonstrate the benefits of using a unified LWT API on top of different underlying native implementations. Within the same platform, different LWT libraries may yield distinct performance for different applications. Even the same application may benefit from different LWT implementations depending on the dataset sizes. Therefore, a unified LWT API such as GLT enables users to select the most appropriate underlying native LWT implementation while avoiding the additional work of implementing the same application using several LWT APIs.

4.5 Overhead Evaluation

We next compare the performance of our test cases implemented directly on top of the low-level libraries with the codes that use the GLT API (both stand-alone and header-only versions). Concretely, we analyze the performance impact of the GLT API with microbenchmarks, and two benchmarks —namely N-Queens and Unbalanced Tree Search (UTS) —. The software and hardware configuration employed was introduced in Sect. 4.4.

4.5.1 Microbenchmarks

We leverage the Callgrind profiling tool [66] to measure the overhead in terms of Instructions Per Call (IPC) of the most frequently used functions of the GLT code for our three reference LWT libraries. These functions are initialization (`Init`), work-unit allocation (`Malloc`), work-unit creation (`Creation`), yield (`Yield`), join (`Join`), and query of number of threads (`Num_thr`).

Figure 4.3 shows our results for the Argobots, Qthreads, and MassiveThreads GLT implementations, comparing the results with the native approaches. The plots expose a common pattern: `Init`, `Malloc`, and `Creation` show a small increment in the number of instructions in both GLT variants (*dynamic* vs *static*); but `Yield`, `Join`, and `num_threads` experience this increment only in the stand-alone version of GLT. These results reflect that the second group of functions contains pure wrappers to the original functions and that the additional function call overhead is added only in case of leveraging a separate GLT library. The library initialization function adds a relatively high number of instructions because of the GLT environment configuration. Nevertheless, this is a one-time overhead introducing merely 10–15% additional instructions compared with the native LWT solutions. The `Malloc` overhead (up to 4 instructions per call) is caused by the type casting of the value returned by the allocation function to the appropriate work-unit pointer. The instructions added in `Creation` are due to the function pointer casting and the return of the work-unit handler. These results confirm that the use of the GLT library as a high-level LWT API introduces fairly low overhead.

4.5.2 N-Queens

We evaluated the overhead in terms of execution time of the GLT API using a translation from an OpenMP version of N-Queens [46]. The number of lines of code needed in the translation are 185 for Argobots code compared with 158 for Qthreads, MassiveThreads, and GLT. Our unified API does not add more lines to the code; indeed, it even reduces the number compared with Argobots. The reason is the automatic environment setup described in Section. 4.3.

In the base OpenMP implementation (Listing 4.9), a single thread creates the first set of tasks (to place a queen in a cell) and executes a `taskwait`. Each task creates more tasks and waits for their termination.

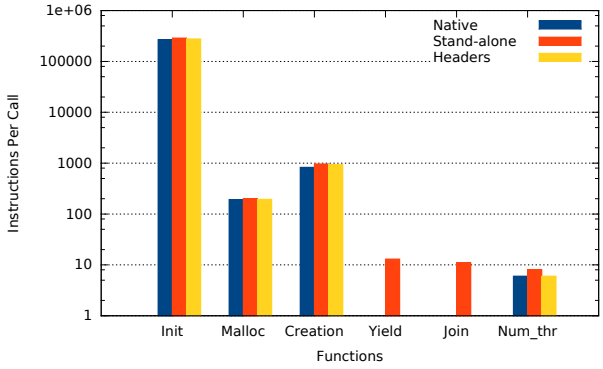
Our implementation of this algorithm using LWTs follows the same philosophy (the pseudocode with GLT functions can be found in Listing 4.10). The main thread creates the first work-units, and each of these is placed into other threads' queue until each thread has at least one work-unit to be executed. Once that is completed, each thread creates its own work. The threads wait for the finalization using the *join* function.

Table 4.3 summarizes the average overhead of several thread configurations (from 1 to 72 threads), for three problem sizes—10, 11, and 12 queens—and the reference LWT libraries. While the average overhead for the stand-alone version varies from 0.28% to 0.56%, for the header-only GLT deployment this overhead is less than 0.1%.

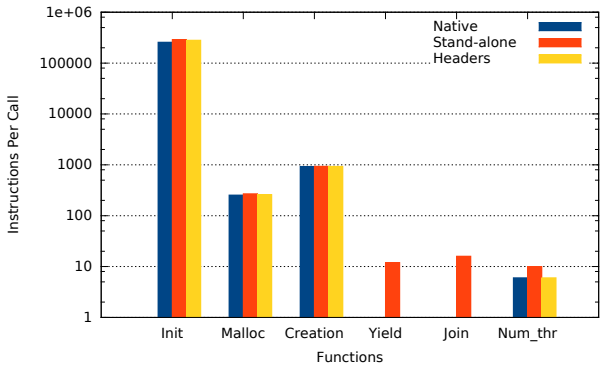
These results showcase the low overhead introduced by the use of the GLT API. The results also show a constant behavior that indicates that the overhead is not caused by the problem size. The largest cost with respect to the native implementations is under 0.6%.

4.5.3 UTS Benchmark

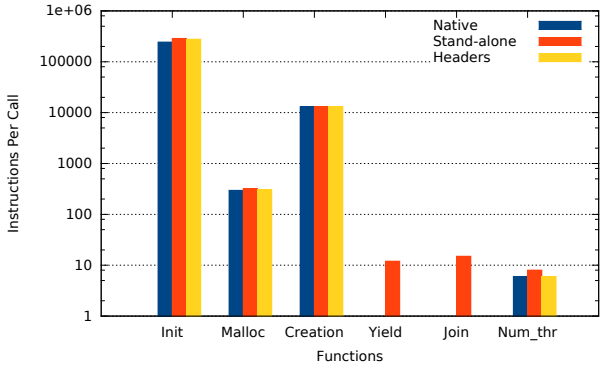
UTS Benchmark [23] is a parallel code that measures the performance attained when executing an exhaustive search on an unbalanced tree. The tree is built at execution time by using a divisible random number generator that splits the structure, making possible the parallel processing while



(a) Argobots.



(b) Qthreads.



(c) MassiveThreads.

Figure 4.3: GLT approaches overhead (IPC) when compared with native libraries.

```
1  void nqueens(int n, int j, char *a, int *solutions, int
       depth)
2  {
3    //Stop condition
4    if (n == j) {
5      *solutions = 1;
6      return;
7    }
8    //General case
9    *solutions = 0;
10   for(int i = 0; i < n; i++) {
11     #pragma omp task
12       nqueens(n, j + 1, b, &csols[i], depth);
13   }
14   #pragma omp taskwait
15   for(int i = 0; i < n; i++)
16       *solutions += csols[i];
17 }
18
19
20 int main(int argc, char * argv [])
21 {
22   #pragma omp parallel
23   {
24     #pragma omp single
25     {
26       nqueens(size, 0, a, &total_count, 0);
27     }
28   }
29 }
```

Listing 4.9: Pseudo-code of the N-Queens application using OpenMP.

still generating a deterministic tree. We translated the original code written with the Pthreads API to our GLT API using 71 code lines for the Argobots implementation and 38 for MassiveThreads, Qthreads, and GLT.

In the original Pthreads implementation, the main thread initializes the tree and places the first (tree) node into its own queue; then all threads execute the same function. First, the next node in the queue is executed, and this node creates more nodes that are pushed into the local queue. If its local queue is empty, a thread tries to steal a certain number of nodes from other queues.

In our implementation, a work-unit is created for each thread, and work-stealing is performed as in the original code. Accessing other threads' queues requires synchronization among threads and is done via *GLT_mutex*.

A pseudo code implementation for threading libraries is shown in Listing 4.11. In this scenario, GLT can leverage the lighter tasklet work-unit because the code does not include any blocking or system call. As discussed in Sect. 4.3, GLT implementations over MassiveThreads and Qthreads employ ULTs instead of tasklets. For reference, we also include the results for native Argobots based on ULTs.

```
void nqueens(void * args) {
  //Stop condition
  if(end())
    add_solution();
  else {
    GLT_ult * ids = glt_ult_malloc(n);
    initialize_args(&args);
    calculate_dest(&dest);
    //Create recursive calls
    for(int i = 0; i < nqueens; i++)
      glt_ult_create_to(nqueens, (void *)&args, &ids[i],
   dest);
    for(int i = 0; i < nqueens; i++)
      glt_ult_join(&ids[i]);
    glt_free(ids);
  }
}

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

  glt_init(argc,argv);
  GLT_ult ult_id;
  initialize_args(&args);
  //Create the first ULT
  glt_ult_create_to(nqueens, (void *)&args, &ult_id, 0);
  glt_yield();
  //Wait until the ULT is completed
  glt_ult_join(&ult_id);
  glt_finalize();
}
```

Listing 4.10: Pseudo-code of the N- application using GLT.

We calculated the average overhead for all the executions of different problem sizes in order to obtain a global vision of the overhead introduced by the GLT API. Table 4.4 shows the average overhead when executing the UTS benchmark with problems T1, T1L, T1XL, and T1XXL (of 4 million, 102 million, 1.6 billion, and 4.2 billion nodes, respectively), on top of the three underlying libraries, modifying the number of threads from 1 to 72. As in the N-Queens case, the difference using the stand-alone (S) and header-only (H) GLT versions is perceivable, being under 0.6% for the former and just slightly above 0.1% for the latter.

The results also show a trend that does not correspond with the problem size; hence it indicates that the overhead is not affected by the size of the problem.

4.6 Pthread-GLT Interaction

As it was discussed in Section 4.3, the GLT API improves the current Pthreads API by adding load balance and block control functionality. However, improving code portability is one goal of this new API. For that reason, we have explored and implemented the interaction among both

GLT Underlying Library (Mode)	Number of Queens		
	10	11	12
Argobots (H)	0.01	0.06	0.04
Argobots (S)	0.28	0.36	0.32
MassiveThreads (H)	0.02	0.01	0.00
MassiveThreads (S)	0.48	0.33	0.49
Qthreads (H)	0.08	0.08	0.09
Qthreads (S)	0.43	0.51	0.56

Table 4.3: GLT average (%) time overhead executing the N-Queens application using headers (H) and stand-alone (S) GLT implementations over the three underlying libraries.

```
1  void parTreeSearch(void * args) {
2    while(!done()){
3      if(local_work()){
4        examine_node(args->stack);
5        continue;
6      }
7      do_steal();
8    }
9  }
10
11 int main(int argc, char * argv []) {
12   thread * thread_ids = thread_malloc(n);
13   initialize_uts_tree(&tree);
14   initialize_args(&args);
15   for(int i = 0; i < n; i++)
16     thread_creation(parTreeSearch, (void *)&args, &
     tasklet_id[i], i);
17   thread_yield();
18   for(int i = 0; i < n; i++)
19     thread_join(&tasklet_id[i]);
20   thread_free(ids);
21 }
```

Listing 4.11: Pseudo-code of the UTS benchmark for threading libraries.

APIs. On the one hand, we have implemented the GLT API over the Pthreads library with the aim of comparing LWT codes with OS threads. Table 4.5 illustrates this relationship.

Functions `glt_ult_create_to` and `glt_tasklet_create_to` are implemented by a combination of `pthread_create` and `pthread_setaffinity_np` calls.

On the other hand, following the MassiveThreads' approximation, we have also implemented the Pthreads API with the functionality offered by GLT. With this implementation, we can automatically execute codes written with Pthreads on top of LWT solutions. Table 4.6 shows some function relationships between the Pthreads API and the GLT API. Since it is impossible to determine if a new thread will execute a blocking code, all creation functions in the Pthreads API are translated using ULT equivalent functions.

GLT Underlying Library (Mode)	Problem Size			
	T1	T1L	T1XL	T1XXL
Argobots Task (H)	0.06	0.00	0.01	0.00
Argobots Task (S)	0.08	0.36	0.39	0.28
Argobots ULT (H)	0.03	0.01	0.01	0.00
Argobots ULT (S)	0.24	0.55	0.22	0.53
MassiveThreads (H)	0.11	0.00	0.08	0.05
MassiveThreads (S)	0.45	0.50	0.45	0.18
Qthreads (H)	0.00	0.01	0.02	0.06
Qthreads (S)	0.30	0.55	0.58	0.29

Table 4.4: GLT average time (%) overhead executing the UTS benchmark using headers (H) and stand-alone (S) GLT implementations over the three underlying libraries.

GLT	Pthreads
glt_init	–
glt_tasklet_create	pthread_create
glt_ult_create	pthread_create
glt_tasklet_create_to	pthread_create + pthread_setaffinity_np
glt_ult_create_to	pthread_create + pthread_setaffinity_np
glt_yield	pthread_yield
glt_tasklet_join	pthread_join
glt_ult_join	pthread_join
glt_finalize	–

Table 4.5: Mapping between some GLT functions and their equivalent in the Pthreads API.

Pthreads	GLT
pthread_create	glt_ult_create
pthread_yield	glt_yield
pthread_join	glt_ult_join
pthread_cancel	glt_ult_cancel

Table 4.6: Mapping between the Pthreads API and the GLT API.

For illustrating the usability of implementing the Pthreads API on top of GLT, we have executed the UTS benchmark (originally implemented on top of the Pthreads API) with different LWT solutions via GLT. Figure 4.4 shows the execution time of the benchmark with the T1XL problem size. As it was discussed in the previous section, UTS generates a set of threads once, and then these threads interact for completing the search algorithm. For this benchmark, Argobots and MassiveThreads (GLT(ABT) and GLT(MTH), respectively) perform close to the Pthreads execution. Since this algorithm uses *mutex* for the interaction among threads, Qthreads (GLT(QTH)) drops the performance because internally each *mutex* function requires additional memory access synchronization.

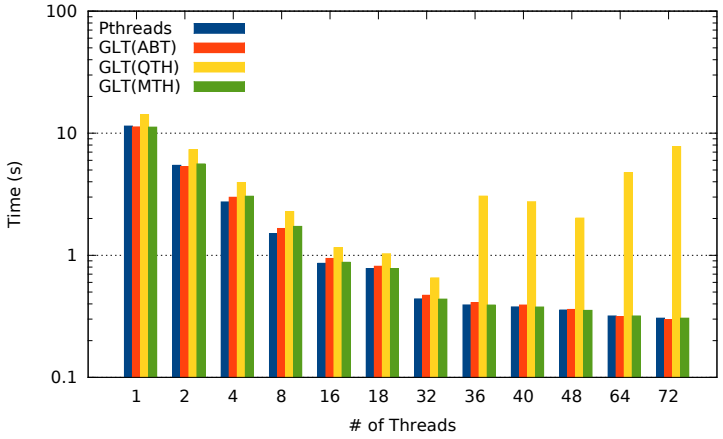


Figure 4.4: UTS benchmark (T1XL size) execution time implemented with the Pthreads API and executed with GLT.

4.7 Summary

In this chapter we have introduced the GLT API [6]. This library proposes a unified API for LWT solutions that is the first attempt to standardize LWT APIs and PMs. We have presented GLT implemented on top of the major general-purpose LWT solutions for HPC: Argobots, MassiveThreads, and Qthreads.

We have discussed the GLT PMs and decomposed the API's modules, presenting semantic mapping between the GLT API and the LWT solutions. Moreover, we have highlighted the limitations of Pthreads API and we have also justified the need for a unified LWT API from the point of view of portability and functionality.

Our performance evaluation, based on stand-alone and header-only implementations of the GLT API, demonstrates the low performance overhead of this approach. We have demonstrated this overhead with a set of microbenchmarks that measure the instructions per call added with GLT. Moreover, we have assessed the overhead by comparing the execution time of two applications where the stand-alone implementation produced an average overhead under 0.6%, while the header-only version showed an average overhead below 0.1%.

CHAPTER 5

Lightweight Threads for High-Level Parallel Programming Models

In this chapter we present and analyze our implementation of the high-level, directive-based PMs OpenMP [42] and OmpSs [30] on top of the GLT API, called GLTO and GOmpSs, respectively. These implementations validate the usability of the unified API and make the use of LWTs easier to programmers.

For each PM, we first depict an analytical study of the interaction between high-level directives and the GLT library. Then, we explain the design details for each PM. We completed this study with the experimental performance evaluation of that relationship in different OpenMP and OmpSs scenarios.

5.1 OpenMP over GLT (GLTO)

In this section we review the design decisions that were made in order to adapt the LLVM [13] OpenMP runtime to the use of LWTs.

Our implementation is based on the BOLT [2] project which is, in turn, based on LLVM. We selected this starting point because both the runtime and the `clang` compiler [3] are open source. In addition, this runtime can be linked with code generated with the Intel compiler.

5.1.1 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`, `gcc` or `icc` compilers. Figure 5.1 shows that an OpenMP code compiled with these tools can be linked with the traditional OpenMP runtimes 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 LWT library without modifying the OpenMP runtime code.

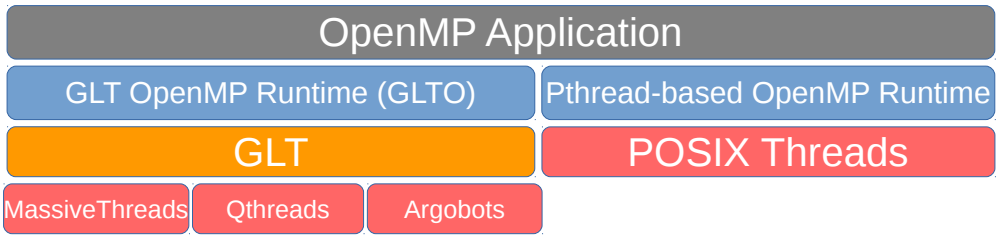


Figure 5.1: Software stack choices of an OpenMP code.

5.1.2 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.

In GLTO, just the underlying threading library varies from the traditional Pthreads-based approach lying on top of the unified GLT API. Figure 5.2 compares both OpenMP implementations when translate a `#pragma omp parallel` directive. Figure 5.2a shows the original Intel OpenMP implementation. In this case, the master thread creates a team of Pthreads if needed, then it assigns the work to be done in parallel and executes its own work. Once this work is completed, the master joins the other threads in the team. In the GLTO implementation (Figure 5.2b), the master GLT_thread creates one GLT_ults for each GLT_thread for the execution of the parallel code, then executes its own work and joins the Glt_ults. There is no need of creating a GLT_thread's team because these are spawned when the library is loaded.

Complying with the OpenMP specifications [58], our GLTO implementation responds to the definition of the `OMP_NUM_THREADS` environment variable creating as many GLT_threads as OpenMP threads are requested by the user. As depicted in Figure 5.3, GLT_threads are bound to CPU cores. These are in charge of executing GLT_ults created at runtime. Standard-compliant dynamic adjustment of threads via the num_threads clause and the `omp_set_num_threads` library routine is also possible.

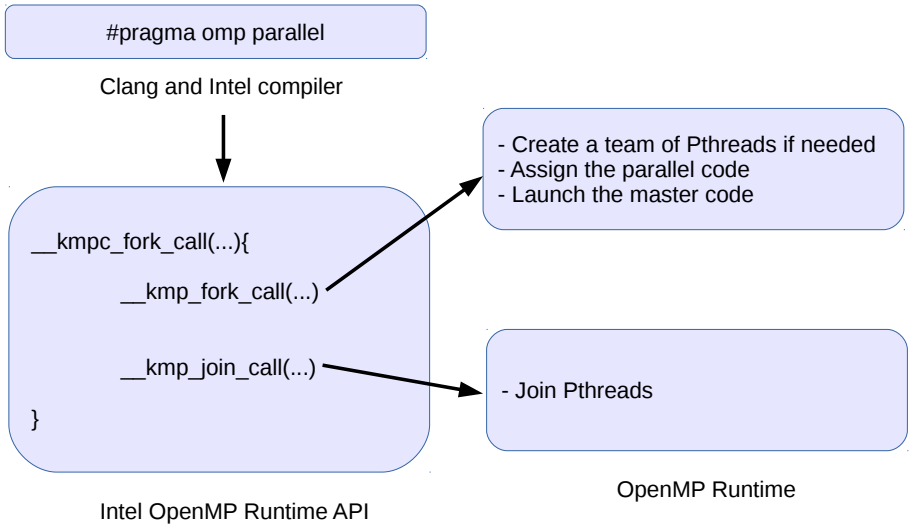
GLT_ults act as Pthreads do inside the POSIX-based OpenMP solutions when work-sharing constructs are invoked. The left-hand side of Figure 5.3 shows that each OpenMP Thread is transformed into a GLT_ult in that scenario.

When exploiting task-parallelism (right-hand side of Figure 5.3), each OpenMP task is also transformed into a GLT_ult. However, due to the different data structures used by the OpenMP runtime for OpenMP thread and OpenMP task, inside the GLTO implementation the behavior of the GLT_ult differs when acting as an OpenMP thread or an OpenMP task.

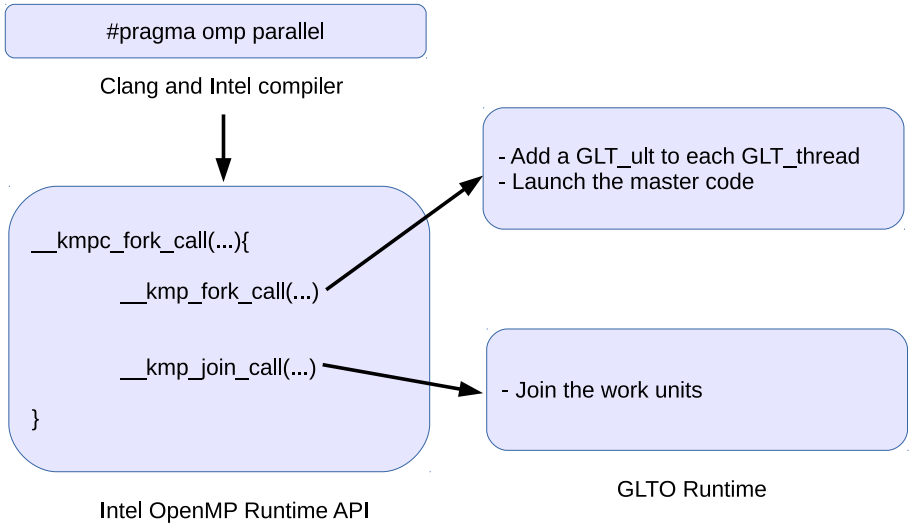
In the next subsections we discuss in more detail the operation modes of GLTO in each scenario.

5.1.3 GLTO Work-sharing Construct

For work-sharing constructs (e.g. `#pragma omp parallel`), our OpenMP solution mimics the mechanism that the GNU and Intel runtimes feature. The master thread assigns the function pointer of the parallel code 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.



(a) Pthread-based.



(b) GLT-based.

Figure 5.2: Internal mechanism for mapping a `#pragma omp parallel` directive with both solutions.

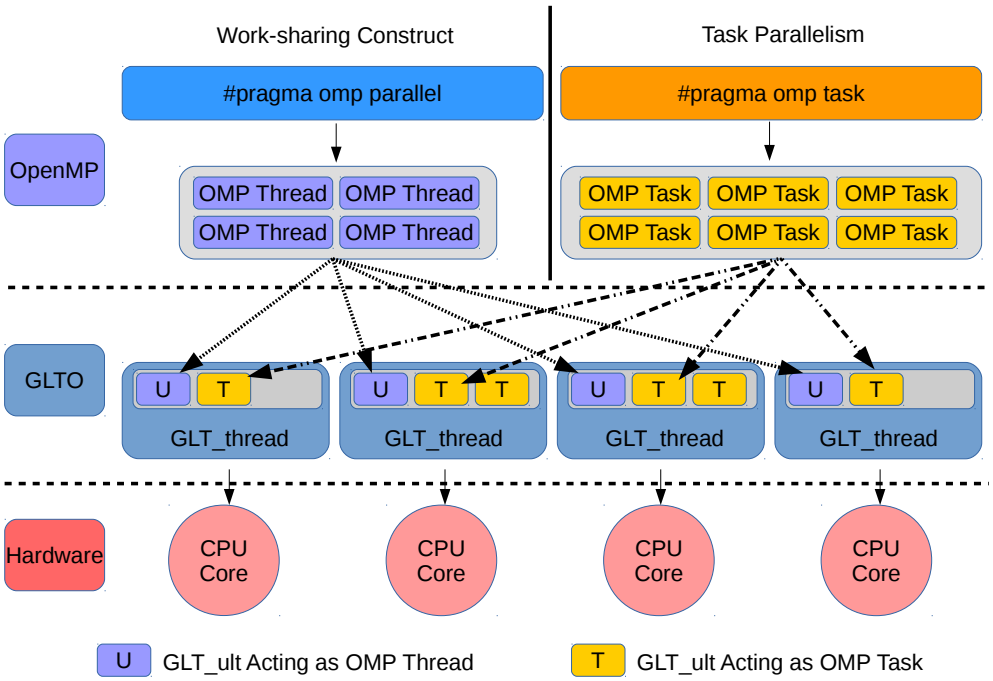


Figure 5.3: Relationship between OpenMP code and the GLTO implementation.

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.

5.1.4 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.

```
1  #pragma omp parallel
2  {
3    #pragma omp master
4    {
5      for(int i = 0; i < N; i++)
6      {
7        #pragma omp task
8        {
9          code(i);
10       }
11     }
12   }
13 }
```

Listing 5.1: OpenMP task parallelism inside a master region.

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

Listing 5.2: OpenMP task parallelism inside a parallel region.

As demonstrated later in our experimentation (Section 5.3), it is in these scenarios where LWTs may 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 (Listing 5.1), 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 (Listing 5.2), each GLT_thread creates its own tasks and executes them.

5.1.5 GLTO Nested Parallelism

Although nested parallel codes are rare, 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 (e.g. MKL from Intel). That code features nested parallelism and current Pthread-based OpenMP solutions tend to offer low performance. These solutions create teams of threads for both outer and inner parallel levels. This approach may oversubscribe the node if more threads than CPUs are spawned.

For tackling the oversubscription scenario, 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_ults for the nested code. This mechanism avoids the oversubscription that impairs performance when the Pthread-based OpenMP solutions are used because the number of GLT_threads remains equal to the number of CPUs.

5.1.6 GLTO Specific Implementation Issues

Although GLT offers a unified 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 may 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.

	GNU	Intel	GLTO
OpenMP Constructs	62	62	62
Used Tests	123	123	123
Successful Tests	118	118	121/122
Failed Tests	5	5	2/1

Table 5.1: Results of the OpenUH OpenMP Validation Suite 3.1 for the OpenMP runtimes.

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 difference cannot be fairly measured.

5.2 GLTO Functionality Validation

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

The OpenUH OpenMP Validation Suite 3.1 is leveraged to test the OpenMP 3.1 specification. It consists of 123 benchmark tests that analyze 62 OpenMP constructs, including task parallelism. The suite employs an automatic approach to run different types of tests in *normal*, *cross*, and *orphan* modes [68]. In addition to show the percentage of the OpenMP specification supported, it may be used to detect and fix bugs inside the runtime code due to the structure of the tests.

Table 5.1 displays the results of executing the Validation Suite with the GNU, Intel, and GLTO runtimes. `gcc 6.1` was used for the GNU runtime and `icc 17.0.1` for Intel and GLTO. GLTO is combined with the `icc` compiler because it can accommodate a larger number of tests than the `clang` tool does. Those results expose that, while the Intel and GNU runtimes pass 118 tests, our OpenMP implementation succeeds in 121 or 122, depending respectively on the use of Argobots/Qthreads or MassiveThreads as the underlying library. The failed tests for GLTO over Argobots/Qthreads are the `omp_taskyield` and the `omp_task_untied`. The reason is that, once a task is bound to a GLT_thread, there is no work stealing, so the task is resumed in the same GLT_thread and the test counts the number of tasks that have been created and started by one GLT_thread and resumed by another GLT_thread. Although the GLTO mechanism is accepted by the specification, the test appears as failed. If we use MassiveThreads, which allows work stealing, only `omp_taskyield` fails, because there are not enough tasks that change from one GLT_thread to another. The same tests fail for the GNU and the Intel runtimes, which indicates that they do not integrate any mechanism for migrating tasks from one OMP thread to another once the tasks have been created. With these runtimes, the tests fail in the *normal* and *orphan* modes. The other failed test is the `omp_task_final`, because the task marked as *final* is not directly executed as the specification indicates.

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

5.3 GLTO Performance Evaluation

In this section we evaluate OpenMP codes with both Pthreads- and GLT-based solutions. First, we perform the comparison with a compute-bound application, then with a nested parallel microbenchmark, and finally using a task-parallel application.

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, and LWT libraries were compiled with `gcc 6.1`. The Intel OpenMP implementation and GLTO were compiled with `icc 16.0`.

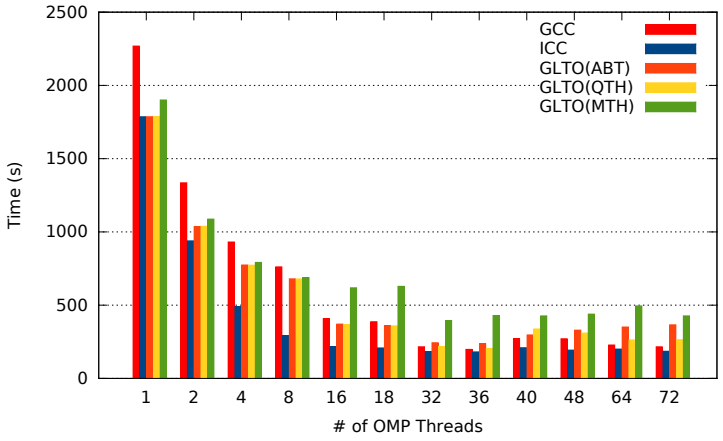
The OpenMP environment variables were set to the values that reported higher performance for each scenario. `OMP_NESTED` and `OMP_BIND_PROC` were set to true for all tests. The former was asserted in order to measure the actual nested management, because otherwise the OpenMP runtime treats nested parallelism as one level of parallelism and sequential code. The boundary variable 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 OpenMP 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.

5.3.1 OpenMP in a Compute-Bound Code

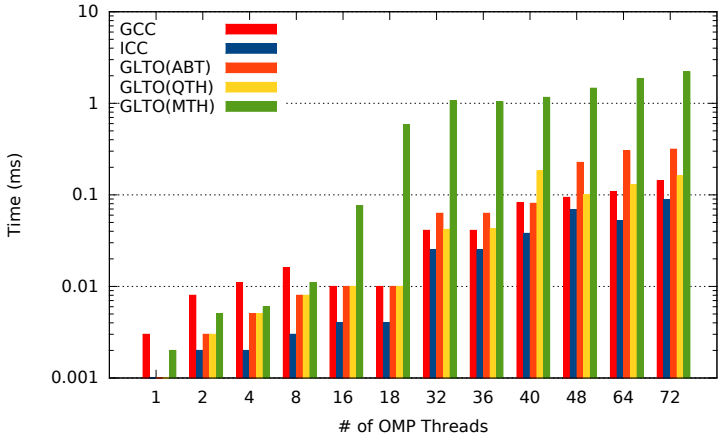
Our first case study reflects the currently most frequent target for OpenMP. It mainly consists of an iterative code that is executed a certain number of times. This code configuration is highly favorable for OpenMP, and often allows the runtimes to exploit a substantial fraction of the hardware parallelism. To study this scenario, we have chosen the CloverLeaf mini-app [4], 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 so that we can also demonstrate the integration of the GLTO implementation with this programming language.

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 5.4a depicts the average of 50 executions of the application for each of the OpenMP solutions using the clover_bm4.in problem instance. 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 Pthread-based OpenMP and LWT-based runtimes relies on the creation of GLT_ults. 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_ults as GLT_threads.

In order to analyze this time gap we have measured the time spent in the work assignment step inside the OpenMP runtime with a microbenchmark that measures the time spent distributing and joining the work. Figure 5.4b shows the difference among OpenMP implementations, demonstrating that the non-LWT solutions deploy the most efficient mechanism. Moreover, Figure 5.4b depicts a behavior variation when more than 18 threads are used. This is because the master thread assigns work to threads that are placed in a different NUMA node. Although the single-iteration



(a) CloverLeaf mini-app.



(b) Work assignment mechanism.

Figure 5.4: (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.

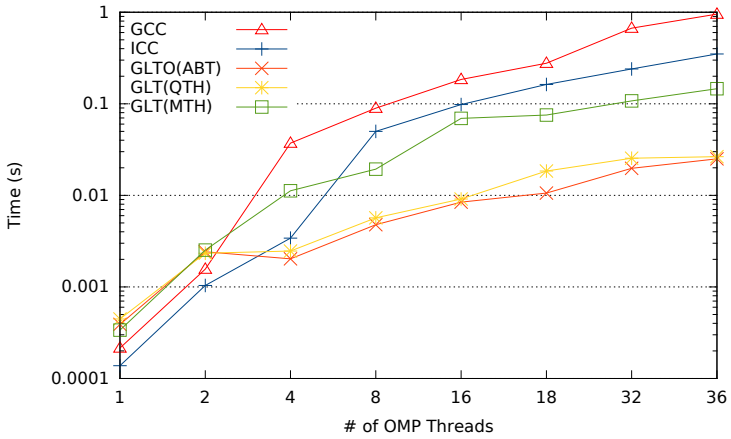
time difference among implementations is barely noticeable, repeating this operation over 336,000 times of the entire CloverLeaf execution yields a nonnegligible total time difference that is shown in Figure 5.4a between Pthreads- and GLT-based OpenMP implementations.

5.3.2 OpenMP with Nested Parallelism

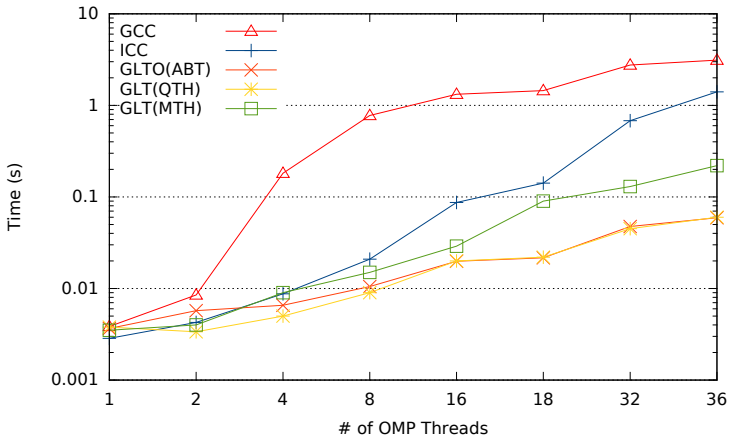
Nested parallelism is not a common OpenMP pattern, but it may appear hidden to the user. Moreover, an increasing number of cores may allow programmers to introduce several levels of parallelism in order to extract all the computational power of future hardware.

Due to the suboptimal design of the nested parallelism mechanism in current OpenMP implementations, it has been impossible 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.



(a) 100 iterations in the outer loop.



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

Figure 5.5: Execution time for the nested parallel code on top of OpenMP runtimes increasing the number of OpenMP threads.

Figure 5.5a reveals the performance difference among the OpenMP implementations when the outer and inner loop comprise 100 iterations, and Figure 5.5b 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 and Qthreads. The performance of GLTO over MassiveThreads is affected by the design issue discussed in Section 5.1.6. In this case, the action of the master thread has a strong negative impact in the overall execution time because it needs to yield in order to execute the inner loop code. Since GLTO over MassiveThreads does not allow this, the work of the master thread needs to be stolen by the remaining threads.

The detected 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 inner loop thread. On

the other hand, the Intel implementation mimics GNU for the outer loop, but the Intel solution reuses the idle threads for the inner loop. 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 notable performance improvement against the Pthread solutions.

5.3.3 OpenMP in Task Parallelism

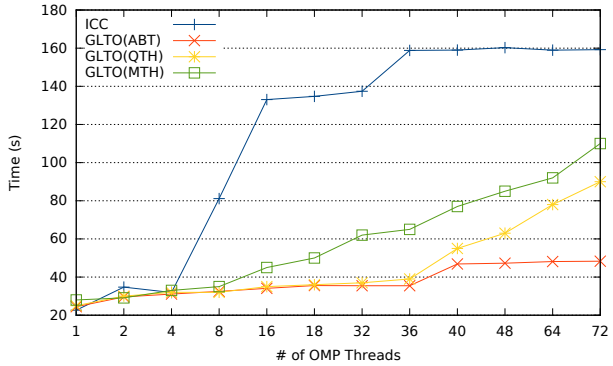
To study the performance in this scenario, we employ the Conjugate Gradient (CG) benchmark. The CG method is an algorithm for the numerical solution of symmetric positive definite systems of linear equations. We have converted the OpenMP `#pragma omp parallel for` directives in the original implementation of CG [25] 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 the 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 the original CG implementation uses the Intel Math Kernel Library [49] and, therefore, since the measured time is the total execution, the comparison between this library and other GNU-available solutions would not be fair.

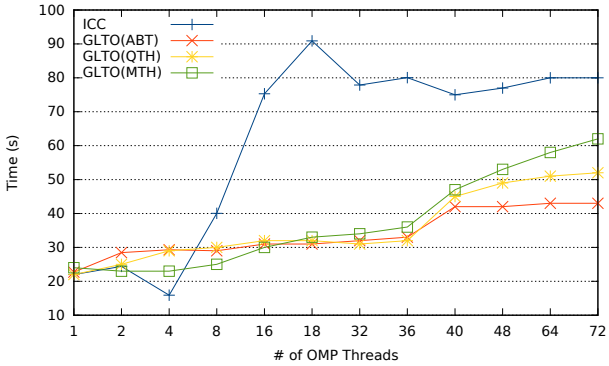
Figure 5.6 displays the results for granularities of 10, 20, 50, and 100 rows per task. The labels ICC, GLTO(ABT), GLTO(QTH), and GLTO(MTH) refer to Intel OpenMP, and GLT on top of Argobots, Qthreads, and MassiveThreads, respectively. Those results reflect the average time of 1,000 executions. Since a smaller number of tasks implies less runtime overhead, 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 5.6a and 5.6b, respectively). For this benchmark, only GLTO on top of Argobots maintains an acceptable performance for a granularity of 50 (Figure 5.6c). These behaviors are depicted in next paragraphs.

In contrast with the previous scenarios, the Intel OpenMP runtime outperforms the GLTO implementations for the coarse-grained problem (Figure 5.6d). 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.

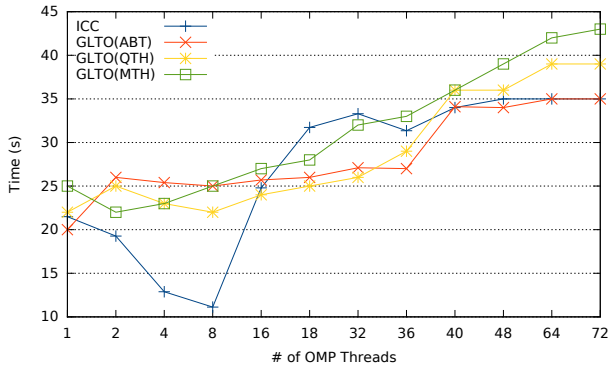
If we compare the GLTO options among them, we observe the effect of different implementation details of the underlying libraries. On the one hand, GLTO(ABT) exhibits almost flat performance lines for the 4 scenarios, which means that the interaction among the GLT_threads is almost non-existent. On the 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 among GLT_threads and the latter because of the mutex-protected access to each word in memory.



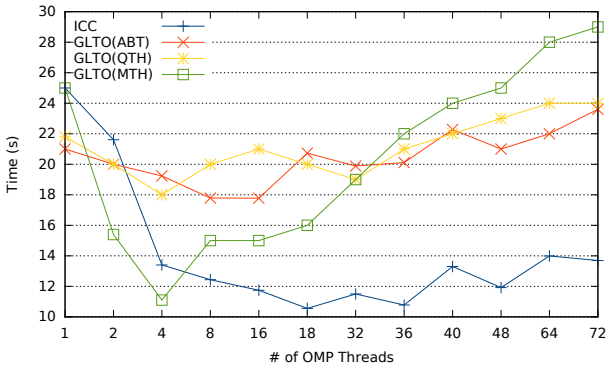
(a) 10 rows per task (1,488 tasks).



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



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



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

Figure 5.6: Execution time of CG with different task granularities on top of OpenMP runtimes increasing the number of OpenMP threads.

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 5.6a) and up to 8 for granularities of 20 (Figure 5.6b) and 50 (Figure 5.6c) rows per task. Once this 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.

We have analyzed those issues in detail by measuring both the number of queued tasks and the cut-off mechanism separately. The contention affects this test because the producer thread creates the tasks into its own task queue. At the same time, the consumer threads that are idle try to gain access to that queue, steal a task and execute it. Therefore, if we increment the number of threads that execute tasks, the number attempts of stealing increases.

Table 5.2 summarizes the percentage of the number of queued tasks for each granularity size. It is relevant to note that a reduced number of non-queued tasks benefits the overall performance. Our results suggest that the OpenMP task management needs additional development effort.

The cut-off mechanism is triggered once a certain number of tasks is queued—256 in the case of the Intel OpenMP runtime—and then the new tasks are executed directly as a sequential code. This mechanism prevents from queuing an elevate number of tasks, hence avoiding performance drops because of task management cost. This cut-off value cannot be modified without recompiling the OpenMP code. To show the impact in performance, we have modified this value to 4,096 and

Task Granularity	# OpenMP 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	93	100	100	100	100
100	100	100	100	100	100	100	100	100

Table 5.2: Percentage of queued tasks for each task granularity configuration.

16. Figure 5.7 shows the execution time of 4,000 fine-grained tasks with different cut-off values. With this experiment, we can show how this value may affect the performance. In the case of the cut-off value 4,096, all tasks are queued (because this number is not reached) and this configuration obtains the worst performance. 256 and 16 cut-off values maintain an acceptable performance up to 8 and 16 threads, respectively. Once that number of threads is used, the consumers avoid the cut-off mechanism and the performance decreases, because 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. The change in the trend of the lines when more than 40 OpenMP threads are employed corresponds to the overhead caused by the work-stealing contention.

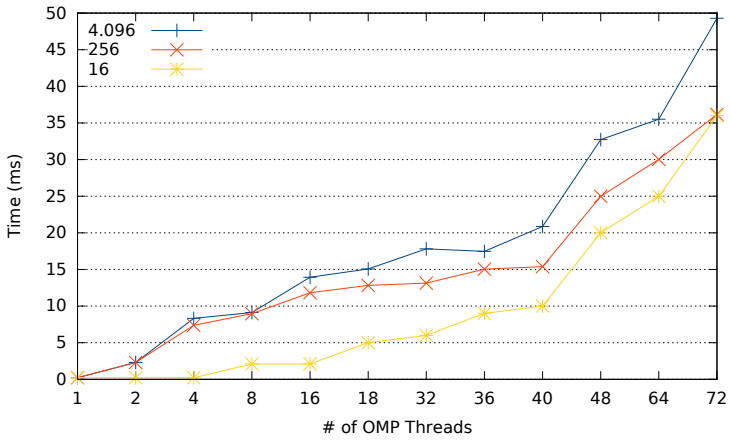


Figure 5.7: Different values for the Intel OpenMP cut-off mechanism.

In summary, our results indicate that, compared with LWT-based solutions, Pthreads-based implementations cannot deal successfully with the fine-grained parallel paradigm. In that case, a LWT-based approach should be selected to favor performance.

5.4 OmpSs over GLT (GOmpSs)

In this section we justify the design decisions that we made in order to adapt the OmpSs runtime to the use of LWTs, in order to provide another high-level PM use case over our GLT propossal.

5.4.1 GOmpSs Interactions

GOmpSs offers a complete implementation of OmpSs version 16.06.3. OmpSs allows to select the underlying LWT implementation by means of an environment variable thanks to its modular implementation (see Figure 5.8). We have leverage this feature in order to allow that the user selects GLT as an option. With this work, OmpSs applications can run on top of Argobots, Qthreads, or MassiveThreads (under GLT) in addition to the native Nanos++ solution. Therefore, once an OmpSs application has been built with the `mercurium` compiler [28], the underlying threading library can be selected by means of environment variables.

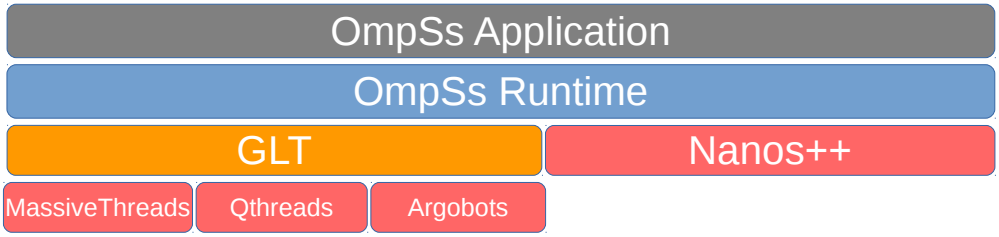


Figure 5.8: Software stack choices of an OmpSs code.

5.4.2 GOmpSs Implementation Details

As in the GLTO implementation, GLT_threads are bound to CPU cores and 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 may be modified via the `GLT_NUM_THREADS` environment variable or the `--smp-workers` runtime argument corresponding to the GLT or Nanos++ implementations, respectively.

5.4.3 GOmpSs Task Parallelism

As introduced in Section 2.4.2, OmpSs is a task-oriented PM and it is not designed for work-sharing constructs, although these are supported. For that reason, our study of both the OmpSs and GOmpSs runtimes is focused on the anotation directives related to tasks for creation (`#pragma omp task`, `#pragma omp taskloop`) and synchronization (`#pragma omp taskwait`).

A task is generated from a `#pragma omp task` directive in the OmpSs runtime. This directive is translated into an OmpSs call that creates a pending task. The runtime evaluates the task dependencies (if any), and once these are accomplished, it promotes the OmpSs task to the "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.

The same algorithm is followed by GOmpSs; however, the generated task is a GLT_ult. Figure 5.10 shows the relationship between an OmpSs task and a GLT_ult. 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 by 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 no need for a dispatch policy or a certain scheduling. In that scenario, the use of a shared queue among the GLT_threads helps with the load balance.

In contrast 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

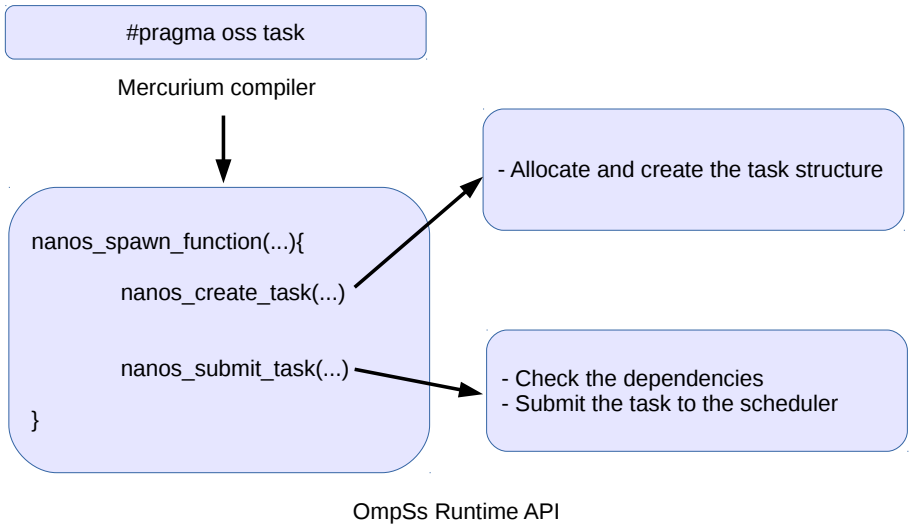


Figure 5.9: Internal mechanism for a `#pragma oss/omp task` directive.

also considered an OmpSs task. Therefore, it may be resumed by any of the GLT_threads once a synchronization point is reached.

5.5 GOmpSs Performance Evaluation

The default environment values of OmpSs have been maintained and the performance-oriented OmpSs library is employed. The hardware and software employed is that introduced in Section 5.3. OmpSs and GOmpSs were compiled with `gcc 6.1`.

5.5.1 GOmpSs 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 if our implementation adds any overhead in this procedure. We implemented a microbenchmark that creates a certain number of tasks and then joins them. Figures 5.11a and 5.11b show the average execution time of 100 executions of creating and joining 1,000 and 10,000 empty tasks without dependencies. 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 OmpSs implementation over Argobots, Qthreads, and MassiveThreads, respectively.

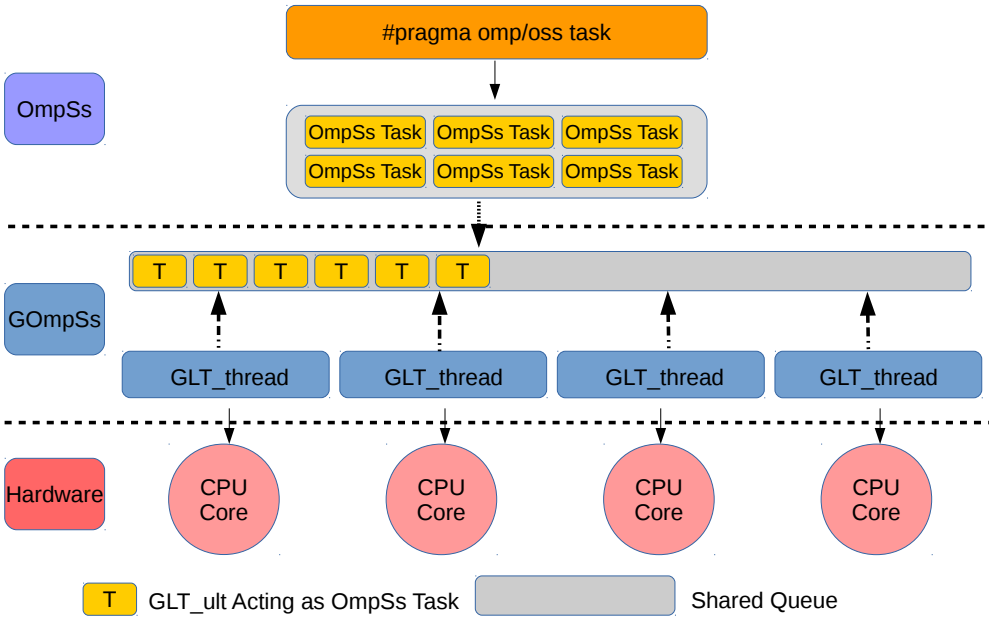


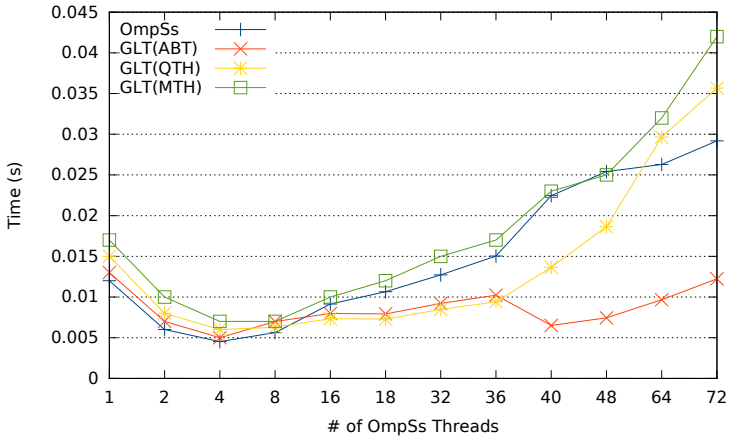
Figure 5.10: Relationship between OmpSs code and the GOmspSs implementation.

The obtained 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 these 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 lowest performance because the internal work-stealing requires additional 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 (Section 5.3.1). GLT(ABT) is the most efficient 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.

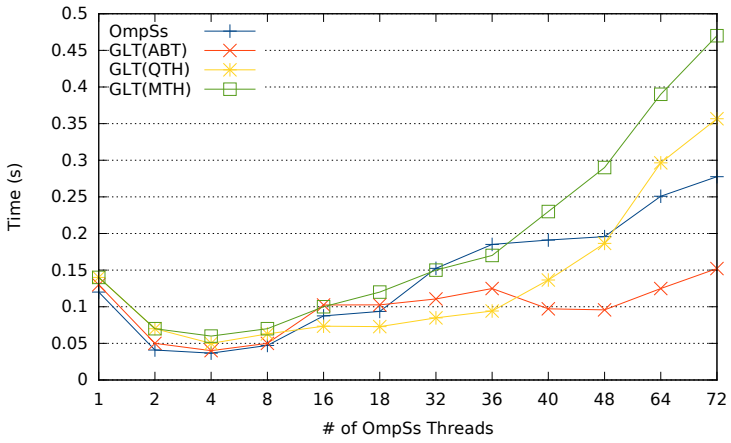
We also evaluated GOmpSs with a production application. We selected the SparseLU Decomposition application from [27]. This application performs an LU decomposition over a square sparse matrix that is allocated by blocks of contiguous memory. We used two different matrix sizes: the default size of 3,200x3,200 elements (Figure 5.12a), and 12,800x12,800 elements (Figure 5.12b), in both cases using double precision. These execution of these problems spawns 1,500 and 89,000 tasks, respectively.

Figures 5.12a and 5.12b show the average of 100 executions for the SparseLU Decomposition and reveal that the time gap among all the OmpSs implementations is almost negligible.

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



(a) 1,000 OmpSs Tasks.



(b) 10,000 OmpSs Tasks.

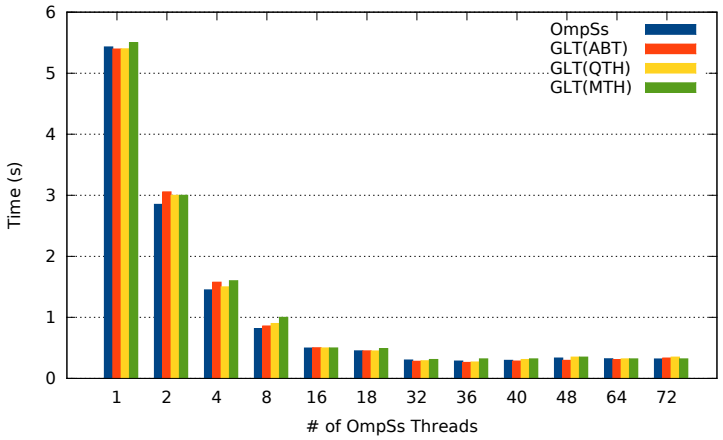
Figure 5.11: Execution time for creating and joining OmpSs tasks on top of OmpSs runtimes increasing the number of OmpSs threads.

5.6 Summary

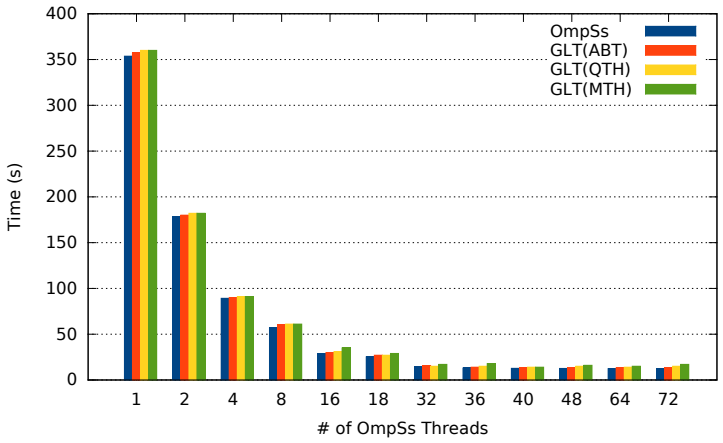
In this chapter we have presented two directive-based PMs, OpenMP and OmpSs, implemented on top of the GLT API, named GLTO [7] and GOmpSs, respectively. These runtimes allow us to execute codes written in OpenMP and OmpSs on top of different underlying LWT solutions without modifying the code.

We discussed the design decisions taken during the implementation of both runtimes, and we showed how these 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, our 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



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



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

Figure 5.12: Execution time for the SparseLU application on top of OmpSs runtimes increasing the number of OmpSs threads.

parallelism scenario with OpenMP, LWTs attain better performance than Pthreads do with fine-grained tasks.

In the case of task parallelism using OmpSs, our implementation performs close to the original runtime (implemented via 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, attaining the highest performance when Argobots is used as the underlying library.

Our results suggest that in general LWTs are highly appropriate to leverage fine-grained tasks, which may be well described by employing high-level PMs. We have also demonstrated that the semantics exposed by GLT are appropiate to act as a threading interface for high-level PMs.

CHAPTER 6

Conclusions

6.1 Conclusions and Main Contributions

The main goal of this dissertation was *to study, design, develop and analyze a unified API that joins, under a unique set of semantics, the characteristic features of current LWT libraries.*

At the conclusion of this work, the main contributions of this dissertation are the following:

- The analysis of several threading solutions from a semantic point of view, identifying the strong points of each threading solution.
- The design and implementation of a unified LWT API, named Generic Lightweight Threads, that groups the required LWT functionality for HPC under the same PM.
- The evaluation of the overhead introduced by the GLT API using several microbenchmarks and applications.
- The design and implementation of the OpenMP and OmpSs runtimes on top of the GLT API, called Generic Lightweight Threads OpenMP (GLTO) and Generic Lightweight Threads OmpSs (GOmpSs), respectively.

The main contribution of this dissertation is the proposal of a unified API for LWT solutions, that is aimed to be the first step towards the standardization of this type of threading solutions following the examples of OpenMP and MPI. This unified API has been the basis for the remaining parts of this dissertation.

As a part of this thesis, we first analyzed several existing LWT solutions in order to extract common functional patterns as well as to evaluate the usability of these solutions for HPC codes.

An additional contribution of this thesis is the development of high-level PMs on top of the unified API that fulfill two necessities: to demonstrate the usability of GLT and LWT solutions in widely-accepted PMs, and to remove the portability issue by easing the use of LWT solutions to experimented high-level PM developers.

The following subsections discuss the contributions and summarize the corresponding conclusions in more detail.

6.1.1 Threading Libraries

We have presented a deep analysis of a set of threading solutions including both OS threads and LWTs. Furthermore, we have performed a PM decomposition of the threading libraries analyzing their features. Moreover, we have introduced the GLT unified API, the first effort in the path to a LWT standarization.

We have experimentally proved that the use of LWT approaches for fine-grained parallel codes is beneficial, because these libraries can deal with common parallel code patterns that are commonly accelerated with OpenMP pragmas, offering a performance level that is, at least, as good as that reached with Pthreads and the OpenMP runtimes implemented by GNU and Intel. 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.

As a general conclusion LWTs improve performance in scenarios that are becoming more popular such as task parallelism or nested parallel structures. These scenarios are aimed to tackle the problem of leveraging the computational power of exascale systems.

6.1.2 GLT API

This library proposes a unified API for LWT solutions that is the first attempt to standardize those PMs. We have implemented GLT on top of the major general-purpose LWT solutions for HPC: Argobots, MassiveThreads, and Qthreads.

We have discussed the GLT PM and presented the API modules. Furthermore, we have presented an example of the semantic mapping between the GLT API with the LWT solutions. Using two microbenchmarks we have also justified the need for a unified LWT API from the point of view of portability.

Our performance evaluation, based on stand-alone and header-only implementations of the GLT API, demonstrates the low performance overhead of this approach. We have demonstrated this overhead with a set of microbenchmarks that measure the instructions per call added by GLT. Moreover, we have assessed the overhead by comparing the execution time of two applications where the stand-alone implementation produced an average overhead under 0.6%, while the header-only version featured an average overhead below 0.1%.

In conclusion, we have demonstrated the portability benefits that a unified API for LWT libraries can offer to programmers translating their applications from OpenMP and Pthreads to the GLT API.

6.1.3 High-level Programming Models

We have designed, developed and analyzed two directive-based PMs, OpenMP and OmpSs, implemented on top of the GLT API, named GLTO and GOmpSs, respectively.

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 scenarios and LWTs for nested parallelism.

In the task-parallel scenario with OpenMP, LWTs attain higher performance than Pthreads with fine-grained tasks.

In the case of task parallelism using OmpSs, our implementation performs close to the original runtime (implemented via an ad-hoc solution) when executing applications and improves the time spent in fine-grained task management when more than 18 threads are used, attaining the best performance when Argobots is used as the underlying library. These results reinforce our findings within the OpenMP PM. From these experiments we conclude that, in general, LWTs are highly appropiate to leverage fine-grained tasks, which may be well described by employing high-level PMs.

6.2 Related Publications

The contributions of this dissertation are supported by publications in different peer-reviewed national and international conferences and journals. In this section, the publications related to each contribution are listed and classified as directly-related to the content of the dissertation, indirectly-related or unrelated.

6.2.1 Directly Related Publications

6.2.1.1 Chapter 2. Background

The first step in designing a unified API was to review the existing threading solutions in order to extract common features from their PMs [33]. In this paper, threading libraries are decomposed and each PM is deeply analyzed.

Castelló, A., Mayo, R., Seo, S., Balaji, P., Quintana-Ortí, E. S., Peña, A. J. analysis of lightweight thread libraries for high-performance computing. *Submitted to IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2018). — Journal [33]

> In this paper, we analyze in detail the most representative threading libraries including Pthread- and LWT-based solutions. In addition, to examine the suitability of LWTs for different use cases, we develop a set of microbenchmarks consisting of commonly found OpenMP patterns in current parallel codes, and we compare the results using threading libraries and OpenMP implementations. Moreover, we study the semantics offered by threading libraries in order to expose the similarities among different LWT application programming interfaces and their advantages over Pthreads. This study reveals that LWT libraries outperform solutions based on operating system threads in cases where tasks and nested parallelism are required.

6.2.1.2 Chapter 3. State of the Art

The work presented in [34] studies different LWT libraries from a semantical point of view and presents a performance evaluation with common OpenMP parallel code patterns such as nested and fine-grained task parallelism.

Conference Proceedings [34]

Castelló, A.,Peña, A. J., Seo, S. Mayo, R., Balaji, P., Quintana-Ortí, E. S. A review of lightweight thread approaches for high performance computing. *IEEE International Conference on Cluster Computing (CLUSTER)* (2016).

> 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 to each other and that they are as efficient as OS threads in some simple scenarios while outperforming them in many cases.

6.2.1.3 Chapter 4. Generic Lightweight Threads (GLT)

The design, implementation details, and functionality of the unified API is presented in [37]. The work presented in this paper offers a detailed description in the common API creation process. Moreover, a demonstration of the usability of a common API is shown. An overhead study using microbenchmarks and applications is performed to verify the possible overhead introduced by this additional software layer.

Conference Proceedings [37]

Castelló, A., Seo, S. Mayo, R., Balaji, P., Quintana-Ortí, E. S., Peña, A. J. GLT: A unified API for lightweight thread libraries. *IEEE International European Conference on Parallel and Distributed Computing (EURO-PAR)* (2017).

> In this paper we introduce the design of a unified LWT API, named GLT, that groups the functionality of popular LWT solutions for HPC under the same PM. GLT is presented as a proof of concept in order to spark a joint effort from the community to design a standard LWT API. We implement GLT on top of Argobots, MassiveThreads, and Qthreads. Using the GLT API, application programmers can develop a single code for different LWT approaches.
>
> Our experiments demonstrate the feasibility of a GLT implementation, which does not exert any perceivable negative performance impact on applications. In our experiments, the average performance overhead when using static and dynamic GLT approaches, instead of the original LWT libraries, is 0.08% and 0.6%, respectively.

6.2.1.4 Chapter 5. Lightweight Threads for High-Level Parallel Programming Models

This chapter aims to make more accessible the use of LWT libraries. With that goal, in [39], GLTO, an OpenMP implementation on top of the GLT API, is presented. In that work, the semantical match between OpenMP pragmas and GLT procedures is analyzed. A performance analysis comparing Intel and GNU OpenMP implementations with GLTO via different applications and microbenchmarks is conducted in this paper.

Conference Proceedings [39]

Castelló, A., Seo, S. Mayo, R., Balaji, P., Quintana-Ortí, E. S., Peña, A. J. GLTO: On the adequacy of lightweight thread approaches for OpenMP implementations. *Proceedings of the International Conference on Parallel Processing (ICPP)* (2017).

> In this paper we present GLTO: our design and implementation of an OpenMP runtime on top of the GLT API. We test our OpenMP implementation with the OpenUH OpenMP Validation Suite 3.1 [68]. Based on GLTO, we analyze the most common OpenMP patterns and discuss how LWTs deal with them, in comparison with traditional Pthread-based approaches. We evaluate our OpenMP implementation and compare its performance with those obtained when using the GNU and Intel OpenMP runtimes in four different scenarios: basic parallel code, *for* loop based code, nested parallelism, and task parallelism. Our study reveals that none of the solutions obtains the best performance in all the scenarios, but that there are important gaps among them.

In order to expand the possibility of the use of LWTs, we have also implemented OmpSs on top of the GLT API, namely GOmpSs [32]. Although OmpSs is currently implemented on top of Nanos++, a LWT library, with this effort, LWT solutions are able to execute OmpSs code. Therefore, we have extended the work in [39] by adding OmpSs to the analysis.

Castelló, A., Mayo, R., Sala, K., Beltran, V., Balaji, P., Peña, A. J. On the adequacy of lightweight thread approaches for high-level parallel programming models. *Future Generation Computer Systems (FGCS)* (2018). Journal[32]

> High-level parallel 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 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 GLT API. GLT exposes a common API for 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 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.

6.2.2 Indirectly Related Publications

Although the thesis is focused on the use of a common API, we have been involved in the Argobots and BOLT development team. That work resulted in the following publications:

Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Carns, P., Castelló, A., Genet, D., Herault, T., Iwasaki, S., Jindal, P., Kale, S., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y., Taura, K., Beckman, P. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2017). Journal [63]

In addition, GLTO and GLT have been employed for incrementing nested parallelism performance in linear algebra solutions. This collaboration results in the following publication:

JOURNAL [40] CATALAN, S., CASTELLÓ, A., IGUAL, F.D., RODRÍGUEZ-SÁNCHEZ, R., QUINTANA-ORTÍ, E. S. Programming parallel dense matrix factorizations with look-ahead and OpenMP. *Submitted to Cluster Computing (CC)* (2018).

6.2.3 Other Publications

The publications listed in this section refer mainly to the collaboration in the development of the Remote CUDA (rCUDA) technology. This includes publications about distinct approaches in order to extend rCUDA with the use of high-level PMs.

The publications related to that parallel work are listed below:

CONFERENCE PROCEEDINGS [35] CASTELLÓ, A., PEÑA, A. J., MAYO, R., BALAJI, P., QUINTANA-ORTÍ, E. S. Exploring the suitability of remote GPGPU virtualization for the OpenACC programming model using rCUDA. *IEEE International Conference on Cluster Computing (CLUSTER)* (2015).

CONFERENCE PROCEEDINGS [31] CASTELLÓ, A., MAYO, R., PLANAS, J., QUINTANA-ORTÍ, E. S. Exploiting task-parallelism on GPU clusters via OmpSs and rCUDA virtualization. *International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (RePARA)* (2015).

JOURNAL [36] CASTELLÓ, A., PEÑA, A. J., MAYO, R., PLANAS, J., BALAJI, P., QUINTANA-ORTÍ, E. S. Exploring the interoperability of remote GPGPU virtualization using rCUDA and directive-based programming models. *The Journal of Supercomputing (JoS)* (2016).

CONFERENCE PROCEEDINGS [50] ISERTE, S., CLEMENTE-CASTELLÓ, F. J., CASTELLÓ, A., MAYO, R., QUINTANA-ORTÍ, E. S. Enabling GPU Virtualization in Cloud Environments. *International Conference on Cloud Computing and Services Science (CLOSER)* (2016).

6.3 Open Research Lines

The use of LWT solutions in HPC is still far from being mainstream. Thus, several research questions remain open after the conclusion of this thesis, some of which are detailed next:

- Creation of a LWT-aware MPI implementation that overlaps communication and computation code automatically.
- Development of a module that dynamically modifies the number of OS threads with the aim of reducing power consumption.
- Use of GLTO and GLT for malleable linear algebra libraries [40].
- Use of GLT as a convenient layer to benchmark and compare various User-Level Threading interfaces and the Fult scheduler [43].

CAPÍTULO 7

Conclusiones

7.1 Conclusiones y contribuciones principales

El objetivo principal de esta tesis era *estudiar, diseñar, desarrollar y analizar una interfaz común que uniera, bajo una misma semántica, las características de las bibliotecas de hilos ligeros actuales*. Tras la conclusión de este trabajo, las principales contribuciones de esta tesis son las siguientes:

- El análisis de las distintas bibliotecas de hilos desde un punto de vista semántico, identificando los puntos fuertes de cada solución.
- El diseño e implementación de una interfaz unificada, llamada Generic Lightweight Threads, que agrupa la funcionalidad necesaria de una biblioteca de hilos para CAP en un mismo modelo de programación.
- La evaluación del sobrecoste introducido por la interfaz GLT con distintos microbenchmarks y aplicaciones.
- El diseño e implementación de los modelos de programación OpenMP y OmpSs sobre la interfaz GLT, llamados Generic Lightweight Threads OpenMP (GLTO) y Generic Lightweight Threads OmpSs (GOmpSs), respectivamente.

La principal contribución de esta tesis es la propuesta de una interfaz común para bibliotecas de hilos ligeros que podría ser el primer paso hacia la estandarización de este tipo de bibliotecas siguiendo el ejemplo de OpenMP o MPI. Esta interfaz común ha sido la base para el resto de partes de la tesis.

Como parte del trabajo, primero hemos analizado las ya existentes bibliotecas de hilos ligeros con el objetivo de extraer funcionalidad común, así como evaluar la utilidad de estas soluciones para códigos de CAP.

Una contribución adicional de esta tesis es el desarrollo de modelos de programación de alto nivel soble la interfaz común y que corresponde a dos necesidades primarias: demostrar la usabilidad de GLT y soluciones de hilos ligeros en modelos de programación altamente aceptados y eliminar los posibles problemas de portabilidad facilitando el uso de los hilos ligeros a programadores experimentados en modelos de programación de alto nivel.

Las siguientes subsecciones ofrecen las contribuciones y resumen las conclusiones más detalladamente.

7.1.1 Bibliotecas de hilos

Se ha realizado un análisis profundo de un conjunto de soluciones basadas en hilos que incluyen hilos del sistema operativo e hilos ligeros. Además, se ha presentado una descomposición de los distintos modelos de programación destacando sus características. También se ha explicado la interfaz común GLT que es el primer paso hacia la estandarización de hilos ligeros.

Se ha comprobado, mediante experimentación, que el uso de bibliotecas de hilos ligeros para paralelismo de grano fino es viable, porque estas bibliotecas pueden lidiar con patrones de códigos paralelos que normalmente son acelerados con directivas OpenMP, ofreciendo un rendimiento que es, al menos, tan bueno como el ofrecido por las bibliotecas de Pthreads. Además, se han detectado algunas decisiones de implementación que afectan negativamente el rendimiento como el paralelismo anidado y el uso del robo de trabajo en el caso de Intel.

Las bibliotecas de hilos ligeros mejoran el rendimiento en escenarios que están siendo más populares como el paralelismo de tareas y las estructuras paralelas anidadas. Estos escenarios están destinados a abordar el problema de extraer todo el poder computacional de los sistemas exascala.

7.1.2 GLT

GLT propone una interfaz común para bibliotecas de hilos ligeros, siendo el primer acercamiento a la estandarización de estos modelos de programación. Además, hemos implementado GLT sobre tres bibliotecas de hilos ligeros de propósito general para CAP: Argobots, MassiveThreads y Qthreads.

Se ha expuesto el modelo de programación de la GLT y detallado los módulos que forman su interfaz. También se ha presentado un ejemplo de mapeado semántico entre la interfaz de GLT y las distintas bibliotecas existentes. Se ha justificado, desde el punto de vista de la portabilidad, la necesidad de una interfaz unificada utilizando dos microbenchmarks.

Nuestra evaluación de prestaciones, basada en las dos implementaciones de GLT, demuestra el despreciable sobrecoste que introduce esta interfaz. Se ha justificado con un conjunto de microbenchmarks que miden el número de instrucciones por llamada añadido por GLT. Además, comparado el tiempo de ejecución de dos aplicaciones, obteniendo sobrecostes por debajo del 0,6 % para la versión dinámica y no superiores al 0,1 % en la versión estática.

Como conclusión, se ha demostrado el beneficio que ofrece una interfaz unificada, traduciendo aplicaciones escritas en OpenMP y/o Pthreads a GLT.

7.1.3 Modelos de programación de alto nivel

Se han diseñado, desarrollado y analizado dos modelos de programación de alto nivel basados en directivas como son OpenMP y OmpSs, implementados con la interfaz unificada GLT, llamados GLTO y GOmpSs, respectivamente. Como GLT está actualmente implementada sobre Argobots, MassiveThreads y Qthreads, GLTO y GOmpSs permiten la ejecución de códigos escritos en OpenMP y OmpSs sobre estas bibliotecas de hilos ligeros sin modificar el código de la aplicación.

Se han discutido las decisiones tomadas durante el transcurso de la implementación de ambos modelos y se ha detallado cómo se comportan bajo distintos patrones de código paralelo. También se han comparado las actuales implementaciones de OpenMP (GNU e Intel) y de OmpSs con nuestras versiones en distintos escenarios paralelos: estructuras de trabajo compartido (bucle for y paralelismo anidado) y en paralelismo de tareas.

Para cada escenario, se ha mostrado la diferencia de rendimiento y se han analizado las razones para la disparidad de resultados. En el caso de estructuras de trabajo compartido, los resultados indican que ninguna de las implementaciones de OpenMP es la mejor opción porque cada una obtiene beneficios en distintos casos: Pthreads para sólo cómputo (bucle for) y bibliotecas de hilos ligeros para paralelismo anidado. En el caso de paralelismo de tareas con OpenMP, las bibliotecas de hilos ligeros obtienen un mejor rendimiento que Pthreads para tareas de grano fino.

En el uso de paralelismo de tareas con OmpSs, nuestra solución obtiene unos resultados cercanos a la implementación original (que utiliza una biblioteca de hilos ligeros llamada Nanos++) en tiempos de ejecución de la aplicación. Además, mejora el tiempo empleado en la gestión de las tareas cuando se utilizan más de 18 hilos, alcanzando el mejor rendimiento cuando Argobots es utilizada como biblioteca. Estos resultados refuerzan aquellos demostrados al utilizar tareas de OpenMP. En general, nuestra investigación corrobora que los hilos ligeros son altamente recomendados para lidiar con tareas de grano fino.

7.2 Publicaciones relacionadas

Las contribuciones de esta tesis están respaldadas por la publicación de su contenido en distintos congresos y revistas revisadas por pares tanto de carácter nacional como internacional. En esta sección se listan las publicaciones relacionadas con cada contribución y se clasifican como directamente relacionadas con el contenido de la tesis, indirectamente relacionadas y no relacionadas.

7.2.1 Publicaciones directamente relacionadas

7.2.1.1 Chapter 2. Background

El primer paso en el diseño de una interfaz común era analizar las soluciones existentes de hilos para poder extraer las características comunes de sus modelos de programación [33]. En este artículo las bibliotecas de hilos se analizan en profundidad.

Castelló, A., Mayo, R., Seo, S., Balaji, P., Quintana-Ortí, E. S., Peña, A. J. Analysis of lightweight thread libraries for high-performance computing. *Enviado a IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2018). — Revista [33]

7.2.1.2 Chapter 3. State of the Art

El trabajo presentado en [34] estudia distintas bibliotecas de hilos ligeros desde un punto de vista semántico y presenta una evaluación de prestaciones utilizando patrones comunes de código paralelo escrito en OpenMP, como por ejemplo paralelismo anidado y de tareas.

Castelló, A.,Peña, A. J., Seo, S. Mayo, R., Balaji, P., Quintana-Ortí, E. S. A review of lightweight thread approaches for high performance computing. *IEEE International Conference on Cluster Computing (CLUSTER)* (2016). — Actas Congreso [34]

7.2.1.3 Chapter 4. Generic Lightweight Threads (GLT)

El diseño, detalles de implementación y funcionalidad de la GLT se presenta en [37]. El trabajo expuesto en ese artículo ofrece una descripción detallada de la creación de la interfaz unificada. Además, se presenta la utilidad de la interfaz y se realiza un estudio del posible sobrecoste introducido por la misma utilizando microbenchmarks y aplicaciones.

Actas Congreso [37] Castelló, A., Seo, S. Mayo, R., Balaji, P., Quintana-Ortí, E. S., Peña, A. J. GLT: A unified API for lightweight thread libraries. *IEEE International European Conference on Parallel and Distributed Computing (EURO-PAR)* (2017).

7.2.1.4 Chapter 5. Lightweight Threads for High-Level Parallel Programming Models

Este capítulo afronta el esfuerzo de hacer más accesible el uso de las bibliotecas de hilos ligeros. Con ese objetivo, [39] presenta GLTO, una implementación de OpenMP sobre la interfaz GLT. En ese trabajo se explica el mapeado semántico entre las directivas de OpenMP y los mecanismos de GLT. Además, se ofrece un análisis de rendimiento comparando la implementaciones de OpenMP de Intel y GNU con GLTO utilizando distintas aplicaciones y microbenchmarks.

Actas Congreso [39] Castelló, A., Seo, S. Mayo, R., Balaji, P., Quintana-Ortí, E. S., Peña, A. J. GLTO: On the adequacy of lightweight thread approaches for OpenMP implementations. *Proceedings of the International Conference on Parallel Processing (ICPP)* (2017).

Con el propósito de expandir el uso de las bibliotecas de hilos ligeros se ha implementado también OmpSs sobre GLT, llamado GOmpSs [32]. A pesar de que OmpSs está actualmente implementado sobre Nanos++, una biblioteca de hilos ligeros personalizada, con este trabajo todas las bibliotecas que implementen la interfaz común podrán ejecutar código OmpSs. Por lo tanto, se ha extendido el trabajo presentado en [39] con el análisis de OmpSs.

Revista [32] Castelló, A., Mayo, R., Sala, K., Beltran, V., Balaji, P., Peña, A. J. On the adequacy of lightweight thread approaches for high-level parallel programming models. *Future Generation Computer Systems (FGCS)* (2018).

7.2.2 Publicaciones indirectamente relacionadas

A pesar de que la tesis se ha centrado en el uso de la interfaz común, también se ha trabajado en el desarrollo de Argobots y BOLT. El trabajo resultante de estas colaboraciones es el siguiente:

Revista [63] Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Carns, P., Castelló, A., Genet, D., Herault, T., Iwasaki, S., Jindal, P., Kale, S., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y., Taura, K., Beckman, P. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2017).

Además, GLTO y GLT se han utilizado para incrementar el rendimiento del paralelismo anidado en soluciones de álgebra lineal. Esta colaboración ha obtenido como resultado la siguiente publicación:

Revista [40] Catalan, S., Castelló, A., Igual, F.D., Rodríguez-Sánchez, R., Quintana-Ortí, E. S. Programming parallel dense matrix factorizations with look-ahead and OpenMP. *Enviado a Cluster Computing (CC)* (2018).

7.2.3 Otras publicaciones

Las publicaciones listadas a continuación corresponden mayormente a la colaboración en el desarrollo de la tecnología rCUDA. Estas publicaciones tratan distintas mejoras para extender rCUDA con el uso de modelos de programación de alto nivel.

Las publicaciones relacionadas con este trabajo se listan a continuación:

Castelló, A., Peña, A. J., Mayo, R., Balaji, P., Quintana-Ortí, E. S. Exploring the suitability of remote GPGPU virtualization for the OpenACC programming model using rCUDA. *IEEE International Conference on Cluster Computing (CLUSTER)* (2015), pp. 92–95. Actas Congreso [35]

Castelló, A., Mayo, R., Planas, J., Quintana-Ortí, E. S. Exploiting task-parallelism on GPU clusters via OmpSs and rCUDA virtualization. *International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (RePARA)* (2015), pp. 160–165. Actas Congreso [31]

Castelló, A., Peña, A. J., Mayo, R., Planas, J., Balaji, P., Quintana-Ortí, E. S. Exploring the interoperability of remote GPGPU virtualization using rCUDA and directive-based programming models. *The Journal of Supercomputing (JoS)* (2016). Revista [36]

Iserte, S., Clemente-Castelló, F. J., Castelló, A., Mayo, R., Quintana-Ortí, E. S. Enabling GPU Virtualization in Cloud Environments. *International Conference on Cloud Computing and Services Science (CLOSER)* (2016), pp. 249–256. Actas Congreso [50]

7.3 Líneas de investigación abiertas

El uso de las bibliotecas de hilos ligeros en la CAP está lejos de ser un estándar. Por lo tanto, algunos aspectos de la investigación permanecen abiertos tras la finalización de esta tesis. Algunas líneas abiertas de investigación se detallan a continuación:

- Implementación del modelo de programación MPI consciente del uso de bibliotecas de hilos ligeros para solapar computación y comunicación automáticamente.
- Desarrollo de un módulo que permita modificar el número de hilos del sistema operativo con el objetivo de reducir el coste energético.
- Aplicación de GLTO y GLT en bibliotecas de álgebra lineal maleables [40].
- Uso de GLT como banco de pruebas para comparar distintas bibliotecas de hilos ligeros y el planificador Fult [43].

APPENDIX A

Generic Lightweight Thread API

In this Appendix, the GLT user guide [38] and the complete GLT API is documented. All the functionality is ordered depending on the function module and each parameter is listed and explained.

Generic Lightweight Thread (GLT) Library

2.5

Generated by Doxygen 1.8.6

Mon Nov 13 2017 12:41:34

Contents

Chapter 1

README

GLT (Generic Lightweight Threads). Common API for Lightweight Thread Implementations.

- Developed by:
  - Adrian Castello (adcastel@uji.es) at Universitat Jaume I
- Supervised by:
  - Antonio J. Peña (antonio.pena@bsc.es) at Barcelona Supercomputing Center
  - Rafael Mayo Gual and Enrique S. Quintana-Ortí ({mayo,quintana}@uji.es)
  - Sangmin Seo and Pavan Balaji ({sseo,balaji}@anl.gov) at Argonne National Laboratory

GLT Release 2.5

GLT is a common API for HPC lightweight thread (LWT) libraries. It supports MassiveThreads, Qthreads, and Argobots as underlying LWT solutions. Moreover, GLT over Pthread is implemented with comparative purpose.

In addition, GLT can be used as POSIX threads API since version 2.5.

1. Getting Started
2. How to use GLT
3. How to cite GLT
4. Reporting Problems

1. Getting Started

The following instructions take you through a sequence of steps to get GLT installed and compiled.

(a.1) You will need the following prerequisites.

```
- REQUIRED: This tar file GLT-2.5.tar.gz
- REQUIRED: A C compiler (gcc is sufficient)
```

(a.2) At least one of these libraries:

```
- Argobots library.
- Qthreads library.
- MassiveThreads library.
```

(b) Unpack the tar file and go to the top level directory:

```
tar xzf GLT-2.5.tar.gz
cd GLT
```

If your tar doesn't accept the z option, use

```
gunzip GLT-2.5.tar.gz
tar xf GLT-2.5.tar
cd GLT
```

(c) Define environment variables:

```
The definition of the HOME_ARG, HOME_QTH, and HOME_MTH environment
variables with the path to Argobots, Qthreads, and MassiveThreads
libraries respectively is required.
```

(d) Build GLT:

```
cd src

for csh and tcsh:

  make [arg|qth|mth|pth] |& tee m.txt

for bash and sh:

  make [arg|qth|mth|pth] 2>&1 | tee m.txt
```

2. How to use GLT

I. GLT offers two library approaches:

(a) Dynamic library. Once the step 1 is completed, a libglt.so file can be found in each underlying library folder. The glt.h file needs to be included in the user's code and the -lglt flag added to the compilation order.

(d) Static library. In order to use this performance-oriented implementation fast_glt.h file may be included in the user's code and the -DFASTGLT flag added to the compilation order.

II. Using Pthreads API with GLT

GLT also offers the use of code written with pthreads just including "glt_pthreads.h" instead of "pthread.h"

3. How to cite GLT

To cite GLT in your work, please use the following for now: A. Castelló, A.J. Peña, S. Seo, R. Mayo, P. Balaji, E.S. Quintana-Ortí. GLT: A common API for lightweight thread libraries. www.hpca.uji.es/GLT. 2016

4. Reporting Problems

If you have problems with the installation or usage of GLT, please send an email to adcastel@uji.es.

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Module Documentation

3.1 Library functions

Functions

- void glt_start (void)

  Entry point for the GLT dynamic library.
- void glt_end (void)

  Ending point for the GLT dynamic library.
- void glt_init (int argc, char ∗argv[])

  GLT initialization function.
- void glt_finalize ()

  GLT finalization function.

3.1.1 Detailed Description

These functions start/stop and open/close the underlying GLT libraries. They are used in dynamic and static implementations.

3.1.2 Function Documentation

3.1.2.1 void glt_end (void)

Ending point for the GLT dynamic library.

`glt_end()` is the last called function when the GLT dynamic library is unloaded

3.1.2.2 void glt_finalize ( )

GLT finalization function.

`glt_finalize()` destroys the GLT environment. It is not mandatory and should be the last GLT function call.

3.1.2.3 void glt_init (int *argc*, char ∗ *argv[]*)

GLT initialization function.

`glt_init()` sets the GLT environment up. It is mandatory and needs to be the first GLT function call.

Parameters

in	*argc*	
in	*argv*	

3.1.2.4 void glt_start (void)

Entry point for the GLT dynamic library.

`glt_start()` is the first called function when the GLT dynamic library is loaded

3.2 Barrier functions

Functions

- void glt_barrier_create (int num_waiters, GLT_barrier *barrier)
  Creates a barrier.
- void glt_barrier_free (GLT_barrier *barrier)
  Destroys a barrier.
- void glt_barrier_wait (GLT_barrier *barrier)
  Waits into a barrier.

3.2.1 Detailed Description

These functions manage the GLT barriers for the ULTs.

3.2.2 Function Documentation

3.2.2.1 void glt_barrier_create (int *num_waiters*, GLT_barrier * *barrier*)

Creates a barrier.

`glt_barrier_create()` creates a barrier for ULTs.

Parameters

in	*num_waiters*	Indicates the number of ULTs requested to continue
in,out	*barrier*	Hande to newly created `GLT_barrier`

3.2.2.2 void glt_barrier_free (GLT_barrier * *barrier*)

Destroys a barrier.

`glt_barrier_free()` destroys a barier for ULTs.

Parameters

in	*barrier*	Handle to the target `GLT_barrier`.

3.2.2.3 void glt_barrier_wait (GLT_barrier * *barrier*)

Waits into a barrier.

`glt_barrier_wait()` Executed by a ULT, it waits until the number of waiters is achieved.

Parameters

in	*barrier*	Handle to the target `GLT_barrier`.

3.3 Condition functions

Functions

- void glt_cond_create (GLT_cond *cond)
  Creates a condition.
- void glt_cond_free (GLT_cond *cond)
  Destroys a condition.
- void glt_cond_signal (GLT_cond cond)
  Sends a signal for a condition.
- void glt_cond_wait (GLT_cond cond, GLT_mutex mutex)
  A ULT waits in this point for a signal.
- void glt_cond_broadcast (GLT_cond cond)
  Broadcast a signal for a condition.

3.3.1 Detailed Description

These functions manage the GLT conditions for the ULTs.

3.3.2 Function Documentation

3.3.2.1 void glt_cond_broadcast (GLT_cond *cond*)

Broadcast a signal for a condition.

`glt_cond_broadcast()` broadcasts a signal for ULTs.

Parameters

in	*cond*	Handle to the target `GLT_condition`.

3.3.2.2 void glt_cond_create (GLT_cond * *cond*)

Creates a condition.

`glt_cond_create()` creates a condition for ULTs.

Parameters

in,out	*cond*	Hande to newly created `GLT_condition`

3.3.2.3 void glt_cond_free (GLT_cond * *cond*)

Destroys a condition.

`glt_cond_free()` destroys a condition for ULTs.

Parameters

in	*cond*	Handle to the target `GLT_condition`.

3.3.2.4 void glt_cond_signal (GLT_cond *cond*)

Sends a signal for a condition.

`glt_cond_signal()` sends a signal for ULTs.

Parameters

in	*cond*	Handle to the target `GLT_condition`.

3.3.2.5 void glt_cond_wait (GLT_cond *cond*, GLT_mutex *mutex*)

A ULT waits in this point for a signal.

`glt_cond_wait()` a ULT waits at this point for a signal to access the mutex.

Parameters

in	*cond*	Handle to the target `GLT_condition`.
in	*mutex*	Handle to the target `GLT_mutex`.

3.4 Mutex functions

Functions

- void glt_mutex_create (GLT_mutex *mutex)
  Creates a mutex.
- void glt_mutex_lock (GLT_mutex mutex)
  Locks a mutex.
- void glt_mutex_unlock (GLT_mutex mutex)
  Unlocks a mutex.
- void glt_mutex_free (GLT_mutex *mutex)
  Destroys a mutex.
- int glt_mutex_trylock (GLT_mutex mutex)
  Tries to lock a mutex.
- void glt_mutex_spinlock (GLT_mutex mutex)
  Locks a mutex without contextswitch.

3.4.1 Detailed Description

These functions manage the GLT mutexes for the ULTs.

3.4.2 Function Documentation

3.4.2.1 void glt_mutex_create (GLT_mutex * *mutex*)

Creates a mutex.

`glt_mutex_create()` creates a mutex for ULTs.

Parameters

`in,out`	*mutex*	Hande to newly created `GLT_mutex`

3.4.2.2 void glt_mutex_free (GLT_mutex * *mutex*)

Destroys a mutex.

`glt_mutex_free()` destroys a mutex for ULTs.

Parameters

`in`	*mutex*	Handle to the target `GLT_mutex`.

3.4.2.3 void glt_mutex_lock (GLT_mutex *mutex*)

Locks a mutex.

`glt_mutex_lock()` locks (if possible) a mutex.

Parameters

in	*mutex*	Handle to the target GLT_mutex.

3.4.2.4 void glt_mutex_spinlock (GLT_mutex *mutex*)

Locks a mutex without contextswitch.

glt_mutex_spinlock() locks (if possible) a mutex.

Parameters

in	*mutex*	Handle to the target GLT_mutex.

3.4.2.5 int glt_mutex_trylock (GLT_mutex *mutex*)

Tries to lock a mutex.

glt_mutex_trylock() tries to lock a mutex.

Parameters

in	*mutex*	Handle to the target GLT_mutex.
out	*locked*	GLT_bool with the value 1 if the mutex has been locked or 0 if it was not possible.

3.4.2.6 void glt_mutex_unlock (GLT_mutex *mutex*)

Unlocks a mutex.

glt_mutex_unlock() unlocks a mutex.

Parameters

in	*mutex*	Handle to the target GLT_mutex.

3.5 Work-units functions

Functions

- GLT_ult * glt_ult_malloc (int number_of_ult)

  ULT allocation.
- GLT_tasklet * glt_tasklet_malloc (int number_of_tasklets)

  ULT allocation.
- void glt_ult_create (void(*thread_func)(void *), void *arg, GLT_ult *new_ult)

  ULT creation.
- void glt_ult_create_to (void(*thread_func)(void *), void *arg, GLT_ult *new_ult, int dest)

  ULT creation in a given destination.
- void glt_tasklet_create (void(*thread_func)(void *), void *arg, GLT_tasklet *new_ult)

  Tasklet creation.
- void glt_tasklet_create_to (void(*thread_func)(void *), void *arg, GLT_tasklet *new_ult, int dest)

  Tasklet creation.
- void glt_yield ()

  ULT yields to another ready ULT.
- void glt_yield_to (GLT_ult ult)

  ULT yields to an specific ULT.
- void glt_ult_join (GLT_ult *ult)

  Joins an specific ULT.
- void glt_tasklet_join (GLT_tasklet *tasklet)

  Joins an specific Tasklet.
- void glt_ult_free (GLT_ult *ult)

  ULT free memory.
- void glt_tasklet_free (GLT_tasklet *tasklet)

  Tasklet free memory.
- void glt_ult_get_id (GLT_ult_id *id, GLT_ult ult)

  Return the unique id of a ULT.
- void glt_workunit_get_thread_id (GLT_thread_id *id)

  Return the unique id of a thread.
- void glt_ult_migrate_self_to (GLT_thread_id id)

  Migrates the current ULT to another thread ready queue.
- void glt_ult_self (GLT_ult *ult)

  Obtains the current ULT handle.
- void glt_tasklet_self (GLT_tasklet *tasklet)

  Obtains the current Tasklet handle.
- void glt_ult_cancel (GLT_ult ult)

  Cancels an specific ULT.
- void glt_tasklet_cancel (GLT_tasklet tasklet)

  Cancels an specific Tasklet.
- void glt_ult_exit ()

  Exits the current ULT.

3.5.1 Detailed Description

These functions create, map, schedule, join, and execute work-units.

3.5.2 Function Documentation

3.5.2.1 void glt_tasklet_cancel (GLT_tasklet *tasklet*)

Cancels an specific Tasklet.

`glt_tasklet_cancel()` cancels a given `GLT_tasklet`.

Parameters

in	*tasklet*	Handle to the target `GLT_tasklet`.

3.5.2.2 void glt_tasklet_create (void(∗)(void ∗) *thread_func*, void ∗ *arg*, GLT_tasklet ∗ *new_ult*)

Tasklet creation.

`glt_tasklet_create()` creates a `GLT_tasklet`.

Parameters

in	*thread_func*	Is the function pointer to be executed by the `GLT_tasklet`.
in	*arg*	Are the arguments for thread_func.
out	*new_ult*	Handle to a newly created `GLT_tasklet`.

3.5.2.3 void glt_tasklet_create_to (void(∗)(void ∗) *thread_func*, void ∗ *arg*, GLT_tasklet ∗ *new_ult*, int *dest*)

Tasklet creation.

`glt_tasklet_create()` creates a `GLT_tasklet`.

Parameters

in	*thread_func*	Is the function pointer to be executed by the `GLT_tasklet`.
in	*arg*	Are the arguments for thread_func.
out	*new_ult*	Handle to a newly created `GLT_tasklet`.
in	*dest*	Number of the `GLT_thread` that should execute the newly created `GLT_tasklet`.

3.5.2.4 void glt_tasklet_free (GLT_tasklet ∗ *tasklet*)

Tasklet free memory.

`glt_tasklet_free()` frees the alloctaed memory.

Parameters

in	*tasklet*	Is the pointer to the allocated memory.

3.5.2.5 void glt_tasklet_join (GLT_tasklet ∗ *tasklet*)

Joins an specific Tasklet.

`glt_tasklet_join()` joins a given `GLT_tasklet`.

Parameters

in	*tasklet*	Handle to the target GLT_tasklet.

3.5.2.6 GLT_tasklet* glt_tasklet_malloc (int *number_of_tasklets*)

ULT allocation.

glt_tasklet_malloc() allocates memory for a given number of GLT_tasklet.

Parameters

in	*number_of_tasklets*	Indicates the total number of GLT_tasklets that needs to be allocated.

Returns

The pointer to the allocated memory.

3.5.2.7 void glt_tasklet_self (GLT_tasklet * *tasklet*)

Obtains the current Tasklet handle.

glt_tasklet_self() returns the current GLT_tasklet handler.

Parameters

out	*tasklet*	Handler of the the current GLT_tasklet.

3.5.2.8 void glt_ult_cancel (GLT_ult *ult*)

Cancels an specific ULT.

glt_ult_cancel() cancels a given GLT_ult.

Parameters

in	*ult*	Handle to the target GLT_ult.

3.5.2.9 void glt_ult_create (void(*)(void *) *thread_func*, void * *arg*, GLT_ult * *new_ult*)

ULT creation.

glt_ult_create() creates a GLT_ult.

Parameters

in	*thread_func*	Is the function pointer to be executed by the GLT_ult.
in	*arg*	Are the arguments for thread_func.
out	*new_ult*	Handle to a newly created GLT_ult.

3.5.2.10 void glt_ult_create_to (void(*)(void *) *thread_func*, void * *arg*, GLT_ult * *new_ult*, int *dest*)

ULT creation in a given destination.

glt_ult_create_to() creates a GLT_ult in a particular destination.

Parameters

in	*thread_func*	Is the function pointer to be executed by the GLT_ult.
in	*arg*	Are the arguments for thread_func.
out	*new_ult*	Handle to a newly created GLT_ult.
in	*dest*	Number of the GLT_thread that should execute the newly created GLT_ult.

3.5.2.11 void glt_ult_exit ( )

Exits the current ULT.

glt_ult_exit() cancels the current GLT_ult.

3.5.2.12 void glt_ult_free (GLT_ult * *ult*)

ULT free memory.

glt_ult_free() frees the alloctaed memory.

Parameters

in	*ult*	Is the pointer to the allocated memory.

3.5.2.13 void glt_ult_get_id (GLT_ult_id * *id*, GLT_ult *ult*)

Return the unique id of a ULT.

glt_ult_get_id() returns the id of a given GLT_ult.

Parameters

in	*ult*	Handle to the target GLT_ult.
out	*id*	Identifier if the the target GLT_ult.

3.5.2.14 void glt_ult_join (GLT_ult * *ult*)

Joins an specific ULT.

glt_ult_join() joins a given GLT_ult.

Parameters

in	*ult*	Handle to the target GLT_ult.

3.5.2.15 GLT_ult* glt_ult_malloc (int *number_of_ult*)

ULT allocation.

glt_ult_malloc() allocates memory for a given number of GLT_ult.

Parameters

in	*number_of_ult*	Indicates the total number of GLT_ult that needs to be allocated.

Returns

The pointer to the allocated memory.

3.5.2.16 void glt_ult_migrate_self_to (GLT_thread_id *id*)

Migrates the current ULT to another thread ready queue.

`glt_ult_migrate_self_to()` moves the current `GLT_ult` to another `GLT_thread` ready queue.

Parameters

in	*The*	identifier of the the `GLT_thread` destination.

3.5.2.17 void glt_ult_self (GLT_ult * *ult*)

Obtains the current ULT handle.

`glt_ult_self()` returns the current `GLT_ult` handler.

Parameters

out	*ult*	Handler of the the current `GLT_ult`.

3.5.2.18 void glt_workunit_get_thread_id (GLT_thread_id * *id*)

Return the unique id of a thread.

`glt_workunit_get_thread_id()` returns the id of the current `GLT_thread`.

Parameters

out	*id*	Identifier of the the current `GLT_thread`.

3.5.2.19 void glt_yield ( )

ULT yields to another ready ULT.

`glt_yield()` puts the current `GLT_ult` in the ready queue and allows another ready `GLT_ult` to be executed.

3.5.2.20 void glt_yield_to (GLT_ult *ult*)

ULT yields to an specific ULT.

`glt_yield_to()` puts the current `GLT_ult` in the ready queue and allows an specific ready `GLT_ult` to be executed.

Parameters

in	*ult*	Handle to the target `GLT_ult`.

3.6 Util functions

Functions

- double glt_get_wtime ()
  Returns the current time.
- void glt_timer_create (GLT_timer *timer)
  Creates a timer.
- void glt_timer_free (GLT_timer *timer)
  Destroys a timer.
- void glt_timer_start (GLT_timer timer)
  Inits a timer.
- void glt_timer_stop (GLT_timer timer)
  Stops a timer.
- void glt_timer_get_secs (GLT_timer timer, double *secs)
  Obtains the elapsed time.
- int glt_get_thread_num ()
  Obtains the number of the current thread.
- int glt_get_num_threads ()
  Returns the total number of threads.

3.6.1 Detailed Description

These functions return values from the environment set up and simplify the use of timers.

3.6.2 Function Documentation

3.6.2.1 int glt_get_num_threads ( )

Returns the total number of threads.

`glt_get_num_threads()` returns the number threads.

Returns

The number of c\ GLT_threads.

3.6.2.2 int glt_get_thread_num ( )

Obtains the number of the current thread.

`glt_get_thread_num()` returns the number of the current thread.

Returns

The number of the current c\ GLT_thread.

3.6.2.3 double glt_get_wtime ( )

Returns the current time.

`glt_get_wtime()` returns the time.

Returns

The time in seconds.

3.6.2.4 void glt_timer_create (GLT_timer * *timer*)

Creates a timer.

`glt_timer_create()` creates a timer.

Parameters

in,out	*timer*	Hande to newly created `GLT_timer`.

3.6.2.5 void glt_timer_free (GLT_timer * *timer*)

Destroys a timer.

`glt_timer_free()` destroys a timer.

Parameters

in	*timer*	Handle to the target `GLT_timer`.

3.6.2.6 void glt_timer_get_secs (GLT_timer *timer*, double * *secs*)

Obtains the elapsed time.

`glt_timer_get_secs()` given a timer. It calculates the elapsed time in seconds.

Parameters

in	*timer*	Handle to the target `GLT_timer`.
out	*secs*	Seconds.

3.6.2.7 void glt_timer_start (GLT_timer *timer*)

Inits a timer.

`glt_timer_start()` inits a timer.

Parameters

in	*timer*	Handle to the target `GLT_timer`.

3.6.2.8 void glt_timer_stop (GLT_timer *timer*)

Stops a timer.

`glt_timer_stop()` stops a timer.

Parameters

in	*timer*	Handle to the target `GLT_timer`.

3.7 Key functions

Functions

- void glt_key_create (void(∗destructor)(void ∗value), GLT_key ∗newkey)
  Creates a key.
- void glt_key_free (GLT_key ∗key)
  Destroys a key.
- void glt_key_set (GLT_key key, void ∗value)
  Sets new value to a key.
- void glt_key_get (GLT_key key, void ∗∗value)
  Gets value from a key.

3.7.1 Detailed Description

These functions manage the GLT keys for the ULTs.

3.7.2 Function Documentation

3.7.2.1 void glt_key_create (void(∗)(void ∗value) *destructor,* GLT_key ∗ *newkey*)

Creates a key.

`glt_key_create()` creates a key.

Parameters

in	*destructor*	Hande to newly created `GLT_ult`.
out	*newkey*	Hande to newly created `GLT_key`.

3.7.2.2 void glt_key_free (GLT_key ∗ *key*)

Destroys a key.

`glt_key_free()` destroys a key for ULTs.

Parameters

in	*key*	Handle to the target `GLT_key`.

3.7.2.3 void glt_key_get (GLT_key *key,* void ∗∗ *value*)

Gets value from a key.

`glt_key_get()` Gets value from a key.

Parameters

in	*key*	Handle of the target c\ GLT_key.
out	*value*	obtained value.

3.7.2.4 void glt_key_set (GLT_key *key,* void ∗ *value*)

Sets new value to a key.

`glt_key_set()` Sets value to a key.

Parameters

in	*key*	Handle of the target c\ GLT_key.
in	*value*	to be set.

3.8 query functions

Functions

- int glt_can_event_functions ()
  Obtains if the event module is available.
- int glt_can_future_functions ()
  Obtains if the event future is available.
- int glt_can_manage_scheduler ()
  Obtains if the scheduler module is available.

3.8.1 Detailed Description

These functions check the availability of advanced features.

3.8.2 Function Documentation

3.8.2.1 int glt_can_event_functions ( )

Obtains if the event module is available.

`glt_can_event_functions()` Gets the availability of the event module.

3.8.2.2 int glt_can_future_functions ( )

Obtains if the event future is available.

`glt_can_future_functions()` Gets the availability of the future module.

3.8.2.3 int glt_can_manage_scheduler ( )

Obtains if the scheduler module is available.

`glt_can_manage_scheduler_functions()` Gets the availability of the scheduler module.

3.9 Scheduler functions

Functions

- void glt_scheduler_config_free (GLT_sched_config *config)

  Destroys the scheduler configuration.
- void glt_scheduler_create (GLT_sched_def *def, int num_threads, int *threads_id, GLT_sched_config config, GLT_sched *newsched)

  Creates a new scheduler.
- void glt_schededuler_create_basic (GLT_sched_predef predef, int num_threads, int *threads_id, GLT_sched_config config, GLT_sched *newsched)

  Creates a new scheduler with predefined behaviour.
- void glt_scheduler_free (GLT_sched *sched)

  Destroys a scheduler.
- void glt_scheduler_finish (GLT_sched sched)

  Finalizes a scheduler.
- void glt_scheduler_exit (GLT_sched sched)

  Stops a scheduler.
- void glt_scheduler_has_to_stop (GLT_sched sched, GLT_bool *stop)

  Requires to a scheduler to stop.
- void glt_scheduler_set_data (GLT_sched sched, void *data)

  Sets new data to a scheduler.
- void glt_scheduler_get_data (GLT_sched sched, void **data)

  gets data from a scheduler.
- void glt_scheduler_get_size (GLT_sched sched, size_t *size)

  gets the current size from the scheduler.
- void glt_scheduler_get_total_size (GLT_sched sched, size_t *size)

  gets the total size from the scheduler.

3.9.1 Detailed Description

These functions manages the configurable scheduler.

3.9.2 Function Documentation

3.9.2.1 void glt_schededuler_create_basic (GLT_sched_predef *predef*, int *num_threads*, int * *threads_id*, GLT_sched_config *config*, GLT_sched * *newsched*)

Creates a new scheduler with predefined behaviour.

`glt_schededuler_create_basic()` creates a new scheduler for some threads with a predefined behaviour.

Parameters

in	*def*	Handle of the target c\ GLT_sched_predef.
in	*num_threads*	number of `GLT_thread` affected by this scheduler.
in	*threads_id*	pointer to an array of c\ GLT_threads_id.
in	*config*	Handle of the target c\ GLT_sched_config.

out	*newsched*	Handle of new c\ GLT_sched.

3.9.2.2 void glt_scheduler_config_free (GLT_sched_config * *config*)

Destroys the scheduler configuration.

`glt_scheduler_config_free()` deletes the scheduler configuration.

Parameters

in	*config*	Handle of the target c\ GLT_sched_config.

3.9.2.3 void glt_scheduler_create (GLT_sched_def * *def*, int *num_threads*, int * *threads_id*, GLT_sched_config *config*, GLT_sched * *newsched*)

Creates a new scheduler.

`glt_scheduler_create()` creates a new scheduler for some threads.

Parameters

in	*def*	Handle of the target c\ GLT_sched_def.
in	*num_threads*	number of `GLT_thread` affected by this scheduler.
in	*threads_id*	pointer to an array of c\ GLT_threads_id.
in	*config*	Handle of the target c\ GLT_sched_config.
out	*newsched*	Handle of new c\ GLT_sched.

3.9.2.4 void glt_scheduler_exit (GLT_sched *sched*)

Stops a scheduler.

`glt_scheduler_exit()` Stops a scheduler.

Parameters

in	*sched*	Handle of the target c\ GLT_sched.

3.9.2.5 void glt_scheduler_finish (GLT_sched *sched*)

Finalizes a scheduler.

`glt_scheduler_finish()` finalizes a scheduler.

Parameters

in	*sched*	Handle of the target c\ GLT_sched.

3.9.2.6 void glt_scheduler_free (GLT_sched * *sched*)

Destroys a scheduler.

`glt_scheduler_free()` destroys a scheduler.

Parameters

in	*sched*	Handle of the target c\ GLT_sched.

3.9.2.7 void glt_scheduler_get_data (GLT_sched *sched*, void ** *data*)

gets data from a scheduler.

`glt_scheduler_get_data()` gets data from a scheduler.

Parameters

in	*sched*	Handle of the target c\ GLT_sched.
out	*data*	obtained.

3.9.2.8 void glt_scheduler_get_size (GLT_sched *sched*, size_t * *size*)

gets the current size from the scheduler.

`glt_scheduler_get_size()` gets size from a scheduler.

Parameters

in	*sched*	Handle of the target c\ GLT_sched.
out	*size*	obtained.

3.9.2.9 void glt_scheduler_get_total_size (GLT_sched *sched*, size_t * *size*)

gets the total size from the scheduler.

`glt_scheduler_get_total_size()` gets the total size from a scheduler.

Parameters

in	*sched*	Handle of the target c\ GLT_sched.
out	*size*	obtained.

3.9.2.10 void glt_scheduler_has_to_stop (GLT_sched *sched*, GLT_bool * *stop*)

Requires to a scheduler to stop.

`glt_scheduler_has_to_stop()` Requires a scheduler to stop.

Parameters

in	*sched*	Handle of the target c\ GLT_sched.
out	*stop*	shows the answer of the scheduler.

3.9.2.11 void glt_scheduler_set_data (GLT_sched *sched*, void * *data*)

Sets new data to a scheduler.

`glt_scheduler_set_data()` Sets data to a scheduler.

Parameters

in	*sched*	Handle of the target c\ GLT_sched.
in	*data*	to be set.

3.10 Event functions

Functions

- void glt_event_add_callbac (GLT_event_kind event, GLT_event_cb_fn ask_cb, void *ask_user_arg, GLT_event_cb_fn act_cb, void *act_user_arg, int *cb_id)

  Creates an event and adds the callback function.
- void glt_event_del_callback (GLT_event_kind event, int cb_id)

  Deletes an event.

3.10.1 Detailed Description

These functions manage the GLT events for the ULTs.

3.10.2 Function Documentation

3.10.2.1 void glt_event_add_callbac (GLT_event_kind *event*, GLT_event_cb_fn *ask_cb*, void * *ask_user_arg*, GLT_event_cb_fn *act_cb*, void * *act_user_arg*, int * *cb_id*)

Creates an event and adds the callback function.

`glt_event_add_callbac()` creates an event eith its callback function.

Parameters

in	*event*	Kind of event.
in	*ask_cb*	callback to ask whether the event can be handled.
in	*ask_user_arg*	user argument for ask_cb.
in	*act_cb*	callback to notify that the event will be handled.
in	*act_user_arg*	user argument for act_cb.
out	*cb_id*	callback ID.

3.10.2.2 void glt_event_del_callback (GLT_event_kind *event*, int *cb_id*)

Deletes an event.

`glt_event_add_callbac()` creates an event eith its callback function.

Parameters

in	*event*	Kind of event.
in	*cb_id*	callback ID.

3.11 Future functions

Functions

- void glt_future_create (int nbytes, GLT_future *newfuture)
  Creates a future.
- void glt_future_free (GLT_future *future)
  Destroys a future.
- void glt_future_wait (GLT_future future, void **value)
  Waits for a future.
- void glt_future_set (GLT_future future, void *value, int nbytes)
  Sets a future value.

3.11.1 Detailed Description

These functions manage the GLT futures for the ULTs.

3.11.2 Function Documentation

3.11.2.1 void glt_future_create (int *nbytes*, GLT_future * *newfuture*)

Creates a future.

`glt_future_create()` creates a key.

Parameters

in	*nbytes*	size in bytes of the memory buffer.
out	*newfuture*	Hande to newly created `GLT_future`.

3.11.2.2 void glt_future_free (GLT_future * *future*)

Destroys a future.

`glt_future_free()` destroys a future for ULTs.

Parameters

in	*future*	Handle to the target `GLTfuture`.

3.11.2.3 void glt_future_set (GLT_future *future*, void * *value*, int *nbytes*)

Sets a future value.

`glt_future_set()` sets a value in the eventual's buffer and releases all waiting ULTs.

Parameters

in	*future*	Handle to the target `GLT_future`.
in	*value*	Pointer to the memory buffer containing the data that will be copied to the memory buffer of the future.

in	*nbytes*	number of bytes to be copied

3.11.2.4 void glt_future_wait (GLT_future *future*, void ** *value*)

Waits for a future.

`glt_future_wait()` blocks the caller ULT until the eventual eventual is resolved.

Parameters

in	*future*	Handle to the target `GLT_future`.
out	*value*	pointer to the memory buffer of the `GLT_future`.

3.12 GLT object list

Variables

- GLT_ult
  The user level thread abstraction.
- GLT_tasklet
  The tasklet abstraction.
- GLT_thread
  The thread abstraction.
- GLT_mutex
  The mutex abstraction.
- GLT_barrier
  The barrier abstraction.
- GLT_cond
  The condition abstraction.
- GLT_timer
  The timer abstraction.
- GLT_bool
  The boolean abstraction.
- GLT_thread_id
  The thread id abstraction.
- GLT_ult_id
  The ult id abstraction.
- GLT_key
  The key abstraction.
- GLT_event
  The event abstraction.
- GLT_future
  The future abstraction.
- GLT_sched
  The scheduler abstraction.
- GLT_sched_config
  The scheduler configuration abstraction.
- GLT_sched_def
  The scheduler definition abstraction.
- GLT_sched_predef
  The scheduler predefinition abstraction.

3.12.1 Detailed Description

Index

Acronyms

API Application Programming Interface. xvii, xviii, 1–3, 6, 7, 10, 13, 15, 16, 22, 47–51, 53, 54, 56–59, 61–65, 67, 68, 71, 79, 82, 85–89

BLAS Basic Linear Algebra Subprograms. 25

CAP Computación de Altas Prestaciones. xx, 91, 92, 95

CG Conjugate Gradient. 76

CPU Central Processing Unit. 5, 12, 23, 25, 39, 42, 47–49, 56, 68, 71, 73, 75, 81

ES Execution Stream. 13–15, 21, 22, 25, 28, 34, 35, 48, 50, 56

FEB Full-Empty Bit. 11, 13, 35, 51, 53

FIFO First-In-First-Out. 8, 10

GLT Generic Lightweight Threads. 2, 3, 47–51, 53–65, 67, 68, 71–74, 76, 78, 79, 82, 83, 85, 86, 88–95

GLTO Generic Lightweight Threads OpenMP. 2, 67, 68, 70–73, 75, 76, 79, 82, 85, 86, 88–92, 94, 95

GNU GNU is Not Unix. 6, 16, 17, 23, 25–27, 30–33, 36–39, 41, 44, 53, 68, 72, 73, 75, 76, 82, 86, 88, 89, 92, 94

GOmpSs Generic Lightweight Threads OmpSs. 2, 67, 79–82, 85, 86, 89, 91, 92, 94

HPC High-Performance Computing. xvii, xviii, 2, 7, 30, 41, 47, 48, 51, 65, 85, 86, 88, 90

IPA Interfaz de Programación de Aplicaciones. xix, xx

IPC Instructions Per Call. 59, 60

KSE Kernel Schedule Entity. 6, 48

LIFO Last-In-First-Out. 8, 10, 12, 14

LWT Lightweight Thread. xvii, xviii, 1–3, 5–8, 11, 13, 15, 17, 21–23, 25, 31, 36, 42–44, 47–51, 53, 55–59, 63, 65, 67, 68, 71–73, 76, 78–83, 85–90

MPI Message Passing Interface. 8, 85, 90, 91, 95

NUMA Non-Uniform Memory Access. 28, 42, 73

OS Operating System. xvii, 1–3, 5, 6, 8, 11, 13, 21, 23–25, 30, 36, 38, 42, 43, 47, 48, 63, 68, 88, 90

PM Programming Model. xvii, xviii, 1–3, 5–11, 13–17, 21–23, 43, 47–49, 51, 65, 67, 78–83, 85–90

Pthread POSIX Thread. ix, x, xvii–xx, 1, 2, 6, 7, 10, 16, 21–25, 27, 30, 33, 35–38, 40–44, 47, 48, 51, 61–65, 67, 68, 70, 71, 73–76, 78, 82, 83, 86, 87, 89, 92, 93

rCUDA Remote CUDA. 90, 94

ULT User-Level Thread. 6, 8, 9, 11–15, 21–25, 28, 33, 35, 37, 38, 41–43, 47–51, 56–58, 61, 63, 68

UTS Unbalanced Tree Search. 58, 59, 62, 64, 65

Bibliography

[1] ARM OpenMP. https://developer.arm.com/products/software-development-tools/hpc.

[2] BOLT: A Lightning-Fast OpenMP Implementation. http://bolt-omp.org/.

[3] Clang project. http://clang.llvm.org/.

[4] CloverLeaf miniapp. http://uk-mac.github.io/CloverLeaf/.

[5] Flang OpenMP. https://github.com/flang-compiler/flang.

[6] Generic Lightweight Threads API. http://github.com/adcastel/GLT.

[7] Generic Lightweight Threads OpenMP. https://github.com/adcastel/glto-runtime.

[8] GNU C Library. https://www.gnu.org/software/libc/.

[9] GNU Portable Threads. http://www.gnu.org/software/pth/.

[10] GOMP: GNU OpenMP. https://gcc.gnu.org/projects/gomp/.

[11] Intel OpenMP. https://software.intel.com/en-us/parallel-studio-xe.

[12] Lahey/Fujitsu OpenMP. http://www.lahey.com/.

[13] LLVM project. http://openmp.llvm.org/.

[14] NAG OpenMP. https://www.nag.com/.

[15] OpenMP 4.5 Specification. http://www.openmp.org/.

[16] OpenUH Research Compiler OpenMP. https://github.com/uhhpctools/openuh.

[17] Oracle OpenMP. http://www.oracle.com/technetwork/server-storage/developerstudio/overview/index.html.

[18] PGI OpenMP. https://www.pgroup.com/.

[19] Programming with Solaris Threads. https://docs.oracle.com/cd/E19455-01/806-5257/6je9h033n/index.html.

[20] Pthreads API. https://computing.llnl.gov/tutorials/pthreads/.

[21] Stackless Python. http://www.stackless.com.

[22] Texas Instruments OpenMP. http://www.ti.com/.

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

[24] TOP500 Supercomputer Sites. http://www.top500.org/.

[25] Aliaga, J. I., Anzt, H., Castillo, M., Fernández, J. C., León, G., Pérez, J., and Quintana-Ortí, E. S. 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.

[26] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing 37*, 1 (1996), 55–69.

[27] BSC. BSC application repository. pm.bsc.es/projects/bar.

[28] BSC. Mercurium compiler. pm.bsc.es/mcxx.

[29] BSC. Nanos++. https://pm.bsc.es/projects/nanox/.

[30] BSC. The OmpSs programming model. http://pm.bsc.es/ompss/.

[31] Castelló, A., Mayo, R., Planas, J., and Quintana-Ortí, E. S. Exploiting task-parallelism on GPU clusters via OmpSs and rCUDA virtualization. In *Trustcom/BigDataSE/ISPA, 2015 IEEE* (2015), vol. 3, IEEE, pp. 160–165.

[32] Castelló, A., Mayo, R., Sala, K., Beltran, V., Balaji, P., and Peña, A. J. On the adequacy of lightweight thread approaches for high-level parallel programming models. *Future Generation Computer Systems (FGCS) 84* (2018), 22 – 31.

[33] Castelló, A., Mayo, R., Seo, S., Balaji, P., Quintana-Ortí, E. S., and Peña, A. J. Analysis of lightweight thread libraries for high-performance computing. *Submitted to IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2018).

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

[35] Castelló, A., Peña, A. J., Mayo, R., Balaji, P., and Quintana-Ortí, E. S. Exploring the suitability of remote GPGPU virtualization for the OpenACC programming model using rCUDA. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on* (2015), IEEE, pp. 92–95.

[36] Castelló, A., Peña, A. J., Mayo, R., Planas, J., Quintana-Ortí, E. S., and Balaji, P. Exploring the interoperability of remote GPGPU virtualization using rCUDA and directive-based programming models. *The Journal of Supercomputing (JoS)* (Jun 2016).

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

[38] Castelló, A., Seo, S., Mayo, R., Balaji, P., Quintana-Ortí, E. S., and Peña, A. J. *GLT User's Guide*. Universitat Jaume I, 2017.

[39] Castelló, A., Seo, S., Mayo, R., Balaji, P., Quintana-Ortí, E. S., and Peña, A. J. GLTO: On the adequacy of lightweight thread approaches for OpenMP implementations. In *Proceedings of the International Conference on Parallel Processing (ICPP)* (Bristol, UK, August 2017).

[40] Catalán, S., Castelló, A., Igual, F. D., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. Programming parallel dense matrix factorizations with look-ahead and OpenMP. *Submitted to Submitted to Cluster Computing (CC)* (2018).

[41] Cuvillo, J. d., Zhu, W., Hu, Z., and Gao, G. R. TiNy threads: A thread virtual machine for the Cyclops64 cellular architecture. In *Proceedings of the Fifth Workshop on Massively Parallel Processing* (April 2005).

[42] Dagum, L., and Menon, R. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering 5*, 1 (1998), 46–55.

[43] Dang, H.-V., Snir, M., and Gropp, W. Towards millions of communicating threads. In *Proceedings of the 23rd European MPI Users' Group Meeting (EUROMPI)* (2016), ACM.

[44] Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys)* (October 2006), pp. 29–42.

[45] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters (PPL) 21*, 02 (2011), 173–193.

[46] Duran González, A., Teruel, X., Ferrer, R., Martorell Bofill, X., and Ayguadé Parra, E. Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *38th International Conference on Parallel Processing (ICPP)* (2009), pp. 124–131.

[47] Flores, I. *Operating System for Multiprogramming with a Variable Number of Tasks*. Allyn and Bacon, 1973.

[48] Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., and Yang, G. The Sunway TaihuLight Supercomputer: System and applications. *Science China Information Sciences 59*, 7 (2016), 072001.

[49] Intel Corp. Intel Math Kernel Library (MKL) 11.0. `http://software.intel.com/en-us/intel-mkl`.

[50] Iserte, S., Clemente-Castelló, F. J., Castelló, A., Mayo, R., and Quintana-Ortí, E. S. Enabling GPU virtualization in cloud environments. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016)* (Rome, Italy, April 2016), pp. 249–256.

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

[52] Kale, L. V., and Krishnan, S. Charm++: a portable concurrent object oriented system based on c++. In *ACM Sigplan Notices* (1993), vol. 28, ACM, pp. 91–108.

[53] Kale, L. V., Yelon, J., and Knuff, T. Threads for interoperable parallel programming. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing (LCPC)* (August 1996), pp. 534–552.

[54] Laboratory, S. N. Qthreads schedulers. `github.com/Qthreads/qthreads/blob/master/SCHEDULING`.

[55] Microsoft MSDN Library. Fibers. `https://msdn.microsoft.com/en-us/library/ms682661.aspx`.

[56] Nakashima, J., and Taura, K. MassiveThreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, vol. 8665. Springer Berlin Heidelberg, 2014, pp. 222–238.

[57] Nichols, B., Buttlar, D., and Farrell, J. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[58] OpenMP Architecture Review Board. OpenMP application programming interface version 4.5. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, Nov. 2015.

[59] Pérache, M., Jourdren, H., and Namyst, R. MPC: A unified parallel runtime for clusters of NUMA machines. In *Proceedings of the IEEE International European Conference on Parallel and Distributed Computing (EURO-PAR)* (2008), pp. 78–88.

[60] Pheatt, C. Intel® threading building blocks. *Journal of Computing Sciences in Colleges 23*, 4 (2008), 298–298.

[61] Saltzer, J. H. *Traffic control in a multiplexed computer system.* PhD thesis, Massachusetts Institute of Technology, 1966.

[62] Schmager, F., Cameron, N., and Noble, J. Gohotdraw: Evaluating the Go programming language with design patterns. In *Evaluation and Usability of Programming Languages and Tools* (2010), ACM, p. 10.

[63] Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Carns, P., Castelló, A., Genet, D., Herault, T., Iwasaki, S., Jindal, P., Kale, S., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y., Taura, K., and Beckman, P. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2018).

[64] Stein, D., and Shah, D. Implementing lightweight threads. In *USENIX Summer* (1992).

[65] Taura, K., Tabata, K., and Yonezawa, A. StackThreads/MP: Integrating futures into calling standards. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (1999), pp. 60–71.

[66] Valgrind Developers. Callgrind: A call-graph generating cache and branch prediction profiler.

[67] von Behren, R., Condit, J., Zhou, F., Necula, G. C., and Brewer, E. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)* (October 2003), pp. 268–281.

[68] Wang, C., Chandrasekaran, S., and Chapman, B. An OpenMP 3.1 validation testsuite. In *International Workshop on OpenMP* (2012), pp. 237–249.

[69] Wheeler, K. B., Murphy, R. C., and Thain, D. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 2008 Workshop on Multithreaded Architectures and Applications (MTAAP)* (April 2008).

[70] Wikipedia. Os/360 and successors. `en.wikipedia.org/wiki/OS/360_and_successors#MVT`.