GLTO: una Implementación de OpenMP sobre Hilos Ligeros

Adrián Castelló[1], Rafael Mayo[1], Sangmin Seo[2], Pavan Balaji[2],
Enrique S. Quintana-Ortí[1], Antonio J. Peña[3]

Resumen— Resumen— **OpenMP es la interfaz de programación de aplicaciones estándar para el paralelismo a nivel de nodo. Las soluciones más populares de OpenMP se basan en implementaciones con POSIX threads (Pthreads) que ofrecen un rendimiento excelente en paralelismo de grano grueso y encajan perfectamente con el hardware actual. Sin embargo, una nueva tendencia en runtimes/ aplicaciones apunta al uso de paralelismo de grano fino en conjunto con paradigmas de planificaciones dinámicas. Se ha demostrado que los hilos ligeros (HLs) o hilos de nivel de usuario son más apropiados para estos nuevos paradigmas. Hemos diseñado y desarrollado GLTO, una implementación de OpenMP sobre la nueva interfaz Generic Lightweight Threads (GLT). GLT expone una interfaz común para soluciones HL y ofrece la posibilidad de ejecutar la misma aplicación/runtime sobre distintas librerías de HLs. En este artículo, utilizamos GLTO para analizar distintos escenarios donde las implementaciones de OpenMP pueden beneficiarse tanto del uso de HLs como de Pthreads. Nuestro estudio revela que ninguna implementación es superior a las otras en todos los escenarios y además, existe una diferencia importante entre ellas.**

Palabras clave— **GLT, Hilos ligeros, OpenMP, POSIX Threads, Modelos de programación.**

I. Introducción

En los últimos años, el número de núcleos por procesador ha aumentado periódicamente alcanzando cifras como los 260 núcleos del supercomputador Sunway TaihuLight supercomputer [1] que ocupa el primer puesto en la lista TOP500 [2] desde junio de 2016.

La tendencia de esta lista indica que los futuros sistemas exaescala ofrecerán un paralelismo masivo a nivel de nodo, contando con miles de núcleos por procesador. Sin embargo, extraer todo el poder computacional del hardware requerirá de bibliotecas y modelos de programación (MP) eficientes. Uno de los modos más populares para obtener un aceptable paralelismo a nivel de nodo consiste en utilizar la interfaz de programación de aplicaciones (API, del inglés Application Programming Interface) de POSIX threads (Pthreads) [3], o a través de MP basados en directivas, como por ejemplo OpenMP [4] u OmpSs [5].

Estos MP están implementados sobre la API de Pthreads, que encaja perfectamente con el hardware actual y los códigos de paralelismo de grano grueso. Sin embargo, falla con los nuevos paradigmas de código, como el paralelismo de grano fino y la planificación dinámica, debido a su alto coste de gestión.

Durante los últimos años se han implementado algunas bibliotecas de hilos ligeros (HLs) para tratar con estos nuevos paradigmas de código [6]. Cada solución de HLs tiene su propio MP y su propio entorno. Algunas se implementan para un sistema operativo (SO) específico, como Windows Fibers [7] y Solaris Threads [8]. Otros como ConverseThreads [9] y Nanos++ [10] soportan un específico MP de alto nivel: Charm++ [11] y OmpSs [5], respectivamente. También hay soluciones de propósito general, como MassiveThreads [12], Qthreads [13], y Argobots [14]. La Generic Lightweight Threads (GLT) API [15] es un esfuerzo para unificar estas soluciones de HLs bajo un único MP con el fin de potenciar la productividad y portabilidad con un sobrecoste mínimo demostrado en [16]. Esta API ofrece una funcionalidad común para soluciones de HLs y actualmente está implementada sobre MassiveThreads, Qthreads, y Argobots. Por lo tanto, cualquier runtime/aplicación escrita con la API de GLT no necesitará ninguna modificación para ejecutarse sobre estas tres soluciones de HLs.

En este artículo presentamos GLTO: nuestro diseño e implementación del runtime OpenMP sobre la API de GLT. Nuestro OpenMP se basa en el proyecto BOLT [17], que a su vez se basa en el proyecto LLVM [18] (el runtime de OpenMP de LLVM comparte el código de la implementación de OpenMP de Intel [19]). Además, verificamos nuestra solución con el test de validación OpenUH OpenMP Validation Suite 3.1 [20].

Centrados en GLTO, analizamos los patrones más comunes de código OpenMP y explicamos cómo los HLs los afrontan comparándolos con las implementaciones tradicionales basadas en Pthreads. Evaluamos nuestra implementación de OpenMP y comparamos los resultados con los obtenidos al utilizar los runtimes de OpenMP de GNU y de Intel[1] en cuatro escenarios distintos: código paralelo básico, bucle *for*, paralelismo anidado y paralelismo de tareas. Nuestro estudio revela que ninguna de las soluciones de OpenMP obtiene el mejor rendimiento en todos los escenarios y que existe una diferencia importante entre ellas.

En resumen, las contribución principal de este artículo es un análisis de los patrones de código OpenM que pueden beneficiarse de implementa-

[1]Universitat Jaume I, Castellón, e-mails: {adcastel, mayo, quintana}@uji.es.
[2]Argonne National Laboratory, US, e-mail: {sseo,balaji}@anl.gov
[3]BSC, Barcelona, e-mail: antonio.pena@bsc.es.

[1]Puesto que BOLT se encuentra en fase de desarrollo, no es justa una comparativa con GLTO por el momento.

ciones basadas en HLs o en Pthreads.

El resto del artículo se organiza de la siguiente forma. Sección II revisa el trabajo relacionado. Sección III ofrece información sobre OpenMP y GLT. Sección IV detalla la implementación de GLTO. Sección V valida nuestra solución. Sección VI ofrece un análisis de los distintos escenarios evaluados. Sección VII expone las lecciones aprendidas en nuestro trabajo. Sección VIII contine las conclusiones y el trabajo futuro.

II. Trabajo Relacionado

Actualmente, el estándar OpenMP es soportado por una gran cantidad de compiladores, tanto de código abierto como privados. A pesar de que la especificación más reciente de OpenMP es la versión 4.5 [21], algunos compiladores no soportan la totalidad de su funcionalidad. Por ejemplo, el compilador del proyecto LLVM (`clang 3.9`) soporta todas las funcionalidades de OpenMP 4.5, a excepción del offloading. Por contra, el compilador de Intel `icc 16.0` soporta completamente la especificación 4.0 de OpenMP y, tanto el `icc 17.0` como el compilador de GNU `gcc 6.1`, lo hacen con la especificación 4.5. Hay otros compiladores que se encuentran un paso por detrás de estas conocidas soluciones. Por ejemplo `pgcc` [22], el compilador de Portland Group, y OpenUH [23], soportan solamente hasta la versión 3.1 de la especificación de OpenMP.

Soportar una especificación de OpenMP significa que cada runtime puede tener sus propias características porque pueden ser específicas para un determinado hardware o patrón de código. De entre todas las soluciones, los más ampliamente aceptados son los ofrecidos por GNU e Intel, llamados `libgomp` e Intel OpenMP runtime, respectivamente. En algunos casos, el mismo código del runtime es compartido entre distintas implementaciones de OpenMP, como ocurre con la solución de Intel, que puede ser utilizada también por el compilador de código abierto `clang`. En el campo de las bibliotecas de HLs, los trabajos presentados en [6], [9], [12], [13], [14] presentan distintas implementaciones de HLs y analizan su rendimiento. El trabajo presentado en [24] realiza un análisis de distintas soluciones de HLs desde un punto de vista semántico y evalúa sus prestaciones.

La relación entre HLs y el runtime de OpenMP se ha estudiado en el pasado. En [25] y [26], el paralelismo anidado se analiza y se resuelve mediante soluciones de HLs. También se ha estudiado en [27] y [28] el efecto de OpenMP cuando se utilizan arquitecturas de acceso a memoria no unificado (NUMA, del ingés Non-Unified Memory Access) para paralelismo de tareas.

III. Background

En esta sección repasamos el MP de OpenMP y describimos la implementación de GLT y su interacción con las bibliotecas de HLs.

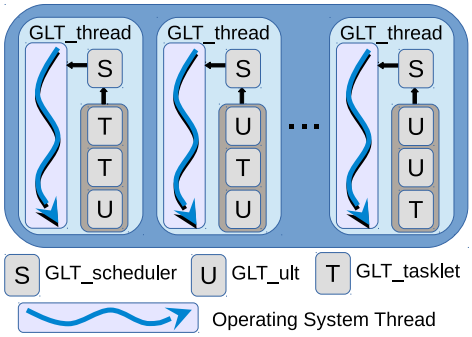

Fig. 1: MP de la biblioteca GLT.

A. OpenMP

La API de OpenMP ofrece una programación multiplataforma y de memoria compartida. Las implementaciones actuales abarcan la mayoría de las arquitecturas y de los sistemas operativos (SO). OpenMP es un MP basado en directivas que acelera el código secuencial mediante el uso de "pragmas". Intel y GNU ofrecen implementaciones de OpenMP sobre la API de Pthreads para aprovechar el paralelismo.

Los runtimes de OpenMP se pueden dividir en dos partes principales: el paralelismo de trabajo compartido y el paralelismo de tareas. Mientras que en el paralelismo de trabajo compartido todas las implementaciones de OpenMP siguen el mismo patrón, en el paralelismo de tareas se pueden encontrar diferencias en su diseño. Más concretamente, la versión de GNU utiliza una única cola compartida entre todos los hilos donde las tareas son almacenadas para posteriormente ser ejecutadas. Sin embargo, la solución ofrecida por Intel incorpora una cola privada para cada hilo e implementa un mecanismo de robo de trabajo (WS, del ingés Work-Stealing) para el balanceo de carga de trabajo. En ambas soluciones la gestión del trabajo compartido y de las tareas no comparten más código que las estructuras básicas de los hilos, ya que las tareas aparecieron por primera vez en la especificación 3.0 de OpemMP.

B. Generic Lightweight Threads

GLT es una API común que ha sido diseñada con el propósito de unificar, bajo una sóla semántica, distintas soluciones de HLs. Actualmente está implementada sobre las siguientes soluciones de propósito general: MassiveThreads, Qthreads, y Argobots. A pesar de que la API de GLT no soporta por completo todas las funcionalidades de las bibliotecas de HLs originales, su conjunto de funciones es suficiente para implementar el runtime de OpenMP.

La API de GLT abstrae las semánticas de cada componente del entorno bajo la misma terminología. La Figura 1 ilustra el MP ofrecido por esta API. Concretamente, `GLT_thread` se refiere al hilo del SO, mientras que `GLT_ult` representa el HL o hilo de nivel de usuario (ULT, del inglés User-Level Thread). Además, el `GLT_tasklet` corresponde a una unidad de trabajo que es más ligera que los propios HLs

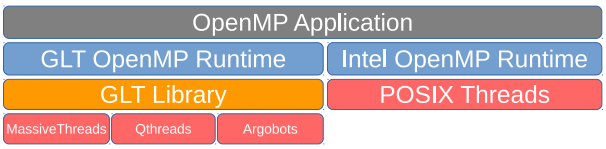

Fig. 2: Pila de bibliotecas disponibles para un código OpenMP.

puesto que no posee su propia pila (deshabilitando migraciones o intercambios) y se utilizan en código de cómputo. Como los `GLT_tasklet`s solamente los soporta Argobots, cuando ejecutamos otra solución de HLs, se utilizan `GLT_ult`s. El `GLT_scheduler` depende de la solución de HLs que se esté utilizando, puesto que cada biblioteca cuenta con su propia implementación. Esta libertad solamente afecta al rendimiento pero nunca al resultado final de la ejecución de la aplicación.

El uso de esta API intermedia entre el código de usuario y la biblioteca de HLs permite al programador probar y elegir distintas soluciones de HLs con solamente una versión de código. Esta característica aumenta la portabilidad entre hardware y código.

IV. OpenMP sobre GLT

En esta sección justificamos las decisiones de diseño adoptadas para adaptar el runtime OpenMP de LLVM al uso de las bibliotecas de HLs.

Como se ha comentado en la Sección I, nuestra implementación de OpenMP se basa en el proyecto BOLT, que a su vez se basa en el código de LLVM. Seleccionamos ese punto de inicio porque, tanto el compilador `clang` [29] como el runtime, son de código abierto. Además, este runtime se puede enlazar con el código generado por el compildor de Intel.

A. Interacciones de GLTO

GLTO ofrece una implementación completa de la especificación de OpenMP 4.0 para códigos escritos en C, C++ y Fortran y puede ser enlazado con código generado por los compiladores `clang` e `icc`. La Figura 2 muestra que un código compilado con esas herramientas se puede enlazar con el runtime original de Intel que utiliza Pthreads, o bien con el runtime de GLTO y ejecutado sobre cualquier biblioteca de HLs. Esta flexibilidad ofrecida por la GLTO puede ayudar a los desarrolladores de dos formas distintas: si una biblioteca de HLs implementa la API de GLT, un código OpenMP se puede ejecutar sobre ella; en el caso de que un cierto código se beneficie de alguna característica de una concreta solución de HLs, el usuario puede cambiar la biblioteca sin modificar el código de la aplicación.

B. Detalles de implementación de GLTO

Las bibliotecas de HLs se componen de dos niveles de hilos. El más bajo está formado por los hilos del SO. Estos hilos son planificados y gestionados por el SO (como por ejemplo los Pthreads). El más

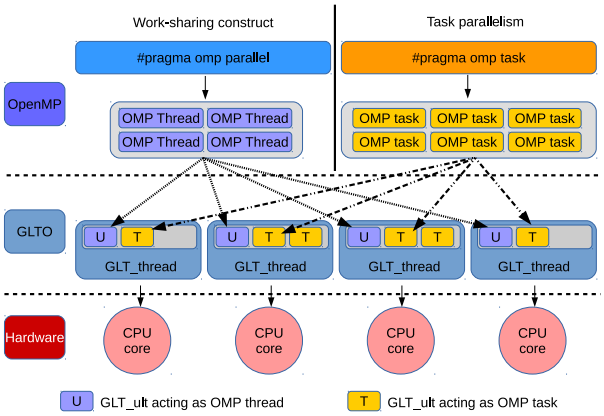

Fig. 3: Relación entre código OpenMP y la implementación de la GLTO.

alto lo forman los HLs que se gestionan (creación, planificación y ejecución) en el espacio de usuario y son ejecutados por los hilos del SO. Esta gestión no involucra al SO y, por lo tanto, el sobrecoste añadido es mucho menor en comparación con la gestión de los hilos del SO.

Cumpliendo con la especificación 4.5 de OpenMP [21], GLTO reacciona al uso de la variable de entorno `OMP_NUM_THREADS` para crear tantos `GLT_thread`s como hilos OpenMP elija el usuario. Como se muestra en la Figura 3, los `GLT_thread`s son fijados a los núcleos del procesador con una relación 1:1 y se crean cuando se carga la biblioteca. Los `GLT_thread`s serán los responsables de ejecutar los `GLT_ult`s que se crearán en tiempo de ejecución. También es posible modificar el número de hilos que participan en la ejecución mediante el uso de la cláusula `num_threads` y la función `omp_set_num_threads`.

Los `GLT_ult`s actuan como los Pthreads en las actuales implementaciones de OpenMP cuando se utiliza paralelismo de trabajo compartido. La parte izquierda de la Figura 3 muestra que cada hilo OpenMP se convierte en un `GLT_ult` en ese escenario.

En el caso de paralelismo de tareas (parte derecha de la Figura 3), cada tarea OpenMP también se convierte en un `GLT_ult`. Sin embargo, debido a las diferencias entre las estructuras de un hilo OpenMP y una tarea OpenMP, en la implementación de GLTO, el comportamiento de los `GLT_ult`s difiere si actúan como hilo o como tarea.

En las siguientes subsecciones explicamos más detalladamente cómo actúa GLTO en cada escenario.

C. Paralelismo de Trabajo Compartido

Para el paralelismo de trabajo compartido, nuestra solución de OpenMP imita el mecanismo seguido por GNU e Intel. El hilo principal asigna un puntero a función a cada hilo del runtime y entonces, una vez ha realizado su trabajo, el hilo principal espera la finalización de los otros hilos. Cuando esta sincronización se lleva a cabo, el hilo principal finaliza la región paralela y continua ejecutando el código secuencial hasta encontrar otra región paralela.

En GLTO, el trabajo se asigna creando un `GLT_ult` con el puntero a función por cada `GLT_thread`, y el hilo principal espera la finalización de ese trabajo. Al igual que en las versiones basadas en Pthreads, el hilo principal también continúa con la ejecución del código secuencial.

D. Paralelismo de Tareas

En contraste con el paralelismo de trabajo compartido, el paralelismo de tareas depende de una implementación de OpenMP a otra. La razón principal es que las tareas se añadieron a la especificación en la versión 3.0 de OpenMP y entonces, las implementaciones existentes las adoptaron con la cautela de no afectar al rendimiento de la parte de trabajo compartido. Por lo tanto, aunque forman parte del mismo runtime, son dos aproximaciones distintas.

Como se demostrará en la sección de evaluación, en este escenario es donde los HLs pueden mejorar las prestaciones, especialmente en el caso de tareas de grano fino. GLTO contempla dos escenarios posibles cuando se utiliza paralelismo de tareas. En caso de que éstas se creen en una región master o single, indica que un solo `GLT_thread` las generará mientras el resto de `GLT_thread`s participará en su ejecución. Si GLTO detecta este escenario, las tareas se repartirán siguiendo un patron circular entre todos los `GLT_thread`s. Si por el contrario, las tareas se generan en una región paralela, cada `GLT_thread` creará sus propias tareas en su propia cola.

E. Paralelismo Anidado

A pesar de que el paralelismo anidado es difidil de encontrar, este tipo de paralelismo puede aparecer implícitamente. Por ejemplo, cuando un código contiene un bucle `for` y dentro de éste se llama a una biblioteca externa que también está paralelizada con directivas OpenMP. Ese código generará paralelismo anidado y las implementaciones basadas en Pthreads presentan un rendimiento bajo.

GLTO afronta el paralelismo anidado aplicando la siguiente política. Para el nivel más externo, se divide el trabajo como en el paralelismo de trabajo compartido. Para el nivel anidado, cada `GLT_thread` generará y ejecutará los `GLT_ult`s necesarios para el nuevo nivel. Este mecanismo evita el problema de oversubscription (generar más hilos que núcleos) que lastra las implementaciones de OpenMP basadas en Pthreads.

F. Desbalanceo de Carga

La configuración por defecto de GLTO puede verse afectada por códigos paralelos irregulares. Concretamente, si hay un desbalanceo entre las tareas de OpenMP o el paralelismo anidado no es regular (por ejemplo, el primer nivel tiene menos hilos que el segundo), el rendimiento de GLTO puede verse afectado. Sin embargo, en esos casos, sacamos provecho del MP de la API de GLT que permite modificar el número de colas de trabajo. Más concretamente, utilizando la variable de entorno `GLT_SHARED_QUEUES`, todos los `GLT_thread`s comparten una sola cola de trabajos acabando con el problema del desbalanceo de carga.

TABLA I: Resultados del test OpenUH OpenMP Validation Suite 3.1 para las implementaciones de OpenMP.

	GNU	Intel	GLTO
Construcciones OpenMP	62	62	62
Tests utilizados	123	123	123
Tests correctos	118	118	121/122
Tests incorrectos	5	5	2/1

V. Validación de GLTO

En esta sección, mostramos los resultados de la prueba de validación y los comparamos con los obtenidos con las implementaciones de OpenMP de GNU e Intel.

El OpenUH OpenMP Validation Suite 3.1 está orientado a analizar la especificación 3.1 de OpenMP. Consiste en 123 tests que analizan 62 construcciones de OpenMP incluyendo paralelismo de tareas. El test eejecuta distintos modos de tests: *normal*, *cross*, y *orphan* [20].

La Tabla I muestra los resultados al ejecutar el test Validation Suite con los runtimes de GNU, Intel, y GLTO. El compilador `gcc 6.1` se utilizó para el runtime de GNU, y el `icc 16.0.1` para Intel y GLTO. Los resultados indican que, mientras Intel y GNU pasan 118 test, nuestra implementación de OpenMP pasa de manera satisfactoria 121 o 122 dependiendo si la biblioteca de HLs utilizada es Argobots/Qthreads o MassiveThreads, respectivamente.

Los tests no superados en GLTO sobre Argobots y Qthreads son los de las directivas `omp_taskyield` y `omp_task_untied`. La razón es que una vez una tarea se une a un determinado `GLT_thread`, al no haber WS, la tarea es reactivada por el mismo `GLT_thread` y el test cuenta el número de tareas pausadas por un `GLT_thread` y reactivadas por otro `GLT_thread` distinto. Sin embargo, nuestra implementación satisface el estándar porque no es obligatorio el cambio de hilo. Si utilizamos MassiveThreads, que sí que permite WS, solo falla el test `omp_taskyield`, porque no hay suficientes tareas que cambien de `GLT_thread`s. Las implementaciones de OpenMP de GNU e Intel fallan, además de los mismos tests que GLTO, pero en los modos *normal* y *orphan* así como el test `omp_task_final`.

Todos los fallos se concentran en la parte de paralelismo de tareas. Este aspecto indica que las soluciones introducidas no son tan sólidas como en el paralelismo de trabajo compartido. Esta situación coincide con el hecho de que las tareas se añadieron como un mecanismo separado.

VI. Evaluación de Prestaciones

En esta sección primero describimos el hardware y las bibliotecas utilizadas. Posteriormente presentamos los resultados para los distintos escenarios de código paralelo.

A. Hardware ay Bibliotecas

Los resultados de esta sección se han obtenido en una máquina de 36 núcleos (72 hilos) divididos en 2 procesadores Intel Xeon E5-2699 v3 (2.30 GHz) de 18 núcleos y 128 GB de RAM. Las bibliotecas son Intel OpenMP Runtime 20160808, GOMP 6.1, GLT 01-2017, Argobots 01-2017, Qthreads versión 1.10, y MassiveThreads versión 0.95. GLT, GOMP, y las bibliotecas HLs se han compilado con `gcc 6.1`. El runtime Intel OpenMP y GLTO se han compilado con el `icc 16.0`.

Las variables de entorno de OpenMP se inicializaron para obtener el mejor rendimiento en cada ecenario. `OMP_NESTED` y `OMP_BIND_PROC` se inicializaron a verdadero para todos los tests. El primero para activar el paralelismo anidado, puesto que por defecto está deshabilitado, y el segundo para evitar la migración de hilos entre los núcleos. Además para las implementaciones basadas en Pthreads, la variable `OMP_WAIT_POLICY` se inicializó con el valor `active` para paralelismo de trabajo compartido y con el valor `default` en paralelismo de tareas. En el primer caso, mantener activos los hilos mejora el rendimiento al completar el trabajo. En el paralelismo de tareas, el valor activo reduce el rendimiento ya que incrementa el sobrecoste por el mecanismo de WS.

B. OpenMP como Generador de Hilos

Una forma de utilizar OpenMP es añadiendo solamente una directiva `#pragma omp parallel` que envuelva a todo el código. Los hilos de OpenMP pueden controlarse como Pthreads, distinguiéndose entre ellos por su identificador. De esta forma, el ususario se aprovecha de OpenMP para crear y sincronizar los hilos. La responsabilidad del programador reside en dividir el trabajo entre los hilos.

En este experimento hemos utilizado el UTS Benchmark [30], un código paralelizado con OpenMP que mide el rendimiento alcanzado mediante una búsqueda exhaustiva en una árbol desbalanceado. El árbol se genera en tiempo de ejecución permitiendo un procesamiento paralelo mientras se genera el árbol determinístico.

La Figura 4 muestra el rendimiento cuando el UTS se ejecuta sobre las distintas implementaciones de OpenMP utilizando e tamaño de problema T1XXL, que corresponde al mayor tamaño que cabe en la memoria. Las barras etiquetadas como GCC e ICC corresponden a las soluciones de GNU e INtel, respectivamente; GLTO(ABT), GLTO(QTH), y GLTO(MTH) corresponden a la ejecución con GLTO sobre Argobots, Qthreads y MassiveThreads, respectivamente. Los resultados muestran unos rendimientos parecidos entre casi todas las soluciones. La

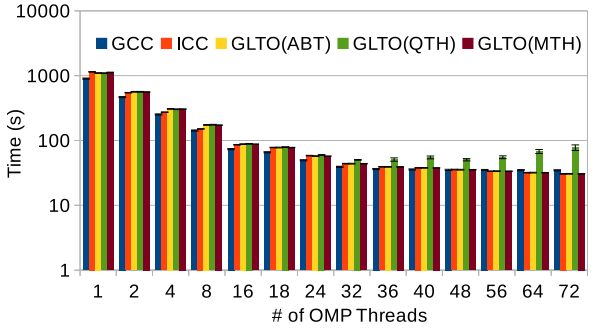

Fig. 4: Tiempo de ejecución para el test UTS benchmark (tamaño T1XXL) con distintos runtimes OpenMP e incrementando el número de hilos.

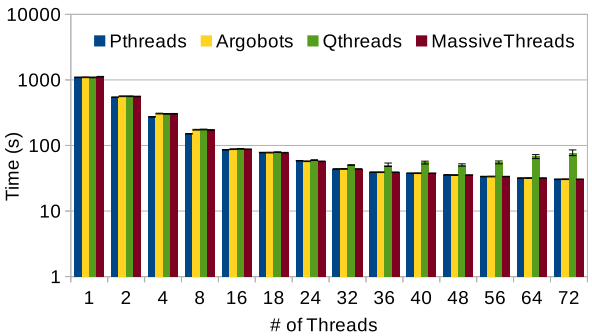

Fig. 5: Tiempo de ejecución para el test UTS benchmark (tamaño T1XXL) con las bibliotecas Pthreads e HLs incrementando el número de hilos.

razón para este comportamiento es que OpenMP solamente interviene en la creación del entorno y la interacción entre los hilos se controlan en el código. La diferencia entre GCC y el resto de soluciones se debe al código generado por el compilador.

Hay una pérdida de rendimiento cuando GLTO se utiliza sobre Qthreads. Para asegurar un mal diseño del runtima de GLTO, se ha traducido la versión original del UTS escrita en Pthreads [31] a las soluciones de HLs. Los resultados se muestran en la Figura 5 y revelan que el incremento de tiempo viene por la misma biblioteca de Qthreads. La principal razón reside en que cuando dos hilos se ejecutan en el mismo núcleo de la máquina, Qthreads utiliza mecanismos de sincronización para todas las palabras de memoria. Este hecho añade contención que se incrementa al aumentar el número de hilos.

En resumen, la elección de una implementación de OpenMP en este escenario no es crítica en el rendimiento.

C. OpenMP en Código Limitado por Cómputo

Este caso representa el código más frecuentemente acelerado con OpenMP. Básicamente consiste en un código iterativo que se ejecuta un determinado número de veces. Este código es muy favorable al MP ofrecido por OpenMP y es donde el runtime puede exprimir todo el poder del hardware paralelo. Para evaluar este escenario, hemos seleccionado la mini aplicación CloverLeaf [32], que resuelve las ecua-

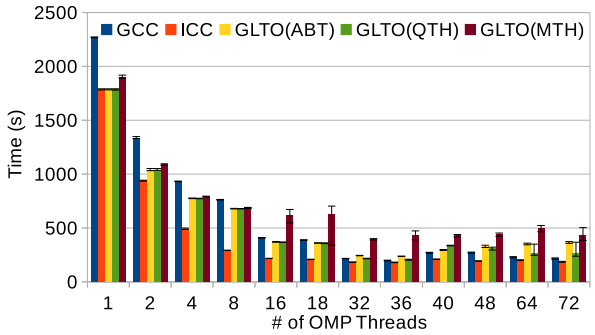

Fig. 6: Tiempo de ejecución de la mini aplicación CloverLeaf (tamaño clover_bm4.in) con distintos runtimes OpenMP e incrementando el número de hilos.

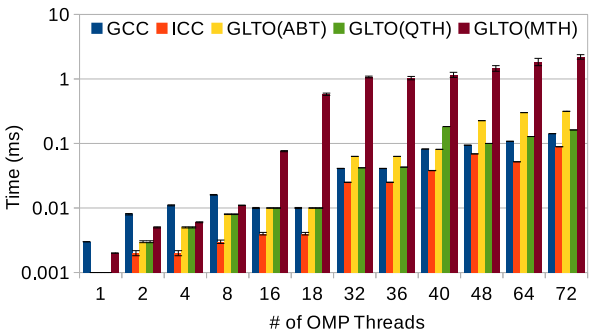

Fig. 7: Tiempo de ejecución de la asignación de trabajo en los runtimes de OpenMP incrementando el número de hilos.

ciones de Euler en una malla Cartesiana utilizando un método de precisión de segundo orden. Cada célula de la maya almacena 3 valores: energía, densidad y presión y además un vector de velocidad. La principal parte de la aplicación es un bucle for que se ejecuta 2.955 veces. El bucle se divide en distintas funciones y cada una de ellas calcula un valor de las células utilizando directivas `#pragma omp parallel for`. Concretamente, 144 bucles for paralelos son ejecutados las 2.955 veces, resultando un total de 336.870 bucles paralelos. La Figura 6 muestra la media de 50 ejecuciones de la aplicación con el tamaño clover_bm4.in. En este escenario la variación es mayor en el caso de MAssiveThreads causado por el mecanismo de WS. Además, las implementaciones de GNU e Intel (etiquetadas como GCC e ICC, respectivamente) obtienen mejores prestaciones. Como se ha comentado anteriormente, INtel y GNU envían el puntero a función a los hilos, mientras que las implementaciones de GLTO crean los `GLT_ults` con el puntero.

Para analizar la diferencia de tiempo, hemos evaluado el tiempo empleado en la asignación de trabajo de los runtimes de OpenMP. La Figura 7 muestra la diferencia entre las implementaciones de OpenMP, demostrando que las basadas en Pthreads (GNU e Intel) ofrecen un mecanismo más eficiente. A pesar de que la diferencia de tiempo es prácticamente imperceptible, repetir la operación más de 336.000 veces representa una diferencia de tiempo importante.

Al contrario que ocurren el el escenario presentado en la Sección VI-B, el buen diseño del paralelismo de trabajo compartido beneficia a las soluciones basadas en Pthreads.

D. OpenMP en Paralelismo Anidado

El paralelismo anidado no es un patrón común de OpenMP pero puede aparecer sin que el usuario sea consciente. Sin embargo, el constante incremento en el número de núcleos puede ofrecer a los programadores la introducción de diversos niveles de paralelismo para extraer todo el poder computacional del futuro hardware.

Debido al defectuoso diseño del paralelismo

```
1  #pragma omp parallel for
2  for (int i = 0; i < N; i++) {
3   #pragma omp parallel for \
4         firstprivate(i)
5   for (int j = 0; j < N; j++){
6     null_code(i,j);
7   }
8  }
```

Listing 1: Ejemplo de código OpenMP con paralelismo anidado.

anidado en las implementaciones actuales de OpenMP, es difícil encontrar aplicaciones que apliquen este patrón. Para estudiar este escenario, hemos implementado un test que mide el sobrecoste de gesión que produce el paralelismo anidado en los runtimes de OpenMP. Este test se compone de dos bucles `for` acelerados con directivas `#pragma omp parallel for` sin ningún código para que no interfiera en la medición del sobrecoste. Un ejemplo de código anidado se muestra en el Listing 1.

La Figura 8 revela la diferencia de rendimiento entre as distintas implementaciones de OpenMP cuando el bucle externo y el interno cuentan con 100 iteraciones cada uno, y la Figura 9 cuando tienen 1.000 iteraciones. Los tiempos de ejecución de las soluciones basadas en Pthreads (GNU e Intel) son de al menos un orden de magnitud mayor que cuando se utiliza GLTO.

El problema que presentan Intel y GCC es debido

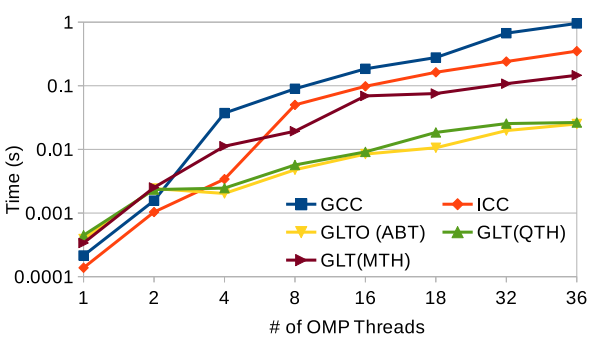

Fig. 8: Tiempo de ejecución de código con paralelismo anidado con runtimes de OpenMP y 100 iteraciones por cada bucle for e incrementando el número de hilos.

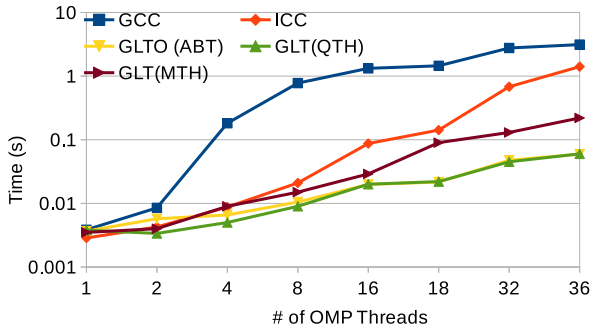

Fig. 9: Tiempo de ejecución de código con paralelismo anidado con runtimes de OpenMP y 1.000 iteraciones por cada bucle for e incrementando el número de hilos.

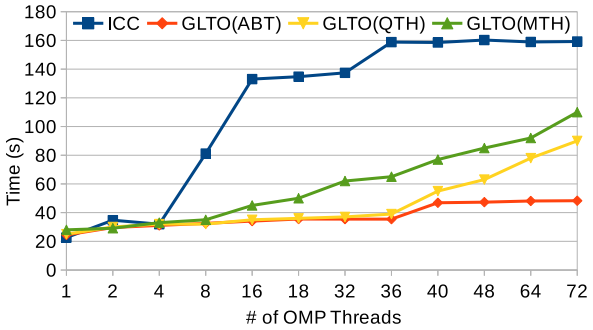

Fig. 10: Tiempo de ejecución del GC con granularidad 10 y runtimes OpenMP incrementando el número de hilos.

TABLA II: Número de hilos creados y reutilizados para cada implementación de OpenMP en paralelismo anidado con 100 iteraciones para cada bucle.

Runtime OpenMP	Hilos Creados	Hilos Reutilizados	`GLT_ults` Creados
GCC	3.536	0	—
Intel	1.296	2.240	—
GLTO	36	0	3.500

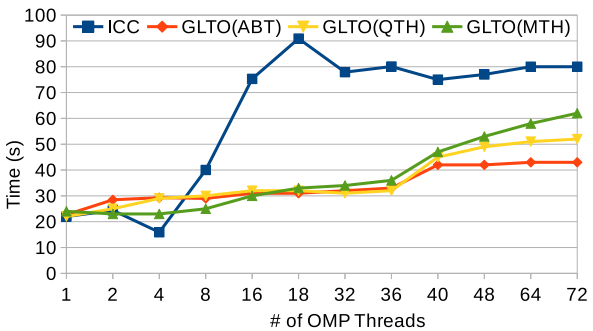

Fig. 11: Tiempo de ejecución del GC con granularidad 20 y runtimes OpenMP incrementando el número de hilos.

al problema de oversubscription. Por un lado, la solución GNU genera un número de hilos para el bucle externo, y para cada una de las iteraciones del bucle externo, genera un nuevo equipo de hilos para dividirse el bucle interno. Esta implementación no reutiliza hilos ociosos para salvar el contexto de los hilos del bucle interno. Por otro lado, la solución de Intel imita a GNU para el bucle externo pero sí reutiliza hilos para el bucle interno. Sin embargo, OpenMP de Intel generará equipos para el bucle interno. En el caso de GLTO solamente se crean `GLT_ults`, tanto para el bucle externo como para el interno, sin incurrir en oversubscription.

La Tabla II resume el número total de hilos creados cuando la variable de entorno `OMP_NUM_THREADS` se inicializa a 36 en el escenario de 100 iteraciones para cada bucle. El número total de hilos del SO excede los 72 núcleos de la máquina y su gestión afecta el rendimiento. En este escenario (36 hilos y 100 iteraciones por bucle) GNU genera 100 x 35 hilos para completar los equipos de hilos para el paralelismo anidado que se componen de 1 hilo del bucle externo y 35 nuevos. Si incluimos los 36 del equipo del primer nivel, tenemos un total de 3.536 hilos. En el caso de Intel, solamente se crean 36 equipos de 36 hilos (1.296) una vez y luego son reutilizados. A pesar de que GLTO crea 3.500 `GLT_ults`, éstos son más ligeros que los hilos del SO y no causan oversubscription porque solamente hay 36 `GLT_threads`.

En resumen, para códigos de paralelismo anidado, el uso de GLTO mejora el rendimiento en comparación con las versiones basadas en hilos del SO.

E. OpenMP en Paralelismo de Tareas

Para analizar este escenario, hemos seleccionado una aplicación de gradiente conjugado (GC). En el campo matemático, el método del GC es un algoritmo para la solución numérica de sistemas de equaciones lineales. Hemos convertido la implementación de [33] basada en directivas `#pragma omp parallel for` en directivas `#pragma omp task`. En nuestra implementación, un hilo actúa como productor mientras el resto de hilos son consumidores. La matriz de entrada es la `bmwcra_1` con un total de 14.878 filas. Nuesta transformación nos permite ajustar la granularidad de las tareas y poder observar cómo afecta al rendimiento. En este trabajo mostramos los resultados para granularidades de 10, 20, 50, y 100 filas por tareas, que producen 1.488, 744, 298, y 149 tareas, respectivamente. Por lo tanto, estudiamos el efecto de tres parámetros: número de hilos, granularidad y cantidad de tareas.

Al contrario que en otros escenarios, no hemos incluido el OpenMP de GNU por dos razones. La primera es que el GC original hace uso de la biblioteca Intel Math Kernel Library [34] y utilizarla de forma secuencial afectaría al rendimiento con GNU. En segundo lugar, GOMP tiene un entorno totalmente distinto al de Intel para el paralelismo de tareas mientras que Intel y GLTO lo comparten.

De la Figura 10 a la 13 muestran los resultados con granularidades de 10, 20, 50, y 100 filas por

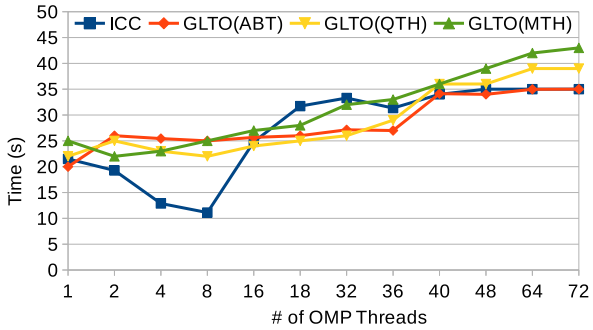

Fig. 12: Tiempo de ejecución del GC con granularidad 50 y runtimes OpenMP incrementando el número de hilos.

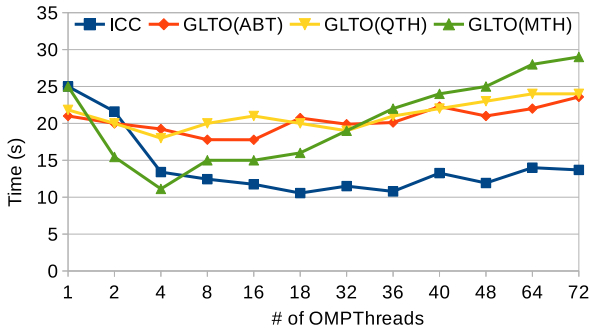

Fig. 13: Tiempo de ejecución del GC con granularidad 100 y runtimes OpenMP incrementando el número de hilos.

tarea. Estos resultados reflejan la media de 1.000 ejecuciones. Ya que un menor número de tareas implica menos sobrecoste del runtime, tiene sentido que el tiempo de ejecución se reduzca cuando pasamos de paralelismo de grano fino a grueso. Sin embargo, el tiempo de ejecución de GLTO es mucho menor que OpenMP de Intel para granularidades de 10 y 20 (Figuras 10 y 11, respectivamente). En este test, solamente cuando se utiliza Argobots como base, GLTO mantiene un rendimiento aceptable para granularidad de 50 (Figura 12). Si comparamos las opciones de GLTO entre ellas, observamos el efecto de las distintos detalles de implementación de las bibliotecas de HLs. Por una parte, GLTO(ABT) mantiene el rendimiento a pesar de incrementarse el número de hilos puesto que la interacción entre `GLT_threads` es casi inexistente. Sin embargo, GLTO(MTH) y GLTO(QTH) sufren contención. El primero por el WS y el segundo por la sincronización en cada palabra de memoria.

En el runtime de OpenMP de Intel, la diferencia de tiempo entre tareas de grano fino y de grano grueso es crítica. Sin embargo, este OpenMP muestra un buen rendimiento hasta 4 hilos en el escenario de granularidad 10 (Figura 10) y hasta 8 hilos para el problema de granularidad 20 (Figura 11) y 50 (Figura 12). Una vez este número de hilos se alcanza, el rendimiento del runtime de Intel cae. Esta bajada está causada por dos motivos: 1) la contención por el mecanismo de WS; y 2) un mecanismo interno del runtime de

TABLA III: Porcentaje de tareas encoladas para distintas granularidades.

Hilos	Granularidad			
OpenMP	10	20	50	100
1	100	100	100	100
2	80	93	84	100
4	88	81	63	100
8	90	97	39	100
16	94	100	100	100
18	94	100	100	100
32	95	100	100	100
36–72	100	100	100	100

que controla el número de tareas encoladas llamado cut-off. En este escenario, el hilo productor crea tareas en su propia cola mientras que los hilos consumidores ganan acceso a esa cola para robar tareas pendientes. Además, el mecanismo de cut-off se dispara una vez un cierto número de tareas son almacenadas en la cola—256 en el runtime de Intel— y entonces las nuevas tareas son ejecutadas directamente como código secuencial. Es conveniente remarcar que el tiempo de gestión de una tarea que no se encola es menor que una que ha de pasar por el planificador.

En cuanto a la granularidad más gruesa, al contrario que en los otros escenarios, el runtime de intel mejora el rendimiento de la GLTO (Figura 13). A pesar de que las tareas pasan todas por el planificador, el tiempo empleado en la ejecución de cada tarea reduce el WS al mínimo disminuyendo el problema de contención. En este caso, el comportamiento del OpenMP de Intel es parecido al del bucle `for`, y por lo tanto, el reparto de trabajo de GLTO disminuye el rendimiento. Como excepción, GLTO sobre MassiveThreads (GLTO(MTH)) mejora al resto de opciones hasta 4 hilos debido a su interno WS. Aunque el resultado mejore al reducir el número de tareas (incrementando su granularidad), parece contradictorio; se debe destacar que en las otras granularidades del problema, el código funciona mejor en secuencia (1 hilo) que en paralelo. Otro claro indicativo del rendimiento que ofrecen actualmente el paralelismo a nivel de tarea.

Para verificar el comportamiento del mecanismo de cut-off, hemos analizado ambas razones midiendo el número de tareas encoladas y el mecanismo de cut-off de forma independiente. Si la creación de tareas es más rápida que su consumo, el mecanismo de cut-off se dispara y el rendimiento no disminuye. Por contra, si la creación es más lenta que el consumo, el tamaño de la cola no llega al umbral y todas las tareas pasan por el runtime y baja el rendimiento. La Tabla III resume el porcentaje de las tareas que son encoladas para cada granularidad de tareas. Es remarcable que un reducido número de tareas no encoladas beneficia al rendimiento. Este hecho sugiere que el trabajo realizado en la gestión del paralelismo de tareas debe ser mejorado.

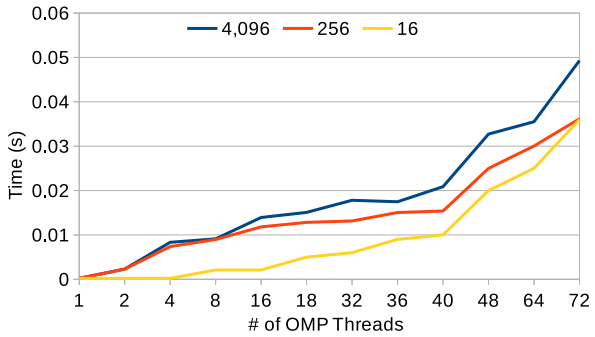

Fig. 14: Tiempo de ejecución para 4.000 tareas con tres valores distintos de cut-off con el runtime de OpenMP de Intel incrementando el número de hilos.

Adicionalmente, hemos implementado un test donde un hilo genera 4.000 tareas de OpenMP. Hemos ejecutado ese test con tres valores distintos que inician el mecanismo de cut-off: el valor por defecto (256), otro donde todas las tareas son encoladas (4.096) y una configuración donde prácticamente todas las tareas son ejecutadas directamente (16). La Figura 14 muestra el efecto de estos valores con el incremento del número de hilos de OpenMP que participan en el proceso. El tamaño más grande (etiquetado como 4,096) muestra el sobrecoste de la contención porque todas las tareas pasan por la cola de tareas y ocurre el WS. Por el contrario, el valor más pequeño ofrece un buen rendimiento hasta los 16 hilo,s ya que en ese punto el productor pasa a ser más lento que los consumidores y, por tanto, las tareas empiezan a encolarse y a aparecer la contención. Hasta los 8 hilos, el tiempo de ejecución es casi el mismo que al ejecutarse con 1 hilo. A pesar de que este valor obtiene mejor rendimiento que el valor por defecto, 16 tareas son muy pocas como límite de la cola.

En resumen, los resultados del OpenMP de Intel indican que no puede abordar el paralelismo de tareas de grano fino tan eficientemente como las soluciones basadas en HLs. Por lo tanto, para este tipo de escenarios, GLTO sería la implementación elegida.

VII. Selección de la Implementación de OpenMP Apropiada

Revisando los resultados presentados en este artículo, dos selecciones de OpenMP están bien definidas. Concretamente, los escenarios de código limitado por cómputo compuestos por bucles `for` beneficia a las implementaciones basadas en Pthreads, gracias a su mejor mecanismo de reparto de trabajo. Por el contrario, el paralelismo anidado se beneficia del uso de los HLs debido a su menor sobrecoste y, además, previene el problema de oversubscription.

Cuando se utiliza OpenMP como generador de hilos, la diferencia de rendimiento entre las distintas soluciones es inapreciable.

Para aplicaciones con paralelismo de tareas, hay distintos aspectos a tener en cuenta. Mientras que tareas de grano fino benefician a las implementaciones sobre HLs, las tareas de grano grueso benefician a las basadas en Pthreads. Además, el rendimiento para granularidades intermedias depende de dos factores: el númeor de hilos y la diferencia entre el tiempo empleado en la creación y en la ejecución de las tareas. Un número de hilos reducido obtiene mejor rendimiento en OpenMP de Intel debido a la menor contención en el WS. El segundo factor indicará si se alcanza el valor del cut-off o no. Las soluciones basadas en HLs no experimentan una caída de prestaciones en ninguno de los dos casos, puesto que solamente se ven afectados por la interacción entre los hilos del SO. De entre todas las soluciones de HLs, la que mejor prestaciones obtiene es Argobots.

VIII. Conclusiones

Hemos presentado una nueva implementación de OpenMP sobre la GLT API llamada GLTO [35]. GLT presenta una API común para bibliotecas de HLs y está implementada sobre Argobots, MassiveThreads, y Qthreads. LA GLTO nos permite comparar códigos escritos en OpenMP sobre distintas bibliotecas de HLs.

Hemos explicado las decisiones de diseño de la GLTO y hemos mostrado cómo el runtime afronta los distintos escenarios de OpenMP. Además, hemos validado nuestra implementación con el test OpenUH OpenMP Validation Suite 3.1, obteniendo mejores resultados que los runtimes de referencia GNU e Intel.

También hemos presentado una comparación justa entre nuestra implementación y los runtimes de GNU e Intel en 4 escenarios distintos: generador de hilos, código limitado por cómputo, paralelismo anidado y paralelismo de tareas. Para cada caso hemos explicado las diferencias existentes tanto a nivel de runtime como de rendimiento.

Nuestro estudio revela que no hay una solución OpenMP dominante. Mientras que las implementaciones clásicas ganan en código limitado por cómputo, las basadas en HLs ganan en paralelismo de tareas de grano fino y paralelismo anidado.

Como trabajo futuro, se quiere implementar el MP OmpSs sobre la API de GLT y estudiar la interacción entre MPI y los HLs.

Agradecimientos

Los investigadores de la Universitat Jaume I de Castelló son financiados por el proyecto TIN2014-53495-R de MINECO y FEDER, Beca FPI de la Generalitat Valenciana con el programa Vali+d 2015. Antonio J. Peña está cofinanciado por el Ministerio de Economía y Competitividad de España bajo la beca Juan de la Cierva número IJCI-2015-23266. Este trabajo ha sido financiado parcialmente por el U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research (SC-21), con el contrato DE-AC02-06CH11357.

Referencias

[1] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen Yang, "The sunway taihulight supercomputer: system and applications," *Science China Information Sciences*, vol. 59, no. 7, pp. 072001, 2016.

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

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

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

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

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

[7] Microsoft MSDN Library, "Fibers," .

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

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

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

[11] Laxmikant V Kale and Sanjeev Krishnan, *CHARM++: A portable concurrent object oriented system based on C++*, vol. 28, ACM, 1993.

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

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

[14] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Phil Carns, Adriaán Castelló, Damien Genet, Thomas Herault, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, and Pete Beckman, "Argobots: A lightweight threading/tasking framework," https://collab.cels.anl.gov/display/ARGOBOTS/, 2016.

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

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

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

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

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

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

[21] OpenMP Architecture Review Board, *OpenMP Application Programming Interface Version 4.5*, Nov. 2015.

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

[23] Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng, "Openuh: an optimizing, portable openmp compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, 2007.

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

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

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

[27] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst, "Forestgomp: an efficient openmp environment for numa architectures," *International Journal of Parallel Programming*, vol. 38, no. 5, pp. 418–439, 2010.

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

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

[30] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P Sadayappan, and Chau Tseng, "UTS: An unbalanced tree search benchmark," in *Languages and Compilers for Parallel Computing*, pp. 235–250. Springer, 2006.

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

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

[33] José I. Aliaga, Hartwig Anzt, Maribel Castillo, Juan C. Fernández, Germán León, Joaquín Pérez, and Enrique S. Quintana-Ortí, "Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors," *Conc. and Comp.: Practice and Experience*, vol. 27, no. 4, pp. 885–904, 2015.

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

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