Universidad Zaragoza
1542

Trabajo Fin de Grado

Análisis y Estudio de los Mecanismos de Sincronización Far Atomics en Multiprocesadores ARM

Analysis and Study of the Far Atomics Synchronization Mechanisms in ARM Multiprocessors

Autor

José Manuel Vidarte Llera

Directores

Darío Suárez Gracia

Alejandro Valero Bresó

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2021

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

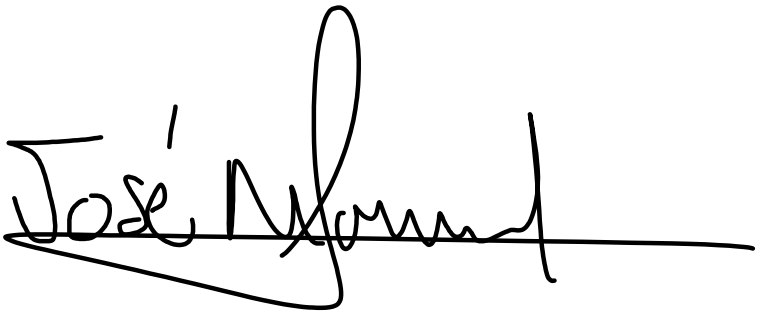
D./Dª. José Manuel Vidarte Llera,

con nº de DNI 73425128C en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Análisis y Estudio de los Mecanismos de Sincronización Far Atomics en Multiprocesadores ARM,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 17/11/2021



Fdo: José Manuel Vidarte Llera

AGRADECIMIENTOS

Dar las gracias a Darío y Alejandro por sus buenos consejos y por la buena atención recibida durante la realización de este proyecto.

RESUMEN

La latencia de acceso a memoria en aplicaciones paralelas con memoria compartida resulta un aspecto crítico en su desempeño en rendimiento. Este hecho resulta cada vez más común debido a la consolidación del paradigma multiprocesador y a la continua demanda de mayores recursos computacionales y de memoria por parte de las aplicaciones paralelas. En el ámbito de los multiprocesadores, esta tendencia supone que año tras año, además de desarrollar nuevos nodos tecnológicos más pequeños, también se implementen diversas mejoras a nivel de arquitectura y microarquitectura de estos chips.

Hace unos años, ARM introdujo su nueva microarquitectura ARMv8.1, la cual incluye un número considerable de cambios en su repertorio de instrucciones, destacando la extensión LSE (*Large System Extensions*) cuyo objetivo es optimizar las instrucciones atómicas, denominando a estas nuevas operaciones como *far atomics*.

El presente trabajo se ha enfocado principalmente en realizar un análisis de las operaciones *far atomics* respecto a otras más convencionales como son las operaciones *load-link/store-conditional*, haciendo uso de la *suite* de benchmarks INNCABS, la cual consta de aplicaciones con tareas paralelas de grano fino.

La primera contribución de este trabajo es un análisis en profundidad acerca de la capacidad del *runtime* de C++ para la creación y gestión de tareas paralelas en aplicaciones muy demandantes, como es el caso de los benchmarks INNCABS, comparando esta solución frente a la utilización de una biblioteca de terceros como son los *Threading Building Blocks* (TBB) de Intel. TBB gestiona estas funciones mediante un *thread pool*. El uso de una biblioteca externa al estándar C++11 como TBB resulta muy conveniente para la gestión de tareas de grano fino, puesto que el *runtime* de C++ supone una sobrecarga muy grande que lastra el rendimiento de estas aplicaciones. En concreto, las implementaciones en C++ mediante la directiva `std::async` requieren lanzar un proceso por tarea, mientras que con TBB la cantidad de procesos (hilos) se puede regular, siendo óptimo un proceso por cada núcleo físico del sistema, provocando una mejora en los tiempos de ejecución de hasta tres órdenes de magnitud en algunos escenarios.

Por otro lado, la comparativa de *far atomics* frente a *load-link/store-conditional* no muestra suficientes diferencias favorables en rendimiento hacia el nuevo conjunto de operaciones atómicas. No obstante, en este documento se razonan ciertas hipótesis en detalle por las que se cree que se han obtenido estos resultados. Por ejemplo, una primera limitación puede ser el procesador objeto de estudio, el cual utiliza la primera versión del conjunto de instrucciones LSE y quizás no disponga en realidad de soporte

far atomics. Otras hipótesis son la falta de contención entre la sincronización de las tareas paralelas o la limitación de enlazar los benchmarks con una única versión de librería *glibc* con soporte *far atomics*.

Índice

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

En este primer capítulo se pretende introducir el interés y la motivación tras este estudio sobre las operaciones far atomics, así como la metodología y la organización que se ha seguido para cumplir con los objetivos de este trabajo.

1.1. Motivación

Hoy en día, la paralelización de aplicaciones resulta prácticamente indispensable de cara a aprovechar los recursos hardware de los que disponen la gran mayoría de dispositivos modernos, ya que en general, incluyen varios núcleos físicos. La forma de acceder a la memoria compartida por parte de los procesos que componen las aplicaciones paralelas puede ser determinante en el rendimiento que desempeñan estas aplicaciones al ejecutarse sobre distintos tipos de sistemas. En especial, estos accesos a memoria son muy importantes y resultan críticos en aplicaciones que requieren acceder a memoria de forma exclusiva, esto es, mediante operaciones atómicas. Por ello, resulta de interés conocer las últimas implementaciones sobre los mecanismos de acceso a memoria sobre hardware moderno y evaluar si realmente suponen un avance respecto a otros mecanismos más convencionales.

El presente trabajo pretende analizar las nuevas operaciones atómicas *far atomics* introducidas en la microarquitectura ARMv8.1, la cual es una extensión de ARMv8 que añadió un gran número de cambios en el repertorio de instrucciones, como es la inclusión de lo que ARM denomina como LSE (*Large System Extensions*). Estas extensiones están dirigidas a optimizar las instrucciones atómicas, reemplazando las antiguas instrucciones exclusivas *load-store* por instrucciones simples como CAS (*Compare-And-Swap*) o SWP (intercambio), mediante una operación *far atomic*. Estas extensiones son conocidas por aumentar de forma inherente el rendimiento de las aplicaciones que utilizan accesos atómicos a memoria.

1.2. Objetivos y Alcance

El principal objetivo de este estudio es verificar si las nuevas instrucciones *far atomics* de ARM realmente suponen un impacto notorio en el rendimiento de las aplicaciones paralelas, en concreto, de tareas de grano fino. Para ello, se ha escogido la *suite* de benchmarks INNCABS, sobre la cual se analiza la utilización de estas instrucciones respecto a otras más convencionales como *load-link/store-conditional*. Además, dado que la suite está compuesta en su mayoría por aplicaciones con mucha carga a nivel de creación de tareas, también se analiza la eficiencia de C++ con la gestión de tareas de grano fino y se estudia si existe alguna alternativa que realice mejor esta función. El objetivo de este análisis sobre la gestión de tareas es reducir al máximo la sobrecarga que esto produce, de manera que el tiempo obtenido al ejecutar las aplicaciones sea en su mayor parte debido al desarrollo de las tareas en sí y no a su gestión y planificación.

1.3. Organización y Desarrollo del Trabajo

El desarrollo del presente trabajo se inicia con la realización de un estudio sobre el funcionamiento de los mecanismos de coherencia y consistencia en memoria compartida, para posteriormente comprender con mayor facilidad y en detalle el funcionamiento de las instrucciones atómicas típicas como *load-link/store-conditional* y las nuevas instrucciones *far atomics* objeto de estudio. Seguidamente se instala un compilador con soporte a *far atomics*, en este caso, GCC 10 en su version 10.1.0 (estas operaciones también son soportadas por GCC 7 aunque de forma menos eficiente, mejorándose en versiones 10.1 o superiores), y se compila la *suite* de benchmarks INNCABS.

Una vez establecido el entorno experimental, se realiza un análisis sobre las aplicaciones de la *suite* para comprobar si se pueden desarrollar mejoras con el objetivo de aumentar el rendimiento y poder analizar de manera más adecuada el impacto del uso de unas operaciones atómicas u otras. Sobretodo, resulta conveniente analizar el desempeño que realiza C++ con la gestión de multitud de tareas de grano fino y estudiar si existe alguna alternativa que ofrezca mayor rendimiento, como es el uso de un *thread pool* para este cometido. En este sentido, se utiliza la biblioteca TBB de Intel para implementar algunas de las aplicaciones de la *suite* con un *thread pool*.

Por otro lado y de forma paralela a la implementación de mejoras, se desarrollan una serie de *scripts* para automatizar la ejecución de los experimentos y la obtención de resultados. Para obtener una representación gráfica de los experimentos también se desarrollan varios *scripts* en Python.

Por último, la experimentación del trabajo también se ha llevado a cabo, en parte, al mismo tiempo que se implementan mejoras y se desarrollan los *scripts*, aunque se ha experimentado más intensivamente tras disponer de una versión final de las aplicaciones para obtener los resultados y conclusiones finales.

Para conocer detalles sobre el tiempo dedicado al proyecto véase el Anexo E.

1.4. Estructura de la Memoria

El resto del documento se ha estructurado en cinco capítulos adicionales. En el Capítulo 2 se detalla una descripción previa de las instrucciones atómicas *load-link/store-conditional* y *far atomics*, así como un estudio del soporte software que ofrece C++ a la ejecución paralela de tareas de grano fino y posibles implementaciones, junto con una introducción a la *suite* de benchmarks INNCABS. El Capítulo 3 describe las características principales de los sistemas utilizados para la experimentación y la metodología que se ha seguido para realizar los experimentos y obtener los resultados más relevantes, así como las tecnologías utilizadas. El Capítulo 4 se dedica a profundizar en las mejoras implementadas en las aplicaciones a evaluar. En el capítulo 5 se muestran y analizan todos los experimentos realizados, tanto la evaluación de las mejoras implementadas como el impacto en el rendimiento de las instrucciones *far atomics*. Por último, el Capítulo 6 recoge las conclusiones obtenidas tras el análisis de los resultados, posibles extensiones que podrían realizarse en un futuro sobre este trabajo y un diagrama de Gantt que muestra la organización junto con la dedicación del autor.

Capítulo 2

Antecedentes

Este capítulo explica los conceptos necesarios para facilitar la comprensión del presente trabajo, centrándose en qué son y qué problema resuelven las operaciones atómicas como load-link/store-conditional o far atomics, y por qué son necesarias. También se detalla cómo se gestiona la concurrencia de tareas en un sistema multiprocesador, en particular, la gestión que realiza C++ y otras posibles implementaciones como la utilización de un thread pool.

2.1. Operaciones Atómicas

Para escribir código multi-hilo, un programador debe saber cómo sincronizar varios hilos, lo que generalmente incluye la ejecución de operaciones de forma atómica. Algunas de estas instrucciones atómicas son *read-modify-write* (RMW), *test-and-set*, *fetch-and-increment* o *compare-and-swap*, entre otras. Estas instrucciones resultan críticas para una correcta sincronización y son utilizadas para implementar herramientas de más alto nivel como los *spin-locks* y otras primitivas. Para que estas instrucciones sean atómicas, las *loads* y *stores* deben ejecutarse de forma sucesiva siguiendo el orden total de las operaciones requeridas por la *consistencia secuencial*. Un modelo de consistencia en memoria define qué comportamientos están permitidos en la ejecución de programas concurrentes con memoria compartida, en concreto, la consistencia secuencial fuerza a que cada núcleo cumpla con la ejecución de las instrucciones en orden de programa para asegurar un estado consistente en memoria. Para conocer más detalles sobre mecanismos de coherencia y consistencia en memoria véase el Anexo A.

Una manera sencilla, aunque muy poco eficiente, de implementar operaciones atómicas consiste en que el núcleo que quiere realizar esta operación bloquee el acceso a memoria al resto de núcleos y realice sus operaciones de lectura y escritura sobre ésta. Existen otras implementaciones más agresivas que mejoran el rendimiento, por ejemplo,

```
loop:
    ldaxr   w2, [@lock]
    cbnz    w2, loop
    mov     w3, #1
    stxr    w4, w3, [@lock]     # stxr devuelve estado 0 en w4
            # si el store fue exitoso, 1 en caso contrario.
    cbnz    w4, loop            # Si el resultado del store no es 0
            # lo vuelve a intentar, si es 0 ha adquirido el mutex.
        # SECCION CRITICA
    stlr    wzr, [@lock]        # Libera el mutex.
```

Listing 2.1: Ejemplo de implementación de un *spin-lock* mediante operaciones *load/store exclusive*.

para el caso de la operación RMW. Si el bloque se encuentra en estado *Shared*, el núcleo puede lanzar la lectura de forma especulativa mientras el controlador de cache gestiona la petición de cambio del bloque a modo *Modified* para que el núcleo pueda escribir. Los estados *Shared* y *Modified* componen una implementación básica del modelo de consistencia secuencial con coherencia en cache en el que un bloque en estado *Shared* indica que varios núcleos lo contienen en sus caches privadas luego solo se puede leer, y el estado *Modified* indica que se tiene la copia más reciente y se puede escribir sobre él. Véase el Anexo A para saber más sobre esta implementación.

A continuación, la explicación se centra en la implementación de las operaciones atómicas en ARMv8, presentes en la máquina objeto de estudio en este trabajo. ARM es una de las arquitecturas con una semántica de consistencia más relajada.

2.1.1. *Load-link/Store-conditional*

La arquitectura de ARMv8 implementa la operación atómica RMW sin bloqueos mediante el par de instrucciones *load-link/store-conditional*[1]. La instrucción *load-link* o *load-exclusive* (LDXR) carga un valor de una dirección de memoria e intenta solicitar un bloqueo exclusivo sobre la dirección. Por otro lado, la instrucción *store-conditional* o *store-exclusive* (STXR) escribe un nuevo valor en esa dirección sólo si el bloqueo fue conseguido por la *load* anterior y éste se ha mantenido, es decir, si no se han ejecutado operaciones de acceso a memoria sobre esa dirección desde la ejecución de *load-link*. Puede verse un ejemplo de implementación de un *spin-lock* mediante estas instrucciones en el *Listing* 2.1 [2].

2.1.2. *Far Atomics*

[1] https://developer.arm.com/documentation/den0024/a/The-A64-instruction-set/Memory-access-instructions/Synchronization-primitives

```
loop:
    ldr     w2, [@lock]
    cbnz    w2, loop
    add     w3, w2, #1
    casal   w2, w3, [@lock]     # Compare and swap atomico,
            # compara el mutex con el 0 inicial y si se mantiene,
            # escribe un 1 para adquirirlo.
        # SECCION CRITICA
    stlr    wzr, [@lock]        # Libera el mutex.
```

Listing 2.2: Ejemplo de implementación de un *spin-lock* mediante operaciones *far atomics*.

Con la introducción de la arquitectura ARMv8.1 se añade soporte para instrucciones atómicas que pueden usarse como una alternativa a las instrucciones *load-exclusive/store-exclusive*, por ejemplo, para facilitar la implementación de actualizaciones atómicas de memoria en sistemas grandes[2]. Estas operaciones proporcionan una alternativa para que una actualización atómica de registros con un contenido de memoria se pueda realizar a nivel de cache e incluso más lejos en el sistema de memoria, mientras que las instrucciones *load-exclusive/store-exclusive* siempre cargan los datos en la cache L1. Se les denomina comúnmente *far atomics* ó *near atomics* para el caso en el que los datos residen en la cache L1. Inicialmente, la operación atómica se envía al controlador de cache L1 como *near atomic*. Si falla o el bloque está en estado *Shared*, entonces la operación se envía como *far atomic* al nivel inferior de la jerarquía de memoria.

Debido a la alta utilización de las instrucciones atómicas en la paralelización de tareas, en este trabajo se pretende analizar el impacto que puede suponer en el rendimiento el uso de estas nuevas operaciones *far atomics* respecto al uso de otras instrucciones atómicas tradicionales como son las *load-link/store-conditional*, observando de esta forma si realmente suponen un avance o en qué tipo de tareas puede ser conveniente su uso. El *Listing* 2.2 muestra un ejemplo de implementación de un *spin-lock* mediante instrucciones *far atomics*.

2.2. Soporte Software a la Ejecución Paralela de Tareas

Asumiendo que se dispone de un sistema con múltiples núcleos físicos, en esta sección se van a tratar algunas ideas básicas sobre el paralelismo, así como los mecanismos de soporte software que permiten la ejecución paralela de tareas y cómo

[2] https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/the-armv8-a-architecture-and-its-ongoing-development

se gestiona su planificación.

En arquitectura de computadores, la Ley de Amdahl es una fórmula que expresa la aceleración teórica en términos de latencia a la ejecución de una tarea con una carga de trabajo fija en un sistema cuyos recursos pueden mejorar, en el caso de la computación paralela, la cantidad de núcleos hardware que se utilizan para ejecutar esa tarea. Esta ley viene expresada mediante la Fórmula 2.1:

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.1}$$

$S_{latency}$ es la aceleración teórica de la ejecución de toda la tarea, s es la aceleración de la parte de la tarea que es paralelizable y p es la proporción del tiempo de ejecución de la parte de la tarea que es paralelizable respecto al tiempo total.

Tras la explicación anterior, es fácil deducir que la clave para conseguir un buen *speedup* es paralelizar todas partes de una tarea que lo permitan, manteniendo a los hilos de nivel de usuario ocupados con tareas continuamente. Para esto, es necesario disponer de tareas de grano más fino, así se pueden tener muchas tareas en ejecución al mismo tiempo y de esta forma maximizar la paralelización. Los sistemas operativos modernos proporcionan hilos a nivel de usuario que abarcan un contador de programa y una pila (un hilo que incluye su propio espacio de direcciones se suele denominar proceso). El kernel del sistema operativo incluye un *scheduler* que ejecuta los hilos en los núcleos físicos del procesador. La aplicación, sin embargo, generalmente no tiene control sobre el mapeo entre hilos y núcleos físicos, luego no puede controlar la planificación de los hilos.

Una forma de eliminar el vacío entre los hilos de nivel de usuario y los núcleos a nivel de sistema operativo es presentar al programador un modelo de tres niveles como se muestra en la Figura 2.1. En el nivel más alto, los programas multihilo descomponen una aplicación en un número de tareas que varía de forma dinámica. En el nivel medio, un *scheduler* a nivel de usuario mapea estas tareas en hilos de sistema operativo. En el último nivel, el kernel mapea estos hilos a los núcleos físicos del procesador. Es este último nivel el que está fuera del control de la aplicación.

Tradicionalmente, los sistemas UNIX, plataforma sobre la que se ha desarrollado el presente estudio, ofrecen soporte a un único hilo de ejecución por proceso, aunque en versiones más modernas se incluye soporte para múltiples hilos a nivel de kernel por proceso. Debido a que las versiones tradicionales no tienen soporte para multihilo, las aplicaciones necesitan ser escritas mediante funciones de librería a nivel de usuario, siendo la más popular la librería *pthread*, que mapea cada hilo en un proceso a nivel de kernel.

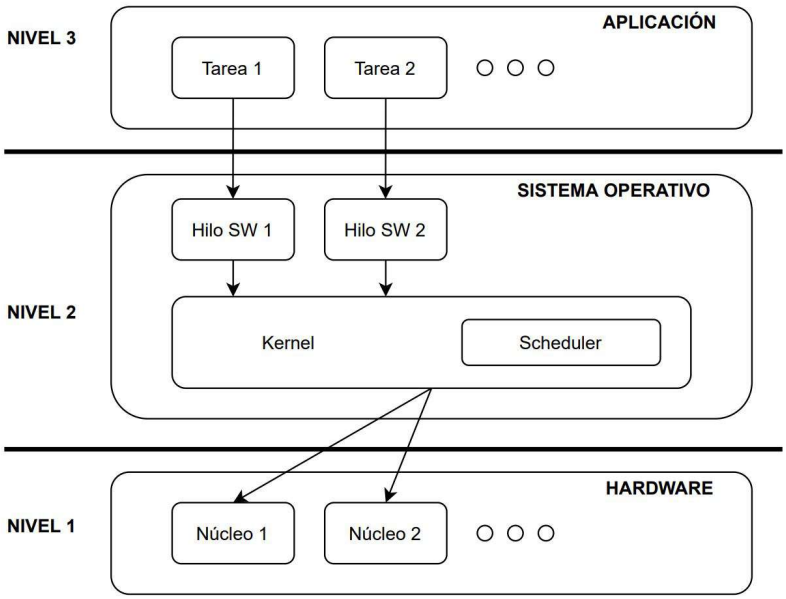


Figura 2.1: Esquema de niveles en la creación y ejecución de tareas.

Linux propone una solución que no distingue entre hilos y procesos, en la que los hilos a nivel de usuario son mapeados en procesos a nivel de kernel. Múltiples hilos a nivel de usuario que constituyen un único proceso de usuario son mapeados a procesos a nivel de kernel que comparten el mismo ID de grupo (esto permite que compartan recursos como ficheros y memoria y evita cambios de contexto cuando el *scheduler* cambia entre procesos del mismo grupo).

Un nuevo proceso se crea en Linux clonando el proceso actual, de forma que comparte recursos como, por ejemplo, la memoria virtual. De esta forma, los procesos funcionan como hilos dentro de un único proceso. Sin embargo, no se define ningún tipo de estructura de datos independiente para cada hilo. En vez de utilizarse la llamada al sistema común `fork()`, los procesos en Linux se crean utilizando la llamada al sistema `clone()`, de forma que copian los atributos del proceso actual (`fork()` es implementada en Linux como `clone()` con todos sus *flags* sin marcar).

Mientras que los procesos clonados que forman parte del mismo grupo pueden compartir el mismo espacio de memoria, estos no pueden compartir las mismas pilas de usuario, luego la llamada `clone()` crea espacios de pila separados para cada proceso.

2.2.1. Ejecución Concurrente con C++

Todo programa de C++ tiene al menos un hilo, que es iniciado por el *runtime* de C++ (es el hilo que ejecuta la función *main*). A partir de este, el programa puede lanzar hilos adicionales que tengan otra función como punto de entrada, los cuales se ejecutan concurrentemente entre ellos. Cuando el programa acaba (la función de punto

de entrada retorna) el hilo creado también acaba. Desde el programa en el cual se lanzan estos hilos también puede esperarse a que acaben.

La forma de lanzar nuevos hilos en C++ [3] es mediante la creación de un objeto `std::thread`, especificando la tarea a ejecutar en ese nuevo hilo. Al crearse el hilo, la función de punto de entrada se copia en la estructura del nuevo hilo y se invoca desde allí. Una vez lanzado el hilo es necesario especificar si se va a esperar a que termine o se deja que se ejecute por su cuenta. Si no se decide antes de que el objeto `std::thread` sea destruido, entonces el programa se termina (el destructor de `std::thread` invoca a `std::terminate`).

El *Listing* 2.3 hace referencia a un ejemplo de creación de un hilo. Si se desea esperar a que la ejecución de los hilos acabe, ha de invocarse la función `join()` sobre cada uno, así se asegura que los hilos acaban antes de que termine el programa principal. Por otro lado, puede hacerse que los hilos pasen a ser ejecutados en segundo plano invocando la función `detach()`. Una vez invocada esta función, el hilo pasa a no poder ser referenciado por un objeto `std::thread`, luego ya no se le puede esperar, y se cede su propiedad y control al *runtime* de C++, que asegura que los recursos asociados a este hilo se liberan correctamente cuando termina su ejecución.

El hecho de esperar a que acaben varios hilos de manera activa con `join()` puede ser bastante ineficiente en algunos casos, como por ejemplo, cuando un programa que lanza varios hilos con tiempos de ejecución muy dispares tiene esperar a que todos acaben para seguir con su ejecución, penalizado al programa con el hilo que más tarda (el programa principal podría estar realizando otra tarea mientras espera). Para paliar este efecto, la librería estándar de C++ propone el concepto de *future* como método para la ejecución asíncrona de tareas.

Si un hilo necesita esperar a un evento específico, de alguna manera recibe un *future* que representa ese evento. De esta forma, un hilo puede consultar periódicamente sobre el *future* cada poco tiempo para ver si el evento ha ocurrido mientras sigue ejecutando otras tareas. Alternativamente, un hilo después de lanzar un *future* puede ejecutar otras tareas hasta que requiera que ocurra el evento para poder continuar (en vez de consultarlo periódicamente), únicamente en ese momento consultará que el *future* esté *ready*. Ambas situaciones representan un caso de ejecución asíncrona (la finalización de la tarea no se requiere inmediatamente), en la que utilizar un modelo como *fork-join* resulta innecesario y poco eficiente. En este caso resulta más interesante utilizar la directiva `std::async`, que lanza una tarea de forma asíncrona y en vez de devolver un objeto `std::thread` sobre el que esperar retorna un objeto `std::future`, el cual devolverá el resultado eventualmente. En el caso en el que se requiera del resultado en cierto momento, la llamada `get()` sobre el *future* bloquea el hilo hasta que el *future*

```
void func(){}

int main(){
    bool wait_thread = true;
    ...
    std::thread th(func);
    if(wait_thread) th.join();
    else{
        th.detach();
        assert(!th.joinable());
    }
}
```

Listing 2.3: Ejemplo de creación de un hilo.

está *ready*, devolviendo el resultado.

El evento más básico al que puede aplicarse el recurso *future* es al resultado de un cálculo que ha sido lanzado en segundo plano en otro hilo. Un inconveniente que tienen los `std::thread` es que no proporcionan ningún mecanismo sencillo para devolver un valor de una tarea (debe hacerse mediante variables compartidas), sin embargo, esto se simplifica con *futures* y la directiva `std::async`.

2.2.2. Posibles Implementaciones de `std::async`

Para el caso de la *suite* de benchmarks INNCABS[3] utilizada en el presente trabajo (véase la Sección 2.3), la paralelización se realizada mediante `std::future` y la directiva `std::async`, en concreto, este análisis se centra en la ejecución con la política `std::launch::async`, la cual fuerza la creación de un nuevo hilo para cada tarea y se ejecutan de forma asíncrona. Para la obtención de resultados se utilizan sobre los *futures* las funciones `wait()` (si no se requiere del resultado para continuar) o `get()` (si es necesario esperar al resultado para continuar la ejecución).

Ya que el número de hilos que crean algunas aplicaciones de la *suite* es muy elevado, también se va a explorar otras posibles implementaciones que puedan reducir el sobrecoste que produce la creación excesiva de hilos, como es la utilización de un *thread pool*.

Hilos de Sistema Operativo. Por defecto, como ya se ha comentado, las aplicaciones de la *suite* fuerzan la creación de un nuevo hilo para cada tarea. Esto se traduce, a nivel de sistema operativo (Linux), en la invocación a la llamada al sistema `clone()`, la cual crea un nuevo proceso clonando el proceso `main()`, copia la función de punto de entrada en la estructura del proceso clonado y la invoca.

[3] https://github.com/PeterTh/inncabs

A partir de aquí, es el sistema operativo el encargado de planificar la ejecución de los procesos que han sido lanzados por la aplicación, los cuales marcarán el *future* a `std::ready` cuando hayan finalizado y tras su espera en el proceso `main()` serán destruidos.

Tareas de Biblioteca: Caso TBB. Otra posible implementación sobre la creación de hilos para la ejecución paralela de tareas es el uso de un *thread pool*. En este caso se va a analizar el ejemplo concreto de la librería TBB (*Thread Building Blocks*, la cual fue diseñada para soportar de manera eficiente el paralelismo de forma anidada, concurrente y secuencial, así como para mapear dinámicamente este paralelismo en una plataforma en concreto.

Todos los algoritmos genéricos de TBB comprenden una serie de patrones paralelos que, a diferencia de `std::async`, son de más alto nivel. Todos estos patrones comienzan desde un solo hilo de ejecución y cuando se encuentra uno de estos algoritmos paralelos, éste divide el trabajo asociado al algoritmo entre múltiples hilos. Cuando todas las partes del trabajo se han realizado, la ejecución se vuelve a unir y continúa la ejecución en el único hilo inicial. Hasta este punto, el funcionamiento de los algoritmos paralelos de TBB parece ser similar al modelo tradicional *fork-join*. Sin embargo, es principalmente en la creación y planificación de hilos donde ambas implementaciones difieren. Con TBB, cuando un hilo encuentra un algoritmo paralelo, por ejemplo, la paralelización de un bucle, éste crea un número suficientemente grande de hilos (con llamadas al sistema `clone()`) para la cantidad de núcleos lógicos del sistema y la cantidad de tareas que se quieren lanzar a ejecución a la vez. Una vez se ha creado la *pool* de hilos, el rango del bucle se divide en trozos de iteraciones y cada uno de estos se convierte en una tarea que es planificada una a una sobre los hilos que participan en la ejecución del algoritmo. Este ejemplo sería el caso de la directiva `tbb::parallel_for`.

La librería TBB es la encargada de la planificación de las tareas, a diferencia de con la librería estándar C++, en la que es el sistema operativo el encargado de la planificación. Por defecto, TBB inicializa su *scheduler* con el que es generalmente el número apropiado de hilos a utilizar. Se crean tantos hilos de trabajo como el número de núcleos lógicos del sistema menos uno, dejando de esta manera uno de los núcleos disponible para la ejecución del hilo principal de la aplicación. Como la distribución de las tareas sobre estos hilos de trabajo se realiza de manera muy eficiente por el *scheduler* de TBB, lo más recomendable suele ser disponer de un hilo software por cada núcleo lógico del sistema.

Las dos características de TBB responsables de su componibilidad son su *global thread pool* y las *task arenas*. La Figura 2.2 muestra los componentes de TBB y cómo

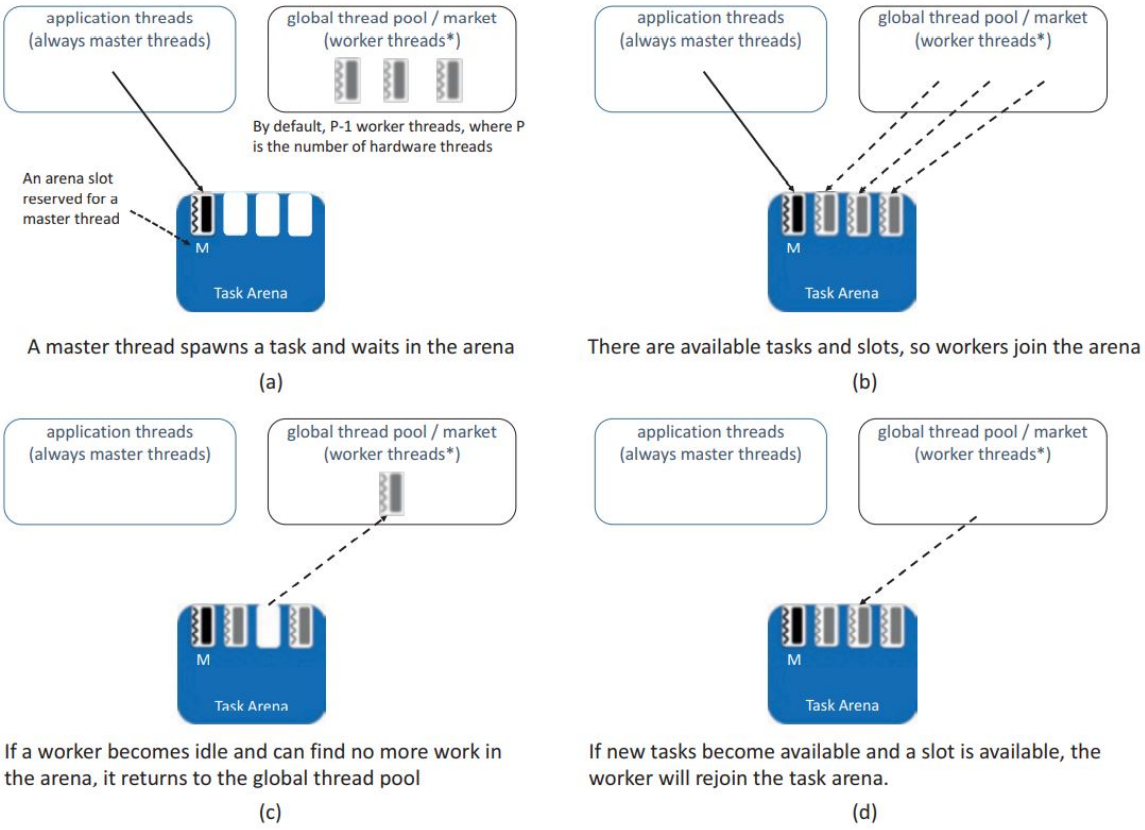


Figura 2.2: Ejemplo de un único hilo principal en el que TBB crea $N - 1$ hilos de trabajo (siendo N el número de núcleos lógicos del sistema). Figura tomada de [1].

se comportan. Inicialmente, cada hilo del *global thread pool* está dormido esperando una oportunidad para participar en el trabajo paralelo y a cada hilo de aplicación que usa TBB se le otorga su propia *task arena* para aislar su trabajo del trabajo de otros hilos de la aplicación `(a)`. Cuando un hilo ejecuta un algoritmo paralelo de TBB, éste lanza un *dispatcher* vinculado a su *task arena* hasta que el algoritmo finaliza. El hilo principal de la aplicación se une a la *task arena* y comienza a crear tareas. En ese momento, los hilos de trabajo se despiertan y migran a la *task arena* `(b)` en la que su *dispatcher* participará en la ejecución de tareas que han sido creadas por el hilo principal mientras las haya. Cuando ya no hay más trabajo en esa *arena*, el hilo de trabajo vuelve a la *pool* global `(c)`, aunque volverá a ella si el hilo principal vuelve a crear nuevas tareas `(d)` [1].

2.3. Suite de Benchmarks INNCABS

El conjunto de aplicaciones *The Innsbruck C++11 Async Benchmark Suite* (INNCABS) se trata de una *suite* diseñada para evaluar el rendimiento y la

```
template< class Function, class... Args >
std::future< typename std::result_of< Function(Args...) >::type >
async( std::launch policy, Function&& f, Args&&... args );
```

Listing 2.4: Definición de la plantilla de la función *async*.

escalabilidad del paralelismo en las implementaciones del estándar C++11 [4]. Además, la variedad de benchmarks engloba distintos tipos de estructuras de paralelización, tareas con diferente granularidad y requisitos de sincronización. Se ha elegido esta *suite* como benchmarks de prueba para el desarrollo de este trabajo ya que se pretende evaluar el impacto de las instrucciones *far atomics* en las características de programación concurrente de grano fino de C++11.

La *suite* INNCABS está compuesta por 14 benchmarks. La mayoría de estos benchmarks están basados en adaptaciones del código de otras *suites* como la Barcelona OpenMP Task Suite[4], utilizada ampliamente para testear la paralelización de tareas. La experimentación de este trabajo se centra en cuatro aplicaciones de INNCABS como son *Round*, *QAP*, *Floorplan* y *NQueens*. Se ha decidido utilizar estas cuatro aplicaciones en concreto debido a que a priori son las que más operaciones atómicas realizan al requerir una mayor sincronización durante la ejecución. Además, todas ellas ejecutan tareas de granularidad baja o moderada, favoreciendo que el impacto de las mejoras en los mecanismos de sincronización sea más notable en el tiempo de ejecución. En el Anexo B puede verse una descripción detallada de las aplicaciones seleccionadas.

En particular, las primitivas `std::async` y `std::lock` son las que resultan más interesantes a evaluar desde el punto de vista de la calidad de una implementación de librería, puesto que el estudio de estas primitivas permite apreciar mayores diferencias respecto a otros mecanismos de librería estándar. Generalmente, la librería de hilos estándar mapea directamente en hilos del sistema operativo, lo que provoca que el rendimiento se vea afectado mayormente por el sistema operativo y no tanto por el control de la implementación de la librería. Sin embargo, el comportamiento de `std::async` es distinto, ya que permite lanzar una función dada (f) con unos argumentos (*args*) conforme a una política establecida (*policy*) y devuelve una instancia de la clase `std::future`, la cual se puede utilizar para encolar los resultados asíncronamente[5]. El parámetro *policy* es de tipo `std::launch`. De acuerdo con este parámetro, la política puede lanzar un nuevo hilo para ejecutar la tarea de forma asíncrona (`std::launch::async`) o ejecutar la tarea en el mismo hilo que la invoca cuando se requiere su resultado por primera vez (`std::launch::deferred`)[6]. El

[4] https://github.com/bsc-pm/bots
[5] https://en.cppreference.com/w/cpp/thread/async
[6] https://en.cppreference.com/w/cpp/thread/launch

Listing 2.4 muestra la definición de *std::async*.

Capítulo 3

Metodología

Este capítulo muestra las características principales de las máquinas utilizadas en la experimentación e incluye una descripción detallada del método establecido para el análisis de los resultados. Además, también se nombran las herramientas utilizadas para el desarrollo y la automatización de los experimentos y para la representación gráfica de los resultados.

3.1. Descripción de las Máquinas Utilizadas

Dado que la arquitectura ARM es la que ofrece soporte a las instrucciones *far atomics*, objetivo de estudio de este trabajo, se ha utilizado la máquina *Pilgor* facilitada por el Departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza. El acceso a esta máquina se realiza mediante conexión remota a través del protocolo SSH. Además, también se utiliza una máquina local con arquitectura x86 como banco de pruebas adicional para testear el desarrollo de nuevas características en las aplicaciones, así como la realización de otros experimentos. La Tabla 3.1 muestra un resumen de las principales características técnicas de ambos sistemas.

Puesto que la mayoría de experimentos se realizan sobre *Pilgor*, la Figura 3.1 muestra en detalle la arquitectura del procesador de este sistema. Puede apreciarse cómo los núcleos están distribuidos en clústers (CCL), donde cada CLL cuenta con cuatro núcleos. Estos núcleos comparten el vector de etiquetas de la cache L3. En total, se implementan ocho clústers por chip (SCCL). Además, también cabe destacar que la conexión entre los clústers de núcleos de los dos chips se realiza mediante un bus dedicado denominado *InterChip Bus*. Este último punto es muy relevante de cara al análisis de las aplicaciones, ya que la necesidad de comunicar datos entre los dos chips podría suponer un impacto negativo en el rendimiento.

Característica	*Pilgor*	Máquina local
Procesador	Kunpeng 920-4826	i7-8550U
Núm. Sockets	2	1
Arquitectura	ARM 64 bits	x86 64 bits
Litografía	7nm de TSMC	14nm de Intel
Núm. núcleos/procesador	48	4
SMT[1]	No	Sí
Frecuencia del procesador	2.6 GHz	1.80 GHz (4.00 GHz turbo)
Cache L1 (privada)	6 MiB (64 KiB/núcleo)	256 KiB (64 KiB/núcleo)
Cache L2 (privada)	24 MiB (512 KiB/núcleo)	1024 KiB (256 KiB/núcleo)
Cache L3 (compartida)	48 MiB	4096 KiB
Memoria RAM	320 GB DDR4	16 GB DDR4
Frecuencia memoria	2933 MHz	2400 MHz
Canales de memoria[2]	8	2
Sistema operativo	CentOS 8.4.2105	Ubuntu 18.04 LTS

Tabla 3.1: Especificaciones técnicas de las máquinas utilizadas.

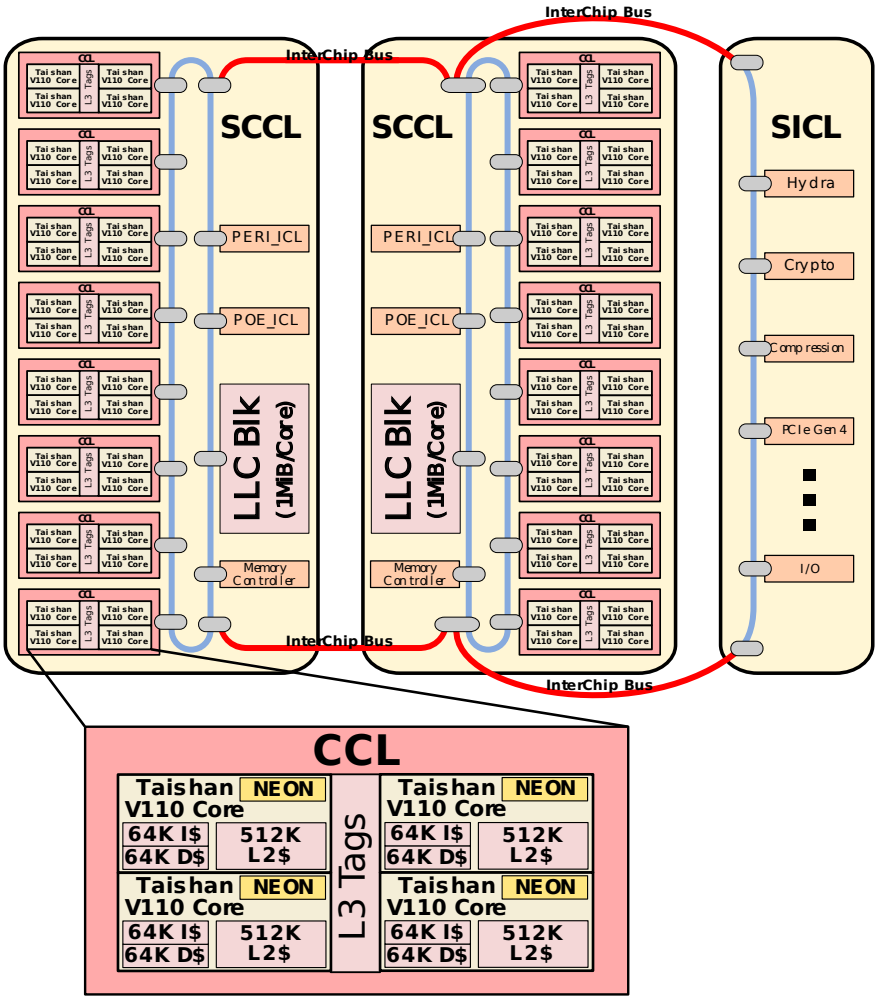


Figura 3.1: Arquitectura del procesador de *Pilgor*. Figura tomada de https://en.wikichip.org/wiki/hisilicon/microarchitectures/taishan_v110

3.2. Realización de Experimentos y Obtención de Resultados

En este trabajo se diferencia entre dos tipos de experimentos para evaluar el impacto de la utilización de las instrucciones *far atomics* frente a el uso de otras instrucciones atómicas más convencionales, así como la implementación de la creación y gestión de tareas del estándar C++11 frente al uso de una librería como TBB. Por un lado, se prueba la escalabilidad de las aplicaciones manteniendo un tamaño fijo del problema (cantidad de hilos software que se crean) y variando la cantidad de hilos hardware que se disponen, y por otro, se varía el tamaño del problema en proporción a la cantidad de hilos hardware disponibles.

En ambos enfoques, es necesario controlar la cantidad de hilos hardware de los que dispone la aplicación en cada ejecución. Desactivar hilos hardware de un sistema implica la necesidad de disponer de permisos de administración, por lo que para el caso de la máquina *Pilgor* no ha sido posible. Como alternativa, se propone la utilización de una herramienta de alto nivel como es la afinidad de los hilos a un núcleo lógico en concreto. Esto se consigue mediante la función `pthread_setaffinity_np()` la cual fuerza a que el proceso que la ejecuta migre al núcleo lógico que se le indica. El *Listing* 3.1 se muestra la implementación de esta característica.

En el caso de TBB, establecer la afinidad es un poco más complejo ya que requiere crear una clase que hereda de `tbb::task_scheduler_observe` y modificar su función `on_scheduler_entry()` para que realice la migración a un núcleo lógico cuando la tarea es planificada por TBB, como se aprecia en el *Listing* 3.2.

Para la experimentación en la máquina local (x86), a pesar de tener permisos de administración, también se utiliza el mismo mecanismo con el fin de mantener la metodología. Además, el hecho de forzar a que las tareas siempre migren a los mismos núcleos lógicos puede favorecer a que los resultados de las ejecuciones tengan menor variabilidad, ya que sin ello el sistema operativo puede realizar colocaciones muy diversas de los procesos en los núcleos entre distintas ejecuciones.

Para comprobar la correcta implementación de este mecanismo se ha utilizado la herramienta *perf*, en concreto *perf sched* que ofrece la función de grabar en un fichero los eventos del *scheduler* que ocurren durante la ejecución de una aplicación. Con un análisis de los eventos `sched:sched_wakeup_new`, se observa en qué núcleo lógico (entre

[1] *Simultaneous Multithreading* es una tecnología que permite un mejor aprovechamiento de los recursos de un núcleo mediante la existencia de varios contextos independientes (hilos) de ejecución sobre éste. En el caso de la máquina local se trata de la tecnología multi-hilo de Intel, similar a SMT, consistente en dos hilos hardware por núcleo.

[2] Tecnología para memorias que permite un incremento del rendimiento gracias al acceso simultáneo a dos o más módulos distintos de memoria.

```
void set_affinity(size_t i) {
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(i, &cpuset);
  pthread_setaffinity_np(pthread_self(), sizeof(cpuset), &cpuset);
}
```

Listing 3.1: Afinidad de hilos software a un núcleo lógico.

```
class pinning_observer : public tbb::task_scheduler_observer{
public:
  pinning_observer(){ bserve(true);}

  void on_scheduler_entry(bool is_worker){
        // AFINIDAD
  }
};
```

Listing 3.2: Afinidad de hilos software a un núcleo lógico con TBB.

0 y N, siendo N la cantidad de núcleos totales menos uno) del sistema se lanza cada hilo de la aplicación y con `sched:sched_migrate_task` se observa entre qué núcleos lógicos migran estos mismos hilos, luego se comprueba observando estos eventos en cada caso si los hilos de la aplicación migran como se ha establecido.

Por otro lado, para realizar y comparar resultados entre versiones de las aplicaciones con soporte y sin soporte de instrucciones *far atomics* se ha añadido un *flag* específico en el momento de compilación, ya que por defecto se aplica el flag *'-moutline-atomics'* el cual permite el uso de ambos tipos de operaciones, *load-link/store-conditional* y *far atomics*. Compilando con gcc, el *flag '-mno-outline-atomics'* fuerza a que la aplicación no utilice instrucciones *far atomics*, mientras que el *flag '-march=armv8.2-a+fp16+rcpc+dotprod+crypto'* activa las características de Graviton, estableciendo que todas las operaciones atómicas se lleven a cabo mediante instrucciones *far atomics*[3]. Es necesario gcc con versión 10 o superior para tener el soporte más eficiente a las características Graviton. En este trabajo se ha instalado manualmente gcc 10.1.0. Refiérase al Anexo C para más detalles acerca del proceso de instalación.

No obstante, es necesario comprobar que las aplicaciones utilizan el tipo de instrucciones atómicas indicadas mediante el *flag* anterior para asegurar una experimentación correcta. Para ello, se ha utilizado el programa *objdump*, el cual sirve para mostrar diversa información sobre ficheros de objetos en sistemas operativos tipo Unix. Ejecutando la sentencia mostrada en el *Listing* 3.3 se obtiene la cantidad de instrucciones soportadas por Graviton que utiliza una aplicación, mientras que

[3] https://github.com/aws/aws-graviton-getting-started/blob/main/c-c++.md

```
$ objdump -d app | grep -i 'cas\|casp\|swp\|ldadd\|stadd\
    |ldclr\|stclr\|ldeor\|steor\|ldset\|stset\|ldsmax\|stsmax\
    |ldsmin\|stsmin\|ldumax\|stumax\|ldumin\|stumin' | wc -l
```

Listing 3.3: Comprobación de que una aplicación utiliza instrucciones *far atomics*.

```
$ objdump -d app | grep -i 'ldxr\|ldaxr\|stxr\|stlxr' | wc -l
```

Listing 3.4: Comprobación de que una aplicación utiliza instrucciones *load/store exclusive*.

ejecutando *objdump* como en el *Listing* 3.4 se obtiene el número de instrucciones *load/store exclusive* que contiene la aplicación. Con esto se comprueba que las versiones de las aplicaciones compiladas con soporte para Graviton utilizan sus instrucciones en vez de *load/store exclusive* y las compiladas sin soporte no las utilizan. Además, también se ha observado mediante esta herramienta el código compilado en ensamblador para analizar qué instrucciones se utilizan exactamente y en qué zonas del código. Con esta herramienta y otras como *perf*, se observó que las aplicaciones que utilizaban `std:mutex` y `std::atomic` como mecanismo de sincronización (*Round*, *Qap* y *Floorplan*) utilizan la implementación mediante la librería *libpthread* compilada en el sistema. Indagando en ello se ha encontrado que esta librería está compilada con soporte a las operaciones *far atomics*, ya que para bloquear y liberar los `std::mutex` o para acceder a las variables de tipo *atomic*, la librería hace uso de las operaciones *"__arch64_cas4_acq"* (*compare-and-swap*) y *"__arch64_swp4_rel"* (*swap*).

Para forzar que las aplicaciones de la *suite* implementen su sincronización entre procesos sin operaciones *far atomics*, es necesario realizar una nueva instalación de la librería *glibc* utilizando el flag *'-mno-outline-atomics'* para excluir estas operaciones durante la compilación y posteriormente enlazar la *suite* con esta nueva librería. Durante la realización del proyecto, se han realizado varias instalaciones de la librería *glibc* sin soporte *far atomics* aplicando distintos procedimientos. Sin embargo, no se ha conseguido enlazar ninguna de ellas y dado que es una tarea bastante tediosa y complicada no se ha podido completar. Además, algunos métodos de instalación o enlazado requieren modificar ciertos ficheros del sistema sobre los cuales no se tiene permiso, así que esta tarea queda como posible trabajo futuro del proyecto.

Por otro lado, se puede asegurar que a nivel de código de aplicación (sin incluir librerías del sistema), cada benchmark utiliza las operaciones deseadas mediante el *flag* de compilación en cada caso.

Finalmente, para poder analizar los resultados correctamente, a partir de los tiempos de ejecución obtenidos de varios lanzamientos para cada aplicación, se han

calculado una serie de métricas que resultan de interés como son la media, la desviación estándar, el coeficiente de variación y el error estándar. Además, para asegurar una correcta representación de los resultados, se ha calculado un intervalo de confianza del 95 % para la media mediante la Fórmula 3.1:

$$IC_{95\,\%} = \bar{x} \pm SE \times 1{,}96 \tag{3.1}$$

En la fórmula, $\bar{x}$ representa la media, SE la desviación estándar y 1,96 es el cuartil 0,975 de la distribución normal (para una confianza del 95 %) [5]. Este intervalo de confianza calculado solo aplica a muestras grandes, esto es, muestras de tamaño mayor a 30. En el presente estudio se han realizado 100 ejecuciones por cada experimento, ya que supone un tiempo total de experimentación razonable y el hecho de incrementar el tamaño de la muestra ayuda a reducir considerablemente el error.

3.3. Resto de Tecnologías Utilizadas

El desarrollo del presente trabajo requiere de una cantidad considerable de experimentación. Con el objetivo de conseguir una mayor eficiencia en la obtención, presentación y análisis de los resultados más destacables, se ha utilizado una serie de tecnologías y herramientas software relativas a la compilación y lanzamiento de los benchmarks, extracción y representación de resultados, control de versiones, etcétera. Además de GCC, perf sched y objdump, ya mencionadas, se ha hecho uso de las siguientes tecnologías:

- Github: para control de versiones sobre las actualizaciones de la *suite* se ha hecho uso de un repositorio en Github partiendo de una copia del repositorio INNCABS. Además, la herramienta facilita el hecho de mantener las aplicaciones actualizadas entre las distintas máquinas que se han utilizado para el desarrollo y testeo de la suite.

- Google Drive: se ha utilizado el almacenamiento en nube de Google para mantener una copia de seguridad siempre activa sobre todos los ficheros que componen el trabajo, además de la *suite* de benchmarks.

- CMake: esta herramienta se ha utilizado para automatizar el proceso de construcción de la *suite*. CMake resulta muy útil debido a que el proceso es independiente del compilador que se utilice, además de que soporta jerarquías de directorios y aplicaciones que dependen de varias bibliotecas. En el caso que nos

ocupa, un ejemplo de biblioteca dependiente es *Threads* y TBB. La versión de CMake mínima requerida para construir la aplicación es la 3.1.

- SSH: herramienta utilizada para el acceso remoto de forma segura a la máquina *Pilgor*.

- *Scripts* en bash: para la automatización en la ejecución de los benchmarks se han implementado una serie de *scripts* de bash. Estos *scripts* permiten ejecutar múltiples instancias de un benchmark con una configuración en concreto, almacenando los tiempos de ejecución obtenidos en un fichero, para posteriormente poder ser analizados. Esta manera de proceder resulta bastante útil puesto que resulta fundamental lanzar múltiples ejecuciones sobre un mismo benchmark y analizar la variabilidad de los tiempos de ejecución para realizar una valoración acertada del comportamiento del programa.

- *Scripts* en Python: se ha elegido Python para elaborar la automatización de la representación de los resultados, ya algunas bibliotecas como *Mathplotlib* y *Pandas* ofrecen una gran variedad de posibilidades para representar resultados. Estos *scripts* toman los ficheros de salida generados por los *scripts* en bash y a partir de su contenido extraen los datos más relevantes como la media o la mediana, percentiles, etcétera, para poder generar una descripción gráfica conveniente de los resultados obtenidos.

- *strace*: se ha utilizado esta herramienta para inspeccionar las llamadas al sistema que se realizan durante la ejecución de un benchmark y poder comprobar la cantidad de procesos que se crean y si hay algún error durante la ejecución.

- TBB: se ha elegido la librería TBB para la implementación del uso de un *thread pool* en las aplicaciones a analizar. Se ha instalado manualmente mediante CMake[4].

[4] https://github.com/oneapi-src/oneTBB

Capítulo 4

Propuesta de Mejoras en las Aplicaciones

Este capítulo está dedicado a explicar las mejoras que se han implementado sobre los benchmarks para mejorar su rendimiento, con el objetivo de reducir la sobrecarga que producen algunas llamadas al sistema y así favorecer que el impacto del uso de operaciones atómicas con load-link/store-conditional o far atomics sea más notable en el tiempo de ejecución de las aplicaciones.

4.1. Visión General

Durante el estudio de los benchmarks definidos en la sección anterior, se apreciaron aspectos de la implementación a mejorar. En particular, en el caso de *Round*, se han realizado modificaciones para que el tiempo de ejecución esté mayormente influido por la sincronización de procesos en lugar de por el cómputo de las tareas.

En cuanto al resto de benchmarks, se ha considerado una mejora a nivel de creación y gestión de los hilos como es la introducción de un *thread pool* mediante el uso de la biblioteca TBB frente a la gestión de hilos del runtime de C++. Se decidió utilizar TBB con el objetivo de reducir la gran cantidad de hilos que crean estos benchmarks y por tanto, reducir el sobrecoste que conlleva tanto su creación como su gestión por parte del runtime de C++, ya que TBB, por defecto, crea un hilo por cada núcleo lógico del sistema y realiza una gestión a nivel de tareas mucho más eficiente.

Las secciones posteriores ofrecen más detalles acerca de las mejoras realizadas en cada una de las aplicaciones.

4.2. Benchmark *Round*

Se han realizado una serie de mejoras en *Round* de cara a establecer a la sincronización entre hilos como el aspecto determinante en el tiempo de ejecución.

```
void Philosopher::eat() {
    using Lock = std::unique_lock<std::mutex>;
    Lock first;
    Lock second;
    if (flip_coin())
    {
        first = Lock(left_fork_, std::defer_lock);
        second = Lock(right_fork_, std::defer_lock);
    }
    else
    {
        first = Lock(right_fork_, std::defer_lock);
        second = Lock(left_fork_, std::defer_lock);
    }
    std::lock(first, second)
    auto d = get_eat_duration();
    auto end = std::chrono::steady_clock::now() + d;
    while(std::chrono::steady_clock::now() < end);
    eat_time_ += d;
}
```

Listing 4.1: Función `eat()` original.

```
void Philosopher::eat() {
    using Lock = std::unique_lock<std::mutex>;
    Lock first;
    Lock second;
    first = Lock(left_fork_, std::defer_lock);
    second = Lock(right_fork_, std::defer_lock);
    std::lock(first, second)
    std::this_thread::sleep_for(get_eat_duration());
    eat_time_ += d;
}
```

Listing 4.2: Función `eat()` modificada, manteniendo el mismo funcionamiento.

En primer lugar, se ha eliminado la aleatoriedad a la hora de que un filósofo escoja el primer tenedor a su derecha o izquierda, evitando de esta manera la invocación a la sobrecarga de la función que usa una distribución Bernoulli para dicha elección. Por otro lado, se ha cambiado la forma en la que se espera el transcurso del tiempo mientras un filósofo está comiendo, ya que inicialmente se realizaba con una espera activa llamando a una función continuamente que añadía mucha sobrecarga. La inclusión de ambas alternativas provoca una mejora sustancial en el tiempo de ejecución como puede apreciarse en la Sección 5.1. El *Listing* 4.1 muestra la función `eat()` sin modificar, mientras que el *Listing* 4.2 muestra la misma función tras las modificaciones.

Por otro lado, se ha prescindido del tiempo durante el cual un filósofo está comiendo, como puede verse en el *Listing* 4.3 respecto a los anteriores. De esta forma, los mutexes se adquieren y se liberan constantemente, consiguiendo que la necesidad de sincronización entre hilos sea mucho mayor y maximizando el uso de las operaciones

```
void Philosopher::eat() {
    using Lock = std::unique_lock<std::mutex>;
    Lock first;
    Lock second;
    first = Lock(left_fork_, std::defer_lock);
    second = Lock(right_fork_, std::defer_lock);
    std::lock(first, second)
    times_eaten++;
}
```

Listing 4.3: Función `eat()` modificada, filósofo come continuamente tantas veces como se indique.

```
void Philosopher::dine() {
    while(eat_time_ < full) eat(); //Come durante X tiempo
}
```

Listing 4.4: Función `dine()` original ejecutada por cada hilo.

```
void Philosopher::dine() {
    global_barrier_.wait();
    while(times_eaten < times_to_eat) eat(); //Come X veces
    global_barrier_.done_waiting();
}
```

Listing 4.5: Función `dine()` modificada ejecutada por cada hilo.

atómicas que se pretenden evaluar. Con la mejora, la ejecución del programa finaliza cuando todos los filósofos han comido tantas veces como se le indica al programa en vez de cuando cada filósofo come durante cierto tiempo.

Además, para asegurar que todos los filósofos comienzan a solicitar los tenedores al mismo tiempo, se ha implementado un módulo barrera para realizar una sincronización previa tras haber sido creados todos los hilos. Los *Listings* 4.4 y 4.5 reflejan estos cambios sobre la función `dine()`.

4.3. Resto de Benchmarks

Para el resto de aplicaciones, se ha medido empíricamente que la creación de un gran número de hilos mediante la primitiva `std::async` y su gestión genera una gran sobrecarga por parte del *runtime* de C++, constituyendo la mayor parte del tiempo de ejecución de las aplicaciones, e incluso, en algunos casos, la ejecución no termina en un tiempo razonable. A diferencia del resto de benchmarks, este comportamiento no tiene lugar en *Round*, puesto que no se hace uso de tareas recursivas y el número de hilos totales creados es mucho menor.

Con el objetivo de mitigar la sobrecarga provocada por la numerosa creación de

```
for(int i=0; i<problem->size; i++) {
    futures.push_back(std::async(1, [=, &best_known](){
        // TAREA
    };
}
for(auto& f : futures) {
  f.wait();
}
```

Listing 4.6: Implementación original sin *thread pool*.

```
tbb::parallel_for(0, problem->size, 1, [=, &best_known](int i){
    // TAREA
};
```

Listing 4.7: Implementación modificada con *thread pool* (`parallel_for`).

```
int resultado = tbb::parallel_reduce(
  /* the range = */ tbb::blocked_range<int>(0, problem->size,),
  /* the identity = */ 0,
  /* func = */
  [&](const tbb::blocked_range<int> &r, int init) -> int{
    for(int row = r.begin(); row != r.end(); ++row){
            // TAREA (ACTUALIZA RESULTADO EN VARIABLE init)
    }
    return init;
  },
  /* reduction = */
  [](int x, int y) -> int{
    // FUNCION DE REDUCCION
  }
);
```

Listing 4.8: Implementación modificada con *thread pool* (`parallel_reduce`).

hilos y la utilización de un modelo *fork-join* por defecto en la *suite INNCABS* para tareas de grano fino (se encolan los resultados con los *futures* y se espera sobre ellos), se ha sustituido la primitiva `std::async` (como aparece en el *Listing* 4.6) por la implementación de un *thread pool* mediante la librería TBB. Tras la instalación de la librería TBB, las modificaciones en la implementación se resumen en cambiar la forma en la que se lanzan los hilos. Esto se ha realizado mediante el uso de las funciones `tbb::parallel_for` y `tbb::parallel_reduce`, orientadas a la paralelización de bucles. La primera función paraleliza aplicando una función lambda, la misma que antes se invocaba con `std::async`, a los elementos de un rango (en este caso el tamaño del programa), luego ya no es necesario el bucle externo (véase el *Listing* 4.7). Por otro lado, `tbb::parallel_reduce`, además de paralelizar un bucle obtiene un único resultado aplicando una función de reducción a todos los resultados parciales de cada iteración, siendo el parámetro *identity* el valor inicial de este resultado. En este caso, la sintaxis

no es tan simple como en `tbb::parallel_for` ya que a `tbb::parallel_reduce` se le pasa un objeto *Rango* y ha de aplicarse la función a paralelizar a todos elementos de este rango explícitamente (véase el ejemplo recogido en el *Listing* 4.8).

Capítulo 5

Resultados Experimentales

Este capítulo recoge los resultados más relevantes obtenidos tras la experimentación con la suite de benchmarks INNCABS. Esto comprende las mejoras realizadas sobre el benchmark Round comentadas en el capítulo anterior y la utilización de la biblioteca TBB para la creación y gestión de tareas sobre el resto de benchmarks (Qap, Floorplan y Nqueens). Se ha realizado un análisis de la eficiencia de `std::async` *respecto al thread pool de TBB. La última sección trata el análisis sobre el impacto que supone el uso de las instrucciones far atomics frente a load-link/store-conditional en todos los benchmarks estudiados. El método de experimentación ha sido el mismo para todas las pruebas como se ha comentado en la Sección 3.2.*

5.1. Mejoras en el Benchmark *Round*

Tras haber aplicado las diferentes mejoras sobre el benchmark *Round* (detalladas en la Sección 4.2), se ha llevado a cabo el mismo experimento para cada versión mejorada[1]. Se ejecuta *Round* variando el número de filósofos entre 2 y 96, incrementando así la complejidad del problema. Cabe mencionar que existe un tiempo de ejecución mínimo en este benchmark debido a que cada filósofo tiene que comer durante 50 milisegundos para terminar y dado que como máximo puede haber un filósofo comiendo al mismo tiempo de los dos que comparten un tenedor, el tiempo mínimo de ejecución es de 100 milisegundos.

En la Figura 5.1 puede verse como, para las ejecuciones con un número de filósofos pequeño, el tiempo de ejecución es de 100 milisegundos, así que cumple con el tiempo mínimo de ejecución estimado. El hecho de sustituir la llamada `steady_clock` por `sleep_for` añade alrededor de un milisegundo al tiempo mínimo. Es decir, el hecho de dormir la tarea mientras realiza la acción de comer añade una pequeña sobrecarga

[1]Excepto para la versión que elimina el tiempo de comida de los filósofos, lo cual supone modificar por completo el benchmark, puesto que esa versión está orientada al análisis de *far atomics* y no es comparable con las anteriores.

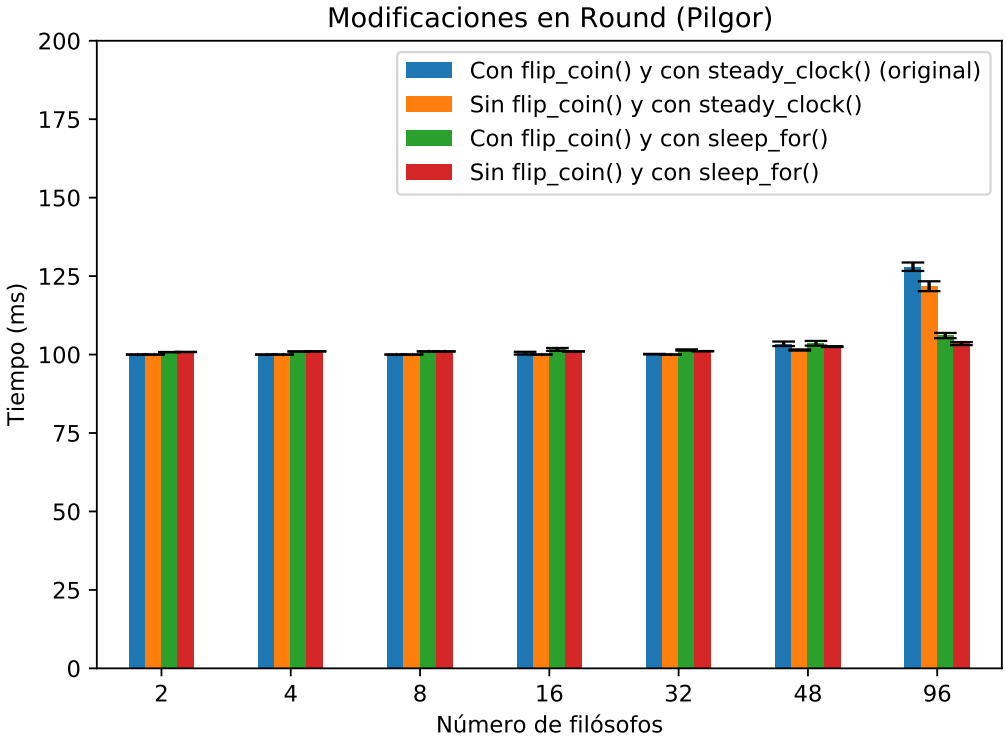


Figura 5.1: Comparativa de mejoras en *Pilgor*.

al total de la ejecución. No obstante, para los casos con más filósofos, el hecho de consultar el reloj constantemente mediante `steady_clock` acaba suponiendo un mayor sobrecoste en tiempo como puede verse claramente en el caso de 96 filósofos.

El hecho de elegir de forma aleatoria mediante `flip_coin()` el orden en el que un filósofo comienza a adquirir los mutexes (tenedores) también influye negativamente en el rendimiento. Este sobrecoste comienza a apreciarse a partir del escenario con 16 filósofos. Si se implementa que cada filósofo intente bloquear primero un mutex determinado (en este caso el tenedor que corresponde a su mano izquierda), esto reduce el tiempo de ejecución y su variabilidad. Se ha observado el código de `std::lock`[2] para verificar que efectivamente el orden en el que se le pasan los mutexes a `Lock()` es el orden en el que los intenta bloquear y en caso de que alguno falle los libera en orden inverso al que los había bloqueado. También se ha añadido un margen de error en cada barra para analizar la variabilidad en cada caso. En esta experimentación, el error medido en cada ejecución es muy pequeño y aumenta sobre todo en el caso de 96 filósofos, al igual que aumenta el tiempo de ejecución, ya que se hace uso de los dos *sockets* del sistema, lo que añade más retardo en la sincronización de los procesos debido a que la comunicación entre los chips es más lenta. Véase el Anexo D para una

[2] https://github.com/gcc-mirror/gcc/blob/4a960d548b7d7d942f316c5295f6d849b74214f5/libstdc%2B%2B-v3/include/std/mutex#L537

comparativa de las mejoras en una máquina x86.

5.2. Implementación de un *Thread Pool*

Como introducción a esta sección y con el objetivo de analizar en detalle la directiva `std::async`, se ha desarrollado un microbenchmark adicional cuya finalidad es únicamente lanzar tantas tareas como se indique como argumento de entrada. Las tareas invocan una función vacía, la cual retorna inmediatamente. Ejecutando este microbenchmark se ha verificado mediante la herramienta *strace* que cada invocación a `std::async` con la tarea a ejecutar deriva en una llamada al sistema `clone()` para la creación de un hilo que la llevará a cabo. Además, mediante esta herramienta también se obtiene el coste medio en tiempo de la llamada al sistema `clone()`, la cual varía bastante en función de la carga del sistema (en este caso, la producida por la aplicación) y por el tamaño de la función a invocar, ya que ha de copiar la función punto de entrada a la estructura del proceso clonado para posteriormente invocarla. Según se ha observado, el coste medio de esta llamada en las distintas aplicaciones puede oscilar desde apenas unos pocos microsegundos hasta casi un milisegundo, lo cual puede suponer una sobrecarga muy grande en ejecuciones de los benchmarks que requieren lanzar una cantidad considerable de hilos, como es el caso de *Qap*, *Floorplan* o *Nqueens*.

A continuación, se van a comparar los benchmarks *Qap*, *Floorplan* y *Nqueens* en sus dos versiones, una implementada con la directiva `std::async` (por defecto) y otra con TBB, para analizar el impacto que supone la utilización de un *thread pool* para la ejecución de tareas de grano fino y su gestión de hilos a nivel de usuario frente a la planificación a nivel de kernel.

La ejecución de estos benchmarks supone la creación de una gran cantidad de hilos, ya que son problemas con un espacio de búsqueda bastante grande y como se ha demostrado, cada invocación a `std::async` deriva en la llamada al sistema `clone()` para la creación de un nuevo proceso. Por este motivo, la mayoría de benchmarks no han podido ejecutarse en la máquina *Pilgor* para los casos con tiempos de ejecución razonables (superiores a pocos milisegundos) debido a una limitación en la cantidad de procesos que el sistema permite crear al usuario, ya que la aplicación se aborta si se supera este límite. Esta limitación fue modificada por parte de José Antonio Gutiérrez Elipe, administrador de sistemas del Departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza, en primer lugar para poder llevar a cabo la experimentación de *Round*[3], aunque resulta insuficiente para experimentar con el

[3]Se trata del benchmark con menos demanda a nivel de creación de procesos.

resto de aplicaciones, puesto que o bien el sistema genera una excepción que aborta la ejecución o el resultado de la ejecución es fallido al no obtener el valor esperado. Por esta razón, el análisis de esta sección se va a realizar sobre la máquina local x86 al permitir la ejecución de casos de prueba más demandantes, apreciando mejor las diferencias entre ambas implementaciones.

Todas figuras que mostradas en esta sección están representadas con diagramas de caja y bigote en los que los bigotes superior e inferior representan los percentiles 95 y 5, respectivamente, mientras que los extremos de las cajas muestran los percentiles 75 y 25. Los puntos huecos representan los *outliers*, valores extremos más allá de los bigotes, y la línea naranja contenida en la caja representa la mediana de los tiempos obtenidos sobre 100 ejecuciones. Además, también se ha añadido la media mediante un punto azul y sobre éste un margen con el error calculado como se ha explicado en la Sección 3.2 con el objetivo de apreciar la variabilidad entre las ejecuciones. A pesar de que este último dibujo añade cierta sobrecarga a la gráfica se ha decidido mantenerlo ya que en algunos casos la variabilidad del tiempo de ejecución es bastante considerable y como la cantidad de ejecuciones realizadas en la experimentación está acotada a 100, se cree de interés conocer un rango para la media que comprenda cualquier ejecución con un 95 % de confianza.

En la Figura 5.2 puede apreciarse una gran mejora sobre las tres aplicaciones en las implementaciones con TBB respecto a `std::async`, reduciendo el tiempo de ejecución a prácticamente cero, estableciendo la cantidad de hilos hardware disponibles a cuatro. Por ejemplo, para *Qap* se observa una reducción de 66 ms a apenas unos pocos microsegundos. Según se ha observado con la utilidad *strace* sobre estas pruebas, en las implementaciones con `std::async` la cantidad de llamadas al sistema que se realizan durante la ejecución es mucho mayor respecto a TBB. También se ha observado con *perf* cómo la carga a nivel de planificación por parte del kernel es muy superior, ya que en TBB las tareas son gestionadas por la biblioteca a nivel de usuario. Por ejemplo, al ejecutar *Qap* para el fichero de entrada *chr06a.dat* en la versión con `std::async`, alrededor de un 60 % del tiempo de cómputo está dedicado a la gestión de llamadas al sistema mediante el kernel, mientras que en la implementación con TBB para el mismo caso solo supone aproximadamente un 15 % del tiempo total. Estas observaciones justifican la gran diferencia en tiempo de ejecución entre ambas implementaciones.

Por otro lado, en la Figura 5.3 se ilustra el caso concreto del benchmark *Nqueens* para un tablero cuadrado de N=9, variando el número de hilos hardware disponibles para cada ejecución entre 1 y 8 hilos. Se ha medido con *strace* que esta ejecución lanza 8393 tareas o crea 8393 hilos. El uso de TBB provoca una mejora muy notable en todos los casos. El error obtenido entre las ejecuciones es despreciable al encontrarse

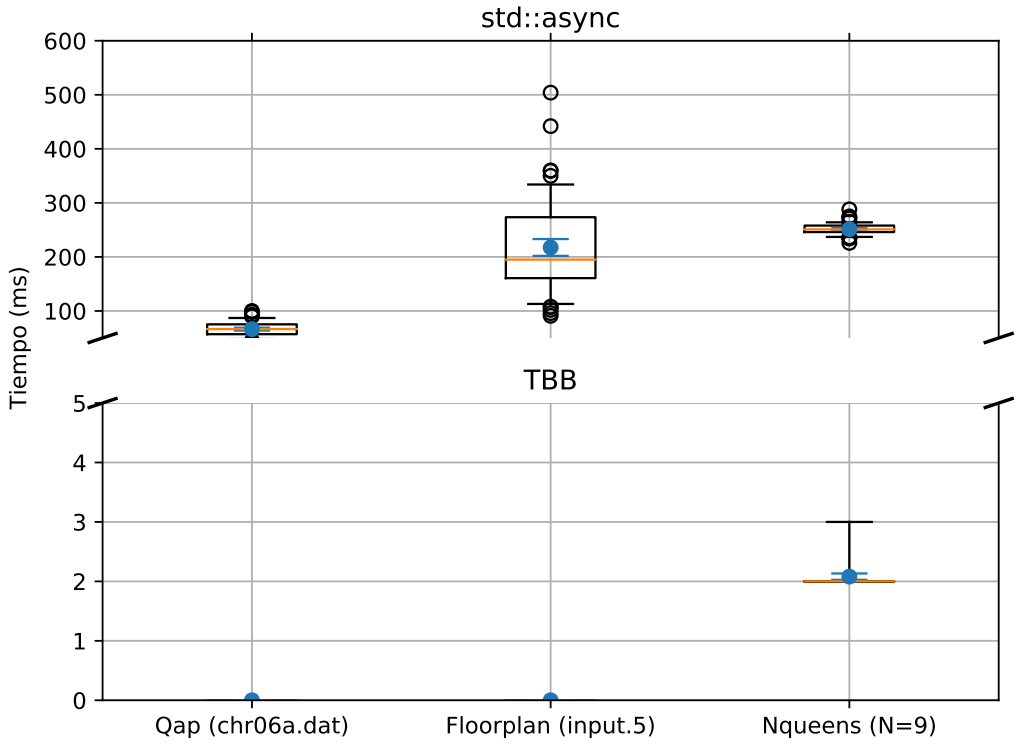


Figura 5.2: Comparativa de implementación de las aplicaciones con `std::async` y TBB en x86 para cuatro hilos hardware.

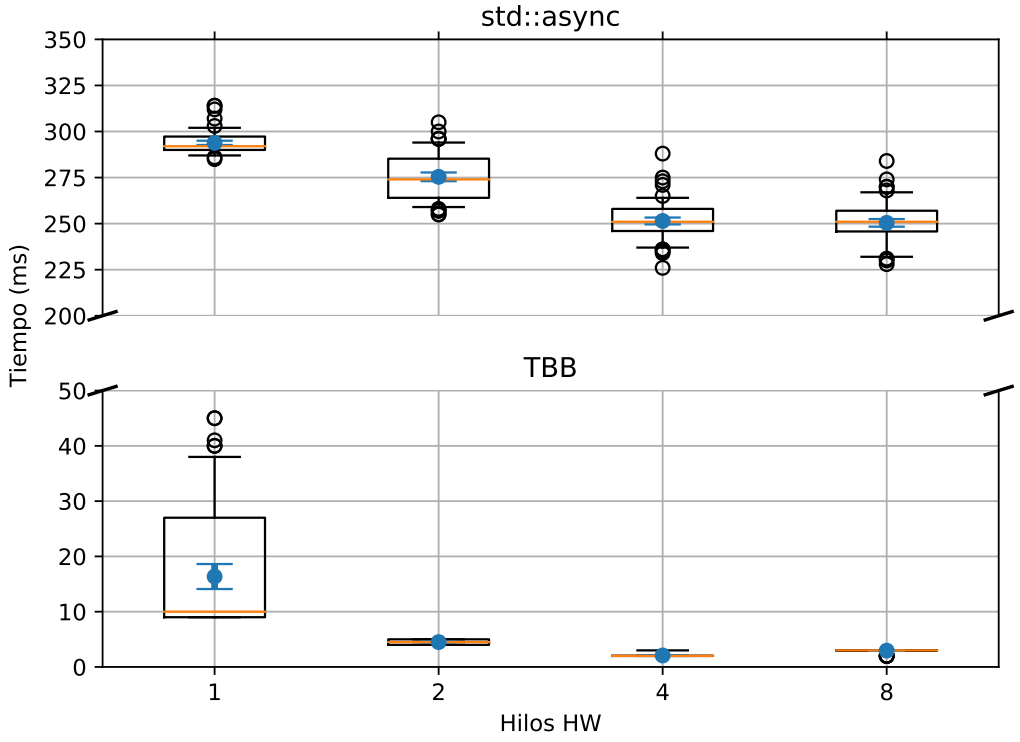


Figura 5.3: Comparación de versiones del benchmark *Nqueens* implementadas con `std::async` y TBB para N=9 en x86.

en la escala de milisegundos, aunque resulta un poco superior para el caso de un hilo hardware, lo que puede deberse simplemente al hecho de tener que ejecutar y planificar todas las tareas sobre un mismo hilo hardware, generando una sobrecarga mayor. Se refiere al lector al Anexo D para conocer en más detalle estas pruebas, así como algunos ejemplos sobre la escalabilidad que proporciona TBB en estos benchmarks para espacios de búsqueda más grandes.

5.3. Evaluación de *Far Atomics*

Para evaluar las instrucciones *far atomics* se experimenta únicamente en *Pilgor*, ya que se trata de la máquina con ISA de ARM que dispone de soporte para este tipo de instrucciones. Dado que tres de las cuatro aplicaciones estudiadas no se pueden ejecutar en esta máquina en su implementación con `std::async` para tamaños grandes de problema, la mayor parte de la experimentación sobre *far atomics* se realiza sobre las versiones de los benchmarks implementadas con TBB. Aprovechando el uso de esta biblioteca, también se va a analizar qué impacto provoca el hecho de compilar la biblioteca de TBB con soporte de *far atomics*. Así pues, se distinguen tres casos de análisis: i) aplicación sin soporte *far atomics*, ii) únicamente la aplicación con soporte *far atomics* y iii) soporte *far atomics* tanto en la aplicación como en la biblioteca TBB.

En la Tabla 5.1 se muestran los resultados obtenidos para los tres casos sobre los benchmarks implementados con TBB fijando el número de hilos hardware a 48, ya que es el número máximo de hilos del que se dispone en un único *socket*. Para cada experimentación se muestra la media ($\bar{x}$) y la desviación típica (SE). Para los benchmarks *Qap* y *Floorplan* se han utilizado los ficheros de entrada 'chr12a.dat' e 'input.15', respectivamente, mientras que *Nqueens* se ha ejecutado con N=12. Los resultados muestran cómo la diferencia entre las distintas versiones es bastante pequeña en todos benchmarks. En *Floorplan* apenas se aprecian diferencias, mientras que en *Qap* y *Nqueens* la tendencia conforme se añaden más instrucciones *far atomics* es a aumentar el tiempo de ejecución, aunque al tratarse de pocos milisegundos no resulta significativo. Por ello, se ha realizado el mismo experimento para *Qap* con un espacio de búsqueda mayor (fichero de entrada 'chr18a.dat') para aumentar la complejidad del problema y así observar si la tendencia es la misma. Con los resultados de la Tabla 5.2 se aprecia con más claridad el hecho de que el uso de estas instrucciones no favorece al rendimiento en esta experimentación realizada.

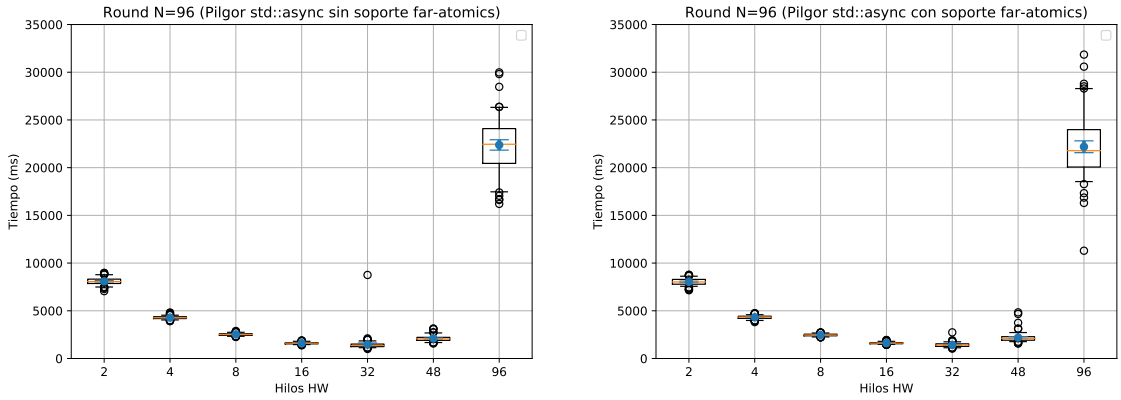
También se ha analizado el impacto de *far atomics* en una implementación con `std::async` para el benchmark *Round*. En este caso tampoco se aprecian diferencias significativas entre las versiones sin y con soporte a *far atomics* como se puede observar

Benchmark	Sin F-A		F-A en App		F-A en App y TBB	
	$\bar{x}$	*SE*	$\bar{x}$	*SE*	$\bar{x}$	*SE*
Qap	44.91	14.88	51.25	16.03	55.15	18.02
Floorplan	1958.29	228.25	1963.44	253.57	1935.29	204.27
Nqueens	107.78	23.59	130.09	26.04	141.78	28.77

Tabla 5.1: Resultados obtenidos (en ms) para benchmarks con TBB y 48 hilos hardware.

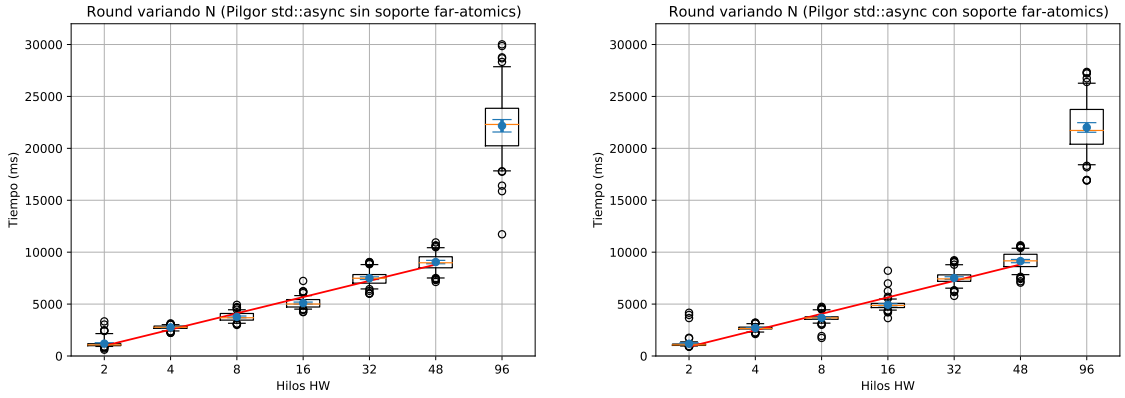
Benchmark	Sin F-A		F-A en App		F-A en App y TBB	
	$\bar{x}$	*SE*	$\bar{x}$	*SE*	$\bar{x}$	*SE*
Qap	31403.42	3320.43	39245.25	3487.32	48001.11	5459.15

Tabla 5.2: Resultados obtenidos (en ms) para *Qap* con TBB, 48 hilos hardware y un tamaño de problema mayor.



(a) Sin soporte *far atomics* (b) Con soporte *far atomics*

Figura 5.4: Ejecución de *Round* en *Pilgor* con 96 filósofos y 1M de bocados/filósofo (bloqueo de ambos tenedores), variando los hilos hardware disponibles.



(a) Sin soporte *far atomics* (b) Con soporte *far atomics*

Figura 5.5: Ejecución de *Round* en *Pilgor* con 1 filósofo por cada hilo hardware disponible y 1M de bocados/filósofos (bloqueo de ambos tenedores).

en las Figuras 5.4a y 5.4b.

Una de las posibles razones a priori por las que se han obtenido estos resultados

puede ser la falta de contención entre los hilos. Esta hipótesis se verifica para los benchmark *Qap* y *Floorplan* mediante *perf* (en *Nqueens* no se observa ya que no tiene sincronización entre tareas), ya que únicamente alrededor de un 0,5-7 % del tiempo de ejecución total está dedicado a la sincronización entre hilos (oscila bastante debido al algoritmo que utilizan y el tamaño del problema). Sin embargo, esto se descarta para el caso de *Round*, ya que en este benchmark la mayor parte del tiempo de ejecución de la aplicación se dedica a intentar bloquear los *mutexes* (alrededor de un 90 %). Por otro lado, viendo la similitud de los resultados entre ambas versiones, se puede concluir que la ausencia de diferencias es causada a que gran parte de la sincronización y sobre la que más accesos se realizan se lleva a cabo mediante objetos de las librerías *libpthread* y *libc*, las cuales están compiladas con soporte a *far atomics*.

Por último, también se ha evaluado la escalabilidad de la aplicación *Round*, ya que se ha comprobado que la carga de sincronización entre sus tareas es muy alta. Las Figuras 5.5a y 5.5b muestran los resultados de un experimento en el cual se comprueba si el benchmark *Round* cumple con la Ley de Gustafson[4] para los casos con y sin soporte *far atomics*. Esta ley anuncia que si un problema con cierto tamaño se resuelve con P procesadores en un tiempo T, si se aumenta el número de procesadores se podrá resolver un problema más complejo en el mismo periodo de tiempo. Para comprobar si esta ley se cumple de forma gráfica, se ha añadido una línea en ambas figuras que representa una regresión lineal para las medias obtenidas en los casos entre 2 y 48 hilos hardware (se excluye el caso de 96 hilos ya que el tiempo se dispara por el uso de dos sockets). Los resultados muestran que el benchmark *Round* no mantiene el tiempo de ejecución conforme aumenta la carga y las características del sistema, ya que el tiempo crece para cada experimento de forma prácticamente lineal excepto cuando se utilizan los dos *sockets* de la plataforma, donde el tiempo de ejecución aumenta mucho más como cabría esperar. Como en el caso anterior, no se aprecian diferencias significativas entre las versiones compiladas con y sin soporte a *far atomics*.

[4] https://es.wikipedia.org/wiki/Ley_de_Gustafson

Capítulo 6

Conclusiones y Trabajo Futuro

Este último capítulo recoge las conclusiones del proyecto, algunas posibles extensiones futuras y un diagrama de Gantt con las principales tareas realizadas y el tiempo dedicado a cada una de ellas.

6.1. Conclusiones

El principal objetivo de este trabajo ha sido analizar el impacto en el rendimiento que supone utilizar las nuevas instrucciones de acceso a memoria *far atomics* respecto a las más tradicionales *load-link/store-conditional*. Para ello se ha utilizado una suite de benchmarks con aplicaciones paralelas compuestas por tareas de grano fino, al ser potenciales beneficiarias de las nuevas instrucciones.

Como en primer lugar se observó que al ejecutar algunas de las aplicaciones seleccionadas para el análisis existía mucha sobrecarga debido a la creación y gestión de las tareas por parte del sistema operativo (implementación de tareas por defecto con `std::async`), y además en su mayoría no podían ejecutarse en la máquina *Pilgor*, se decidió realizar una implementación de la *suite* utilizando un *thread pool*, en concreto, con la biblioteca TBB.

Tras el análisis y comparación entre ambas versiones `std::async` y *thread pool*, resulta conveniente utilizar bibliotecas de terceros, en este caso TBB, para delegar la gestión y planificación de aplicaciones paralelas con tareas de grano fino a un *thread pool*, ya que se consigue una gran mejora en el rendimiento respecto a la gestión que realiza C++ mediante la primitiva `std::async`. También sería recomendable que C++ incluyera su propio *thread pool* para facilitar la implementación de este tipo de aplicaciones al programador.

En la aplicación *Round* también se realizaron una serie de mejoras con el objetivo de, a priori, aumentar la contención entre las tareas y que de esta forma utilicen un mayor número de instrucciones atómicas al intentar bloquear los mutex de forma más

intensiva. A la vista de los resultados obtenidos, el uso de las nuevas instrucciones *far atomics* no supone una mejora sustancial en el rendimiento de las aplicaciones de la suite de benchmarks INNCABS respecto a *load-link/store-conditional* en el procesador Kumpeng 920 (*Pilgor*), a pesar de que como hipótesis de partida del presente trabajo se esperaba obtener una mejoría en tiempo de ejecución.

Hipotéticamente, existen varias razones que podrían justificar la falta de mejora en el tiempo de ejecución por parte de *far atomics*. En primer lugar, esto podría deberse a que los benchmarks estudiados no generan suficiente contención entre los hilos como para que el uso de *far atomics* suponga una mejora notable en el rendimiento. Para poder comprobar si esto sucede, sería necesario disponer de contadores hardware que midan más exactamente el grado de interacción de los hilos entre sí mediante estas operaciones atómicas.

Por otro lado, como para el benchmark *Round* se ha observado mediante *perf* que la aplicación sí genera bastante contención, pero sin embargo los resultados también son bastante similares para *far atomics* y *load-link/store-conditional*, otra hipótesis es que el hecho de que no haya sido posible el enlazado de la *suite* con una nueva biblioteca *glibc* compilada sin soporte *far atomics* implica que no se hayan podido obtener las diferencias entre *far atomics* y *load-link/store-conditional* sobre los objetos que mayor contención provocan, como es el caso de los `std::mutex` y los `std::atomic`. Al tratarse de las variables compartidas mayormente accedidas, son sobre las que más influye la localidad en la cache L1, luego deberían ser las variables sobre las que *far atomics* más deberían destacar respecto a *load-link/store-conditional*.

6.2. Trabajo Futuro

Algunas de las posibles extensiones sobre este trabajo podrían ser las siguientes:

- Realizar una nueva instalación de la librería *glibc* sin soporte a las operaciones *far atomics* en la máquina Pilgor y enlazarla con la *suite* de benchmarks modificada en este proyecto, para ver cómo influye en el rendimiento la implementación de `std::mutex` y `std::atomic` mediante operaciones *load-link/store-conditional* respecto a *far atomics*.

- Repetir la experimentación para implementaciones de *far atomics* más modernas, ya que la máquina Pilgor implementa la primera versión del conjunto de extensiones *LSE*. Comprobar si en futuras microarquitecturas el uso de estas instrucciones sí supone una mejora respecto a *load-link/store-conditional* y respecto a implementaciones de *far atomics* anteriores.

- Indagar más profundamente en el uso real que se está haciendo de estas instrucciones y la contención que producen las aplicaciones de la *suite* INNCABS mediante contadores hardware.
- Realizar una experimentación similar utilizando otras benchmark suites que propongan otro tipo de tareas, con una mayor carga de sincronización entre ellas.

Bibliografía

[1] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Apress, 2019.

[2] Alejandro Valero, Darío Suárez Gracia, Rubén Gran Tejero, Luis M. Ramos, et al. Experimentación preliminar con un trazador de rayos para relacionar niveles de abstracción. In *Proceedings of the XXX Jornadas de Paralelismo. At: Cáceres (Spain)*, 2019.

[3] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Publications, 2012.

[4] Peter Thoman, Philipp Gschwandtner, and Thomas Fahringer. On the Quality of Implementation of the C++11 Thread Support Library. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 94–98, 2015.

[5] Raj Jain. *The Art of Computer Systems Performance Analysis (Techniques for Experimental Design, Measurement, Simulation, and Modeling)*. John Wiley & Sons, 1991.

[6] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, pages 355–364, 1991.

[7] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

Anexos

Anexos A

Coherencia y Consistencia

Hoy en día, casi todos los microprocesadores destinados ya sea a propósito general o a computación de altas prestaciones disponen de varios núcleos hardware y una jerarquía de memoria cache de varios niveles [7]. Generalmente, el primer nivel de memoria cache, con poca capacidad, resulta de acceso privado por cada núcleo (acceso más rápido), mientras que los niveles inferiores de mayor capacidad son de acceso compartido entre algunos ó todos los núcleos (acceso más lento), dependiendo de la organización del chip. El objetivo primordial de esta jerarquía es reducir al máximo el tiempo de acceso a los datos, evitando en la medida de lo posible los costosos accesos a memoria fuera del chip, para incrementar sustancialmente el rendimiento.

Cuando se habla de paralelismo en multiprocesadores, generalmente se hace referencia a un conjunto de procesos que trabajan simultáneamente, ejecutándose en distintos núcleos, con el objetivo de realizar una tarea en conjunto. Esto implica que varios núcleos del multiprocesador compartan cierta información entre ellos, luego el sistema ha de disponer de ciertos mecanismos que garanticen que la información compartida es coherente entre las distintas copias privadas de cada núcleo. Por otra parte, los microprocesadores modernos utilizan ciertas técnicas para aumentar el rendimiento como son la reordenación de instrucciones o la especulación. Estas mejoras fuerzan a que también sea necesario un modelo que asegure que la ejecución global siempre mantiene un estado consistente en memoria. A continuación se ofrece al lector un mayor detalle acerca de los conceptos de coherencia y consistencia.

Coherencia. Los problemas de coherencia en memoria aparecen cuando varios núcleos ejercen accesos de escritura a varias copias de un mismo dato (copias en sus caches privadas). Por ello, es necesario que todas las escrituras realizadas por cada núcleo sean propagadas al resto, para que todas las caches privadas se mantengan coherentes entre ellas. La principal diferencia entre los protocolos que llevan a cabo este control está en cuándo y cómo se realiza ese sincronismo. Los protocolos más

conocidos son:

- Bus snooping: cuando se produce un fallo en cache, el controlador de cache del núcleo que ha fallado arbitra un bus compartido para enviar una petición del dato al resto de núcleos. Este bus asegura que todos los controladores de cache del resto de núcleos observan las peticiones en el mismo orden y pueden coordinar sus propias acciones con las distribuidas del resto de controladores para asegurar que mantienen un estado global consistente. Esto se complica en los buses modernos, ya que utilizan colas de arbitraje que pueden enviar respuestas *unicast* retrasadas por el efecto del *pipeline* o la ejecución fuera de orden. Todas estas optimizaciones conducen a estados de coherencia más transitorios.

- Protocolo basado en directorio: consiste en un controlador en el siguiente nivel de memoria cache que mantiene un directorio o tabla almacenando las direcciones de memoria presentes en las caches. De esta forma, en función de la dirección solicitada, se envía una petición solamente a aquellas caches que cuentan con esa dirección.

Existen dos invariantes que todo protocolo de coherencia debe satisfacer. Por un lado el invariante *Single-Writer, Multiple-Read Invariant* (SWMR) implica que en cualquier momento, o bien existe un único núcleo escribiendo un bloque de memoria o existen varios núcleos leyéndolo, pero nunca se realizan ambas operaciones simultáneamente. Por otro lado, también debe cumplirse el *Data-Value Invariant*. Este invariante implica que las escrituras sobre un bloque de memoria se propagan correctamente, es decir, no hay lecturas posteriores que leen valores obsoletos cuando ya se ha escrito en ese bloque.

Consistencia. En cuanto a la consistencia en memoria, un núcleo puede realizar distintos tipos de reordenaciones sobre las instrucciones de acceso a memoria, siendo un ejemplo el caso *store-store*. Esta reordenación puede producirse, en un núcleo que no disponga de una cola FIFO para escrituras, si durante la ejecución una primera escritura falla en la cache y la segunda acierta, adelantando la segunda a la primera. Al tratarse de accesos a direcciones de memoria distintas, esta situación en ejecución mono-hilo no provoca ninguna inconsistencia en memoria. Sin embargo, en ejecución multi-hilo, el acierto de la segunda escritura podría marcar un *flag*, indicando al resto de núcleos que también se ha realizado la escritura anterior, lo que provocaría que estos núcleos pudieran consumir el contenido de la primera escritura antes de que ésta se haya producido.

Un modelo de consistencia en memoria define las especificaciones de los comportamientos permitidos en la ejecución de programas multi-hilo con memoria compartida. A diferencia del modelo de coherencia, en el que es el *pipeline* del procesador junto con el protocolo de coherencia en memoria el que la asegura de forma invisible al programador, el modelo de consistencia sí es visible al programador.

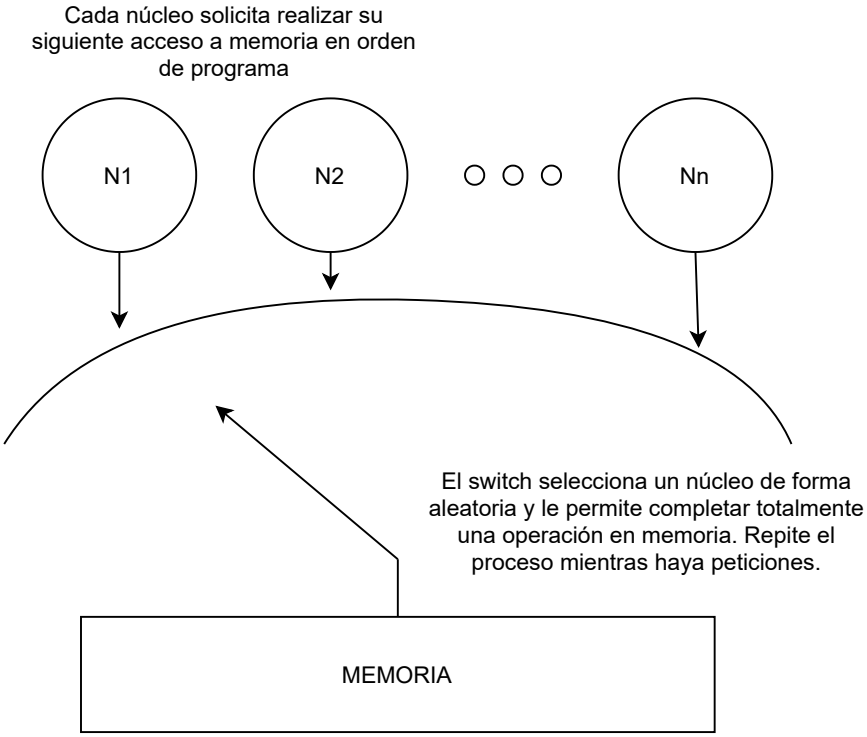


Figura A.1: Implementación del modelo de consistencia secuencial mediante un *switch*.

El modelo de consistencia más intuitivo es el que se conoce como *consistencia secuencial*. Este modelo fuerza que cada núcleo respete el orden de las instrucciones en la ejecución de un programa. Sin embargo, existen otros modelos de consistencia más relajados que permiten la ejecución de ciertas instrucciones en desorden, mejorando así el rendimiento aunque a costa de complicar la tarea del programador. Véase la Figura A.1 para una mejor comprensión del modelo mediante un ejemplo donde n núcleos solicitan varios accesos a memoria en su orden de programa. El *switch* atiende estas peticiones una a una de forma aleatoria entre los núcleos.

Una implementación básica de *consistencia secuencial* con coherencia en cache consiste en que cada bloque de memoria contenido en la cache privada de cada núcleo mantenga una serie de estados. Estos estados pueden ser *Modified*, es decir, el núcleo dispone de la copia más reciente y puede leer y escribir el bloque ó *Shared*, esto es, el núcleo puede únicamente leer este bloque, ya que varias caches de otros núcleos también

disponen de una copia en el mismo estado. La transición entre ambos estados se lleva a cabo mediante las operaciones *GetM* y *GetS*, las cuales solicitan el estado *Modified* ó *Shared*, respectivamente, sobre un bloque en concreto. Cualquier optimización sobre este modelo debe asegurar que ningún resultado (valor devuelto por una instrucción de carga o *load*) viola la *consistencia secuencial*.

Un ejemplo de optimización sobre el modelo de consistencia secuencial es la técnica de pre-búsqueda en cache, la cual es posible debido a que el orden en los accesos a memoria por parte de un programa es el que importa a la hora de cumplir con la *consistencia secuencial* (o cualquier otro modelo) y no el orden en el que se piden los permisos, se modifica el estado o incluso se carga en cache el bloque que se quiere acceder. De esta manera surgen dos variantes de pre-búsqueda en cache:

- *Non-binding prefetch*: un núcleo solicita al sistema coherente de memoria que cambie el estado de un bloque en una o más caches, pero no modifica ningún registro ni dato del bloque. Si su cache no tiene el bloque, lo carga.

- *Binding prefetch*: en este caso, el núcleo también solicita el cambio de estado de un bloque en una o mas caches, pero además carga localmente el bloque en un *buffer* entre las caches L1 y L2 ó en registros, de forma que en caso de escritura, los siguientes accesos sobre la misma dirección de memoria encontrarán el nuevo dato modificado. Esto requiere de un tratamiento adicional en núcleos especulativos, ya que si falla la especulación se deben deshacer los cambios sobre el bloque junto con la ejecución especulativa.

En multiprocesadores también surge la especulación consistente en memoria, que implica que cada núcleo debe verificar que cada especulación que realiza es correcta. Una de las técnicas de verificación presentadas por Gharachorloo *et al.* [6] consiste en, después de que un núcleo haya ejecutado una *load* de forma especulativa, verificar que antes de jubilar la instrucción de acceso a memoria (*commit* de la instrucción), el bloque sobre el que se ha especulado no ha sido expulsado de la cache. Este hecho garantiza que el contenido no ha cambiado desde que la *load* se lanzó a ejecución hasta que se dió por completada la operación. Una petición *GetM* sobre este bloque en la cache L2 indicaría que otro núcleo podría observar esta cache fuera de orden, lo que implicaría un fallo de especulación y el núcleo tendría que deshacerla.

Anexos B

Descripción de Benchmarks de la *Suite* INNCABS

- *Round*: esta aplicación consiste en una implementación del problema conocido como la cena de los filósofos, el cual se establece como un benchmark para evaluar la sobrecarga computacional y de la planificación ejercida por el algoritmo de adquisición de una llave sin bloqueo mútuo implementado en una librería estándar de C++. La aplicación lanza un número N determinado de instancias asíncronas (filófosos) que compiten por la adquisición de dos llaves (tenedores), cada una de ellas compartida mutuamente por cada par de filósofos. Cuando un filósofo adquiere las dos llaves, comienza a realizar su tarea (en este caso sería comer, acción simulada mediante una espera de un determinado número de segundos), liberando las llaves una vez ha terminado. El análisis de este benchmark resulta muy interesante puesto que establece un paralelismo muy claro entre los N procesos que compiten por adquirir los recursos compartidos entre ellos.

- *Qap*: resuelve un caso del problema de la asignación cuadrática usando un algoritmo de ramificación y poda. Esta aplicación, junto con *Floorplan*, resulta muy atractiva para el estudio ya que el proceso de poda y cancelación se lleva a cabo para sub-árboles enteros utilizando operaciones atómicas de *test-and-set*. A priori, el uso del mecanismo far-atomics en estos benchmarks podría favorecer el rendimiento. El requerimiento computacional por tarea es pequeño en ambos benchmarks.

- *Floorplan*: esta aplicación implementa un problema de empaquetado en dos dimensiones. Consiste en encontrar la cuadrícula más pequeña posible en la que se pueden contener una serie de celdas descritas en un fichero de entrada sin que se solapen utilizando el algoritmo de poda y cancelación. Esto resulta en

un procedimiento recursivo de tareas paralelas, con un paralelismo *loop-like*[1] en cada uno de sus nodos. Además, la estructura de árbol generada está muy desequilibrada debido a la poda temprana y agresiva de caminos no viables mediante operaciones atómicas.

- *NQueens*: encuentra la solución al problema de las N reinas para un número N dado. Esta implementación recursiva genera un árbol de invocación asíncrono de altura variable (hasta N-1). El esfuerzo computacional de cada nodo es moderado y no requiere de sincronización con otros hilos, puesto que cada uno es independiente del resto. Solamente se tratan los resultados de los hilos adicionales una vez éstos han terminado. El análisis de este benchmark resulta interesante de cara a estudiar cómo de eficiente es la gestión de los hilos por parte de la librería estándar C++11 mediante la primitiva *async*.

[1] Las estructuras *loop-like* simulan un paralelismo en bucle utilizando la primitiva `async` dentro de un bucle básico `for` o `while`, ejecutándose en el hilo principal.

Anexos C

Instalación de GCC10

Este anexo esta dedicado a la explicación en detalle de la instalación del compilador GCC. Para este proyecto se ha sido necesaria una versión de GCC 10 o superior por lo que se ha decidido utilizar GCC 10.1.0[1]. El primer paso consiste en descargar el fichero .tar que contiene los binarios y extraerlo. A continuación se ha de crear un nuevo directorio en el que se va a realizar la instalación.

Es posible que la compilación requiera de ciertas dependencias, por lo que se recomienda ejecutar el comando del Listing C.1 desde el directorio de extracción del fichero.

```
$ ./contrib/download_prerequisites
```

Listing C.1: Instalación de dependencias.

Este fichero se encargará de instalar todas las dependencias necesarias. A continuación se ha de crear el directorio *build* dentro del paquete descargado y desde él llevar a cabo el proceso de configuración pre-instalación mediante la orden del Listing C.2.

```
$ ../configure --disable-multilib --enable-languages=c,c++
    --prefix=/export/home/a739729/gcc10.1.0 --program-suffix=-10.1.0
```

Listing C.2: Configuración de GCC.

En este caso, se configura para que compile tanto *C* como *C++*, se añade el directorio de instalación */export/home/a739729/gcc10.1.0* y se crea el sufijo *-10.1.0* para distinguir esta versión de GCC de otras ya instaladas en el sistema.

El siguiente paso consiste en compilar el paquete con la configuración que se ha añadido, luego desde el directorio *build* del paquete se ha de ejecutar el Listing C.3.

[1] https://ftp.gnu.org/gnu/gcc/gcc-10.1.0/gcc-10.1.0.tar.gz

```
$ make -j n_hilos
```

Listing C.3: Compilación de GCC.

Siendo `n_hilos` la cantidad de núcleos lógicos del sistema que se quieren destinar a la compilación. Es probable que en este punto el proceso se aborte porque no se disponga de permisos sobre algún fichero del paquete. La solución es darle permisos de ejecución a esos ficheros y volver a lanzar el proceso de compilación. En este caso, fue necesario dar permisos de ejecución a los ficheros *move-if-change* y *libgcc/mkheader.sh* desde la raíz del paquete.

El último paso de la instalación consiste en copiar los ficheros generados en el directorio *build* al directorio que se ha indicado en la configuración. Para ello, se debe ejecutar desde el directorio *build* el comando del Listing C.4.

```
$ make install
```

Listing C.4: Instalación de GCC.

Para finalizar y poder utilizar el compilador es necesario añadir los *paths* de la nueva librería al sistema, paso recogido en Listing C.5. En este caso, se debe cambiar el directorio de instalación según el indicado en la configuración.

```
$ LD_LIBRARY_PATH="/export/home/a739729/gcc10.1.0/lib64"
$ PATH="${PATH}:/export/home/a739729/gcc10.1.0/bin"
```

Listing C.5: Añadir paths de GCC al sistema.

Tras la acción anterior ya se puede compilar un programa utilizando GCC 10.1.0. Tal y como se ha configurado en esta explicación, un ejemplo de uso sería el recogido en el Listing C.6.

```
$ gcc-10.1.0 main.c -o main
```

Listing C.6: Ejemplo de compilación con GCC 10.1.0.

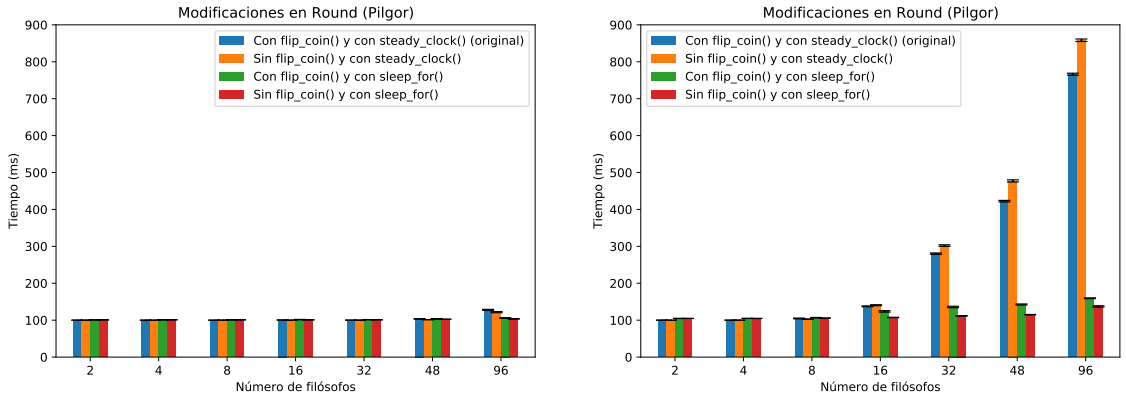
Anexos D

Resultados

En este anexo se presentan las figuras de los benchmarks que no han sido expuestas en el Capítulo 5, siguiendo el mismo orden: mejoras en el benchmark *Round*, implementación de un *thread pool* y evaluación de *far atomics*.

D.1. Mejoras en el Benchmark *Round*

A continuación se muestra en la Figura D.1 la comparación de los resultados de las mejoras sobre el benchmark *Round* en los sistemas *Pilgor* y *x86* en la misma escala en el eje Y.



(a) *Pilgor* (b) *x86*

Figura D.1: Comparativa de mejoras en *Round*.

Resulta notable el aumento del tiempo de ejecución para la primera mejora (sin `flip_coin()` en el caso de *x86*, al contrario que en *Pilgor* en el que sí reduce el tiempo de ejecución. Sin embargo, para el caso en el que se aplican todas mejoras (sin `flip_coin()` y con `steady_clock()`) sí que mejoran los tiempos en ambos sistemas. Una primera hipótesis sobre el impacto negativo de la primera mejora sobre *x86* podría ser que el hecho de que los dos sistemas, además de estar compuestos por distinto hardware

con arquitecturas diferentes, implementan versiones distintas de la biblioteca de C++, dando lugar este resultado.

D.2. Implementación de un *Thread Pool*

Para el benchmark *Qap* fue necesario la creación de ficheros de entrada adicionales a los incluidos en la *suite*, con matrices más pequeñas para poder llevar a cabo la ejecución del mismo en su implementación con `std::async` de forma exitosa sin errores. La Figura D.2 muestra los resultados obtenidos para el fichero *'chr06a.dat'* compuesto por matrices cuadradas de tamaño N=6. Se ha partido el eje Y para poder apreciar más claramente los tiempos obtenidos para la implementación con TBB (parte inferior), ya que son prácticamente cero. El uso de un *thread pool* en esta aplicación supone una mejora muy grande en el tiempo de ejecución. Puede apreciarse, más claramente, en la versión con `std::async` cómo la aplicación reduce su tiempo de ejecución conforme aumentan el número de hilos hardware que puede utilizar.

Puede verse en la Figura D.3 como a pesar de que los tiempos de ejecución son muy bajos debido a la limitación en la creación de hilos en *Pilgor* y a que las tareas son de grano muy fino, el uso de TBB supone igualmente un gran impacto positivo en el tiempo.

Las Figuras D.4a y D.4b muestran la ejecución para un fichero de entrada con matrices cuadradas de tamaño N=12, lo cual implica una complejidad del problema bastante alta, aunque el uso de TBB hace que los tiempos de ejecución no se disparen, mientras que con `std::async` solo es posible llegar a una experimentación exitosa para matrices de tamaño N=4 en *Pilgor* y N=8 en la máquina local como máximo. También es apreciable como la aplicación escala hasta los 96 núcleos en el caso de *Pilgor* en la Figura D.4b y la aplicación tarda más en ejecutarse para los casos de uno a cuatro hilos hardware respecto a la máquina x86.

En el caso de *Floorplan* se ha utilizado el fichero de entrada *'input.5'*, el cual contiene las características de cinco celdas que ha de empaquetar en una cuadrícula lo más pequeña posible sin solaparse. En la Figura D.5 se muestran los resultados para ambas versiones siendo muy notable la mejora mediante TBB. En este caso, con `std::async` puede verse una mayor variación en los tiempos de ejecución respecto al caso anterior y la aplicación no escala con el número de hilos hardware que utiliza sino con el número de núcleos, ya que, como recordatorio, se trata de una máquina con 4 núcleos y 2 hilos por núcleo. Esta aplicación no ha podido ejecutarse de forma exitosa en *Pilgor* en su implementación con `std::async`.

Las Figuras D.6a y D.6b muestran los resultados de *Floorplan* para un fichero de

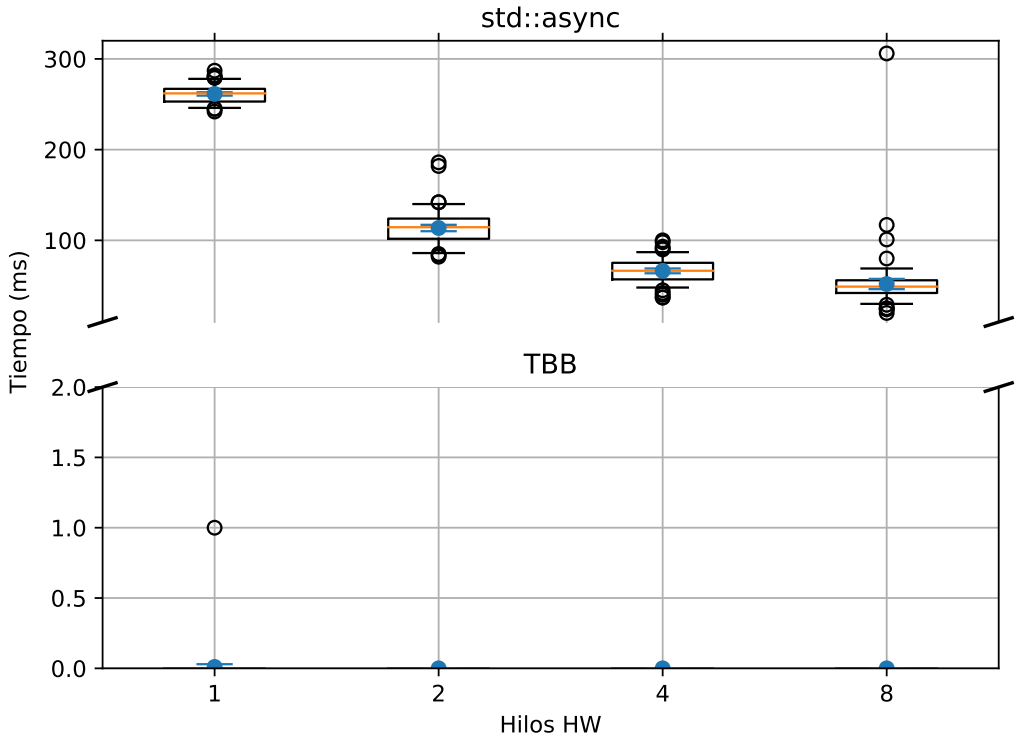


Figura D.2: Comparación en x86 de versiones del benchmark *Qap* implementadas con `std::async` y TBB para el fichero de entrada *'chr06a.dat'*.

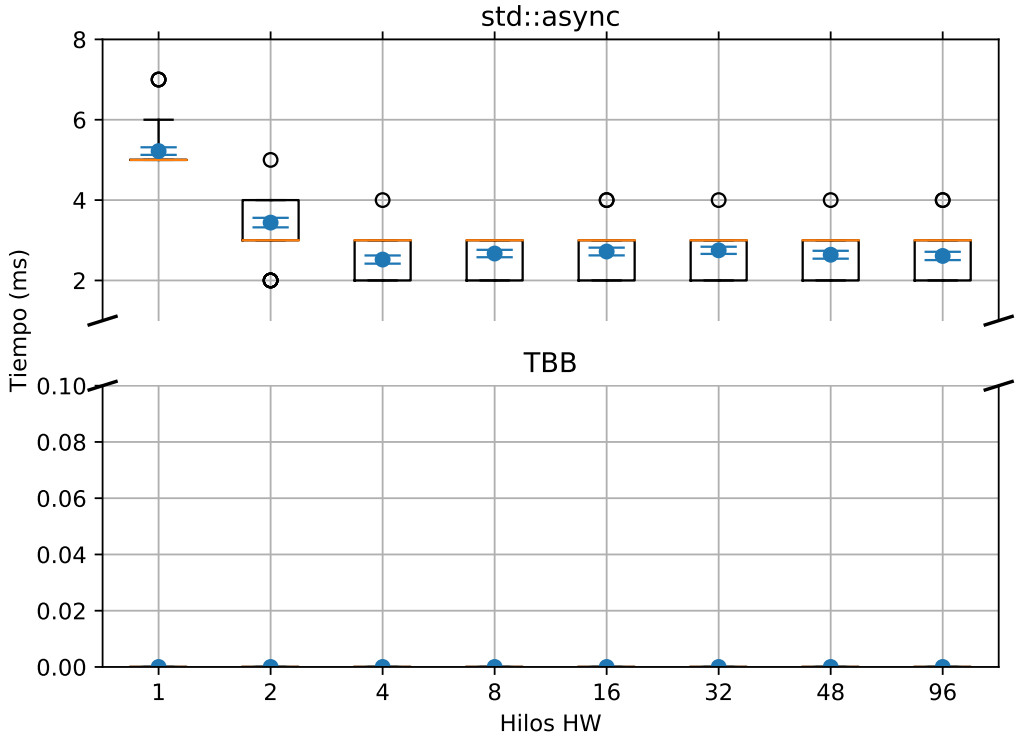
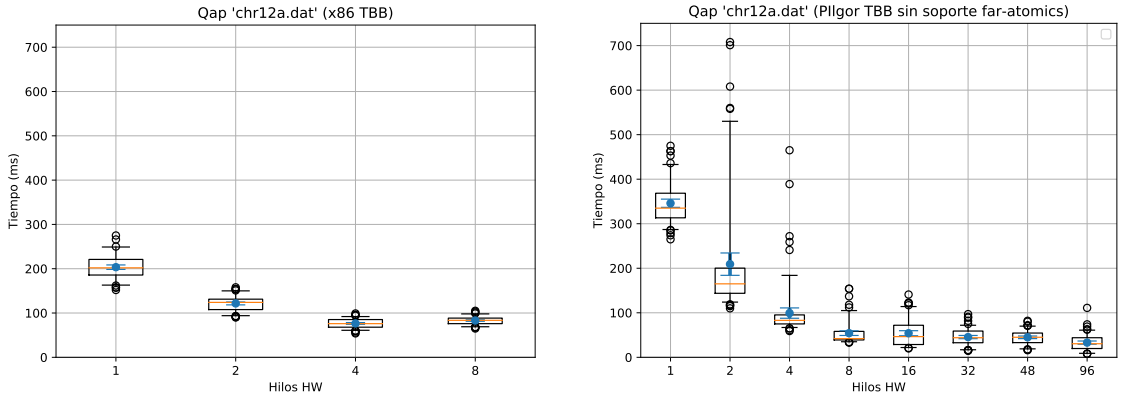


Figura D.3: Comparación en Pilgor de versiones del benchmark *Qap* implementadas con *std::async* y TBB para el fichero de entrada *'chr04a.dat'*, sin soporte *far atomics*



(a) x86 (b) Pilgor

Figura D.4: Ejecución de *Qap* implementada con TBB para el fichero de entrada *'chr12a.dat'*.

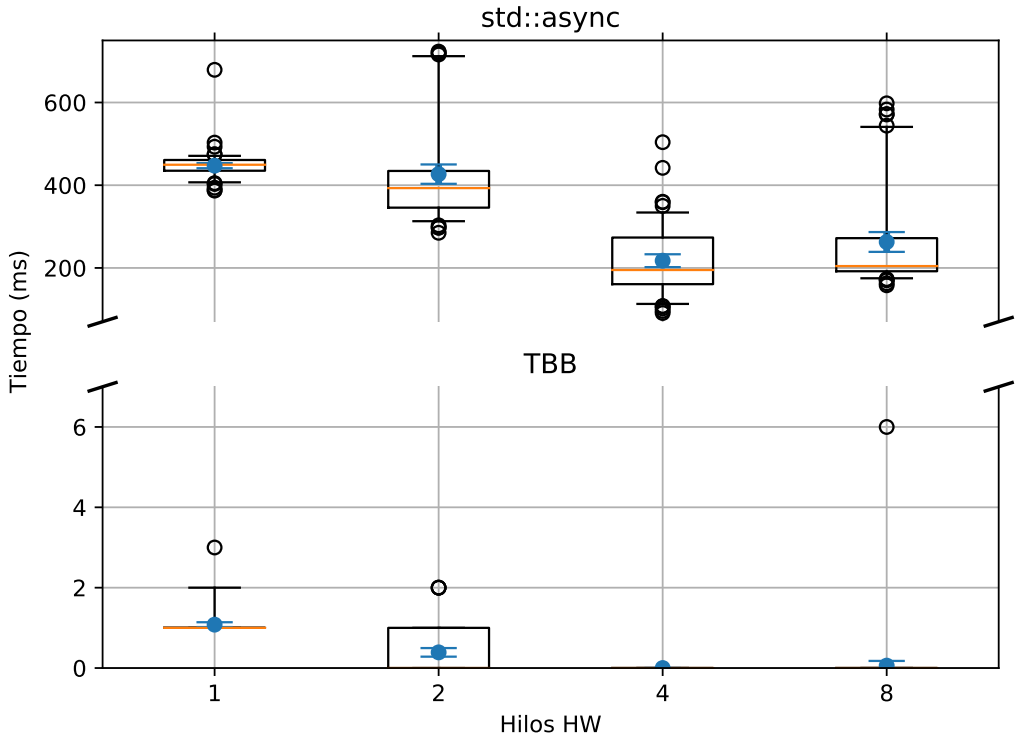
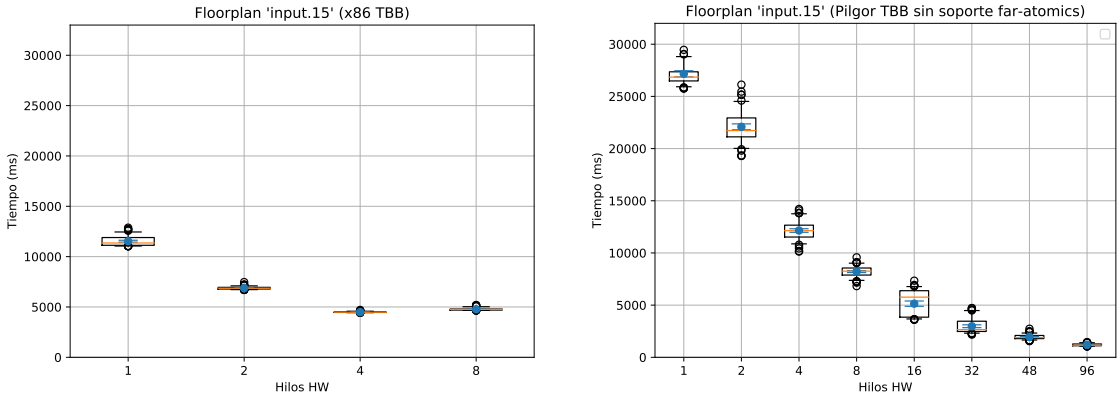


Figura D.5: Comparación en x86 de versiones del benchmark *Floorplan* implementadas con *std::async* y TBB para el fichero de entrada *'input.5'*.

entrada que contiene 15 celdas. Puede observarse como en efecto la aplicación escala con el número de núcleos, escalando hasta 96 en *Pilgor* (cada núcleo tiene un único hilo hardware) y la ejecución en este sistema también es más lenta para los casos de 1 a 8 hilos hardware comparado con los resultados en x86.

La experimentación con mayor tamaño que ha sido posible llevar a cabo en *Pilgor* para *Nqueens* con `std::async` ha sido con N=6. La Figura D.7 muestra una



(a) x86 (b) Pilgor

Figura D.6: Ejecución de *Floorplan* implementado con TBB para el fichero de entrada *'input.15'*.

comparación entre ambas implementaciones. En este experimento para el caso de TBB el uso de varios hilos hardware no escala ya que la complejidad del problema para N=6 es baja luego el hecho de utilizar varios hilos añade más sobrecarga que el lanzar la totalidad de la aplicación en un solo hilo hardware.

Las Figuras D.8a y D.8b exponen los resultados obtenidos para N=12 en la versión implementada con TBB. Viendo el caso de x86 sí que podría confirmarse que la aplicación escala con el número de núcleos disponible y no con el número de hilos hardware y el tiempo de ejecución también es peor en los casos de 1 a 8 hilos en *Pilgor*, ya que la potencia mono-núcleo del chip es menor, aunque consigue reducir el tiempo mínimo de x86 añadiendo más núcleos a la aplicación.

D.3. Evaluación de *Far Atomics*

A continuación se muestran pares de ejecuciones en *Pilgor* para los benchmarks implementados con TBB *Qap*, *Floorplan* y *Nqueens* comparando los resultados obtenidos en detalle para ejecuciones sin soporte *far atomics* con los obtenidos con soporte *far atomics* únicamente en la aplicación y con soporte en la aplicación y en la biblioteca TBB.

En ambas comparaciones sobre las Figuras D.9 y D.10 observa como los tiempos de ejecución empeoran ligeramente cuantas más instrucciones *far atomics* se utilizan en el caso del benchmark *Qap*.

Para *Floorplan*, puede verse en las Figuras D.11 y D.12 cómo las diferencias son más pequeñas y poco significativas, ya que en algunos casos se obtienen tiempos medios de ejecución mejores en los experimentos sin soporte y en otros en los experimentos con soporte. Las diferencias son de pocos milisegundos en todos los casos.

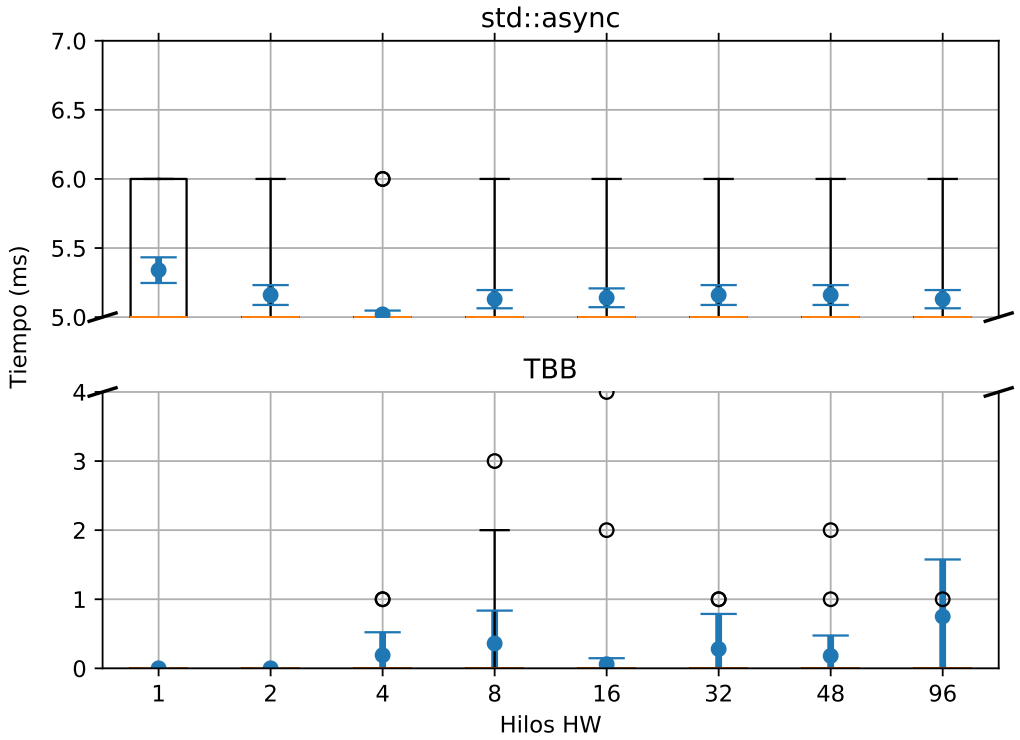
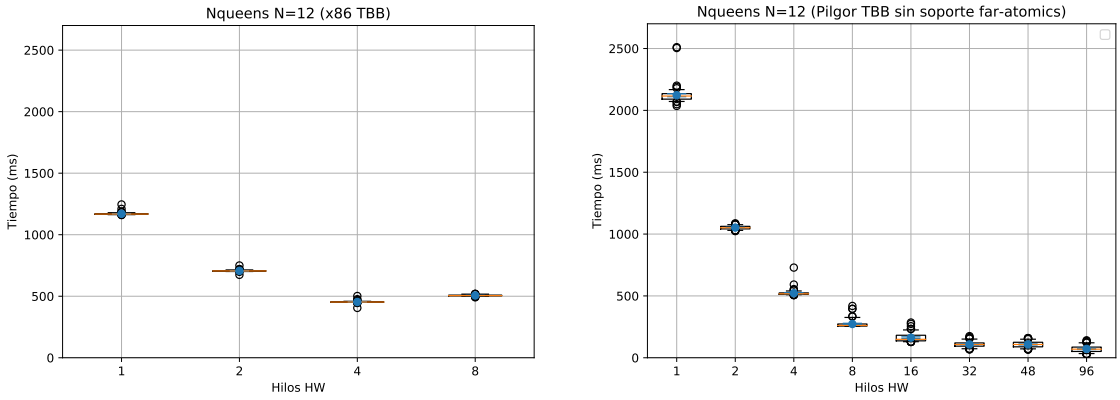


Figura D.7: Comparación en *Pilgor* de versiones del benchmark *Nqueens* implementadas con `std::async` y TBB para N=6, , sin soporte *far atomics*.

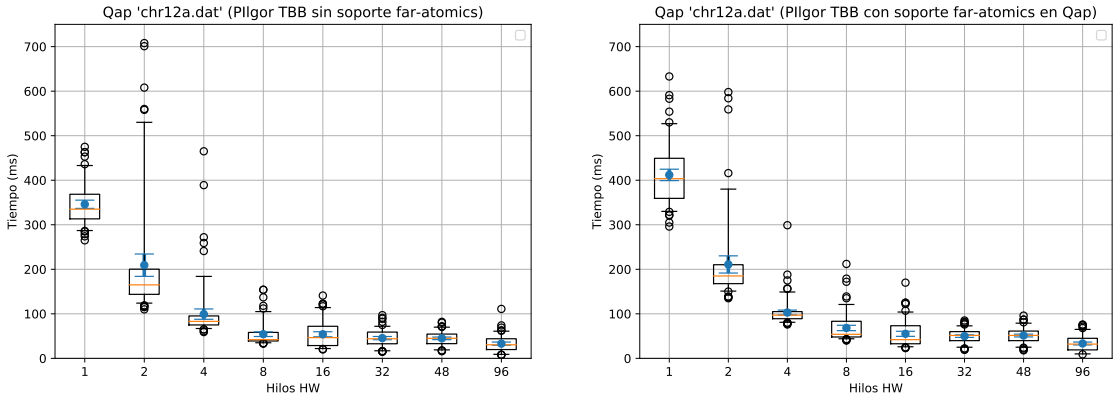
A pesar de que el benchmark *Nqueens* es el que no requiere de sincronización entre procesos, es en el que más influye negativamente el uso de las instrucciones *far atomics* como se aprecia en las Figuras D.13 y D.14.

Por último, por completitud, las Figuras D.15a y D.15b muestran dos ejecuciones del benchmark *Round* en *x86* para N=8 y N=16, variando la cantidad de hilos hardware disponibles del sistema para evaluar la escalabilidad de la aplicación como se ha hecho en Pilgor. Se aprecia como el mejor tiempo de ejecución se consigue para 2 y 4 hilos hardware respectivamente, en ambos casos cuando el número de filósofos que se ejecuta en cada hilo es de 4, es decir, que la aplicación rinde más con una sobre-suscripción de 4 tareas por hilo, luego la aplicación se comporta de manera similar en ambos sistemas, *x86* y *Pilgor*.



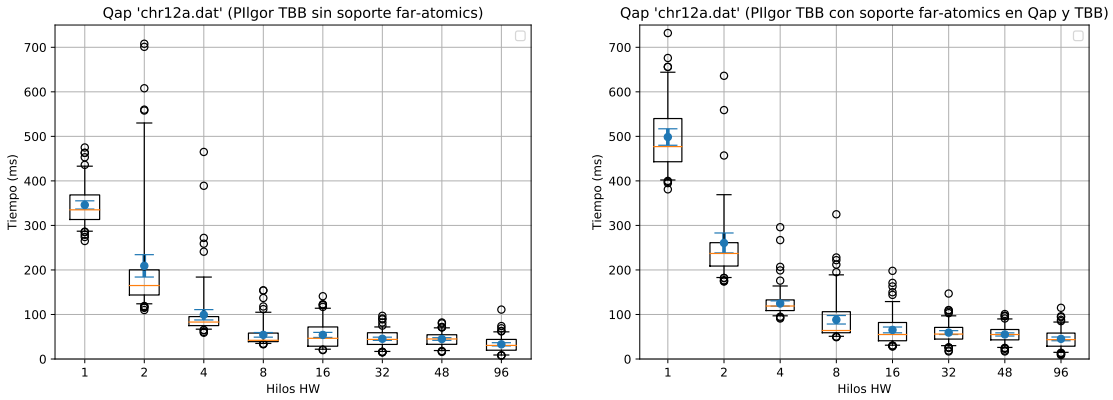
(a) x86 (b) Pilgor

Figura D.8: Ejecución de *Nqueens* implementado con TBB para N=12.



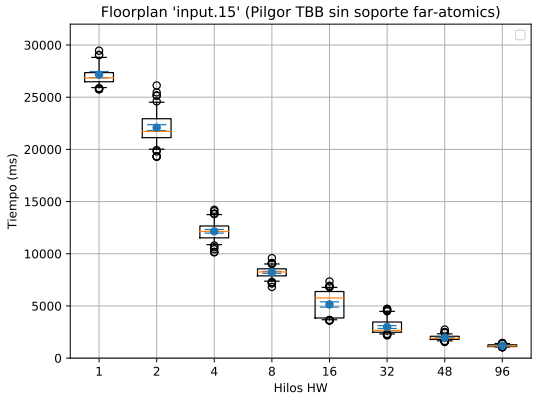
(a) Sin soporte *far atomics*. (b) Con soporte *far atomics* en *Qap*

Figura D.9: Ejecuciones de *Qap* en *Pilgor* con TBB para fichero de entrada 'chr12a.dat' variando los contextos hardware disponibles. Comparación entre versión sin soporte *far atomics* y con soporte en aplicación.

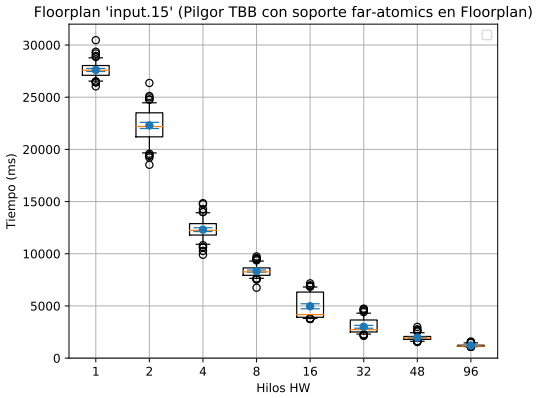


(a) Sin soporte *far atomics*. (b) Con soporte *far atomics* en *Qap* y TBB.

Figura D.10: Ejecuciones de *Qap* en *Pilgor* con TBB para fichero de entrada 'chr12a.dat' variando los contextos hardware disponibles. Comparación entre versión sin soporte *far atomics* y con soporte en *Qap* y TBB.

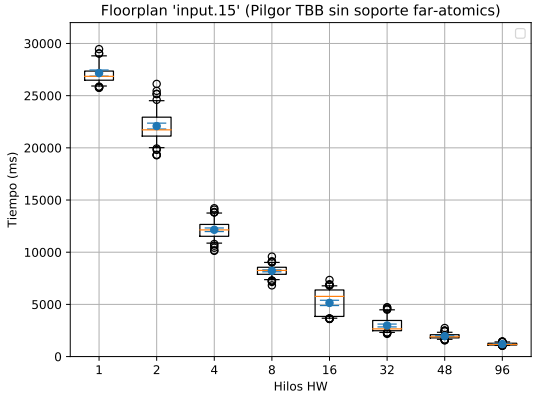


(a) Sin soporte *far atomics*.

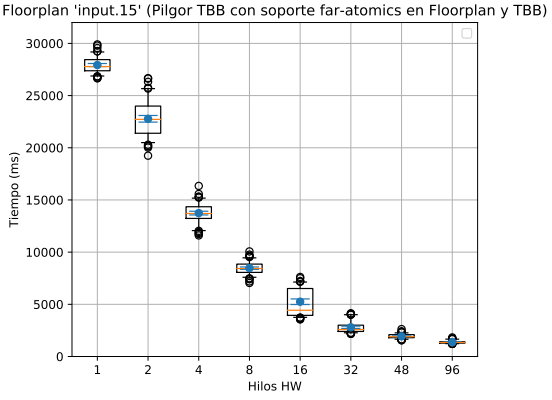


(b) Con soporte *far atomics* en *Floorplan*.

Figura D.11: Ejecuciones de *Floorplan* en *Pilgor* con TBB para fichero de entrada 'input.15' variando los contextos hardware disponibles. Comparación entre versión sin soporte *far atomics* y con soporte en *Floorplan*.

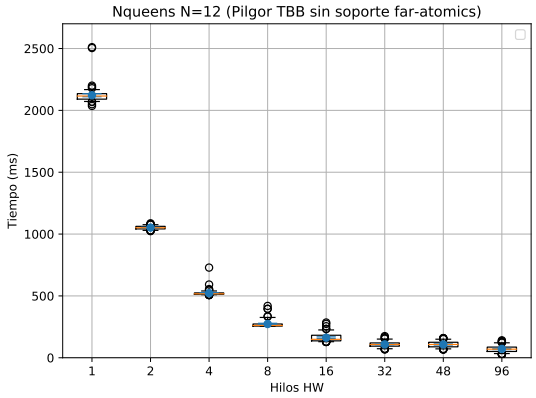


(a) Sin soporte *far atomics*.

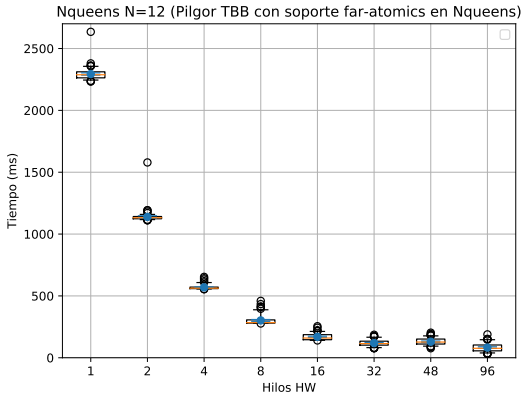


(b) Con soporte *far atomics* en *Floorplan* y TBB.

Figura D.12: Ejecuciones de *Floorplan* en *Pilgor* con TBB para fichero de entrada 'input.15' variando los contextos hardware disponibles. Comparación entre versión sin soporte *far atomics* y con soporte en *Floorplan* y TBB.

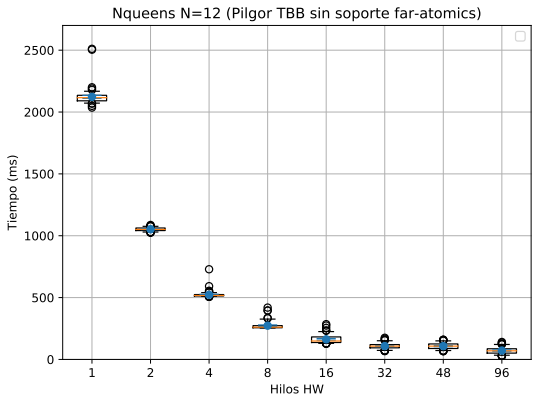


(a) Sin soporte *far atomics*.

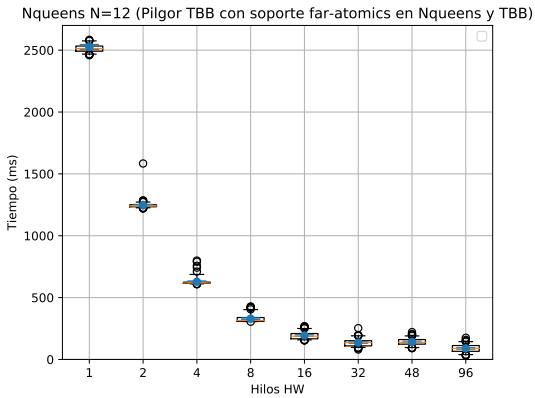


(b) Con soporte *far atomics* en aplicación.

Figura D.13: Ejecuciones de *Nqueens* en *Pilgor* con TBB para N=12 variando los contextos hardware disponibles. Comparación entre versión sin soporte *far atomics* y con soporte en *Nqueens*.

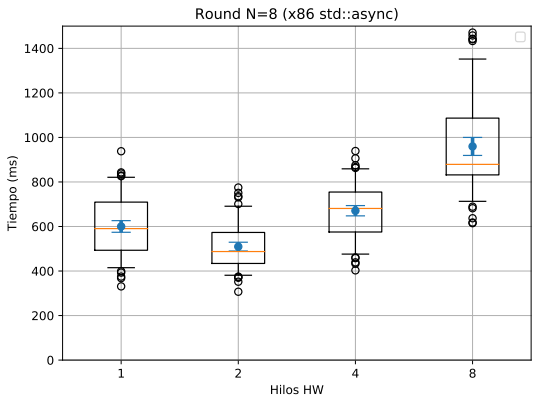


(a) Sin soporte *far atomics*.

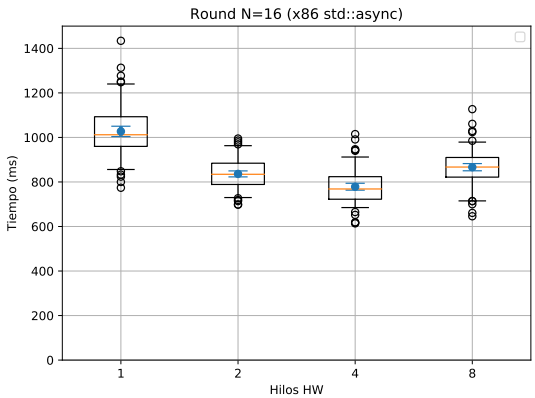


(b) Con soporte *far atomics* en App y TBB.

Figura D.14: Ejecuciones de *Floorplan* en *Pilgor* con TBB para fichero de entrada 'input.15' variando los contextos hardware disponibles. Comparación entre versión sin soporte *far atomics* y con soporte en *Nqueens* y TBB.



(a) N=8.



(b) N=16.

Figura D.15: Ejecuciones de *Round* en *x86* variando los contextos hardware disponibles.

Anexos E

Dedicación

En esta sección se muestra mediante un diagrama de Gantt en la Figura E.1 la organización y el desarrollo del proyecto a lo largo del tiempo, destacando las tareas principales que se han realizado. Es importante mencionar que las tareas de experimentación y análisis mediante las herramientas como *strace* o *perf*, la automatización de experimentos, así como la representación y análisis de resultados se realizan de forma prácticamente iterativa para cada benchmark de la *suite* y por ello aparecen muy solapadas en el tiempo. También cabe destacar que es en estas tres tareas en las que se ha dedicado más tiempo, ya que el trabajo supone una cantidad considerable de experimentación cuyos resultados se deben analizar y mostrar adecuadamente mediante gráficos o cálculos adicionales.

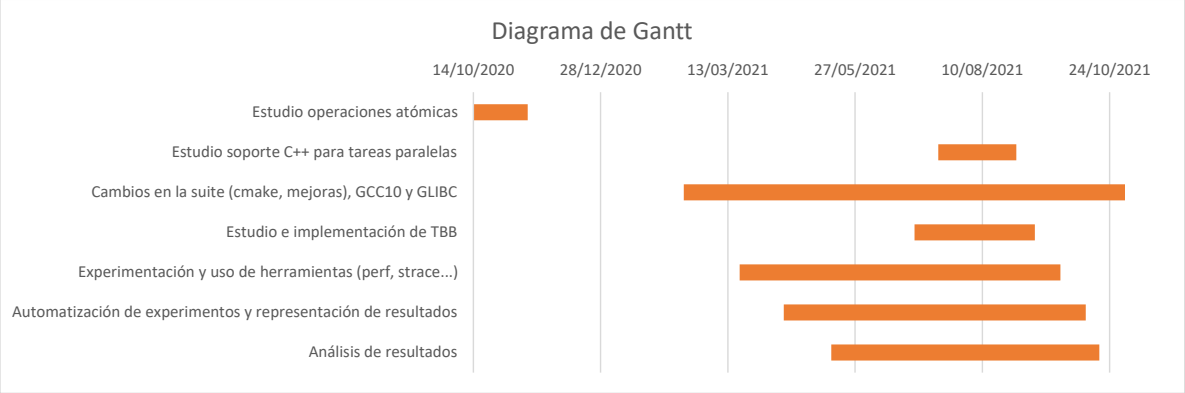


Figura E.1: Diagrama de Gantt desglosando la duración de las tareas principales del proyecto.

Por último, la Tabla 3.1 recoge el tiempo dedicado a cada tarea que se ha comentado, siendo la parte de experimentación y análisis la más destacable junto con la redacción del presente documento. También se ha dedicado bastante tiempo a preparar la *suite* y el entorno de experimentación, así como para la instalación de GCC10 y GLIBC, aunque finalmente esta última no se consiguió enlazar con el proyecto como se ha comentado anteriormente.

Atomics	C++	Suite	TBB	Exp	Aut&Repr	Análisis	Memoria	TOTAL
20	17	55	33	75	35	35	77	347

Tabla E.1: Tiempo dedicado a cada tarea (Horas).