Universidad Zaragoza
1542

Trabajo Fin de Grado

Análisis, Evaluación e Implementación de Mecanismos de Sincronización de Hilos en Múltiples Niveles de Abstracción de un Sistema Informático

Analysis, Evaluation, and Implementation of Thread Synchronization Mechanisms in Multiple Abstraction Levels of a Computer System

Autor

Emanuel Alexandru Georgescu

Directores

Alejandro Valero Bresó

Rubén Gran Tejero

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2021

AGRADECIMIENTOS

Me gustaría en primer lugar agradecer a Rubén y Álex su apoyo, ayuda y guía durante las reuniones y el conocimiento adquirido a lo largo de este trabajo. De nuevo a Álex por sus magníficas clases de Arquitectura de Sistemas.

En segundo lugar, a mis padres por su apoyo, sacrificio y paciencia. A mis amigos por estar siempre ahí.

Finalmente, al Grado en ITyS de Telecomunicación por la formación obtenida.

Análisis, Evaluación e Implementación de Mecanismos de Sincronización de Hilos en Múltiples Niveles de Abstracción de un Sistema Informático

RESUMEN

La mayoría de asignaturas relacionadas con sistemas informáticos recurren a niveles de abstracción para ocultar la complejidad de los niveles subyacentes y centrarse en los conocimientos propios y relevantes de cada nivel. Esta organización implica que, en ocasiones, los niveles de abstracción puedan verse como independientes y sin relación entre sí.

El Departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza ha desarrollado varios proyectos docentes con el objetivo de ofrecer al estudiante una visión global y vertical de un sistema informático, mediante un enfoque práctico que involucra a múltiples asignaturas del Grado de Informática correspondientes a distintos niveles de abstracción. Este enfoque hace uso de una plataforma multihilo común (Raspberry Pi 3B) para implementar distintos componentes de una aplicación paralelizable y protegida con los mecanismos de sincronización referidos como mutexes.

El presente trabajo parte de los proyectos docentes mencionados y se centra en realizar la parte experimental de los mismos, verificando la corrección de mutexes implementados en distintos niveles de abstracción de un sistema informático, experimentando sobre los mismos, creando un entorno de trabajo adecuado para su evaluación y obteniendo los resultados experimentales y las conclusiones más relevantes. Para ello, se diseñan cargas de trabajo consistentes en programas concurrentes protegidos mediante mutexes del nivel de abstracción de Biblioteca, Sistema Operativo y Arquitectura del Lenguaje Máquina, los cuales están construidos mediante funciones de la librería estándar de C++, llamadas al sistema futex y primitivas atómicas de la arquitectura ARMv8, respectivamente.

Tras el diseño de las cargas de trabajo, se crean bancos de pruebas y se obtienen resultados experimentales consistentes en el tiempo de ejecución e incremento de la temperatura del chip por cada tipo de mutex, variando la contención ejercida por las distintas instancias en ejecución. El análisis de los resultados permite establecer un balance entre rendimiento y consumo obtenido frente a la programabilidad del mutex en cada nivel de abstracción.

A partir del análisis de los resultados, este trabajo concluye que el mutex de

Biblioteca proporciona el mejor rendimiento en escenarios reales, mientras que los mutexes de Sistema Operativo y Arquitectura del Lenguaje Máquina basados en espera no activa resultan convenientes en un escenario de alta contención, además de poder contribuir en la reducción de la temperatura del chip.

Índice

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

El presente capítulo introduce la motivación, los objetivos y el alcance que definen al presente trabajo, así como una descripción de la estructura general de la memoria.

1.1. Motivación

Los sistemas informáticos suelen organizarse en distintos niveles de abstracción. Estos niveles son auto-contenidos y permiten esconder la complejidad del sistema completo, facilitando por tanto el uso del mismo. Es decir, un usuario puede trabajar en un nivel determinado sin conocer los detalles de implementación del resto de niveles. Sin embargo, los niveles de abstracción se relacionan entre sí, de manera que cada nivel proporciona un interfaz al resto de niveles. Estos interfaces modelan una abstracción simplificada de la complejidad subyacente y establecen claros límites a través de las distintas partes del sistema [1].

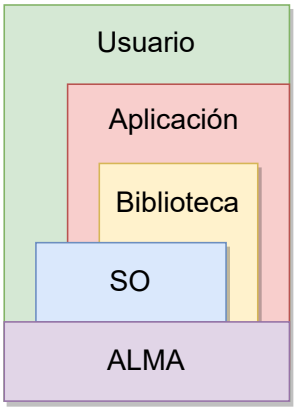


Figura 1.1: Ejemplo de niveles de abstracción de un sistema informático.

La Figura 1.1 muestra un ejemplo de sistema informático organizado en cuatro niveles de abstracción. A través de un enfoque descendente desde el usuario hasta el hardware, estos niveles quedan definidos en Aplicación, Biblioteca, Sistema Operativo (SO) y Arquitectura del Lenguaje Máquina (ALMA). Cada nivel puede relacionarse con aquellos con los que establece una frontera. En otras palabras, cada nivel de abstracción definido interacciona con todos los inferiores. Por ejemplo, en el caso del nivel de Aplicación, este puede usar las funciones de biblioteca, las llamadas al sistema operativo o las primitivas del lenguaje ensamblador disponibles en la ALMA para lograr un mejor aprovechamiento de los recursos a costa de una mayor complejidad y conocimiento del sistema. De manera similar, el usuario interactúa con el nivel de Aplicación, SO (a través de la línea de

comandos) y ALMA del sistema.

La mayoría de asignaturas relacionadas con la ingeniería de computadores, incluyendo a asignaturas tanto del Grado en Informática como del Grado en Ingeniería de Tecnologías y Servicios (ITyS) de Telecomunicación, recurren a abstracciones con el fin de diseñar y explicar los sistemas informáticos. Esto permite ocultar la complejidad de los niveles subyacentes. Las abstracciones ayudan a reforzar el proceso de aprendizaje, ya que facilitan que los estudiantes se centren en los conceptos relevantes y propios del nivel en cuestión, como el paralelismo, la concurrencia, la atomicidad o la consistencia [2, 3, 4]. Sin embargo, debido a esta focalización, los estudiantes también pueden perder la visión global del sistema y no conseguir integrar los niveles de abstracción, viendo cada asignatura como independiente y sin relación con las demás. En concreto, el estudiante puede olvidarse de las implicaciones hardware bajo las abstracciones de alto nivel, sobretodo los aspectos referentes al rendimiento o al consumo/temperatura en el sistema.

1.2. Objetivos y Alcance

El objetivo principal del presente Trabajo Fin de Grado (TFG) es realizar la parte experimental de los proyectos docentes recogidos en la Sección 2.1, logrando una visión integral de un sistema informático. Para ello, se estudia, verifica la corrección e implementan los mecanismos de sincronización llamados mutex, creando entornos de trabajo y bancos de prueba adecuados para evaluarlos y se obtienen y analizan los resultados técnicos de los mismos en los niveles de Biblioteca, SO y ALMA. Para conseguir este propósito se definen los siguientes objetivos:

- Estudiar la sincronización de hilos de ejecución en computadores paralelos de memoria compartida, como es el caso de la plataforma Raspberry Pi 3B (arquitectura ARMv8) y configuración del entorno experimental.

- Utilizar y verificar la corrección de las primitivas de sincronización disponibles en distintos niveles de abstracción software de un sistema informático: Biblioteca, SO y ALMA.

- Diseñar distintas cargas de trabajo y escenarios de contención de cara a evaluar experimentalmente cada solución de sincronización, así como automatizar el lanzamiento de ejecuciones y obtención de resultados.

El presente proyecto describirá cómo se han alcanzado todos los objetivos anteriores. Además, los resultados experimentales y las conclusiones obtenidas fruto de este trabajo

se han publicado recientemente en la revista JCR *Elsevier Journal of Parallel and Distributed Computing* [5].

1.3. Estructura de la Memoria

El resto del documento se organiza como sigue. El Capítulo 2 explica el contexto en el que se realiza el trabajo, así como resume el trabajo actual y otros trabajos relacionados con el presente proyecto. El Capítulo 3 introduce la cola de tareas concurrente protegida mediante semáforos, los define y analiza los mecanismos de sincronización en cada nivel de abstracción. El Capítulo 4 describe el entorno experimental, incluyendo la plataforma hardware, el diseño de cargas de trabajo y las herramientas de medida. El Capítulo 5 muestra y razona los resultados de rendimiento e incremento de temperatura del chip de los mecanismos de sincronización de cada nivel de abstracción. Finalmente, el Capítulo 6 expone las conclusiones principales del trabajo, líneas futuras y dedicación.

Capítulo 2

Contexto

Este capítulo establece el punto de partida del presente trabajo a través de una breve descripción de una serie de proyectos de innovación docente. A continuación, se describe el trabajo actual, así como los trabajos relacionados.

2.1. Trabajo Previo

Para entender y explotar un sistema informático es muy importante alcanzar una visión global de todos los niveles de abstracción del mismo, incluyendo al procesador, sistema operativo y lenguaje de alto nivel. La organización de las asignaturas del Grado de Informática implica la creación de compartimentos estancos, permitiendo al estudiante centrarse en un sólo nivel de abstracción para afianzar los conceptos. Sin embargo, esta organización también puede tender a aislar dichos conceptos y especializar plataformas. Para romper esta tendencia que priva de una visión global de sistema, se han realizado los siguientes proyectos de innovación docente con un enfoque transversal y práctico para el Grado en Ingeniería Informática de la Universidad de Zaragoza, abarcando múltiples asignaturas del mismo:

- Estudio y Diseño de una Plataforma Común de Trabajo para la Mejora del Aprendizaje en el Grado en Ingeniería Informática[1], 2016.
- Experimentación y Difusión de una Plataforma Común de Trabajo para la Mejora del Aprendizaje en el Grado en Ingeniería Informática[2], 2017.
- Plataforma Multi-Asignatura para la Mejora del Aprendizaje en el Grado en Ingeniería Informática: Ensayo con Alumnos[3], 2018.

[1] `https://innovaciondocente.unizar.es/convocatoria2016/ventanas/ver_ficha_proyecto.php?proyecto=270`

[2] `https://innovaciondocente.unizar.es/convocatoria2017/ventanas/ver_ficha_proyecto.php?proyecto=365`

[3] `https://innovaciondocente.unizar.es/convocatoria2018/ventanas/ver_ficha_proyecto.php?proyecto=246`

Asignatura	Curso y semestre	Nivel de Abstracción
Informática Gráfica	4to Otoño	Aplicación: implementación de aplicaciones paralelas que requieren mutexes
Programación de Sistemas Concurrentes y Distribuidos	2do Otoño	Biblioteca: uso de lenguaje estándar de alto nivel para construir mutexes
Sistemas Operativos	2do Otoño	SO: uso de servicios del SO para construir mutexes
Multiprocesadores	3ro Primavera	ALMA: uso de herramientas de la arquitectura del procesador para construir mutexes

Tabla 2.1: Asignaturas del Grado en Ingeniería Informática involucradas en los proyectos de innovación docente.

Cada proyecto se construye sobre el anterior y se centran en: *i)* realizar un estudio de mercado, seleccionando una plataforma común de trabajo, así como realizar el material docente del proyecto, estableciendo un conjunto de asignaturas implicadas, *ii)* realizar la puesta a punto de la plataforma y la implementación de los distintos tipos de mutex y *iii)* realizar los primeros ensayos con estudiantes voluntarios para evaluar la plataforma y refinar el material docente.

Como fruto de los proyectos anteriores, se ha propuesto una serie de laboratorios de distintas asignaturas en los cuales trabajar con cada nivel de abstracción de un sistema informático consistente en un trazador de rayos paralelo. Estos laboratorios abarcan desde el nivel algorítmico del trazador (nivel más alto) hasta las instrucciones atómicas necesarias para garantizar la atomicidad. Cada laboratorio se centra en un sólo nivel de abstracción, pero muestra las interacciones con los niveles restantes. La Tabla 2.1 muestra las asignaturas implicadas.

El nivel de Aplicación implementa un algoritmo trazador de rayos, donde se paraleliza el cómputo dividiendo una imagen en regiones, cada una de las cuales supone una tarea independiente a procesar por un hilo. El nivel de Biblioteca implementa y administra una cola de tareas con acceso concurrente por parte de múltiples hilos, a la cual se accede en exclusión mutua mediante semáforos construidos con mutexes para repartir las tareas. El nivel de SO gestiona el acceso a la cola concurrente por parte de los hilos mediante llamadas al sistema futex. Finalmente, el nivel de ALMA usa instrucciones de código máquina para implementar el acceso a la cola, pudiendo reducir el consumo energético respecto a las soluciones de Biblioteca y SO.

2.2. Trabajo Actual

El presente Trabajo Fin de Grado (TFG) parte de los proyectos docentes mencionados y tiene como fin verificar la corrección de los distintos mecanismos de sincronización basados en mutex, así como obtener resultados experimentales de rendimiento y temperatura de cada mutex, razonando las diferencias entre las distintas soluciones. Para ello, se diseñan cargas de trabajo con distintos grados de estrés o contención en los mecanismos de sincronización. Este conjunto de programas y datos de entrada tienen como objetivo que los estudiantes que participan en los laboratorios propuestos en los proyectos docentes dispongan de un entorno de experimentación apropiado para obtener las mismas conclusiones que se derivan en el presente trabajo.

2.3. Trabajos Relacionados

Existen diferentes formas de enfocar la docencia de las asignaturas relacionadas con computación paralela y distribuida. Algunos enfoques se centran exclusivamente en abstracciones de alto nivel para aligerar la carga algorítmica de las aplicaciones [6, 7], mientras que otros enfoques recurren a abstracciones de bajo nivel, como el lenguaje ensamblador, para entender las implicaciones a bajo nivel en una ejecución paralela [8]. A diferencia de estos enfoques, los proyectos docentes a partir del cual se desarrolla el presente trabajo refuerzan los conocimientos desde el nivel de abstracción más alto hasta el más bajo, involucrando aplicaciones o cargas de trabajo paralelas y complejas presentes en un sistema informático.

Capítulo 3

Mecanismos de Sincronización

Este capítulo presenta y describe el funcionamiento del mecanismo de sincronización conocido como semáforo, el cual se utiliza para proteger el acceso a una cola productor/multiconsumidor en exclusión mutua. El semáforo se implementa mediante mutexes en distintos niveles de abstracción: Biblioteca, SO y ALMA. En concreto, el nivel de Biblioteca hace uso de mutexes del estándar C++, el nivel de SO implementa los mutexes mediante llamadas al sistema, mientras que el nivel de ALMA construye los mutexes directamente con instrucciones de lenguaje ensamblador. Todos los códigos fuente del trabajo pueden consultarse en el repositorio de GitHub: `https://github.com/EmanuelAlexandru/TFG-Emanuel`.

3.1. Cola de Tareas Concurrente

Se usa una cola productor/multiconsumidor para paralelizar el cómputo de la aplicación. La Figura 3.1 muestra cómo el hilo principal de la aplicación, el productor, se encarga de dividir la carga de trabajo en tareas y encolar dichas tareas en la cola. Por su parte, los hilos de trabajo, referidos como consumidores, desencolan y procesan cada tarea en cada núcleo del procesador. En concreto, la figura muestra un instante de tiempo en el que el productor encola la tarea 9, mientras que los consumidores desencolan las tareas 0, 1 y 2. La cola contiene las tareas desde la 3 hasta la 8 esperando a ser desencoladas por parte de los consumidores. La gestión de la cola sigue una política FIFO (*First In, First Out*). Se refiere al lector al Código fuente 4.2 para consultar un

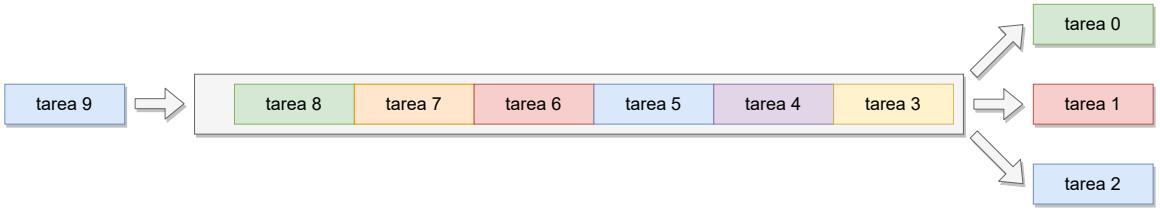


Figura 3.1: Cola de tareas concurrente productor/multiconsumidor.

ejemplo de código fuente de una cola concurrente.

La solución al problema de la sincronización de los hilos en la cola requiere satisfacer una serie de condiciones: i) el hilo productor no debe encolar tareas si la cola está llena, ii) los hilos consumidores no deben extraer tareas si la cola está vacía, iii) se debe proteger el acceso a la cola por parte de los hilos mediante exclusión mutua para preservar el estado de la misma. Estas condiciones se pueden satisfacer mediante el uso de semáforos.

3.2. Semáforos

Un semáforo es un tipo de datos abstracto que permite el uso de un recurso (sección crítica de código) en exclusión mutua cuando varios procesos o hilos compiten por él. Su estado interno cuenta el número de procesos que pueden acceder al recurso. El semáforo cuenta con tres operaciones básicas:

- `init()`: inicializa el estado del semáforo antes que cualquier otro hilo ejecute `wait()` o `signal()`. Si se desea un acceso en exclusión mutua a la sección crítica, el semáforo debe inicializarse a 1, resultando en un semáforo binario.

- `wait()`: si el estado es 0, el hilo permanece bloqueado en una cola del semáforo hasta que sea despertado por otro hilo. Si el estado representa a un número positivo, se permite el acceso a la sección crítica y la función decrementa el contador del estado.

- `signal()`: esta operación permite al hilo señalizar al semáforo una vez haya terminado el acceso a la sección crítica, aunque cualquier hilo puede ejecutarla. La función despierta a un hilo si hay algún otro esperando. De lo contrario incrementa el estado.

Las operaciones `wait()` y `signal()` deben implementarse mediante operaciones atómicas para garantizar su corrección.

3.3. Implementación de Semáforo con Mutexes de Biblioteca

El semáforo del nivel de biblioteca se implementa con mecanismos de sincronización referidos como mutexes y variables de condición de la librería estándar de C++. El Código 3.1 muestra la implementación de este semáforo, incluyendo los métodos `signal()` y `wait()`. La instrucción `std::unique_lock<std::mutex> lck(mtx)`

Código 3.1: Construcción de la clase `Semaphore` mediante mutexes de biblioteca y variables de condición.

```
class Semaphore {
private:
    std::mutex mtx;
    std::condition_variable_any cv;
    int count; //natural asociado al semaforo
    bool initialized;
    ...
}
void Semaphore::signal() {
    std::unique_lock<std::mutex> lck(mtx);
    assert(initialized);
    count++;
    cv.notify_all();
}
void Semaphore::wait() {
    std::unique_lock<std::mutex> lck(mtx);
    assert(initialized);
    while(count == 0) {
        cv.wait(lck);
    }
    count--;
}
```

bloquea al mutex `mtx` cuando la variable `std::unique` se construye y lo desbloquea cuando se destruye. Esta acción protege toda la sección de código que sigue a la instrucción[1], garantizando la atomicidad de ambos métodos. Las instrucciones `cv.wait(lck)` y `cv.notify_all()` son métodos de una variable de condición `cv`. Las variables de condición son primitivas de sincronización que bloquean a uno o más hilos hasta que otro hilo modifica la variable compartida (condición) y notifica a la variable de condición. La invocación `cv.wait(lck)` bloquea al hilo actual en una cola hasta que la variable de condición es invocada. La llamada `cv.notify_all()` despierta a todos los hilos dormidos en la variable de condición.

3.4. Implementación de Semáforo con Mutexes de SO

El nivel de SO construye el semáforo utilizando la llamada al sistema `futex` y primitivas atómicas para la implementación de las primitivas mutex requeridas en el semáforo, prescindiendo de la librería estándar de C++. El Código 3.2 muestra la implementación. Los cambios más reseñables frente a la implementación de semáforo

[1]Se refiere a los métodos completos de `wait()` y `signal()`.

Código 3.2: Construcción de la clase `Semaphore` mediante mutexes de SO.

```
class Semaphore {
private:
    mutex mtx;
    int count; //natural asociado al semaforo
    bool initialized;
    void adormir(int ve); //dormir si ve es igual a count
    void despertar();
    ...
void Semaphore::adormir(int ve){
        syscall(__NR_futex, &(count), FUTEX_WAIT, ve, NULL, 0, 0);
}
void Semaphore::despertar(){
        syscall(__NR_futex, &(count), FUTEX_WAKE, INT_MAX, NULL, 0, 0);
}
void Semaphore::signal() {
    mtx.lock();
    assert(initialized);
    count++;
    despertar();
    mtx.unlock();
}
void Semaphore::wait() {
    mtx.lock();
    assert(initialized);
    while(count == 0) {
        int vr = count;
        mtx.unlock();
        adormir(vr);
        mtx.lock();
    }
    count--;
    mtx.unlock();
}
```

de biblioteca son los siguientes:

- La cola interna de la variable de condición se modela con dos métodos: `adormir()` y `despertar()`, los cuales reemplazan las invocaciones `cv.wait(lck)` y `cv.notify_all()`, respectivamente. Estos métodos se construyen con llamadas al sistema futex[2]. Las llamadas al sistema futex se describen más abajo en la descripción de los mutexes de SO.

- El mutex `std::unique_lock<std::mutex> lck(mtx)` de los métodos `wait()` y `signal()` se reemplaza por una instrucción `mtx.lock()` al comienzo y una instrucción `mtx.unlock()` al final de cada método. Por su parte, la llamada

[2] `https://man7.org/linux/man-pages/man2/futex.2.html`

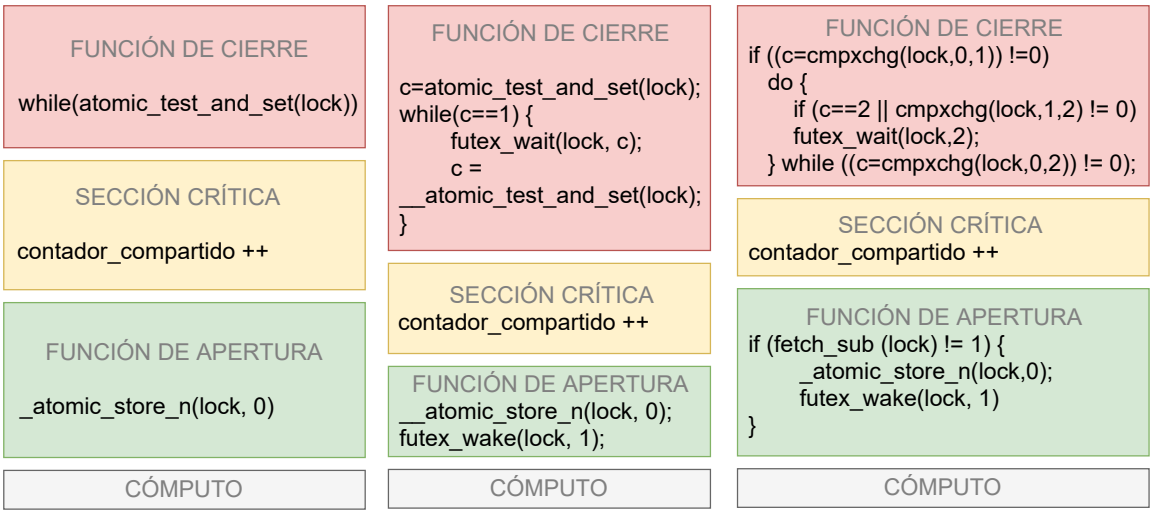
al método `adormir()` en la función `wait()` se encuentra entre las llamadas a `mtx.unlock()` y `mtx.lock()`, de manera que se permita la entrada de un nuevo hilo al método. Esta construcción otorga al método de la atomicidad que necesita.

A partir del código anterior, en el presente nivel de abstracción se propone explotar los servicios del SO para implementar tres versiones distintas del mutex `mtx` integrado en el semáforo. Es decir, cada versión implementará los métodos del mutex `mtx.lock()` y `mtx.unlock()` de manera distinta. La Figura 3.2 muestra las tres versiones de mutex de SO, distinguiendo entre distintas secciones de código. Las secciones coloreadas en rojo y verde representan a las funciones de cierre y apertura de los mutexes, respectivamente. Se refiere al lector a la Sección 4.2.2 para una descripción de las regiones de código anotadas como sección crítica y cómputo. Todas las versiones de mutex utilizan primitivas atómicas incluidas en el lenguaje C para evitar condiciones de carrera entre hilos. A continuación se describe el funcionamiento de cada versión:

- *Spin-lock*: esta versión utiliza la variable compartida `lock` para definir el estado del mutex (0=libre y 1=ocupado) y establece una espera activa para garantizar el acceso en exclusión mutua a la sección crítica. La protección de la variable compartida se hace con las instrucciones atómicas `atomic_test_and_set()` y `atomic_store_n()`, las cuales no se pueden ejecutar por más de un hilo a la vez. La instrucción `atomic_test_and_set()` modifica el estado del mutex a ocupado y devuelve el estado anterior, haciendo que los hilos en contención permanezcan iterando en el método *lock*.

  Cuando el hilo abandona la sección crítica, libera el mutex mediante la instrucción `atomic_store_n()`, permitiendo que el primer hilo en ejecutar `atomic_test_and_set()` pueda acceder a la sección crítica. El derroche de recursos debido a la espera activa es proporcional a la contención. Sin embargo, este mutex es fácil de implementar y muy eficiente en escenarios de poca contención.

- *Sleep* básico: con el objetivo de solventar los problemas de la versión de mutex anterior, la presente implementación de mutex suspende (duerme) a aquellos hilos en contención mediante la llamada al sistema `futex_wait()`. Esta llamada sólo tiene éxito si el estado del mutex es igual al valor esperado. En la función de cierre, si la sección crítica se encuentra ocupada, los hilos que ejecutan `futex_wait()` quedan suspendidos en una cola del sistema. Por el contrario, en caso de que la sección crítica se haya liberado justo antes de ejecutar `futex_wait()`, esta



(a) *Spin-lock* (b) *Sleep* básico (c) *Sleep* avanzado

Figura 3.2: Código de cada implementación de mutex de SO, diferenciando entre funciones de cierre (sección de código en rojo) y apertura (en verde). La sección crítica (en amarillo) y el cómputo (en gris) se describirán en la Sección 4.2.2.

devuelve un error y el hilo intenta acceder a la sección crítica de manera análoga al método *spin-lock*.

En la función de apertura, el hilo que libera la sección crítica cambia el estado del mutex a libre y despierta a un solo hilo. Nótese que, al igual que la función de cierre, la modificación del estado del mutex se realiza con las mismas funciones atómicas utilizadas en la versión *Spin-Lock*. En comparación a *Spin-Lock*, la función de cierre es más eficiente debido a que duerme a los hilos en contención. Sin embargo, la función de apertura realiza la llamada `futex_wake()` sin comprobar si existen hilos dormidos que despertar, resultando en un cambio de contexto innecesario si no hay ningún hilo en dicha situación.

– *Sleep* avanzado: esta versión introduce un tercer estado del mutex para evitar la problemática de *sleep* básico. Así pues, aparte del estado 0 (libre) y 1 (ocupado), el estado 2 corresponde a ocupado y al menos un hilo esperando a acceder a la sección crítica. Las funciones atómicas anteriores dejan de ser útiles con tres estados, por lo que utiliza la función también atómica `cmpxchg(lock, X, Y)` donde `X` es el valor esperado e `Y` el deseado.

En la sección de cierre del mutex, el hilo que ejecuta esta instrucción en el primer condicional recibe en `c` el valor actual del mutex, y actualiza el estado del mutex con el valor deseado sólo si éste tiene el valor esperado. El resultado de esta primera llamada a `cmpxchg()` permite el acceso a la sección crítica sólo si el mutex está libre. De lo contrario, el flujo de ejecución sigue con el bucle `do-while`,

donde, si el estado es 2 (`c==2`) el hilo se duerme con la invocación a `futex_wait()`, y si es 1 (segundo `cmpxchg()`) cambia su estado a 2 y lo duerme igualmente. La condición necesaria para dormirse es que el estado sea 2, por lo que el hilo no se dormirá si el mutex queda liberado entre la primera y segunda llamada a `cmpxchg()` (`futex_wait()` devolverá error), pasando el flujo al `while` para que un tercer `cmpxchg()` compruebe si sigue libre y le permita el acceso la sección crítica, estableciendo `lock` a 2 ya que no se sabe con certeza el número de hilos en espera, o por el contrario, si el mutex está ocupado de nuevo, volver a intentar dormir al hilo con una nueva iteración de `do-while`.

Por su parte, la función de apertura comprueba si existe algún hilo esperando antes de hacer la llamada a `futex_wake()`. Esto lo hace mediante la función atómica `fetch_sub()` que devuelve el estado anterior del mutex y le resta uno.

El mutex *Sleep* avanzado se basa en la implementación de mutex realizada por Ulrich Drepper [9], la cual está integrada en el *kernel* de Linux[10].

3.5. Implementación de Mutexes con Lenguaje Ensamblador

El nivel de ALMA presenta diferencias importantes con respecto al nivel de SO. Hasta ahora, la facilidad de trabajar desde un nivel de abstracción relativamente alto tenía el inconveniente de involucrar al SO y generar una sobrecarga por llamadas al sistema. El compilador, encargado de traducir el modelo de memoria de C al modelo de memora de la ISA de la arquitectura utilizada, convierte las llamadas al sistema futex en un número variable de instrucciones de la ISA (ARMv8 en el caso de Raspberry Pi). En el presente nivel, se crean los mutex mediante instrucciones de la ISA, evitando involucrar al SO y obteniendo diferencias notables en consumo energético y rendimiento.

Para implementar una sección crítica en ensamblador se necesita atomicidad a fin de evitar condiciones de carrera y garantizar la consistencia. ARM no dispone de este bloqueo atómico, por lo que se usan las instrucciones *load-link/store conditional* para crearlo [11]. A continuación se detallan un par de versiones de mutex en ensamblador: una versión mutex con espera activa y otra con *sleep*. Ambas implementaciones reducen el número de estados del mutex a tan sólo dos, a diferencia de la versión *sleep* avanzado de SO. Estas son:

– La versión de mutex con espera activa, la cual se puede verse en el Código 3.3. El funcionamiento de este mutex se basa en el mecanismo *load-link/store-conditional*

Código 3.3: Mutex en ensamblador con espera activa.

```
    .global mi_mutex_lock

mi_mutex_lock:
    .cfi_startproc
    mov w3, 1
mi_mutex_lock_loop:
    ldaxr w2, [x0]
    cbnz w2, mi_mutex_lock_loop
    mov w4, #1
    stlxr w3, w4, [x0]
    cbnz w3, mi_mutex_lock_loop
    ;dmb sy
    dsb sy
    ret
    .cfi_endproc

    .global mi_mutex_unlock
    .align 2
    .type mi_mutex_unlock, %function
mi_mutex_unlock:
    .cfi_startproc
    ;dmb sy
    mov w1, #0
mi_mutex_unlock_loop:
    str w1, [x0]
    dsb sy
    ret
    .cfi_endproc
```

de ARM. Este mecanismo de sincronización de hilos esta basado en las instrucciones `ldaxr` y `stlxr`. La primera instrucción carga (*load*) en un registro (`w2`) la dirección de memoria que indica el estado del mutex (`x0`). Por su parte, la segunda instrucción actualiza (*store*) el valor en esta dirección sólo si ningún otro hilo ha ejecutado `ldaxr` de manera entrelazada. Este esquema permite leer-modificar-escribir de forma atómica, ya que cualquier otro hilo que entrelace la ejecución de `ldaxr` entre la ejecución de `ldaxr-stlxr` del primer hilo abortará el acceso a la sección crítica.

Debido a que la arquitectura de ARM permite ejecución de instrucciones fuera de orden para mejorar la eficiencia de los recursos disponibles, el hilo podría ejecutar instrucciones de la sección crítica antes de finalizar la función de cierre del mutex. Para solucionarlo, se usa la instrucción `dmb sy`. Se trata de un tipo de barrera de memoria que impone orden sobre las instrucciones de memoria antes y después de la barrera. En este caso, se requiere la realización de todas las

Código 3.4: Mutex en ensamblador con *sleep*.

```
    .global mi_mutex_lock

mi_mutex_lock:
    .cfi_startproc
    sevl
mi_mutex_lock_loop:
    wfe
    ldaxr w1, [x0]
    cbnz w1, mi_mutex_lock_loop
    mov w1, #1
    stlxr w2, w1, [x0]
    cbnz w2, mi_mutex_lock_loop

    dsb sy
    ret
    .cfi_endproc

    .global mi_mutex_unlock
    .align 2
    .type mi_mutex_unlock, %function
mi_mutex_unlock:
    .cfi_startproc
    mov w1, #0
    str w1, [x0]
    dsb sy
    sevl
    ret
    .cfi_endproc
```

instrucciones anteriores a la barrera antes de continuar con las siguientes. Esta barrera garantiza la ejecución correcta del programa aunque ejerce cierto impacto sobre el rendimiento.

– El Código 3.4 muestra la versión de mutex avanzada con *sleep*. Este mutex hace uso de dos nuevas instrucciones: `wfe` y `sevl`. La instrucción `wfe` (*wait for event*) establece al núcleo que ejecuta la instrucción en un estado de bajo consumo. La instrucción `sevl` despierta a todos los núcleos dormidos. Con respecto a la anterior implementación, esta versión evita las desventajas de la espera activa al establecer en suspensión aquellos núcleos que están corriendo un hilo que no puede acceder a la sección crítica, ahorrando consumo energético.

Por otro lado, con respecto a la implementación del mutex con llamadas al sistema futex, se debe tener en cuenta que las versiones de ensamblador prescinden del SO para gestionar el acceso a la sección crítica y por lo tanto se evitan todos los costes asociados a los cambios de contexto.

3.6. Resumen

Este capítulo ha introducido la cola de tareas concurrente, un mecanismo que reparte el trabajo de una aplicación paralelizable entre los hilos para que se procesen de forma paralela en los diferentes núcleos del procesador. El acceso concurrente a este recurso se protege mediante semáforos, los cuales se construyen a partir de mutexes de nivel de Biblioteca, SO y ALMA. La Tabla 3.1 resume las ventajas e inconvenientes de los distintos mutexes en cada nivel de abstracción en base al análisis teórico de los códigos vistos en este capítulo.

Tabla 3.1: Resumen comparativo de los mutexes en cada uno de los niveles de abstracción.

Tipo de Mutex	**Ventajas**	**Inconvenientes**
Biblioteca	Programabilidad (versión más fácil de implementar)	Desconocimiento de su funcionamiento interno. Pérdida de control que puede hacer perder oportunidad de rendimiento
Spin-Lock	Fácil de implementar y acceso inmediato a la sección crítica cuando hay poca contención	Malgasta recursos en escenarios con mucha contención
Sleep Básico	Rápido y eficiente con mucha contención debido a que duerme a los hilos que compiten por acceder a la sección crítica	Mayor complejidad de implementación que *Spin-Lock*. La función de apertura hace llamadas al sistema *futex wake* sin comprobar si existen hilos dormidos por despertar, resultando en un derroche de recursos
Sleep Avanzado	Es más rápido y eficiente que *Sleep* Básico ya que tiene un tercer estado que evita las llamadas al sistema *futex wake* cuando estas son innecesarias	Mayor complejidad de implementación respecto a *Sleep* Básico
Spin-Lock de ensamblador	Es más rápido que *Spin-Lock* en dar acceso a la sección crítica cuando hay poca contención ya que está implementado de una forma más eficiente en un nivel inferior	Más difícil de implementar que los mutexes de nivel superior. Requiere conocimientos del repertorio de instrucciones del procesador
Spin-Lock de ensamblador con estado de bajo consumo	Es más rápido que *Spin-Lock* con poca contención y más eficiente enérgicamente al establecer a los núcleos en un estado de bajo consumo cuando los hilos compiten por el acceso a la sección crítica	Más difícil de implementar que *Spin-Lock* de ensamblador. Requiere conocimientos adicionales del repertorio de instrucciones del procesador

Capítulo 4

Entorno Experimental

El presente capítulo comienza con la introducción de la plataforma hardware, exponiendo sus características principales. Posteriormente, se detalla el entorno de desarrollo sobre la placa, incluyendo el sistema operativo, compilador, cargas de trabajo, scripts generados para lanzar experimentos, obtención de tiempo de ejecución y temperatura del chip, entre otros.

4.1. Hardware

La plataforma hardware utilizada es Raspberry Pi 3 Modelo B[1]. La Figura 4.1 muestra una fotografía de la placa. Raspberry Pi es una placa *single-board* escogida por algunos profesores del Departamento de Informática e Ingeniería de Sistemas para la realización del proyecto docente en base a su capacidad multitarea y reducido coste

[1] `https://www.raspberrypi.com/products/raspberry-pi-3-model-b/`

Figura 4.1: Raspberry Pi 3 Modelo B.

Tabla 4.1: Especificaciones técnicas de Raspberry Pi 3B.

Característica técnica	***Especificación***	***Característica técnica***	***Especificación***
Modelo	RPi 3 Model B	Salida de video	HDMI 1.3
Generación	3B	USB	4x USB 2.0
Año	2016	Arranque	USB/Red
SoC	BCM2837	Ethernet	100 Mbps
Frecuencia de reloj	1.2 GHz	Wi-Fi	b/g/n Single Band 2.4GHz
Núcleos/hilos	4/4	Bluetooth	4.1 BLE
Arquitectura	ARMv8 64 bit	Alimentación	Micro-USB
GFLOPS	3.62	Precio	$35
RAM	1 GB	GPIO	40 pines
Sonido	Salida Estéreo de 4 pines y vídeo compuesto		

para realizar experimentos por parte de los estudiantes [5]. Su elección entre otras placas con un coste similar, como por ejemplo Clockwork Pi, Rock64 o Pine A64-LTS se debe a su uso extendido y a la gran disponibilidad de software y material en abierto. Estas características la hacen apropiada en el caso de desear extender su uso a más cursos. La Tabla 4.1 muestra sus características principales.

El procesador BCM2837 está basado en la arquitectura ARMv8. Cuenta con 4 núcleos de ejecución sin *hyperthreading*, con lo cual, el procesador puede ejecutar hasta 4 contextos en paralelo.

4.2. Software

La presente sección se organiza en tres partes diferenciadas. En primer lugar, se enumeran las herramientas o aplicaciones software requeridas para llevar a cabo el proyecto. A continuación, se detallan las cargas de trabajo utilizadas para evaluar las diferentes implementaciones de mutex y obtener los resultados experimentales. Finalmente, se detallan los comandos usados para medir tiempo de ejecución e incremento de temperatura en el procesador tras cada experimento.

4.2.1. Herramientas Software

La Tabla 4.2 resume las principales herramientas software utilizadas en el desarrollo del presente trabajo. El sistema operativo utilizado es Ubuntu Server 18.04.3 LTS (*Long-Term Support*) de 64 bits, el cual integra el compilador gcc 7.5.0. Se necesita la versión de 64 bits debido a que las instrucciones necesarias para construir los mutexes

Tabla 4.2: Principales herramientas software utilizadas en el desarrollo del proyecto.

Tipo	***Programa***
Sistema Operativo	Ubuntu Server 18.04.3 LTS de 64 bits
Acceso remoto	Dataplicity
Transferencia de ficheros	WinSCP
Editor de código fuente	Sublime Text
Figuras	draw.io
Gráficas	MATLAB 2020a
Memoria	Overleaf

del nivel de ALMA pertenecen al conjunto de instrucciones A64 de ARMv8. Además, esta versión para servidor incluye una imagen más pequeña sin interfaz gráfica, lo cual es un requisito importante a la hora de trabajar con Raspberry Pi.

Por otro lado, para una mayor facilidad en la gestión y acceso remoto a la placa, se utiliza Dataplicity como portal de acceso remoto en conjunto con WinSCP para transferir ficheros entre Raspberry Pi y una máquina local mediante interfaz gráfica.

4.2.2. Cargas de Trabajo

En esta sección se describen las regiones de código fuente establecidas como sección crítica y cómputo en el capítulo anterior. Estas regiones de código permiten generar un escenario y condiciones propicias para evaluar tanto el rendimiento como la temperatura alcanzada por cada tipo de mutex.

La sección crítica se implementa con dos programas diferenciados. El primer programa es un contador ascendente que permite evaluar la corrección de los mecanismos de sincronización y valorar la homogeneidad en la participación de los hilos a la hora de computar la suma. El segundo programa es una versión simplificada de la cola de tareas concurrente descrita en el Capítulo 3 con planificación estática. En esta versión de uso de la cola, el escritor llena la cola y finaliza su ejecución antes de que los lectores procedan a acceder a la sección crítica para el desencolado de tareas y posterior procesamiento de las mismas. A diferencia del primer programa, el segundo programa permite obtener los resultados técnicos de los mecanismos de sincronización en los niveles de Biblioteca, SO y ALMA, los cuales se discutirán en el siguiente capítulo.

A continuación, se describen las operaciones principales en la sección crítica de ambos programas:

– Contador ascendente: La sección crítica corresponde a la sentencia *if-else* del Código 4.1. En dichas líneas de código se incrementa la variable compartida `contador_compartido`. Cuando su valor alcanza el límite establecido como argumento del programa (`MAXSUMA`), la variable compartida `nofin` (inicializada

Código 4.1: Sección crítica correspondiente al contador ascendente.

```
char nofin = TRUE;
while (nofin) {
    ...
    //Cierre del mutex
    ...

    if(contador_compartido >= MAX_SUMA) {
        nofin = FALSE;
    }else{
        contador_compartido = contador_compartido + 1;
    }

    ...
    //Apertura del mutex
    ...

    ...
    //Computo
    ...
}
```

como `true`) toma el valor `false` para que todos los hilos no vuelvan a entrar al bucle `while` en la siguiente iteración.

– Cola de tareas concurrente: la sección crítica corresponde a la sentencia `if-else` del Código 4.2 comprendida entre el cierre y la apertura del mutex del método `firstR`. La variable compartida `no_vacia`, inicializada a `true`, indica el estado de la cola. El método `f = bq->first()` lee la primera tarea de la cola, mientras que `bq->dequeue()` desencola la tarea. Cuando la cola se ha vaciado, se ejecuta `no_vacia = false` para que los hilos no vuelvan a entrar al bucle `while`.

La sección de cómputo se modeló inicialmente teniendo solamente en cuenta el tiempo que el hilo estaría ocupado en dicha región. Esto se modeló inicialmente mediante la primitiva `std::this_thread::sleep_for`[2] del lenguaje C++, pero se descartó porque dicha llamada no estresa las unidades funcionales de la CPU y por tanto no se observa un aumento de temperatura en el chip. Por otro lado, también se descartó usar un *benchmark*, ya que esto involucra la creación de un nuevo proceso y la invocación del planificador del sistema operativo, resultando en cambios de contexto que aumentan la sobrecarga de ejecución y limitan las diferencias entre los distintos tipos de mutex.

Para modelar el cómputo de la tarea extraída de forma realista, se le asigna un peso

[2] `https://en.cppreference.com/w/cpp/thread/sleep_for`

Código 4.2: Sección crítica correspondiente a la cola concurrente.

```
void extraer(...) {
    ...
    while(no_vacia) {
        ...
        cbq.firstR(no_vacia, ...);
        ...
        //Computo
        ...
    }
}

void ConcurrentBoundedQueue<T>::firstR(bool &no_vacia, ...) {
    ...
    //Cierre del mutex
    ...
    if ((bq->length() == 0)) {
        if (no_vacia == true) {
            no_vacia = false;
        }

    }
    else {
        f = bq->first();
        bq->dequeue();
    }
    ...
    //Apertura del mutex
    ...
}
```

Código 4.3: Código correspondiente a la sección de cómputo.

```
if(max_rep > 0) {
    rand_int = rand();
    for (int i=0; i < rand_int % max_rep; i++) {
        if (i==0) result = trigFunc((double)rand_int);
        else result=trigFunc(result);
    }
    p_v_trig[ind_thread] += result;
}
```

computacional distinto a cada tarea mediante una serie de operaciones trigonométricas de distinta duración. El Código 4.3, correspondiente a la región de cómputo, muestra cómo la función trigonométrica `trigFunc()` se ejecuta un número variable de veces dado por la operación módulo entre un número pseudo-aleatorio y la variable `max_rep`. Esta última variable es un argumento de entrada al programa cuyo valor da lugar a dos situaciones de contención en los mutexes:

Código 4.4: Código de la función trigonométrica.

```
double trigFunc(double entrada) {
   return sin(entrada*786.12);
}
```

- Contención real: tiene lugar cuando `max_rep > 0`. Se computa cierta carga de trabajo variable antes de volver a la función de cierre de acuerdo con el código anterior. Cuanto mayor es la variable `max_rep`, mayor es la carga de trabajo y menos contención experimentan los hilos para acceder a la sección crítica.

- Contención sintética: tiene lugar cuando `max_rep = 0`. Los hilos intentan acceder a la sección crítica tan pronto salen de ella y sin realizar trabajo alguno, por lo que la aglomeración de hilos en la entrada de la sección crítica genera máxima contención.

Nótese también que se usa una semilla fija al inicio del programa para que los números pseudo-aleatorios obtenidos mediante `rand()` sean siempre los mismos, garantizando que todas las ejecuciones se hagan con las mismas condiciones. Sin embargo, su asignación a los hilos es aleatoria según el orden en que éstos salen de la sección crítica.

La función trigonométrica del Código 4.4 está ideada para producir una alta utilización de las unidades funcionales de la CPU mediante operaciones en coma flotante. Se encadenan sus iteraciones, se usan y exportan posteriormente los resultados devueltos por la llamada para evitar que el compilador optimice los cálculos intermedios desechando los resultados no usados.

4.2.3. Herramientas de Medida

En el presente trabajo se hace uso de BASH *scripting* para crear los bancos de pruebas y automatizar su lanzamiento. Los *scripts* `script_auto.sh` (véase Anexo A.2) y `script_banco_pruebas.sh` (véase Anexo A.1) hacen uso de las siguientes funciones para la medición del rendimiento y el incremento de temperatura de cada mecanismo de sincronización mutex:

- Para medir el tiempo de ejecución se toma en consideración cuánto tardan los lectores en desencolar y procesar las tareas de la cola concurrente con planificación estática definida en la Sección 4.2.2. El Código 4.5 muestra la realización de la medida. En ella, la función `gettimeofday()`[3] devuelve el número de segundos y

[3] `https://man7.org/linux/man-pages/man2/settimeofday.2.html`

microsegundos transcurridos desde el *Epoch* (1 de enero de 1970).

Código 4.5: Medición del tiempo de ejecución mediante `gettimeofday()`.

```
...
pIns[0].join(); // finaliza el escritor

gettimeofday(&start, NULL);

for (int ind_thread=0; ind_thread<N_LECTORES; ind_thread++)
    pExt[ind_thread] = thread (&extraer<int>, ref(cbq),
ref(no_vacia), p_vector_comprobacion, max_rep, ind_thread, p_v_trig);

for (int i=0; i<N_LECTORES; i++) pExt[i].join();
// finalizan los lectores
gettimeofday(&end, NULL);
long long seconds = (long long)(end.tv_sec - start.tv_sec);
long long micros = (long long)((seconds * 1000000) + end.tv_usec) -
(start.tv_usec);
double milisF = ((double)micros)/((double)1000);
...
```

El hilo principal del programa la usa para medir el período desde que el escritor hace `join()` (primera llamada en el código) hasta que todos los lectores han hecho `join()` (segunda llamada). Su resta permite obtener el tiempo de ejecución. Inicialmente utilizó la macro *CLOCKS_PER_SEC*[4] de `time.h`, pero se obtuvo un comportamiento anómalo en la versión de Ubuntu considerada.

Es importante destacar que previamente a ejecutar los bancos de pruebas, es necesario fijar la frecuencia del procesador para garantizar la reproducibilidad de los resultados. Por defecto, la frecuencia es variable y se ajusta según diversos factores, entre los que se encuentra la carga de trabajo. Para fijarla es necesario cambiar el modo de operación de la CPU a *performance*, lo cual ajusta la frecuencia de la CPU a su valor máximo de 1.2 GHz. (Anexo A.1).

– La medición de la temperatura se muestra en el Código fuente BASH 4.6. La medida de la termperatura del empaquetado de la CPU se realiza a intervalos regulares antes de ejecutar el programa y una vez al finalizar. La temperatura del chip se mide consultando el fichero `temp` localizado en `/sys/class/thermal/thermal_zone0/`. Se consulta periódicamente la temperatura antes de ejecutar la siguiente iteración del banco de pruebas, dejando enfriar el chip durante el tiempo necesario para que todos los experimentos partan de una misma temperatura inicial. Inicialmente, se trató de utilizar

[4] `https://www.gnu.org/software/libc/manual/html_node/CPU-Time.html`

Código 4.6: Medición de la temperatura del empaquetado de la CPU.

```
  tActual1=$(echo -e "$(echo "scale=1; $(cat /sys/class/thermal/
thermal_zone0/temp)/1000" | bc)\n");
  while (( $(echo "$tActual1 > $tUmbral" |bc -l) ))
    do
        sleep 5s;
        tActual1=$(echo -e "$(echo "scale=1; $(cat /sys/class/thermal/
thermal_zone0/temp)/1000" | bc)\n");
    done

        ...
        //Ejecucion del programa
        ...

    tActual2=$(echo -e "$(echo "scale=1; $(cat /sys/class/thermal/
thermal_zone0/temp)/1000" | bc)\n");
    tempDiff=`echo $tActual2 - $tActual1 | bc`
    #echo "$tActual2 $tActual1";
    echo $tempDiff >> temperatura_$1_$2.txt;
```

`/opt/vc/bin/vcgencmd measure_temp`, el cual obtiene la temperatura con una precisión de milésimas de grado centígrado, pero la orden no funcionaba en esta versión de SO.

Capítulo 5

Resultados Experimentales

En este capítulo se analizan y comparan los resultados experimentales obtenidos al evaluar el rendimiento y la huella térmica de los distintos tipos de mutex atendiendo a los niveles de Biblioteca, SO y ALMA bajo distintas condiciones de contención real y sintética.

5.1. Contador Ascendente: Homogeneidad en la Participación de los Hilos y Resultados Técnicos de los Mutexes de SO

A modo de introducción y verificación de la corrección de los distintos mutexes implementados, esta sección analiza en primer lugar cómo se distribuye la participación de los hilos lanzados en la ejecución de un programa. Como programa de ejemplo se toma el contador ascendente definido en el Capítulo 4.2.2 y cuya sección crítica se detalla en el Código 4.1.

Los gráficos circulares correspondientes a la Figura 5.1 representan cuántas veces participa cada hilo en incrementar la variable compartida `contador_compartido` hasta que esta alcanza `MAX_SUM=100`. Se muestra la participación en el cómputo de cuatro ejecuciones con diferente número de hilos involucrados, asumiendo un escenario de contención real. Nótese cómo en el primer gráfico circular de izquierda a derecha, el hilo 2 acapara alrededor del 75 % de la participación mientras que el hilo 3 aporta menos del 2 %. Este desbalanceo en la participación de cada hilo sigue ocurriendo al aumentar el número de hilos hasta un total de 32, donde se observa cómo sólo participan 3 de los 32 hilos.

La Figura 5.2 muestra los resultados aumentando el tamaño del problema hasta un límite en el contador ascendente de `MAX_SUM=10000`. A diferencia del caso anterior, se observa cómo todos los hilos participan en la suma, existiendo una mayor uniformidad a lo largo de todas las ejecuciones, incluso en el caso con hasta 32 hilos, aunque en este

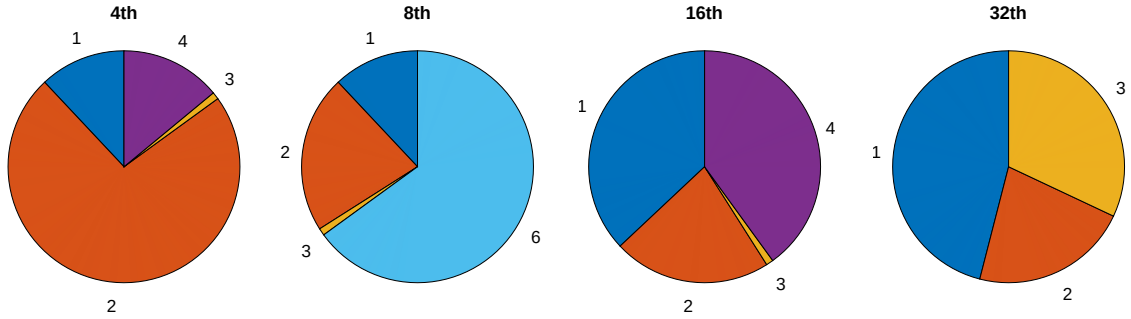


Figura 5.1: Participación de los hilos en la versión de mutex *Sleep* avanzado de SO bajo un escenario de contención real, una suma del contador hasta 100 y variando el número de hilos hasta 32.

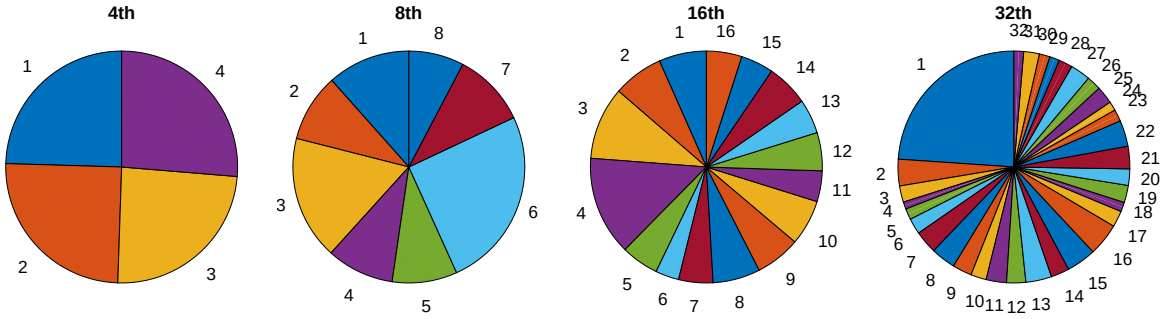


Figura 5.2: Participación de los hilos en la versión de mutex *Sleep* avanzado de SO bajo un escenario de contención real, una suma del contador hasta 10 K y variando el número de hilos hasta 32.

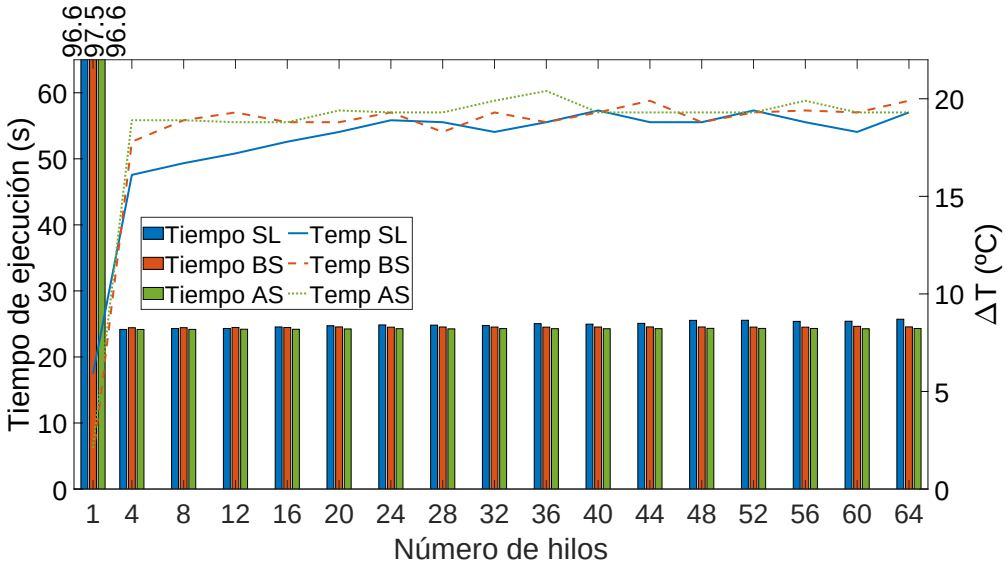
escenario, el hilo 1 acapara casi un cuarto de la participación.

Nótese que aunque los resultados se han limitado a *Sleep* avanzado, las conclusiones son similares respecto a los otros tipos de mutex, ya que la homogeneidad en la participación sólo depende del planificador de procesos del SO y del tamaño del problema. En otras palabras, el desbalanceo observado en la Figura 5.1 se debe a que la duración del experimento es corta en comparación al quantum[1] asignado por parte del planificador a cada tarea, por lo que el experimento finaliza antes de que todos los hilos puedan participar en la resolución del problema. De esta manera, se ha decidido establecer tanto un límite de la suma como un tamaño de tareas de la cola de 1 millón[2] para el resto de experimentos, ya que representa un balance razonable entre homogeneidad en la participación de los hilos y tiempo de ejecución.

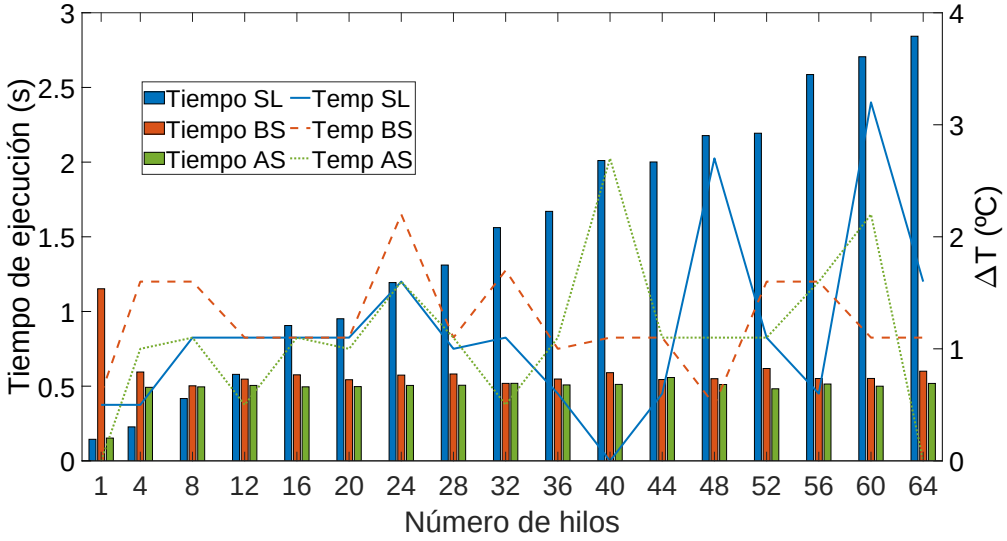
A continuación, se experimenta con el sumador ascendente para obtener los resultados de tiempo de ejecución de temperatura de cada tipo de mutex de SO conforme aumenta el número de hilos involucrados.

[1] `https://en.wikipedia.org/wiki/Round-robin_scheduling`

[2] El tiempo requerido para aumentar en una unidad el contador es muy similar al tiempo necesario para desencolar una tarea en la cola concurrente.



(a) Contención real



(b) Contención sintética

Figura 5.3: Tiempo de ejecución (en segundos) e incremento de temperatura (en grados centígrados) de los mutexes de SO.

Las barras y líneas de la Figura 5.3a muestran el tiempo de ejecución (en segundos) y el incremento de temperatura de la CPU (en grados centígrados), respectivamente, para el experimento con contención real. Este experimento se ha realizado estableciendo el límite `max_rep=1000`, el cual determina la duración de la sección de cómputo del Código 4.3. La nomenclatura usada para los mutexes es: SL = *Spin-lock*, BS = *Sleep* básico y AS = *Sleep* avanzado. A continuación se destaca en primer lugar las conclusiones sobre el tiempo de ejecución, seguidas por el incremento de temperatura, justificando los comportamientos observados:

– El tiempo de ejecución se reduce por cuatro al pasar de 1 a 4 hilos. Esto se debe

a que la sección de cómputo (cuyo peso total es constante[3]), responsable de casi la totalidad del tiempo de ejecución, es completamente paralelizable.

– El rendimiento no mejora a partir de 4 hilos debido al efecto *thread oversubscription* o sobreasignación de hilos, es decir, puesto que es una tarea limitada por cómputo, no hay más paralelismo que aprovechar más allá de 4 hilos puesto que el número de núcleos físicos es 4.

– El mutex que peor rinde de 1 a 12 hilos es SB. Esto se debe a la sobrecarga por la llamada al sistema innecesaria `FUTEX_WAKE` en la función de apertura para despertar a los demás hilos bloqueados cuando no hay ningún hilo esperando (ver Sección 3.4). Por su parte, SL rinde peor de 16 a 64 hilos porque la espera activa es la solución que más penaliza al aumentar la contención.

– Como cabe esperar, SA rinde ligeramente mejor que SB al prescindir de las llamadas `FUTEX_WAKE` cuando no hay hilos a quienes despertar (ver Sección 3.4).

– Respecto a la temperatura, esta es muy baja en el caso de un hilo ya que sólo uno de los cuatro núcleos del procesador está siendo utilizado. La tendencia de la temperatura es ligeramente creciente de 4 a 64 hilos para todos los mutexes, siendo SL la solución con un menor incremento en la mayoría de casos. Esto sugiere que las llamadas al sistema `FUTEX_WAKE` tienen una huella térmica mayor.

El caso con contención sintética de la Figura 5.3b muestra el comportamiento cuando hay sobrecarga por contención. Este escenario elimina la aportación del tiempo de cómputo y maximiza las diferencias de implementación en la sincronización de hilos:

– Las diferencias en los tiempos de ejecución entre distintos mutexes son similares al caso real para un hilo. Por ejemplo, SB rinde aproximadamente 1 segundo peor que los demás mutexes para un sólo hilo tanto en la contención real como sintética. Sin embargo, la diferencia de tiempo relativa de SL frente a las soluciones de SO es mucho mayor, sobretodo al aumentar el número de hilos más allá de 16.

– El mutex que mejor rinde para un sólo hilo y cuando el número de hilos coincide con el de núcleos es SL. Esto se debe a la sobrecarga por la llamada al sistema de las versiones *Sleep*.

[3]La sección de cómputo del Código 4.1 se ejecuta un número de `MAX_SUM=1000000` de veces. Dado que el método `rand()` del Código 4.3 hace uso de una semilla fija y que `max_rep=1000`, la secuencia de repeticiones de la función trigonométrica es siempre la misma en todos los el experimentos. Por tanto, el cómputo total es constante y sólo depende de las constantes `MAX_SUM` y `max_rep`.

– El paso de 1 a 4 hilos mejora el rendimiento para BS. Esto de sebe principalmente a que el cambio de contexto generado por FUTEX_WAKE penaliza menos al escenario de 4 hilos, pudiendo asignar la CPU a uno de estos hilos frente a un proceso distinto como es el caso del escenario con un sólo hilo. Por otro lado, SL y SA empeoran ya que les perjudica el aumento de contención al no haber sección de cómputo que realizar.

– La tendencia de tiempo de cómputo creciente para SL conforme se usan más hilos se debe a que la espera activa penaliza el rendimiento proporcionalmente a la contención. Las versiones *Sleep* se mantienen, viéndose en casi todos los casos que *Sleep Advanced* rinde ligeramente mejor (ver Sección 3.4).

– Se descarta el análisis de la temperatura porque para tiempos de ejecución relativamente cortos el chip no llega a calentarse lo suficiente como para apreciar diferencias significativas entre mutexes distintos.

5.2. Cola de Tareas Concurrente: Comparativa entre Mutexes de Biblioteca y SO

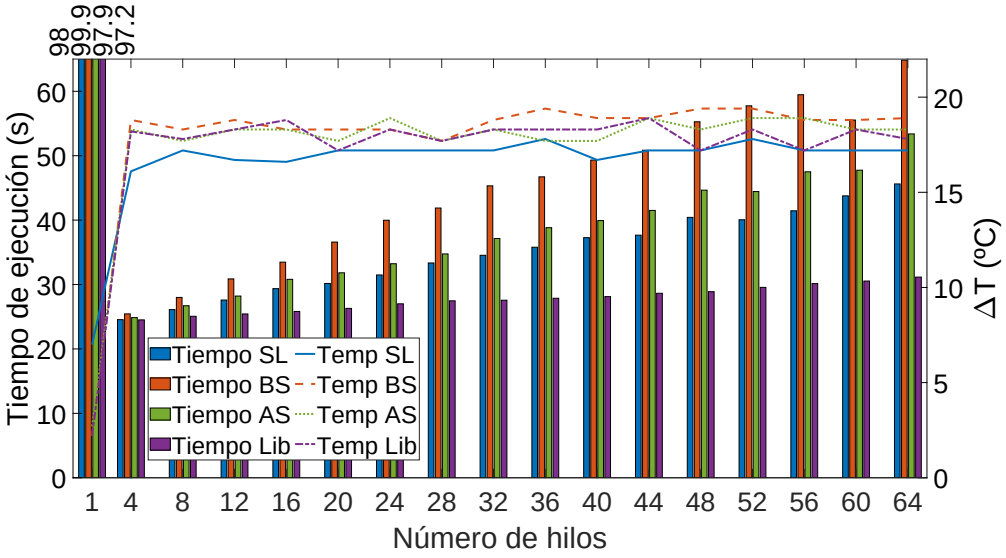
La comparativa entre los mutexes de SO y de la librería estándar C++ de Biblioteca se realiza mediante la cola de tareas concurrente. Como se indicaba en la sección anterior, el tamaño de la cola está definido en un millón de elementos, dando lugar a unos tiempos de ejecución similares respecto al uso del contador ascendente.

La Figura 5.4a muestra los resultados obtenidos bajo una contención real con idéntica configuración a la establecida en la sección anterior (`max_rep=1000`). La nomenclatura *Lib* corresponde a la implementación con mutex de Biblioteca. Se destacan las siguientes observaciones:

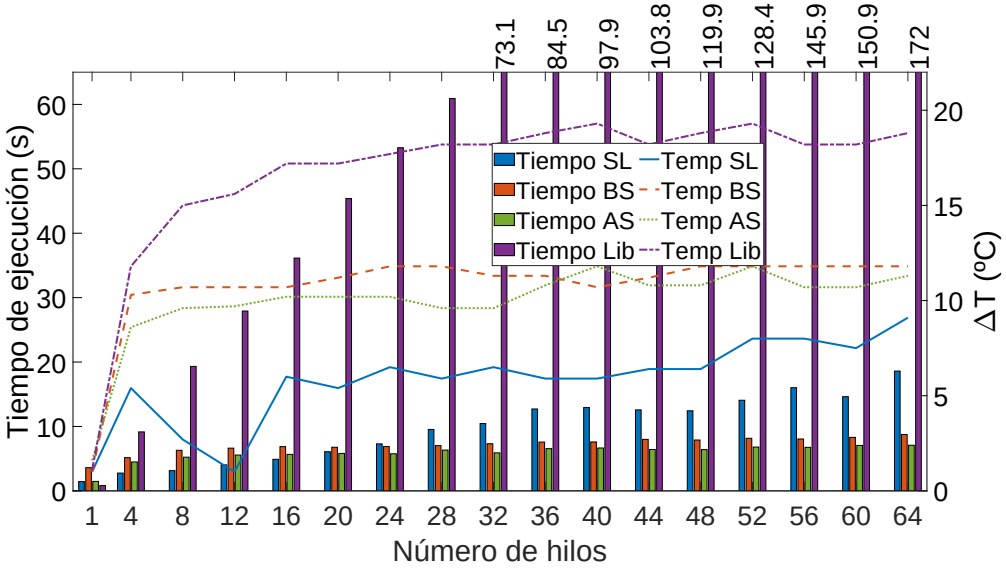
– El mejor rendimiento se obtiene para 4 hilos puesto que se explota completamente el paralelismo con los 4 núcleos disponibles. SL y Lib obtienen los mejores resultados. Esto sugiere que el mutex de Biblioteca está construido con espera activa al menos para este número de hilos[4].

– El mutex de biblioteca es el que mejor rinde a partir de 4 hilos. Esto se puede deber a que la implementación regula el exceso de hilos en caso de *thread oversubscription*, desechando aquellos hilos que resultan innecesarios o

[4]El impacto en el rendimiento de la contención en distintas implementaciones de mecanismos de sincronización es conocida. Por ese motivo las implementaciones de Biblioteca (`std::mutex`) seleccionan la implementación más adecuada al escenario en cuestión.



(a) Contención real



(b) Contención sintética

Figura 5.4: Tiempo de ejecución (en segundos) e incremento de temperatura (en grados centígrados) de los mutexes de SO y Biblioteca.

gestionándolos en una cola propia de biblioteca, y por tanto limitando los efectos de la contención en el rendimiento[5].

– SL rinde mejor que las versiones *Sleep* para 4 o más hilos porque la implementación de la cola a partir de semáforos a nivel de sistema operativo (Código 3.2), a diferencia de la implementación de biblioteca (Código 3.1), suspende a los hilos en los métodos `adormir()` y `despertar()` mediante llamadas al sistema futex, lo cual causa una sobrecarga debido a la involucramiento del

[5]La biblioteca `std::threads` gestiona con independencia del SO los hilos de biblioteca y así evita las penalizaciones por cambio de contexto.

sistema operativo con los cambios de contexto.

- *Sleep* avanzado rinde mejor que *Sleep* básico por la llamada al sistema `futex_wake()` innecesaria que realiza el segundo mutex cuando no existen hilos a los que despertar.

- El incremento de temperatura es muy similar en todas las soluciones, ya que en la contención real se realiza cómputo trigonométrico más allá de acceder a la sección crítica. Sin embargo, el aumento de temperatura más bajo se da en *Spin-Lock*, seguido del mutex de biblioteca, cuya gestión de los hilos calienta más al procesador. Los mutex de SO, como se esperaba, calientan más porque sobretodo, tardan más en ejecutar el programa.

La Figura 5.4b muestra los resultados del escenario sintético. Las conclusiones más relevantes son las que siguen:

- El tiempo de ejecución del mutex de Biblioteca se dispara para 4 o más hilos respecto al resto de soluciones. Esto indica que esta versión de mutex no ha sido diseñada teniendo en cuenta un escenario de contención extrema.

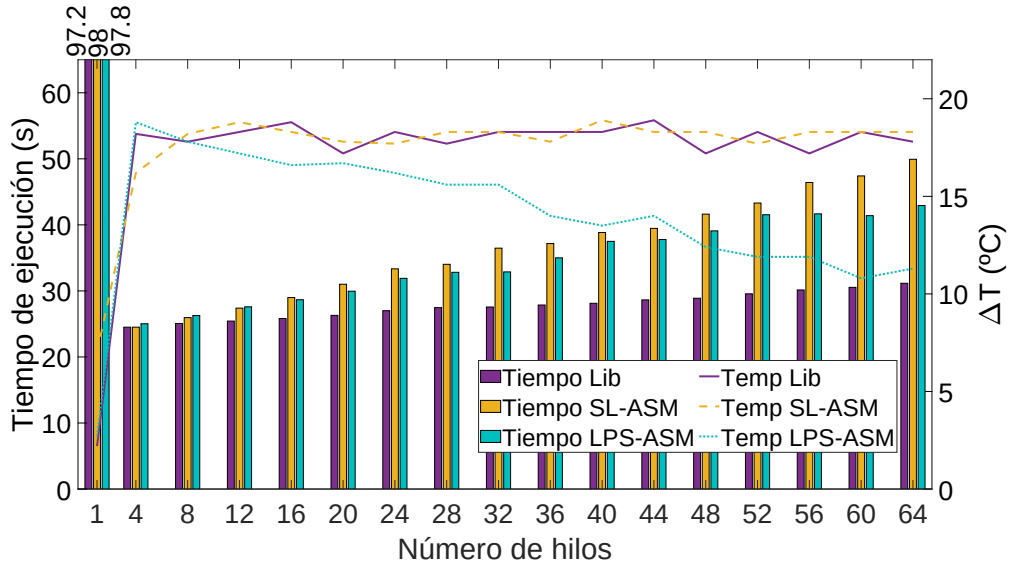
- La espera activa hace que *Spin-Lock* rinda peor que las versiones *Sleep* a partir de 24 hilos, indicando que esta penaliza más que las llamadas al sistema futex de *Basic Sleep*. Por tanto, a partir de este número de hilos merece más la pena escoger una solución basada en *Sleep* que con *Spin-Lock*.

- *Spin-Lock* tiene menor impacto térmico, aun en los casos donde su tiempo de ejecución es mayor respecto a las versiones *Sleep*. La temperatura alcanzada por las versiones de SO es parecida, mientras que en el caso de Lib la temperatura es mucho mayor debido probablemente a su tiempo de ejecución más largo.

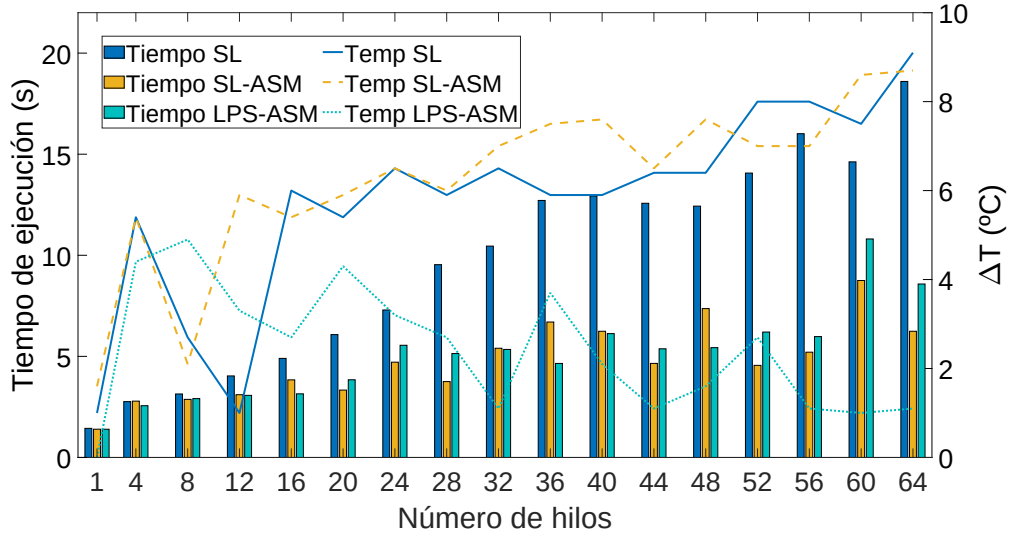
En resumen, el uso de un mutex de Biblioteca es recomendable en un escenario con una alta carga de cómputo (escenario real), posiblemente debido a su construcción híbrida con *Spin-Lock*, regulación de hilos y colas para dormir. Sin embargo, convendría usar cualquier otro mutex de SO para un escenario con menos carga (escenario sintético).

5.3. Cola de Tareas Concurrente: Comparativa entre Mutexes de ALMA y SO

Esta sección continúa utilizando la cola de tareas concurrente con la mismas características que en la sección anterior. En este caso se compara el mutex que mejor



(a) Contención real



(b) Contención sintética

Figura 5.5: Tiempo de ejecución (en segundos) e incremento de temperatura (en grados centígrados) de los mutexes de ALMA frente a los mejores mutexes vistos hasta el momento bajo contención real (Lib) y sintética (SL).

rinde en la sección anterior con un número de hilos igual al número de núcleos físicos (Biblioteca y *Spin-Lock* para contención real y sintética, respectivamente) frente a las versiones de mutex de ensamblador explicadas en la Sección 3.5. Recuérdese que se propone evaluar dos mutexes diferentes en el nivel de ALMA: un mutex con espera activa, referido simplemente como SL-ASM (*Spin-Lock Assembly*) y otro mutex que explota la capacidad de establecer a un núcleo en un modo de operación de bajo consumo con la invocación de la instrucción `wfe` de ARMv8, referido como LPS-ASM (*Low Power State Assembly*)

La Figura 5.5a se corresponde con el escenario de contención real. Se destacan las

siguientes observaciones:

- La versión SL-ASM es la que peor rinde y además incrementa su tiempo de ejecución con el efecto de *thread oversubscription*. LPS-ASM mejora ligeramente respecto a la versión anterior debido a que reduce levemente la contención al establecer a los núcleos en bajo consumo.

- El mutex de Biblioteca sigue siendo la versión que mejor rinde en todos los casos a pesar de que su huella térmica es tan alta como la versión SL-ASM.

- El incremento de temperatura decrece en el caso LPS-ASM conforme aumentan los hilos porque es más probable dejar en modo de operación de bajo consumo a los núcleos al haber más hilos involucrados. Nótese que la temperatura se mantiene constante en el mutex de Biblioteca con independencia del número de hilos, indicando que esta solución no cambia el estado de los núcleos.

La Figura 5.5b muestra los resultados correspondientes al escenario sintético, sustituyendo al mutex Lib por SL. Las conclusiones más relevantes son las siguientes:

- Ambas versiones basadas en espera activa aumentan el tiempo de ejecución con el efecto de *thread oversubscription*. La versión de más alto nivel es la que peor rinde en todos los casos y además aumenta su diferencia respecto a SL-ASM conforme aumenta el número de hilos.

- La huella térmica de SL-ASM es la más baja en la mayoría de casos y la única con una tendencia decreciente (por el mismo razonamiento que en el escenario real).

5.4. Resumen

En resumen, el análisis de los resultados desvela que: i) *i)* la homogeneidad en la participación de los hilos es proporcional al tamaño de problema a resolver, *ii)* el efecto de *thread oversubscription* (mayor número de hilos frente a núcleos físicos de la máquina) penaliza el rendimiento en todas las versiones de mutex, aunque con diferencias notables dependiendo de la versión de mutex y cantidad de contención, *iii)* se recomienda el uso del mutex de biblioteca en un escenario con contención real por su facilidad de implementación y rendimiento superior, *iv)* en un escenario sintético (o de baja carga de cómputo) es más recomendable utilizar un mutex de SO o basado en ensamblador (a costa de un mayor conocimiento y tiempo de implementación) y *v)* las instrucciones de bajo consumo de LPS-ASM resultan determinantes para reducir el consumo energético (temperatura) en escenarios de alta contención.

Capítulo 6

Conclusiones

El presente capítulo resume las conclusiones principales de este trabajo, muestra posibles direcciones futuras y un diagrama de Gantt con las principales tareas del trabajo y las horas de dedicación.

6.1. Desarrollo y Resultados Obtenidos

El presente trabajo se ha estudiado, verificado, implementado y obtenido resultados experimentales de los mecanismos de sincronización mutex en los niveles de Biblioteca, SO y ALMA de un sistema informático. Para ello, se han diseñado cargas de trabajo, incluyendo secciones críticas y de cómputo, y se han utilizado diversas herramientas de medida para obtener los resultados experimentales.

En concreto, se ha diseñado un programa con un contador ascendente para verificar la corrección de los mecanismos de sincronización y obtener el rendimiento e incremento de temperatura de los mutexes de SO. Seguidamente, se ha programado una cola de tareas con planificación estática para comparar el rendimiento entre explotar semáforos construidos con mutexes de Biblioteca y variables de condición frente a usar semáforos construidos con mutexes de SO mediante llamadas al sistema basadas en futexes, diferenciando entre versiones con espera activa y sin espera activa. Finalmente, en el nivel de ALMA, se comparan las versiones de mutex anteriores con otras implementadas a partir de instrucciones en ensamblador que permiten cambiar el modo de operación de los núcleos de la máquina.

A partir de la experimentación se han obtenido las siguientes conclusiones principales: *i)* el uso de un mutex con espera activa es conveniente en escenarios de poca contención, mientras que las versiones sin espera activa son apropiadas para aquellos escenarios con mucha contención, *ii)* el mutex de Biblioteca es el más adecuado bajo contención real debido a su facilidad de implementación y rendimiento superior, *iii)* la versión de mutex en ensamblador que establece un modo de bajo consumo en los

núcleos consigue reducir el incremento de temperatura respecto a las soluciones, con un impacto moderado en el rendimiento.

Por último, este trabajo ha permitido al autor tanto afianzar el trabajo con niveles de abstracción ya conocidos como aprender a trabajar con nuevos niveles de abstracción, resultando en una experiencia de aprendizaje enriquecedora. Mi experiencia personal ha sido muy positiva, al igual que lo han demostrado las encuestas realizadas por estudiantes del Grado en Informática que han podido disfrutar de este proyecto[1].

Finalmente, también cabe destacar que los resultados técnicos del presente trabajo se han publicado en un artículo de la revista *Elsevier Journal of Parallel and Distributed Computing* (JPDC) [5].

6.2. Trabajo Futuro

A partir del trabajo realizado en el presente proyecto, surgen diversas líneas de trabajo futuro:

- Afianzar el uso de una cola concurrente con planificación dinámica. En este escenario, la gestión de la cola contempla el llenado y vaciado simultáneo de las tareas de manera concurrente.
- Utilizar aplicaciones reales con distinta carga computacional a la hora de evaluar los mutexes.
- Estudiar la posibilidad de realizar un proyecto similar en el Grado en ITyS de Telecomunicación que englobe las asignaturas relativas a un sistema informático.
- Estudiar la posibilidad de crear un proyecto de comunicaciones con un enfoque práctico que se realice a lo largo de distintas asignaturas del Grado en ITyS de Telecomunicación, involucrando a múltiples disciplinas de la titulación.

6.3. Dedicación

El proyecto se ha realizado en las fases representadas en la Figura 6.1. La dedicación total aproximada del mismo ha sido de 450 horas. Sin embargo, la carga temporal de la experimentación ha sido grande y la memoria sólo muestra los experimentos más conclusivos, por lo que se ha incluido parte de esta en las tareas *Cola concurrente*

[1] `https://innovaciondocente.unizar.es/convocatoria2018/ventanas/ver_ficha_proyecto.php?proyecto=246`

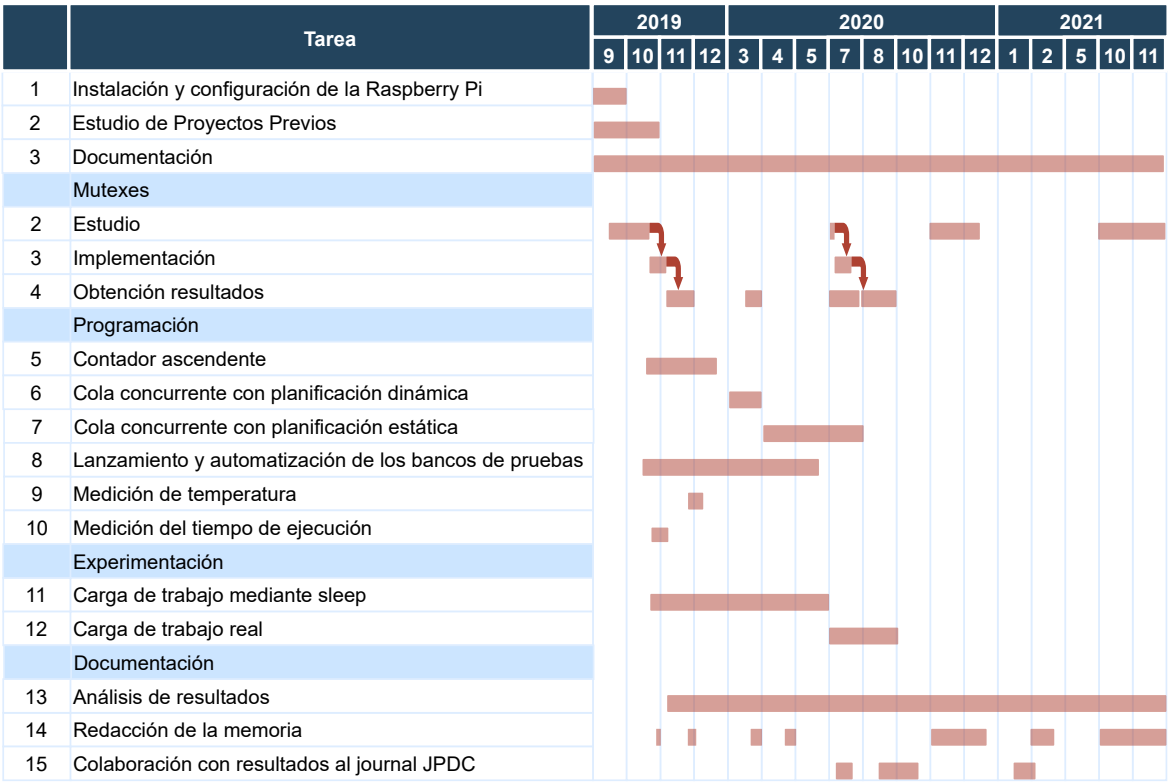


Figura 6.1: Diagrama de Gantt con el desarrollo del trabajo. Los números debajo de los años se refieren a la numeración mensual.

con planificación dinámica y *Carga de trabajo mediante sleep*, las cuales sirvieron para aprender sobre los algoritmos de los mutexes y la construcción de la cola concurrente. También se muestra la parte de colaboración en la obtención de resultados para el artículo publicado en la revista. Debido a períodos de inactividad, el diagrama sólo muestra los meses donde se avanzó en el trabajo.

Capítulo 7

Bibliografía

[1] Jeff Kramer. Is Abstraction the Key to Computing? Communications of the ACM, 50(4):36–42, 2007.

[2] V. Kumar. Introduction to Parallel Computing. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.

[3] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. Morgan & Claypool Publishers, 1st edition, 2011.

[4] ACM/IEEE. Computer Engineering Curricula 2016: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering, 2016.

[5] Alejandro Valero, Rubén Gran-Tejero, Darío Suárez-Gracia, Emanuel A. Georgescu, Joaquín Ezpeleta, Pedro Álvarez, Adolfo Muñoz, Luis M. Ramos, and Pablo Ibáñez. A learning experience toward the understanding of abstraction-level interactions in parallel applications. Elsevier Journal of Parallel and Distributed Systems, 156:38–52, 2021.

[6] D. Ginat and Y. Blau. Multiple Levels of Abstraction in Algorithmic Problem Solving. In Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education, pages 237–242, 2017.

[7] C. Ferner, B. Wilkinson, and B. Heath. Toward Using Higher-Level Abstractions to Teach Parallel Computing. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops, pages 1291–1296, 2013.

[8] B. Levandowski, D. Perouli, and D. Brylow. Using Embedded Xinu and the Raspberry Pi 3 to Teach Parallel Computing in Assembly Programming. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops, pages 334–341, 2019.

[9] U. Drepper. Futexes Are Tricky, 2011. http://people.redhat.com/drepper/futex.pdf.

[10] U. Drepper and I. Molnar. The Native POSIX Thread Library for Linux, 2005. https://akkadia.org/drepper/nptl-design.pdf.

[11] ARM. ARM DS-5 Development Studio Examples, 2018.

Anexos

Anexos A

Entorno Experimental

Este anexo explica los códigos fuente y comandos usados para obtener los resultados experimentales, al igual que el formato de los ficheros de resultados y su representación.

A.1. Comando para establecer el Governor a Performance

Para establecer el modo de operación del procesador a alto rendimiento, fijando la frecuencia máxima a 1.2GHz, es necesario cambiar el Governor de `Ondemand` a `Performance`. Para ello, se lanza `sudo systemctl disable ondemand`. El comando `cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor` muestra el resultado `Performance`, lo que indica que los cambios se han realizado con éxito. Se ha comprobado que este cambio es permanente y persiste entre reinicios.

A.2. Comando para No Interrumpir la Ejecución del Banco de Pruebas en Caso de Desconexión con el Terminal Remoto

En caso de pérdida de la conexión remota, los bancos de pruebas se interrumpirían, perdiendo el progreso de horas de ejecución y necesitando reinciarlas. Para ello se usan las instrucciones `tmux detatch` / `attatch`. Estos comandos mantienen ejecuciones persistentes en servidores, permitiendo desconectarse de la sesión sin interrumpir la ejecución para conectarse posteriormente a ella.

A.3. Scripts que Gestionan el Banco de Pruebas

Esta sección explica cómo se ha generado el banco de pruebas usado para obtener los resultados técnicos del Capítulo 6. Para ello, se muestra y explica el ejemplo de los dos

scripts BASH usados sobre la cola de tareas simplificada para obtener el rendimiento y consumo de los mutexes de Biblioteca y SO:

Código A.1: *Script* BASH encargado de la ejecución y escritura en ficheros de resultados.

```
#USO: ./script3_con_temp.sh TAM_COLA max_rep

#./main_consuspension TAM_COLA N_LECTORES mutex_type(s,n,K) max_rep
# $0 nombre programa, $1 1er argumento...
# $# numero de argumentos despues del comando

tInicial=$(echo -e "$(echo "scale=1; $(cat /sys/class/thermal/
thermal_zone0/temp)/1000" | bc)\n")
tUmbral=`echo $tInicial + 10 | bc`

if (($# != 2)); then
   echo ["USO: ./script3_con_temp.sh TAM_COLA max_rep"];
else
   for i in {1..4} # 4 tipos de mutex
   do
      for j in 1 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 #68 72
#76 80 84 88 92 96 100 104 108 112 116 120 124 128 132 136 140 144 148
#152 156 160 164 168 172 176 180 184 188 192 196 200 #number of threads
      do
         tActual1=$(echo -e "$(echo "scale=1; $(cat /sys/class/
thermal/thermal_zone0/temp)/1000" | bc)\n");
         while (( $(echo "$tActual1 > $tUmbral" |bc -l) ))
         do
            sleep 5s;
            tActual1=$(echo -e "$(echo "scale=1; $(cat /sys/class/
thermal/thermal_zone0/temp)/1000" | bc)\n");
         done

         if ((i==1)); then
            ./main_consuspension $1 $j s $2 >> salida_$1_$2.txt;
         elif ((i==2)); then
            ./main_consuspension $1 $j n $2 >> salida_$1_$2.txt;
         elif ((i==3)); then
            ./main_consuspension $1 $j K $2 >> salida_$1_$2.txt;
         else ./main_master $1 $j s $2 >> salida_$1_$2.txt;
            #internamente ignora el tipo de mutex
         fi

         tActual2=$(echo -e "$(echo "scale=1; $(cat /sys/class/
thermal/thermal_zone0/temp)/1000" | bc)\n");
         tempDiff=`echo $tActual2 - $tActual1 | bc`
         echo $tempDiff >> temperatura_$1_$2.txt;

      done
   done
fi
```

– El primer *script* se puede ver en el Código A.1. Este código realiza un barrido del número de hilos para cada tipo de mutex. Además, también realiza la gestión térmica asegurándose que todas las ejecuciones parten de una misma temperatura, garantizando la reproducibilidad de los resultados y evitando que el calentamiento de la placa penalice el rendimiento debido al *thermal throttling*. Para ello, el *script* sigue los siguientes pasos:

  1. Toma una medida de temperatura en reposo y establece un umbral de 10°C por encima de ella.
  2. Antes de cualquier iteración, mide la temperatura de la CPU cada 5 segundos y no lanza ningún experimento hasta que la temperatura se encuentre por debajo del umbral.
  3. Ejecuta el programa, redireccionando la salida estándar al fichero de texto de resultados `salida_$1_$2.txt`.
  4. Hace una nueva medición de temperatura y guarda en el fichero `temperatura_$1_$2.txt` el incremento de temperatura registrado respecto al paso 2.

  Nótese que el *script* inicialmente realizaba un barrido hasta 200 hilos, pero se decidió limitar a 64 para los resultados finales. Los argumentos $1 y $2 corresponden a `TAM_COLA` y `max_rep` y vienen dados por el siguiente *script*.

– El *script* del Código A.2 recibe como argumento el tamaño de la cola y se encarga de lanzar el *script* anterior haciendo un barrido de 0 a 4000 de la variable `max_rep`,

Código A.2: *Script* BASH encargado de generar el escenario sintético y real.

```
# script automatizado experimentos
# $0 nombre programa, $1 1er arg...
# $# numero de argumentos despues del comando

if (($# != 1)); then
        echo ["USO: ./script_auto TAM_COLA"];
else
        for j in 0 5 25 50 250 500 1000 4000
        do
                echo ["./script3_con_temp"] $1 $j;
                ./script3_con_temp.sh $1 $j;
                echo "sleep 10minutos";
                sleep 600s;
        done
fi
```

obteniendo los resultados con la contención sintética y para distintas cargas de trabajo con la contención real. Después de cada ejecución, este se mantiene inactivo durante 10 minutos para dejar enfriar la placa. Nótese que los resultados técnicos del Capítulo 5 finalmente usaron `max_sum=1000` para la contención real.

A.4. Formato de los Ficheros salida_$1_$2.txt y temperatura_$1_$2.txt

Esta sección muestra un ejemplo de los ficheros de resultados `salida_$1_$2.txt` y temperatura_$1_$2.txt para ver su formato. Estos se pueden ver en el código A.3 y A.4, respectívamente.

Código (A.3) Formato de salida_$1_$2.txt. En orden y separados por tabulación: tipo de mutex (s, n, K, m) correspondientes a *Spin-Lock*, *Sleep* Básico, *Sleep* Avanzado y *Biblioteca*, respectivamente; número de hilos (1 a 64); tiempo de ejecución en segundos; argumento de entrada TAM_COLA; argumento de entrada max_rep.

```
s       1       97979.426000    1000000 1000
s       4       24545.177000    1000000 1000
s       8       26101.761000    1000000 1000
s       12      27592.386000    1000000 1000
...
n       1       99936.059000    1000000 1000
...
K       1       97895.506000    1000000 1000
...
m       1       97240.575000    1000000 1000
...
```

Código (A.4) Formato de temperatura_$1_$2.txt. Hay una sola cola columna con el incremento de temperatura en ºC.

```
7.0
16.1
17.2
16.7
...
2.2
...
2.7
...
2.2
```

Los resultados de tiempo de ejecución son generados por el hilo principal dentro del `main` una vez que los lectores han finalizado el desencolado y el procesamiento de todas las tareas. Este usa la función `fprintf` para escribir por salida estándar (`stdout`) los argumentos de entrada y los resultados del experimento actual mediante el Código A.5.

Código A.5: *Script* de escritura en fichero dentro del método `main`.

```
fprintf(stdout, " %c\t %d\t %lf\t %d\n", mutex_type, N_LECTORES, milisF, TAM_COLA,
max_rep);
fprintf(stderr, "TODO PERFECTO \t");
for (int i=0; i<N_LECTORES; i++) {
    fprintf(stderr, " %f\t", v_trig[i]);
}
fprintf(stderr, "\n");
```

También escribe por salida estándar de error (`stderr`) los resultados de la sección de cómputo para ver el progreso del banco de pruebas en el terminal y además evitar que el compilador optimice los cálculos intermedios desechando los resultados no usados.

A.5. Representación Gráfica de los Ficheros de Resultados Mediante MATLAB

Las gráficas de resultados del Capítulo 5 se han realizado a partir de los ficheros `salida_$1_$2.txt` y `temperatura_$1_$2.txt` mediante MATLAB. El Código A.6 muestra cómo este lee y convierte los ficheros a matrices para luego representar el tiempo en gráficos de barras múltiples y superponer la temperatura con líneas de tendencia.

Código A.6: Código MATLAB de representación de los resultados.

```
%%
clc; close all; clear all;

%% TIEMPO Y TEMPERATURA
%clc;
close all;
clear all;

% PARAMETROS DE REPRESENTACION:
%limites de 65 y 22 para parte real. 3 y 4 para la sintetica.
LIM_EJE_Y1 = 65; %120 para no cortar
LIM_EJE_Y2 = 22;

FONTSIZE = 40;
FONTSIZE2 = 40;
NUM_DECIMALES_BARRAS = 1;

azulClaro = [0, 0.4470, 0.7410];
azulOscuro = [0, 0, 1];
naranjaClaro = [0.8500, 0.3250, 0.0980];
verdeOscuro = [0, 0.5, 0];
amarilloClaro = [0.9290, 0.6940, 0.1250];
naranjaChillon = [1, 0, 0];
moradoClaro = [0.4940, 0.1840, 0.5560];
cyanChillon = [0, 0.75, 0.75];
verdeClaro = [0.4660, 0.6740, 0.1880];
fuxia = [0.75, 0, 0.75];
cyan = [0.3010, 0.7450, 0.9330];
verdePistacho = [0.75, 0.75, 0];
magenta = [0.6350, 0.0780, 0.1840];
verdeOsc = [0.25, 0.25, 0.25];

FileList = dir(fullfile('salida*'));
FileName = fullfile(FileList(1).name);
fid = fopen(FileName);

FileList = dir(fullfile('temperatura*'));
FileName = fullfile(FileList(1).name);
fid2 = fopen(FileName);

%Formato fichero salida: s     1    221.114 1000000 1000
C = textscan(fid, '%c %u %f %u %u');
T = textscan(fid2, '%f');

tiempoMs = C{1,3}(:);
temp = T{1,1}(:);
temp(temp<0)=0;
fclose(fid);
fclose(fid2);
```

```
numColumnas = 17;
tFinal = tiempoMs';
tempFinal = temp';

tFinalS = tFinal/1000;
tS = tFinalS(1:numColumnas)';
tN = tFinalS(numColumnas+1:2*numColumnas)';
tK = tFinalS(2*numColumnas+1:3*numColumnas)';

tempS = tempFinal(1:numColumnas)';
tempN = tempFinal(numColumnas+1:2*numColumnas)';
tempK = tempFinal(2*numColumnas+1:3*numColumnas)';
tempSNKM = [tempS tempN tempK ];

grupos = [1 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64]';
mat = (horzcat(tS, tN, tK));
fig = figure;
orient(fig,'landscape');
fig.WindowState = 'maximized';
b = bar(grupos, mat);
b(1).FaceColor = azulClaro;
b(2).FaceColor = naranjaClaro;
b(3).FaceColor = verdeClaro;

xtips1 = b(1).XEndPoints;
ytips1 = b(1).YEndPoints;
xtips2 = b(2).XEndPoints;
ytips2 = b(2).YEndPoints;
xtips3 = b(3).XEndPoints;
ytips3 = b(3).YEndPoints;
labels1 = string(round(b(1).YData,NUM_DECIMALES_BARRAS));
labels2 = string(round(b(2).YData,NUM_DECIMALES_BARRAS));
labels3 = string(round(b(3).YData,NUM_DECIMALES_BARRAS));

altura_sobre_barras = 1.005*LIM_EJE_Y1;
v_pos_sobre_lim_y1 = find(ytips1 > LIM_EJE_Y1);
if (~isempty(v_pos_sobre_lim_y1))
    text(xtips1(v_pos_sobre_lim_y1),repelem(altura_sobre_barras,length
(v_pos_sobre_lim_y1)),labels1(v_pos_sobre_lim_y1),'HorizontalAlignment',
'left',...'VerticalAlignment','middle','rotation', 90, 'FontSize',
FONTSIZE2)
end

v_pos_sobre_lim_y2 = find(ytips2 > LIM_EJE_Y1);
if (~isempty(v_pos_sobre_lim_y2))
    text(xtips2(v_pos_sobre_lim_y2),repelem(altura_sobre_barras,length
(v_pos_sobre_lim_y2)),labels2(v_pos_sobre_lim_y2),'HorizontalAlignment',
'left',...'VerticalAlignment','middle', 'rotation', 90, 'FontSize',
FONTSIZE2)
end
```

```
v_pos_sobre_lim_y3 = find(ytips3 > LIM_EJE_Y1);
if (~isempty(v_pos_sobre_lim_y3))
    text(xtips3(v_pos_sobre_lim_y3),repelem(altura_sobre_barras,length
(v_pos_sobre_lim_y3)),labels3(v_pos_sobre_lim_y3),'HorizontalAlignment',
'left',...'VerticalAlignment','middle', 'rotation', 90, 'FontSize',
FONTSIZE2)
end

labels = {'1' '4' '8' '12' '16' '20' '24' '28' '32' '36' '40' '44' '48'
'52' '56' '60' '64'};
ticks = horzcat([1], [4:4:64]);
set(gca,'XTick',ticks,'xticklabel',labels, 'FontSize', FONTSIZE);
ylabel('Execution time (s)');
xlabel('Number of threads');

ylim([0 LIM_EJE_Y1]);
yyaxis right;

p=plot(grupos, tempSNKM(:,1), grupos, tempSNKM(:,2), grupos,
tempSNKM(:,3));
p(1).LineWidth = 3; p(1).Color = azulClaro;
p(2).LineWidth = 3; p(2).Color = naranjaClaro;
p(3).LineWidth = 3; p(3).Color = verdeClaro;

yl = ylim;
ylim([0 (yl(2)+2)]);
legend('SL time', 'BS time', 'AS time', 'SL temp', 'BS temp', 'AS temp');
ylabel('\DeltaT ($^{\circ}$C)');
ylim([0 LIM_EJE_Y2]);

ax = gca;
ax.YAxis(1).Color = 'k';
ax.YAxis(2).Color = 'k';
```