UNIVERSIDAD DE ALMERÍA
Departamento de Informática

ALGORITMOS MULTIHEBRADOS ADAPTATIVOS EN SISTEMAS NO DEDICADOS

Tesis doctoral presentada por:
JUAN FRANCISCO SANJUAN ESTRADA

Dirigida por:
Dr. LEOCADIO GONZÁLEZ CASADO
Dr. INMACULADA GARCÍA FERNÁNDEZ

Enero de 2015

D. **Leocadio González Casado**
Profesor Titular
de la Universidad de Almería.
Dr. **Inmaculada García Fernández**
Catedrática de Arquitectura de Computadores
de la Universidad de Malaga.

CERTIFICAN:

Que la memoria titulada, **ALGORITMOS MULTIHEBRADOS ADAPTATIVOS EN SISTEMAS NO DEDICADOS**, ha sido realizada por **D. Juan Francisco Sanjuan Estrada** bajo nuestra dirección en el Departamento de Informática de la Universidad de Almería y concluye la Tesis que presenta para optar al grado de Doctor en Ingeniería Informática.

Almería, 16 de Enero de 2015

Dr. **Leocadio González Casado**
Director y Tutor de la Tesis

Dr. **Inmaculada García Fernández**
Co-directora de la Tesis

A Raquel, Hugo y Alejandro

Agradecimientos

El tiempo dedicado a la realización de esta Tesis Doctoral pasaba año tras año, sin vislumbrar una fecha de terminación, con mucho trabajo que no necesariamente se refleja en el documento final, pero que permite sustentar las bases de la investigación realizada. Sin embargo, la dedicación no ha sido exclusivamente mía, sino en gran parte de mis directores de Tesis los profesores Dr. Leocadio González Casado y Dra. Inmaculada García Fernández, quienes han sufrido mi larga agonía. Por esto, quiero agradecerles a ambos por iluminarme el camino cada vez que me encontraba con una dificultad, cuya amplia experiencia investigadora ha permitido solventar mi ceguera investigadora.

Me gustaría dedicar unas líneas agradeciendo a cada uno de los miembros del grupo de investigación *TIC146 Supercomputación: Algoritmos*, por sus esfuerzos en darme siempre las mayores facilidades y recursos posibles a la hora de investigar, quienes continuamente me animaban año tras año para finalizar la Tesis. No quisiera olvidarme del profesor Dr. José Antonio Martínez García, quien me enseñó el funcionamiento y me facilitó el código fuente de la aplicación *Local-PAMIGO*, intensivamente utilizada en esta Tesis Doctoral.

Agradecer la financiación recibida por distintos organismos que ha permitido subvenciar este trabajo, entre los que destacamos, el Ministerio de Ciencia e Innovación (*TIN2008-01117* y *TIN2012-37483*), la Junta de Andalucía (*P011-TIC7176*) y el Fondo Europeo de Desarrollo Regional (*ERDF*).

Sin menospreciar a nadie, lo principal y más importante es la familia, que te ofrece amor y comprensión a cambio de nada. Así que tengo que reconocer que mi mujer ha sido la principal instigadora en la terminación de la Tesis. Por esto quiero agradecer a Raquel su empeño en motivarme para realizar la Tesis, algo que ella hizo hace ya más de una década. Sus continuas palabras chocaban con mi cabezonería, pero finalmente esas palabras me han hecho creer que la vida de un doctor es mejor.

Finalmente, quiero aprovechar esta oportunidad para agradecer a mi familia, el apoyo constante que siempre me han dado. A mi madre, Maria Francisca, y mis hermanos Mari Trini, Tere y Manolo, por la suerte que he tenido de tenerlos, sin olvidarme mis tías Maina, Aurora y Tere y del resto de mi familia. Especialmente le agradezco a mi mujer Raquel, mis hijos Hugo y Alejandro, por todos los momentos felices que me hacen pasar cuando estamos juntos y que me permiten afrontar cada día con más alegría.

Prólogo

El incipiente aumento de microprocesadores con una arquitectura donde se combina un elevado número de procesadores en el mismo chip, más conocido como *MIC*, de tal forma que el número de núcleos por chip/nodo se está incrementando enormemente en los últimos años. Las principales dificultades que experimentan las aplicaciones cuando se ejecutan en los multiprocesadores actuales se acentúan conforme aumenta el número de procesadores integrados en el sistema. Existen aplicaciones multihebradas que no son eficientes cuando intentan usar todos los recursos de este tipo de sistemas. Las APIs ofrecen soporte a este tipo de aplicaciones para que obtengan un buen rendimiento, sin embargo, el paralelismo de tareas no funciona bien para todas las aplicaciones multihebradas, desafortunadamente los códigos paralelos B&B son un claro ejemplo, por las irregularidades que surgen en la resolución de distintos tipos de problemas.

La computación paralela está ayudada por librerías y herramientas que facilitán la programación paralela, sin embargo, estas librerías no terminan de resolver correctamente el problema que plantean las aplicaciones multihebradas en sistemas no dedicados, dejando toda la responsabilidad al sistema operativo (SO).

Esta tesis pretende avanzar en el ámbito de las APIs auto-configurables centradas en la programación multicore. Para ello, se estudian métodos que permita al SO cooperar con las aplicaciones multihebradas informandoles periódicamente del mejor número de hebras a utilizar según su rendimiento. Las aplicaciones deben adaptar su nivel de paralelismo para mantener un buen rendimiento, teniendo en cuenta la disponibilidad de recursos en tiempo de ejecución, especialmente en sistema no dedicados.

Uno de los factores que inciden directamente en el tiempo total de ejecución de la aplicación en un sistema multicore es el número de hebras. Tradicionalmente, el número de hebras es establecido por el usuario al inicio de la ejecución de la aplicación. Generalmente, el comportamiento de la aplicación depende directamente del número de hebras, que se mantiene invariable durante todo el tiempo de ejecución.

La finalidad de este trabajo de tesis es reducir el tiempo de computación de las aplicaciones multihebradas, para lo cual se propone que el algoritmo multihebrado pueda crear nuevas hebras, o incluso detener alguna hebra activa, de tal forma que el número de hebras activas se adapte dinámicanente a los recursos disponibles en cada momento. Sin embargo, la aplicación debe conocer el número óptimo de hebras en cada instante de tiempo que ofrece el mejor rendimiento posible. El gestor del nivel de paralelismo (GP) determinará periodicamente el número de hebras óptimo e informará a la aplicación para que pueda adaptarse durante la ejecución haciendo frente a las variaciones de las cargas computacionales en el sistema producidas por la propia aplicación, o incluso por otras aplicaciones que se ejecutan en el sistema no dedicado. Los GPs propuestos analizan el comportamiento de las aplicaciones multihebradas en ejecución y adaptan el número de hebras automáticamente, manteniendo el máximo rendimiento posible. Esta cooperación entre el GP, la aplicación y el SO será beneficiosa para una amplia variedad de aplicaciones, entre ellas, aquellas que utilizan técnicas de B&B.

En esta tesis se proponen distintos tipos de GP a nivel de usuario y a nivel de Kernel.

Los GPs a nivel de usuario se caracterizan por su fácil implementación, pero poseen ciertas limitaciones respecto a los criterios de decisión que pueden utilizar para estimar el número óptimo de hebras. Los GPs a nivel de kernel ofrecen diseños de mayor complejidad, ofreciendo diferentes versiones en función de la entidad encargada de estimar el número óptimo de hebras. También se ha diseñado la librería *IGP* que facilita al programador la implementación de aplicaciones multihebradas que interactuan con los distintos GPs.

Todas las experimentaciones han sido realizadas directamente sobre sistemas reales con el SO Linux, tanto dedicados como no dedicados, sin utilizar ningún tipo de simulador. El algoritmo *Ray Tracing* ha sido utilizado para verificar el comportamiento de los distintos GPs en sistemas dedicados, mientras que la aplicación B&B Local-PAMIGO ha permitido analizar distintos criterios de decisión integrados en los diferentes GPs utilizando sistemas dedicados y no dedicados.

Esta tesis se compone de cinco capítulos, comenzando con el Capítulo 1 donde se realiza una breve introducción sobre la adaptabilidad de los algoritmos multihebrados en arquitecturas multicore y analizando las APIs disponibles. También se detallan las características de las aplicaciones multihebradas, centrándonos en las distintas estrategias de creación de hebras y su comportamiento en sistemas no dedicados. Finalmente se detalla la aplicación B&B Local-PAMIGO que permite al usuario adaptar el número de hebras gracias a la estrategia de creación dinámica de hebras implementada.

El Capítulo 2 se centra en las etapas que integran un GP, y el procedimiento seguido para la implementación de las distintos tipos de GPs. Un benchmark basado en el algoritmo *Ray Tracing* diseñado especialmente para analizar el comportamiento de los diferentes GPs a nivel de usuario (*ACW* y *AST*) y a nivel de kernel (*KST*, *SST* y *KITST*) en sistemas dedicados. Se han realizado varios experimentos para estudiar diferentes características que inciden en la adaptabilidad de la aplicación, tales como, la duración de las secciones críticas de la aplicación, el impacto de la reserva dinámica de memoria sobre el rendimiento de la aplicación, la frecuencia de ejecución del GP para reducir su coste, y el mecanismo de detención de hebras.

El Capítulo 3 estudia la selección del mejor criterio de decisión que estima de forma precisa el número óptimo de hebras. Entre los distintos criterios de decisión analizados, resaltamos, el número de procesadores ociosos, rendimiento parcial de la aplicación, minimizar los retardos de la aplicación y estimación del número óptimo de hebras en función de los tiempos de bloqueo de las hebras. Todos los experimentos se han realizado en sistemas dedicados sobre una aplicación real de B&B con los distintos GPs usando diferentes criterios de decisión. El comportamiento irregular de la aplicación B&B ha incidido directamente en la adaptabilidad con diferentes GPs y criterios de decisión, de tal forma que la mejor adaptabilidad de la aplicación multihebrada B&B resolviendo distintos problemas irregulares se ha conseguido habilitando la detención de hebras en el GP a nivel de kernel.

El Capítulo 4 evalúa el comportamiento de la aplicación B&B y su adaptabilidad utilizando distintos GP (*ACW*, *KST* y *KITST*) en un sistema no dedicado. Finalmente, en el Capítulo 5 se exponen las conclusiones finales de la tesis, y el trabajo futuro.

Índice general

Índice de Algoritmos

Índice de figuras

Lista de Tablas

Capítulo 1

Algoritmos Multihebrados Adaptativos en entornos CMP

En este capítulo se ofrece una visión global de los aspectos científicos y tecnológicos que han motivado el desarrollo de este trabajo de tesis. Las cuestiones que se estudian están directamente relacionadas con la idea de optimizar el uso de los recursos computacionales disponibles, es decir de obtener el máximo rendimiento de la arquitectura sobre la que se ejecutan procesos con una alta demanda computacional. La idea global es extremadamente ambiciosa y va mas allá del horizonte científico que se alcanza en este trabajo. Nuestro planteamiento está limitado a arquitecturas multicore y algoritmos multihebrados que conviven con procesos de multiples usuarios; es decir en entornos no dedicados. El hecho de que nuestros procesos tengan que compartir los recursos disponible con otros procesos, nos obliga a dotarlos de un alto grado de adaptabilidad.

Para desarrollar nuestro trabajo hemos partido de la idea de que la adaptabilidad de nuestros algoritmos multihebrados se puede conseguir mediante el establecimiento óptimo del número de hebras activas en cada instante, que dependerá directamente de los recursos del sistema y de la coexistencia de otros procesos.

En este trabajo primero analizamos un modelo simple en el que los recursos computacionales están dedicados a un único algoritmo multihebrado. Bajo estas condiciones nos centraremos en el modelado de varios tipos de gestores del nivel de paralelismo que doten a las aplicaciones multihebradas de la capacidad de adaptarse en cada instante al estado real del sistema en el que se están ejecutando.

En este capítulo se describen las sobrecargas debidas al paralelismo en aplicaciones multihebradas, que suelen crecer con el número de hebras. Existen librerías y herramienas que falicitan la programación de aplicaciones paralelas. Normalmente el nivel de paralelismo de la aplicación es determinado de antemano. Por ejemplo, el número de hebras suele ser estático y decidido por el usuario. Estas librerías no modifican el número de hebras en tiempo de ejecución y por lo tanto no pueden mantener el nivel de eficiencia de la aplicación cuando se ejecuta en entornos no dedicados donde varias aplicaciones compiten por los recursos existentes. En este trabajo de tesis se propone la creación de varios tipos de gestores del nivel de paralelismo (GP) que ayuden a las aplicaciones multihebradas a

adaptarse al estado actual del sistema en el que se ejecutan.

En esta tesis nos centraremos en algoritmos SPMD (Single Program Multiple Data), en los que todas las hebras realizan el mismo programa. Por este motivo, todas las hebras sufren de las mismas sobrecargas debidas al paralelismo, lo que facilita la estimación y gestión de su número para mantener la eficiencia de la aplicación. Las aplicaciones en las que estamos interesados utilizan estructuras de datos irregulares, por lo que no se conoce de antemano la carga computacional asociada a un problema. Una asignación estática entre los elementos de proceso suele resultar en una baja eficiencia. La distribución de la carga entre hebras que se crean dinámicamente permite una mejor adaptación a este tipo de problemas irregulares.

En este capítulo se describen estos conceptos y los problemas a abordar en cada uno de los gestores de paralelismo propuestos. Finalmente, se realiza una breve descripción del contenido de cada uno de los capítulos de la tesis.

1.1. Tendencias en las arquitecturas paralelas

En los últimos tiempos están evolucionando rápidamente las arquitecturas *MIC* (Many Integrated Cores) provocando una revolución tecnológica en la computación paralela. Actualmente, se pueden encontrar en el mercado, por un lado, las arquitecturas multicore basadas en CPUs homogéneos con docenas de procesadores, y por otro lado, las arquitecturas heterogéneas CPU+GPU donde un menor número de procesadores, más lentos y complejos (CPUs), es ayudado por un elevado número de procesadores, más rápidos y sencillos (GPUs). Esta tesis se centra en el desarrollo de aplicaciones que se ejecutarán en arquitecturas multicore.

En el ámbito de las arquitecturas multicore destacan los Chip-Multiprocessors (CMP) que permiten aumentar el rendimiento de aplicaciones cuya carga computacional puede ser dividida entre múltiples hebras [47]. Esto permite distribuir las hebras entre los distintos núcleos del sistema, incrementando de esta forma el paralelismo y mejorando el rendimiento. Cuando el número de hebras es superior al número de núcleos disponibles se produce concurrencia entre las hebras que se ejecutan en el mismo núcleo. Un nivel de concurrencia elevado puede ser beneficioso para aquellas aplicaciones cuyas hebras realizan operaciones de entrada/salida, y perjudicial cuando las hebras necesitan disponer frecuentemente de las unidades de procesamiento.

Actualmente existen microprocesadores con un elevado número de unidades de procesamiento. Por ejemplo, la familia *Intel Xeon Phi coprocessor x100*, también conocida como *Knights Corner*, es considerada la primera generación de la arquitectura *Intel® Many Integrated Core* (Intel® MIC) [53]. Esta arquitectura combina un elevado número de procesadores Intel en un mismo chip. El primero de la serie posee un tamaño igual al de un microprocesador convencional, pero con más de 50 núcleos que generan un total de 1 TFLOP de procesamiento utilizando tecnología 3D Tri-gate de 22nm. Este sistema de procesamiento puede ser aplicado a tareas de alta complejidad computacional en una amplia variedad de campos, tales como la Física, Química, Biología, Servicios Financieros, etc. Los últimos *Knights Corner* disponibles comercialmente desde finales de 2013

permiten múltiples configuraciones, alcanzando los 61 núcleos con 244 hilos de ejecución y un rendimiento de 1.2 TFLOPs, así que, el número de núcleos por chip/nodo se está incrementando en los últimos años. Con este incremento de núcleos, existirán aplicaciones multihebradas que no puedan usar de forma eficiente todos los recursos del sistema, bien por las sobrecargas del paralelismo o porque el problema no tiene una carga computacional suficiente. La aplicaciones que se proponen en esta tesis, podrán adaptar su rendimiento, lo que permitirá mantener un buen grado de eficiencia, una reducción de consumo energético y/o que otras aplicaciones puedan compartir de forma más eficiente el sistema.

1.2. Interfaces de programación paralelos

Actualmente existen diversas herramientas que facilitan la programación de aplicaciones multihebradas sobre arquitecturas multicore, denominadas Interfaz de Programación de Aplicaciones (*API*), tales como *OpenMP* [48], *TBB* [54] o *Cilk* [21], pudiendo obtener un buen rendimiento en las aplicaciones que utilizan bucles, incluso con los bucles anidados.

Las APIs suelen obtener un buen rendimiento en muchos tipos de aplicaciones y normalmente la API no facilita al usuario la posibilidad de realizar una planificación de las tareas. Además, en algoritmos como los de Ramificación y Acotación (B&B: Branch-and-bound, ver Sección 1.7, página 11), la creación de una tarea por sub-problema generado puede resultar en un número alto de tareas, incrementando el coste de su gestión por parte del SO. Por lo tanto, el paralelismo de tareas funciona bien para muchas aplicaciones, pero desafortunadamente no ofrece buenos rendimientos para códigos paralelos de B&B [18, 45].

OpenMP permite una asignación dinámica de las iteraciones de un bucle a un número de hebras variable, mediante un particionado dinámico del bucle. En [34] se analiza y valora el número óptimo de hebras para cada bucle en función del tiempo de ejecución en una aplicación paralela.

Actualmente se están realizando esfuerzos para que los programas que usan OpenMP puedan medir y determinar los costes de su ejecución (ociosidad, trabajo, sobrecarga y espera en semáforos) [37]. Los estudios realizados en esta tesis permitiran informar a las aplicaciones multi-hebradas de su eficiencia y de las posibles causas en la bajada de rendimiento, para que puedan auto-configurarse en tiempo de ejecución.

La Librería de Intel Threading Building Blocks (TBB) utiliza un modelo basado en tareas, donde el cómputo se divide en unidades de trabajo indivisibles [54]. Las aplicaciones que hacen uso de esta librería generalmente crean tantas hebras como núcleos existan, repartiendo las tareas disponibles entre las hebras activas. El balanceo de la carga de trabajo entre las hebras lo realiza un planificador que implementa el concepto de *work-stealing*, similar al de *Cilk*. Básicamente, el balanceo de la carga consiste en que aquellas hebras que terminan todas sus tareas, roban tareas asignadas a otras hebras más sobrecargadas, y aquellas hebras que no consiguen más tareas para ejecutar pasan al estado de dormido.

El planificador de *work-stealing* está distribuido entre todas las hebras activas, de tal forma que todas las instancias del planificador se comunican entre sí, a través de memoria

compartida, tanto en la asignación de tareas entre las hebras como en la decisión de dormir una hebra.

El problema principal de la librería TBB se presenta en sistemas no dedicados, donde varias aplicaciones pueden utilizar distintas instancias de la librería, por lo que el rendimiento disminuye rápidamente. Este problema se ha resuelto implementando una hebra monitor, que periódicamente chequea la carga del sistema y comunica a los planificadores si deben detener o continuar su ejecución [41]. El monitor implementado chequea la carga del sistema a partir de la información disponible en el pseudo-fichero */proc/loadavg*, que junto con el número de procesos en ejecución, determina si el sistema está sobrecargado o no. Si se detecta que el sistema está sobrecargado, entonces se reducirá a la mitad el número de hebras en una de las aplicaciones, mientras que se creará una hebra nueva si el sistema está infra-utilizado. La hebra monitor interactua con el planificador TBB a través de memoria compartida, desactivando o activando flags, según sea el caso, para despertar o detener hebras, a través de la función *pthread_cond_wait()*. Según [41], este mecanismo permite reducir rápidamente la perdida de rendimiento en sistemas sobrecargados, y alcanzar progresivamente un valor óptimo.

En esta tesis se propone diseñar una herramienta que permita ayudar a las aplicaciones multihebradas a obtener buenos niveles de eficiencia tanto en sistemas dedicados como en sistemas no dedicados, de forma que el número de hebras en ejecución se adapte dinámicamente a la disponibilidad de recursos en el sistema.

1.3. Avances en los SO

Generalmente, si se tienen p unidades de procesamiento, las aplicaciones paralelas se ejecutan en p procesos que cooperan en la resolución del mismo problema. Inicialmente, estos procesos se comunicaban mediante paso de mensajes, lo que provocaba estados ociosos en las unidades de procesamiento. Posteriormente, se diseñaron procesos con dos hebras, una encargada de la computación y otra de la comunicación para solapar computación y comunicación, concurrentemente [2].

De forma similar a TBB, el SO realiza *thread-stealing* a la hora de asignar tareas a núcleos de procesamiento. El SO de aquellos núcleos con menos tareas o sin trabajo, roban tareas pendientes de ejecución de los núcleos más cargados. La gestión de las tareas por parte del SO queda fuera del ámbito de esta tesis.

Desde el punto de vista del SO, en el caso concreto de los sistemas CMP, se ha realizado mucho esfuerzo en mejorar el planificador de tareas a ejecutar. Más concretamente se han realizado estudios sobre el balanceo de hebras entre procesadores teniendo en cuenta alguna de las siguientes métricas: a) ratio computación/comunicación de cada hebra, b) carga computacional de cada uno de los procesadores y c) número de hebras asignadas a cada procesador [66]. Muchos de estos trabajos se han realizado sobre simuladores de SO [67].

Desde el punto de vista de la aplicación, el número de hebras que cada proceso genera, en la mayoría de las situaciones, es fijo y viene especificado por el usuario. Sin embargo, también existen trabajos en los que los autores proponen un modelo dinámico de ejecu-

ción, en el que el número de hebras cambia durante la ejecución de la aplicación. Este modelo dinámico pretende maximizar el grado de utilización de las unidades de computo disponibles, basándose en características específicas de la aplicación (speed-up, eficiencia y tiempo de ejecución), ajustando dinámicamente el número de hebras al número de unidades de computo, pero sin la interacción con el SO [14]. Por ejemplo, en [63] se propone un modelo en el que las hebras pueden recibir información de la existencia de un procesador ocioso, de forma que una hebra que disponga de suficiente trabajo pendiente es capaz de tomar la decisión de generar una nueva hebra que realice parte de su trabajo. Para ello [63] diseñó una rutina del SO Linux que informa de la existencia de un procesador ocioso.

En el sistema operativo Linux, toda la información relativa a los procesos se guarda en su descriptor de proceso, representado mediante la estructura *task_struct* definida en el fichero *include/linux/sched.h*. A diferencia de otros sistemas operativos, Linux emplea el nombre de tareas (tasks) para referirse aquellas entidades ejecutables que permiten ser planificadas y ejecutadas de manera independiente. Mientras que el termino proceso se utiliza para describir a un programa en ejecución. Un proceso es la entidad mínima de ejecución a nivel de sistema operativo que puede poseer recursos como ficheros abiertos, o segmentos de datos independientes en memoria. Sin embargo, un proceso puede estar compuesto por varios hilos de ejecución (hebras) que pueden comunicarse a través del segmento de datos del proceso; compartiendo los recursos propiedad del proceso (por ejemplo, descriptores de ficheros abiertos). Sin embargo, los descriptores de proceso, tanto de procesos monohebra como multihebrados, se representan con una estructura del tipo *task_struct*. Por este motivo, el término tarea (task) se utiliza tanto para representar a procesos monohebrados como a las distintas hebras de una aplicación multihebrada.

La investigación presentada en esta tesis complementa pero no pretende sustituir otras metodologías que mejoran la explotación de sistemas multicore, tales como políticas de planificación dinámica o *task-stealing*.

1.4. Estudios previos sobre aplicaciones auto-configurables

Esta tesis pretende profundizar en el ámbito de las API auto-configurables centradas en la programación multicore, donde ya existen aportaciones recientes como puede ser el proyecto *AutoTune*. *AutoTune* desarrolla una herramienta de análisis automático del rendimiento que ofrece recomendaciones para mejorar el rendimiento de la aplicación [43]. Otro ejemplo es la herramienta ISAT diseñada por Intel para buscar automáticamente los valores de diferentes parámetros que inciden significativamente en el rendimiento de la aplicación, tales como el tamaño de la memoria cache en computación matricial, granularidad de las tareas en TBB, o la política de planificación del constructor paralelo OpenMP [38].

1.5. Eficiencia energética

En los últimos años, el estudio y mejora del consumo energético de los multiprocesadores está adquiriendo una especial relevancia. Esto es especialmente interesante en CMPs

puesto que el incremento en el número de procesadores requiere que se diseñen unidades de procesamiento donde el consumo energético esté limitado [25]. Actualmente se están barajando varias alternativas para reducir el consumo. Por un lado, utilizar tecnologías de integración que permitan reducir el voltaje de los procesadores [8], por otro lado, modificar el planificador del sistema operativo para que permita rebalancear periódicamente la carga de trabajo entre distintos procesadores para homogeneizar la temperatura en todo el chip [42]. En este sentido, las aplicaciones multihebradas adaptativas también pueden colaborar para reducir el consumo de energía, por ejemplo, se podría reducir el consumo energetico en base a reducir el rendimiento de las aplicaciones multihebradas. El rendimiento se puede limitar, estableciendo un número de hebras adecuado que permita mantener el nivel de consumo por debajo del umbral deseado.

Esta problemática se complica aún mas en sistemas no dedicados, donde varias aplicaciones se ejecutan simultáneamente, por lo que la asignación de los recursos computacionales y unidades de entrada/salida incide aún más en la necesidad de aplicaciones autoadaptables al sistema.

1.6. Características de las aplicaciones multihebradas

La ejecución eficiente de una aplicación multihebrada en un sistema viene determinada, entre otras cosas, por el número de hebras en ejecución, el cual no tiene por qué permanecer constante durante todo el tiempo de ejecución. Se define la eficiencia una aplicación con n hebras en ejecución, como

$$E_a(n) = \frac{S_a(n)}{n}, \tag{1.1}$$

donde $S_a(n)$ es el rendimiento de la aplicación multihebrada, también denominado *speed-up*. El *speed-up* se define como la relación entre el tiempo real de ejecución de la aplicación monohebrada y el tiempo real de ejecución de la misma aplicación con n hebras, según la siguiente expresión:

$$S_a(n) = \frac{T(1)}{T(n)}, \tag{1.2}$$

El número de hebras óptimo de una aplicación multihebrada es aquel que permite un nivel de eficiencia próximo a la unidad, lo que se traduce en un rendimiento lineal, y por lo tanto en un alto nivel de escalabilidad.

1.6.1. Modelo tradicional de desarrollo

El desarrollo tecnológico que actualmente están experimentando las arquitecturas multicore, respecto al incremento del número de procesadores, dificultan la tarea de los actuales sistemas operativos en la planificación de las aplicaciones. Las aplicaciones multihebradas están diseñadas para obtener el máximo rendimiento en este tipo de arquitecturas. En esta tesis se estudian métodos que permita al sistema operativo cooperar con las aplicaciones

multihebradas informandoles periódicamente sobre cual es el número óptimo de hebras a utilizar para obtener un rendimiento máximo.

El tipo de aplicaciones sobre las que se va a centrar esta tesis, son aplicaciones multihebradas cuyas hebras realizan el mismo tipo de computación sobre distintos datos, es decir, SPMD. De esta forma, todas las hebras de la aplicación deberían estar afectadas por los mismos tipos de retardos, pues ejecutan el mismo fragmento de código. Sin embargo, esta tesis presenta las bases para poder abordar otros tipos de aplicaciones multihebradas más complejas.

Aunque el tipo de arquitecturas en las que se centran los estudios son multiprocesadores de memoria compartida, no se descarta como trabajo futuro la extensión de los métodos propuestos para facilitar la aplicabilidad en sistemas distribuidos.

La incipiente proliferación de arquitecturas multicore con más unidades de procesamiento obligan a desarrollar aplicaciones multihebradas que se adapten a los recursos disponibles de una manera eficiente. Normalmente, el desarrollo de una aplicación multihebrada para un sistema computacional determinado, sigue los siguientes pasos:

1. Desarrollar la aplicación multihebrada.
2. Ensayar varias ejecuciones con diferentes números de hebras para ver la escalabilidad de la aplicación en el sistema.
3. Revisar la aplicación, para determinar posibles causas que provocan la perdida de rendimiento.
4. Reprogramar el código de la aplicación para reducir las causas.
5. Repetir el paso 2 mientras se pueda mejorar.

1.6.2. Sistemas no dedicados

El esquema de diseño descrito en la subsección previa consume mucho tiempo de desarrollo de aplicaciones y además solo es viable cuando se trabaja con sistemas dedicados. Cualquier cambio de configuración en la arquitectura puede acarrear una perdida de rendimiento en la aplicación. Por lo tanto, es importante que estas aplicaciones adapten su nivel de paralelismo para mantener su rendimiento teniendo en cuenta la disponibilidad de recursos en el sistema en tiempo de ejecución. Para conseguir este objetivo, es importante:

1. Determinar cual es la entidad que se encarga de monitorizar el rendimiento de la aplicación.
2. Establecer la frecuencia de monitorización, así como decidir que cantidad de información se debe recopilar para evaluar el rendimiento de la aplicación.
3. Detectar los motivos que limitan el rendimiento de la aplicación.
4. Adoptar las medidas correctas cuando se produce una perdida de rendimiento.

Uno de los principales factores que afectan directamente al tiempo total de ejecución de la aplicación en un sistema multicore es el número de hebras. La sobrecarga del paralelismo

Algoritmo 1.6.1 : Creación estática de hebras implementada en el proceso principal.

```
(1) funct main_static_process(WorkLoad, MaxThreads)
(2)   global_work_load = memory_allocator(WorkLoad);          Reserva memoria
(3)   while (MaxThreads ≠ {∅})
(4)       work_load_i = divide(global_work_load, MaxThreads);   Reparto del trabajo
(5)       create_thread(static_thread, work_load_i);
(6)       decrease(MaxThreads);
(7)   wait_results_threads();                  Esperar a que todas las hebras terminen
(8)   proccess_results();
```

crece cuando aumenta el número de hebras, debido principalmente a las comunicaciones entre las hebras y al uso concurrente de los recursos. Por este motivo, es importante establecer el número de hebras activas que permitan obtener un buen nivel de rendimiento. Las aplicaciones pueden ejecutarse varias veces con un número diferente de hebras para saber cual es el número de hebras apropiado. Sin embargo, esto consume tiempo y no se adapta a los posibles cambios de carga en sistemas no dedicados. Está generalmente extendido que el número de hebras debería ser igual al número de procesadores. Por ejemplo, se ha demostrado que es cierto para sistemas con 4 ó 8 procesadores [52]. Sin embargo, no se cumple para sistemas con un mayor número de procesadores, tal y como se verificó sobre ocho programas *PARSEC* ejecutados sobre un sistema con 24 procesadores. Estos programas no permiten cambiar el número de hebras durante la ejecución, por lo que su eficiencia dependía del número de hebras establecido a priori por el usuario [7].

Las aplicaciones que se ejecutan en sistemas de alto rendimiento (HPC) están normalmente diseñadas con algoritmos multihebrados que pretenden hacer un uso eficiente del sistema, con la finalidad de reducir el tiempo de computación. Existen varios motivos que limitan la ganancia de velocidad de la aplicación paralela. Entre ellos, el más común es el acceso a recursos compartidos, que influye directamente en el retardo, o tiempo en el estado de espera de las hebras, de la aplicación. Este retardo varía normalmente en función del número de hebras en ejecución. Para ello, el algoritmo multihebrado debería poder crear nuevas hebras, o incluso detener alguna hebra, de tal forma que el número de hebras sea el más adecuado en cada momento.

Existen dos tipos de estrategias para la creación de hebras en algoritmos multihebrados: estática o dinámica.

1.6.3. Creación estática

Cuando se conoce de antemano la carga de trabajo, esta se puede repartir de forma estática entre el número estático de hebras determinado por el usuario. También se puede usar un número de hebras estático para estructuras de datos irregulares, pero se requiere de mecanismos de balanceo dinámico de la carga para aumentar la eficiencia de la aplicación. Finalmente, cada una de las hebras termina su ejecución cuando el trabajo total haya concluido.

Algoritmo 1.6.2 : Trabajo computacional de las hebras.

```
(1)  funct static_thread(work_load)
(2)    do
(3)      while (work_load ≠ {∅})
(4)        critical_section(global_parameters);
(5)        if (∃idle_thread)
(6)          half_work_load = divide(work_load);                    Reparto de trabajo
(7)          send_work_load(id_idle_thread, half_work_load);
(8)          compute_work(work_load_i);                             Realizar el trabajo asignado
(9)        send_results();
(10)       work_load = get_work_load(busy_thread);                  Rebalancear trabajo
(11)   while (global_work_load ≠ {∅})
```

Las estructuras básicas para la creación estática de hebras se basan en dos funciones: *main_static_process*() y *static_thread*(). Los pseudocódigos que describen estas funciones se muestran en los Algoritmos 1.6.1 y 1.6.2. El proceso principal de la aplicación multihebrada ejecuta la función *main_static_process*(), donde el usuario informa de la carga de trabajo total de la aplicación, a través del parámetro $WorkLoad$, y por otro lado, del número máximo de hebras que puede crear ($MaxThreads$). Esta función se encarga de repartir todo el trabajo inicial de forma equitativa entre las hebras. Por lo tanto, este reparto se realiza en una fase inicial, incluso previamente a la creación de las hebras (línea 4 del Algoritmo 1.6.1). Posteriormente, cada una de las hebras creadas ejecuta la función *static_thread*() para realizar el trabajo asignado (*work_load*) por el proceso principal (Algoritmo 1.6.2). Sin embargo, cuando alguna hebra se queda sin trabajo, se establece un procedimiento de balanceo dinámico de la carga de trabajo entre las hebras (líneas 7 y 10 del Algoritmo 1.6.2). Una característica de la creación estática de las hebras es que todas las hebras creadas inicialmente, no terminan hasta que se haya completado la totalidad de la carga de trabajo de la aplicación (línea 11 del Algoritmo 1.6.2).

1.6.4. Creación dinámica

Otros algoritmos multihebrados realizan una creación dinámica de hebras, de tal forma que el balanceo dinámico de la carga es inherente a la creación de hebras [74]. Sin embargo, en este caso, una hebra muere cuando termina el trabajo inicialmente asignado, permitiendo que la hebra más sobrecargada cree una nueva hebra. De esta forma, se crean hebras mientras exista trabajo a realizar. Aunque este método pueda parecer menos eficiente, permite decidir si se aumenta o disminuye el número de hebras que se están ejecutando en el sistema. En el caso estático puede suceder que no exista trabajo suficiente para alimentar a todas las hebras. Esto puede ocurrir en la etapa inicial de reparto de la carga cuando se usan estructuras de datos regulares o incluso en etapas intermedias de la ejecución, cuando se usan estructuras de datos irregulares.

Los Algoritmos 1.6.3 y 1.6.4 muestran la estructura general de una creación dínamica

Algoritmo 1.6.3 : El proceso principal, encargado de crear la hebra inicial.

```
(1) funct main_dinamic_process(WorkLoad, MaxThreads)
(2)     global_work_load = memory_allocator(WorkLoad);        Reserva memoria
(3)     create_thread(dinamic_thread, global_work_load);      Crear la 1ª hebra
(4)     decrease(MaxThreads);
(5)     wait_results_threads();                               Esperar a que todas las hebras terminen
(6)     proccess_results();
```

Algoritmo 1.6.4 : Creación dinámica de hebras implementada en las hebras.

```
(1) funct dinamic_thread(work_load)
(2)     while (work_load ≠ {∅})
(3)         critical_section(global_parameters);
(4)         compute_work(work_load_i);                        Realizar el trabajo asignado
(5)         if (MaxThreads ≠ 0)
(6)           half_work_load = divide(work_load);             Reparto de trabajo
(7)           create_thread(dinamic_thread, half_work_load);  Crear hebras sucesivas
(8)           decrease(MaxThreads);
(9)     send_results();
(10)    increase(MaxThreads);
```

de hebras, donde se observa que el reparto de la carga de trabajo inicial se realiza por etapas. En una primera etapa, la función *main_dinamic_process*(), ejecutada por el proceso principal, crea una única hebra a la que se le asigna toda la carga de trabajo (línea 3 del Algoritmo 1.6.3). Posteriormente, en la segunda etapa, la hebra inicial crea una nueva hebra a la que le asigna la mitad de su carga de trabajo. Finalmente, en la etapa enésima, si existe carga de trabajo suficiente, será la hebra más sobrecargada la que cree una nueva hebra y le asigne la mitad de su carga de trabajo (línea 6 del Algoritmo 1.6.4). Este procedimiento se repite hasta alcanzar el número de hebras deseado. Cuando una hebra completa el trabajo asignado, entonces comunica su resultado y termina. Cada vez que se crea una hebra se decrementa el valor de la variable *MaxThreads* (línea 8 del Algoritmo 1.6.4), de forma que, si *MaxThreads* es cero no se generan más hebras. Cuando una hebra termina se incrementa el valor de *MaxThreads*, permitiendo la creación de otra si existe trabajo suficiente (línea 10 del Algoritmo 1.6.4). Una peculiaridad de esta estrategia de creación dinámica de hebras es que el balanceo de la carga de trabajo es intrínseco a la creación de hebras, evitando la necesidad de realizar un balaceo dinámico de la carga cuando el número de hebras es estático (líneas 7 y 10 del Algoritmo 1.6.2). Aunque la creación y terminación de hebras es un proceso rápido, podría tener un coste significativo si se realiza un número relativamente elevado de veces.

El número máximo de hebras ($MaxThreads$) se pasa como argumento de entrada a las funciones *main_static_process*() y *main_dinamic_process*(), como se puede observar en los algoritmos 1.6.1 y 1.6.3. Puede ocurrir que el valor de $MaxThreads$ difiera del número

óptimo de hebras durante la ejecución de la aplicación ya que depende del comportamiento de la aplicación y de la sobrecarga del sistema.

En la creación dinámica de hebras es especialmente útil conocer el número óptimo de hebras en cada instante de tiempo, para obtener el mejor rendimiento posible en función de los recursos disponibles. En esta tesis, el número de hebras optimo durante la ejecución de la aplicación lo determinará un gestor del paralelismo (GP). El GP solo puede interactuar con los algoritmos multihebrados auto-adaptativos, de tal forma que el número de hebras pueda adaptarse durante la ejecución para hacer frente a las variaciones de las cargas computacionales en el sistema producidas por la propia aplicación, o incluso por otras aplicaciones que pueden ejecutarse en entornos no dedicados. De esta forma, el algoritmo multihebrado adaptativo podría crear, detener o terminar hebras, para mantener el número de hebras establecido por el GP.

Por lo tanto, el gestor del nivel de paralelismo (GP) decide el número de hebras a ejecutar, monitoriza la ejecución de las hebras, y mantiene el rendimiento de la aplicación multihebrada al mejor nivel posible, según las características de la aplicación, carga de trabajo y recursos disponibles del sistema.

1.7. Algoritmos de B&B

Para validar y evaluar los modelos presentados en esta tesis, se ha elegido una aplicación en la que existe un alto grado de paralelismo del tipo SPMD. En concreto hemos elegido los algoritmos de B&B como caso de estudio, debido a que permiten la exploración independiente de las ramas del árbol de búsqueda por distintas hebras.

Los algoritmos de ramificación y acotación (B&B) son métodos bien conocidos para resolver problemas NP completos de optimización combinatoria, tales como, planificación del trabajo, asignación de tareas, problemas de camino mínimo, etc. Estos algoritmos realizan una búsqueda exhaustiva de la solución, generando un árbol de búsqueda mediante el particionamiendo recursivo el problema inicial en sub-problemas más fáciles de resolver. Para cada sub-problema se calculan límites de la solución que contienen, lo que permite podar ramas del árbol donde se garantiza que no existe la solución global.

La eficiencia de los algoritmos de ramificación y acotación depende de las siguientes reglas usadas en su desarrollo:

- División: cómo se descompone un problema en subproblemas.
- Acotación: cómo se calculan las cotas de la solución de un problema.
- Selección: cual es el problema que se procesará a continuación.
- Rechazo: cómo se sabe que un problema no contiene la solución global.

En algunas aplicaciones hay que introducir la regla de terminación, que indica cuando un problema no necesita más procesado, principalmente porque se ha llegado a la precisión requerida [6, 12, 27, 28].

Estos algoritmos utilizan estructuras de datos irregulares y necesitan del uso de balanceadores dinámicos de la carga en su paralelización. Por lo tanto, estos algoritmos suponen

un reto a la hora de obtener un buen rendimiento en arquitecturas paralelas. Los estudios de esta tesis parten de una versión paralela del algoritmo B&B que utiliza aritmética de intervalos, denominado *Local-PAMIGO*, la cual ofrece buenos niveles de ganancia en velocidad en sistemas multicore [13].

1.7.1. *Local-PAMIGO*

Local-PAMIGO adapta el número de hebras en tiempo de ejecución ofreciendo buen rendimiento en sistemas multicore. Esta aplicación utiliza creación dinámica de hebras en tiempo de ejecución sin implementar ningún mecanismo explícito de balanceo de la carga dinámico. *Local-PAMIGO* utiliza la librería *POSIX Thread NTPL* [44], lo que a diferencia de las *APIs* descritas anteriormente (ver Sección 1.2, página 3) permite un mayor control sobre la ejecución de la aplicación. En *Local-PAMIGO* el usuario establece como parámetro de entrada, el número máximo de hebras simultaneamente en ejecución. En [59] se estudia como adaptar el número de hebras al rendimiento de la aplicación *Local-PAMIGO* y a los recursos computacionales en tiempo de ejecución, sin la interacción con el usuario. Algunos estudios proponen que el SO debe interactuar con los programas en tiempo de ejecución, de tal forma que se deben desarrollar programas paralelos que automáticamente adapten su comportamiento a la arquitectura y los recursos disponibles en tiempo de ejecución [51]. En este sentido, [60] propone cuatro métodos con diferentes métricas utilizadas para determinar cuando debe crearse una nueva hebra de la aplicación paralela, así como las modificaciones necesarias del Kernel del SO.

Según la clasificación propuesta en [23], *Local-PAMIGO* tiene un nivel de paralelismo que puede ser clasificado como *AMP* (Asynchronous Multiple Pool). En *Local-PAMIGO* el usuario establece inicialmente el máximo número de hebras ($MaxThreads$) como parámetro de entrada, de tal forma, que aquella hebra con suficiente trabajo genera una nueva hebra con la mitad de su carga de trabajo, si observa que el número de hebras activas es inferior a $MaxThreads$. Generalmente, el usuario establece $MaxThreads$ igual al número de núcleos (p) del sistema. *Local-PAMIGO* no es adaptativo, pues se intenta que el número de hebras sea igual a $MaxThreads$ durante toda la ejecución del algoritmo. El valor óptimo de $MaxThreads$ que permite minimizar el tiempo total de ejecución de la aplicación, dependerá de la arquitectura, nivel de paralelismo del algoritmo, tamaño y dificultad del problema, habilidad del programador, etc. Este aspecto se complica cuando se trabaja con sistemas no dedicados, donde otras aplicaciones pueden ejecutarse en cualquier momento.

Se podría pensar que el número de hebras que pueden ser creadas dinámicamente en cada instante debería de coincidir con el número de procesadores ociosos disponibles en el sistema. Sin embargo, la Ley de Amdahl muestra que el rendimiento de la aplicación paralela se determina por la fracción de código que puede ser paralelizado [24]. Por ejemplo, si el 50 % del tiempo de ejecución puede ser paralelizado y el 50 % restante corresponde a una sección crítica (puede ser ejecutado solo por una hebra), entonces la máxima ganancia en velocidad que puede obtenerse es de 2, independientemente del número de hebras de la aplicación y de los recursos del sistema paralelo.

1.8. Problemas a resolver

Sería conveniente que cualquier aplicación multihebrada pueda establecer el número adecuado de hebras en ejecución que permita mantener el máximo rendimiento posible. Este número optimo de hebras no es constante, pues depende de muchos factores que inciden directamente en el rendimiento de la aplicación, tales como: el número de unidades de procesamiento del sistema dedicado, tamaño de la carga computacional del problema a resolver, y las características de la implementación de la aplicación que puede incidir en retrasos no deseados de la aplicación. Esta problemática se acentúa aún más en sistemas no dedicados, donde también se deben analizar la influencia de otras aplicaciones en ejecución que compiten por los recursos disponibles.

Los gestores a nivel de paralismo (GPs) permiten analizar el comportamiento de la aplicación y adaptar el número de hebras al entorno de ejecución, para intentar mantener el máximo rendimiento posible. Algunos de los factores que debe tener en cuenta el GP son:

- Estimación del trabajo computacional pendiente de la aplicación.

  La mayoría de las estimaciones encontradas en la literatura se basan en el número de iteraciones pendientes en un bucle computacional [35, 73]. Esta tesis estudia los algoritmos de optimización global que hacen uso de técnicas de B&B. En estos algoritmos, el número de iteraciones, en términos de nodos del árbol evaluados, y el trabajo computacional por nodo se desconocen a priori. Además, este tipo de algoritmos sufren de anomalías de búsqueda, por ejemplo, el número de nodos evaluados puede ser diferente en versiones paralelas y secuenciales [17]. Algunos estudios recientes no consiguen un buen rendimiento paralelo en algoritmos B&B multihebrados [19, 20, 46].

- Métricas para determinar la disponibilidad de recursos.

  La métrica más extendida para la disponibilidad de recursos se basa en la disponibilidad de procesadores ociosos, aunque también se utiliza el consumo de energía en el sistema [68]. Algunos estudios analizan los efectos en sistemas cuando los recursos son compartidos entre varias aplicaciones [61, 73]. En [50] se comenta textualmente: *Portable parallel programs of the future must be able to understand and measure /any/ computer on which it runs so that it can adapt effectively, which suggests that hardware measurement should be standardized and processor performance and energy consumption should become transparent.* Además, la aplicación puede informar directamente al SO, para que el SO pueda monitorizar las hebras de la aplicación. Esta cooperación entre la aplicación y el SO será beneficiosa para muchas aplicaciones que usan técnicas de B&B [4].

  En [62] se estudia la importancia de diferentes medidas de tiempos relacionados con la disponibilidad de los recursos del sistema para determinar el rendimiento de una aplicación paralela basada en un algoritmo B&B. Esta aplicación paralela permite la creación de nuevas hebras en tiempo de ejecución, sin modificar el planificador del SO.

- Periodicidad en el análisis de la métricas para determinar la creación de nuevas hebras.

  Cualquier cambio en el sistema y/o aplicación tiene que ser detectado tan pronto como sea posible, ya que el objetivo es mantener un buen nivel de rendimiento de la aplicación. Las operaciones para conseguir estos objetivos deberían realizarse solo cuando sean necesarias, para reducir así el consumo de recursos. La mayoría de las estadísticas relativas a la medida de tiempos de la aplicación son realizadas por el SO y las decisiones sobre el nivel de paralelismo se realizan cuando existen procesadores ociosos para evitar el consumo de núcleos que están dedicados a otras tareas [59, 61, 60].

En esta tesis se estudian distintos tipos de GPs estableciendo las ventajas e inconvenientes de cada uno de ellos. Se comienza diseñando varios GPs a nivel de usuario. Estos GPs se caracterizan por su fácil implementación, pero presentan deficiencias en cuanto a velocidad de respuesta y limitaciones respecto a la estimación del número de hebras. Posteriormente, se implementan varios GPs a nivel de kernel, cuyos diseños aumentan progresivamente en complejidad, ofreciendo distintas versiones en función de la entidad encargada de decidir el número óptimo de hebras.

Esta tesis no realiza los experimentos en simuladores de SO, sino que los realiza directamente en el SO Linux. Se ha usado un algoritmo *Ray Tracing* para facilitar el análisis del comportamiento de los distintos GP diseñados [55, 57]. Finalmente, se ha evaluado una versión de Local-PAMIGO (ver Sección 1.7) adaptada para interactuar con los distintos GPs en sistemas dedicados y no dedicados.

1.9. Estructura de la tesis

La introducción planteada en este capítulo se refiere al estado del arte, las motivaciones, la necesidad de diseñar Gestores de Paralelismo y las cuestiones abiertas relacionadas con los *Algoritmos Multihebrados Adaptativos en Entornos No Dedicados*. El resto de la tesis esta organizada de la siguente forma:

En el Capítulo 2 se establecen las fases que integran un gestor del nivel de paralelismo (*GP*) y el procedimiento seguido para su diseño, así como las distintas tipologías de *GP* que se pueden implementar. Posteriormente se analizan e implementan diferentes GPs a nivel de usuario (*ACW* y *AST*) y a nivel de kernel (*KST*, *SST*, y *KITST*). Todos ellos deciden la creación o no de nuevas hebras evaluando el criterio de decisión basado en igualar el número de hebras al número de núcleos del sistema. La adaptabilidad de las aplicaciones multihebradas se completa implementando un mecanismo de detención rotativo entre las hebras, que permite al GP detener y despertar hebras según los objetivos planteados. Finalmente, se analiza el comportamiento de distintos GP sobre un benchmark sintético basado en el algoritmo *Ray Tracing* que permite comparar los GP a nivel de usuario con los GP a nivel de kernel. Además, se realizan diferentes experimentos para establecer la frecuencia de ejecución del GP sin que el tiempo de computación del GP afecte excesivamente a la eficiencia de la aplicación. También se analiza la adaptabilidad de la aplicación

multihebrada a diferentes configuraciones y duraciones de las secciones críticas. Como ejemplo de sección crítica, se estudia el impacto de la reserva dinámica de memoria sobre el rendimiento de la aplicación, y su repercusión en la adaptabilidad eficiente. Finalmente, se completa el análisis de la adaptabilidad de las aplicaciones multihebradas habilitando la detención de hebras por parte del GP.

El Capítulo 3 profundiza en el diseño de criterios de decisión que permitan al GP adaptar de forma precisa las aplicaciones multihebradas al sistema. Se plantean distintos criterios basados en el número de procesadores ociosos, el rendimiento instantáneo de la aplicación, en minimizar los retardos de la aplicación y la estimación del número máximo de hebras en función de los tiempos de bloqueo de las hebras. Los experimentos se realizan sobre algoritmos de ramificación y acotación (B&B) usando diferentes criterios de decisión (rendimiento, procesadores ociosos y minimización de los retardos) en GPs a nivel de usuario (*ACW* y *AST*) y a nivel de kernel (*KST-MST* y *KITST-MST*), cuyos resultados son comparados con los datos de referencia obtenidos a partir del número de hebras estático que presenta un mejor rendimiento.

Finalmente, se realizan diferentes experimentos que permitan evaluar la adaptabilidad de distintos criterios de decisión basados en la estimación del número máximo de hebras en función de los tiempos de bloqueo de las hebras, tales como, *MNT_IBT* (máximo número de hebras en función del tiempo de bloqueo interrumpible), *MNT_NIBT* (máximo número de hebras en función del tiempo de bloqueo no interrumpible), y *MNT_BT* (máximo número de hebras en función del tiempo de bloqueo) sobre un GP a nivel de kernel. Además, se evalúa la adaptabilidad que experimenta la aplicación multihebrada B&B resolviendo distintos problemas irregulares en un sistema dedicado, para lo cual se habilita la detención de hebras en el GP a nivel de kernel.

En la Figura 1.1 se muestran los gestores diseñados a nivel de usuario y a nivel de kernel (izquierda) y los distintos criterios de decisión (derecha) propuestos en esta tesis. También se observan los criterios de decisión que puede utilizar cada uno de los GPs, teniendo en cuenta que todos los GPs propuestos no pueden hacer uso de todos los criterios de decisión propuestos. Esto se debe a que algunos criterios de decisión necesitan un tipo de información no disponible por el GP. En principio, todos los gestores a nivel de kernel pueden implementar cualquier criterio de decisión ya que tienen acceso de toda la información, sin embargo, los GP a nivel de usuario no pueden acceder a la información relativa a los retardos de la aplicación, disponibles exclusivamente a nivel de kernel. Por ejemplo, los criterios de decisión *MNET* (minimizar el tiempo en estados no ejecutables), *MST* (minimizar el tiempo en estado dormido), *MIBT* (minimizar el tiempo de bloqueo interrumpible), *MNIBT* (minimizar el tiempo de bloqueo no interrumpible) y *MWT* (minimizar el tiempo de espera).

En el Capítulo 4 se evaluá el comportamiento de la aplicación B&B adaptativa con diferentes GPs (*ACW*, *KST* y *KITST*) en un sistema no dedicado. La ejecución de varias instancias de la misma aplicación para resolver el mismo problema permite simular el comportamiento de un sistema no dedicado de forma controlada.

Finalmente en el capítulo de conclusiones se resaltan los puntos más relevantes de la investigación realizada, así como los trabajos a realizar como continuación de esta tesis.

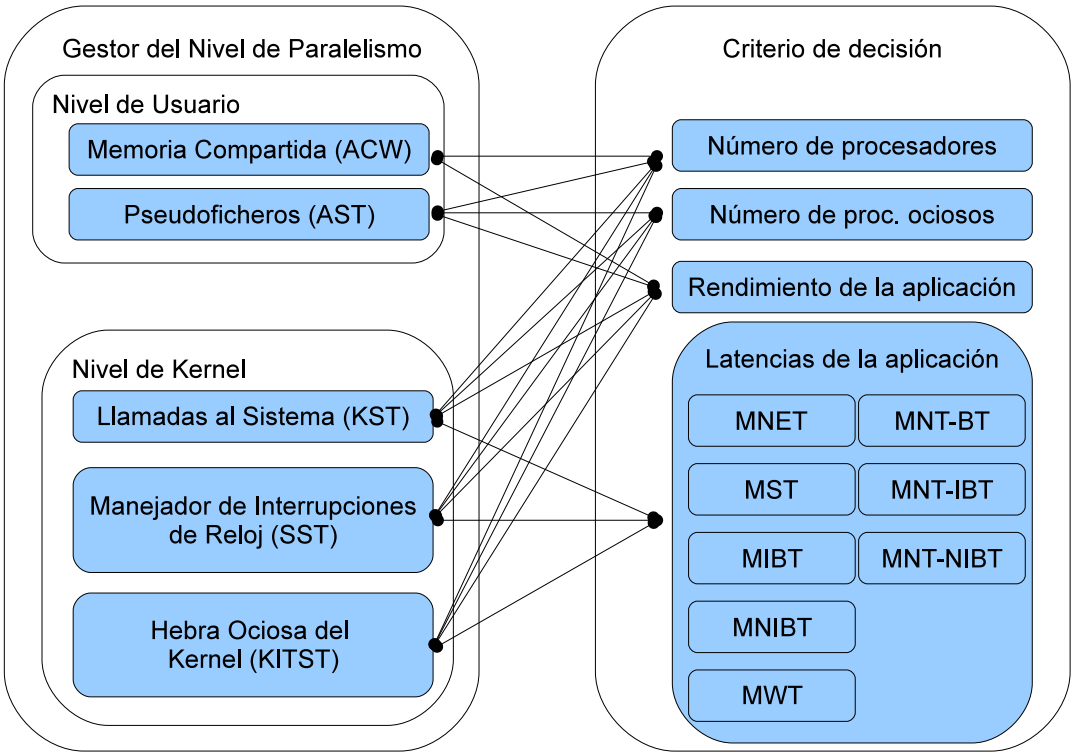


Figura 1.1: Criterios de decisión del GP.

Capítulo 2

Gestores del nivel de paralelismo

En este capítulo se diseñan varios gestores del nivel de paralelismo que informan a las aplicaciones multihebradas sobre el número de hebras que dan lugar a un uso más eficiente de los recursos del sistema. Estos estudios se hacen sobre algoritmos paralelos del tipo *SPMD* (Single Pogram Multiple Data) ya que presentan un modelo repetitivo y predecible de computación que permite una mejor estimación del grado de paralelismo de la aplicación en un sistema dado. Entre las distintas tipologías de gestores de paralelismo (GP) propuestos, se presentan las ventajas e inconvenientes de implementar el GP a nivel de usuario y a nivel de kernel. Además, se exponen diferentes versiones para cada uno de los tipos de GPs propuestos comenzando con los gestores más simples y aumentando su complejidad.

Todos los GP planteados en este capítulo se ensayan sobre un benchmark sintético basado en el algoritmo *ray tracing*, pues este tipo de aplicaciones son fácilmente escalables y permiten acotar las características intrínsecas de una aplicación multihebrada (secciones críticas, reserva de memoria, etc.) para analizar el impacto del GP sobre cada uno de estos factores.

Este capítulo se ha organizado de la siguiente forma: La Sección 2.1 es una breve introducción al problema, donde se describe el modelo de trabajo y sus restricciones. En la Sección 2.2 se describen las características que debe tener cualquier GP, así como el procedimiento que usan los GP propuestos en este trabajo. Todos los GPs planteados en la Subsección 2.2.2 se analizan en las Secciones 2.3 y 2.4, siguiendo su evolución desde las versiones más sencillas a nivel de usuario hasta las versiones más eficientes que se integran en el Kernel del SO. Posteriormente, en la Sección 2.6 se describe la libreria IGP (Interfaz del Gestor del nivel de Paralelismo) utilizada para la comunicación entre la aplicación multihebrada y los distintos tipos de gestores. En la Sección 2.5 se explica la técnica para detener momentáneamente la ejecución de algunas hebras de la aplicación, lo que permitirá una adaptación más eficiente de las aplicaciones multihebradas. La experimentación y evaluación de los distintos GPs planteados se realiza en la Sección 2.7, donde además de analizar las ventajas e inconvenientes de cada uno de los gestores, se determinará cual es el GP más eficiente. Finalmente, en la Sección 2.8 se exponen los resultados de un conjunto de experimentos realizados usando un benchmark. Este benchmark se ha diseñado

expresamente para analizar la adaptabilidad del mejor gestor del nivel de paralelismo, sobre diferentes supuestos centrados en potenciar las distintas tipologías de retardos que suelen afectar al rendimiento de las aplicaciones multihebradas. El capítulo termina con una sección que resume las contribuciones del trabajo tratado en este capítulo y plantea algunas posible futuras líneas de investigacion.

2.1. Introducción

El GP es el responsable de calcular el número óptimo de hebras activas en función de los objetivos propuestos. El criterio de decisión, tradicionalmente más utilizado en HPC, se basa en establecer un número estático de hebras igual al número de unidades de procesamiento del sistema. El parámetro que determina el número de unidades de procesamiento del sistema puede ser facilitado manualmente por el usuario, como argumento de entrada de la aplicación. No obstante, el número de unidades de procesamiento en el sistema se puede obtener de forma automática por la aplicación, mediante la lectura del fichero (*/proc/cpuinfo* del SO Linux).

En este capítulo utilizaremos el criterio de decisión más sencillo que establece el número de hebras igual al número de unidades de procesamiento disponibles en el sistema. Este criterio de decisión generalmente suele obtener buenos niveles de eficiencia, especialmente para aplicaciones multihebradas escalables en entornos dedicados. Se pretende mostrar los distintos tipos de GP cuando la decisión a tomar es muy simple. Uno de los objetivos de la creación adaptativa de hebras será minimizar el tiempo de ejecución de la aplicación, con el menor número de hebras posible, y/o mantener un nivel de eficiencia cercano a la unidad. Sin embargo, tanto la creación estática, como dinámica, de hebras basada en el número de unidades de procesamiento del sistema descrítos en 1.6, presentan las siguientes desventajas:

1. Existen aplicaciones, por ejemplo algoritmos de Ramificación y Acotación, donde el trabajo a realizar no se conoce de antemano, pues se crea y destruye en tiempo de ejecución. Por este motivo para determinar el mayor número de hebras estático que mantenga la eficiencia de la aplicación, la aplicación debe ejecutarse con las distintas alternativas posibles.

2. El elevado número de unidades de procesamiento en los sistemas multicore del futuro, pueden dar lugar a que existan aplicaciones paralelas con un nivel insuficiente de escalabilidad para hacer un uso eficiente de todos las unidades de procesamiento.

3. En entornos no dedicados, donde la aplicación multihebrada se ejecuta concurrentemente con otras aplicaciones, se requiere que la aplicación multihebrada se adapte a los recursos disponibles. En este sentido, habrá que determinar prioridades entre las aplicaciones. En el capítulo cuarto se aborda el caso más sencillo, en el que los algoritmos paralelos adaptativos tienen menor prioridad que las demás aplicaciones y la misma entre ellos.

2.2. Diseño de un gestor adaptativo

El gestor del nivel de paralelismo de una aplicación es el responsable de informar sobre el número óptimo de hebras activas al algoritmo multihebrado, así como de monitorizar las hebras en ejecución y analizar el rendimiento de la aplicación. Por todo ello, el GP se considera un elemento clave para obtener niveles adecuados de eficiencia. Como se comentó en la Sección 2.1, el número de hebras activas afecta a la eficiencia de la aplicación multihebrada en un sistema con unos recursos limitados. Por este motivo, la adaptabilidad del algoritmo multihebrado se centra en disponer de un gestor capaz de realizar una estimación del número óptimo de hebras activas. Todo gestor del nivel de paralelismo debería ser del tipo *LER* (*Ligero*, *Eficaz* y *Rápido*):

- *Ligero*: El tiempo de computación del GP debe ser despreciable frente al tiempo de computo de la aplicación. La ejecución del gestor debería de consumir muy poco tiempo, o por lo menos, un tiempo despreciable frente al tiempo consumido por cada hebra en una iteración. Por este motivo, el gestor debe tener un código reducido, sin secciones críticas bloqueantes, y minimizar el consumo de los recursos computacionales. Una forma adicional de reducir el consumo de recursos por parte del GP consiste en reducir el número de veces que debe ejecutarse el gestor.

- *Eficaz*: Esta característica requiere un criterio de decisión eficaz que establezca el número de hebras óptimas en cada instante. El criterio de decisión debe ser capaz de analizar las limitaciones en cuanto a la escalabilidad de la aplicación, que vendrán establecidas por la disponibilidad de los recursos computacionales del sistema. Sin olvidar, que en entornos no dedicados, también debemos tener presente los efectos de la ejecución concurrente de diferentes aplicaciones.

- *Respuesta rápida*: El GP debe de ser capaz de detectar rápidamente una disminución del rendimiento de la aplicación multihebrada y gestionar la creación, detención y/o eliminación de hebras para mantener un nivel adecuado de la eficiencia. Por lo tanto, la rápida recopilación de información utilizada por el criterio de decisión debe permitir un periodo de muestreo pequeño.

2.2.1. Procedimiento de uso de un GP

El gestor del nivel de paralelismo inicia la monitorización cuando entra en ejecución la primera hebra de la aplicación multihebrada, siempre que esta hebra haya realizado una unidad de trabajo (por ejemplo, una iteración de un bucle). Independientemente de los parámetros escogidos por el GP para establecer el número de hebras adecuado, se deben establecer los procedimientos que los algoritmos multihebrados deben seguir después de ser informados por el GP sobre su nivel de paralelismo. Particularmente, en la creación dinámica de hebras es recomendable que la hebra con una mayor carga de trabajo sea la responsable de crear una nueva hebra, siempre que el número de hebras activas sea inferior al valor de $MaxThreads$ propuesto por el GP.

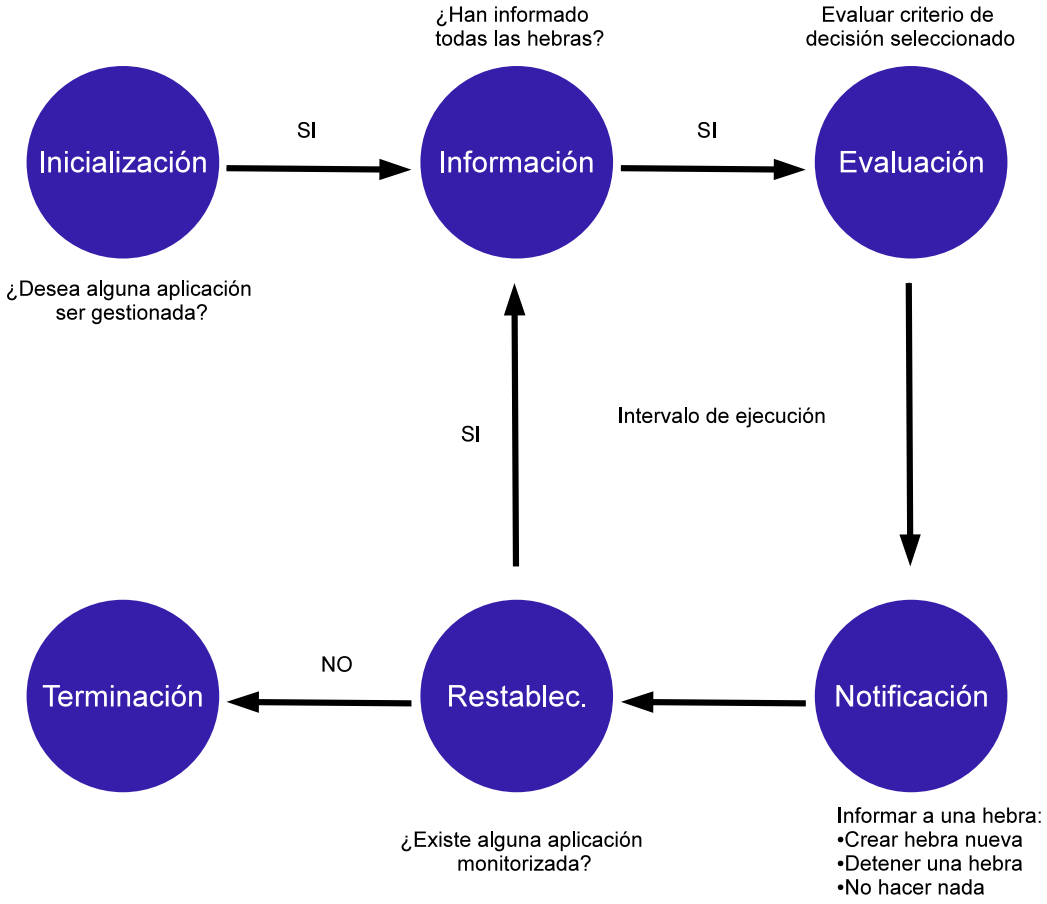


Figura 2.1: Diagrama de fases de funcionamiento de un GP.

La Figura 2.1 muestra el esquema de las distintas fases de funcionamiento en las que puede estar trabajando el GP. El proceso principal y las hebras de la aplicación multihebrada informan al GP a través de la fase de *Inicialización*, su deseo de ser gestionados por el GP. Posteriormente, en la fase de *Información*, cada una de las hebras computacionales informan al GP de los datos requeridos por el criterio de decisión, normalmente las unidades de trabajo realizadas desde el último informe. Una vez que todas las hebras activas han informado, el GP ejecuta la fase de *Evaluación* donde valora el criterio de decisión seleccionado, que puede informar sobre crear una nueva hebra, detener una hebra activa o no hacer nada. La decisión tomada es notificada a la hebra seleccionada por el GP en la fase de *Notificación*. A continuación, en la fase de *Restablecimiento*, el GP resetea todos los datos de la última evaluación realizada antes de comenzar un nuevo intervalo de evaluación con la fase de *Información*. El GP ejecuta la fase de *Terminación*, una vez que, tanto las hebras activas, como el proceso principal, han terminado su carga de trabajo e informan al GP su deseo de terminar. A continuación se detallan las distintas fases en

las que puede encontrarse el gestor del nivel de paralelismo para mantener un nivel de eficiencia adecuado:

1. *Fases de Inicialización y Terminación*: La fase de *Inicialización* permite al proceso principal de una aplicación multihebrada activar el GP para que gestione la ejecución adaptativa de las hebras. En la fase de *Inicialización*, cada una de las hebras computacionales tienen que notificar al GP cual es la carga de trabajo asignada inicialmente. Sin embargo, si la aplicación crea otras hebras satélites sin carga computacional, destinadas a realizar otro tipo de tareas (por ejemplo, operaciones de entrada/salida, rebalanceo de carga, etc...), estas no tienen que ejecutar la fase de *Inicialización*.

   Por otro lado, cuando una hebra o el proceso principal finaliza su ejecución, debe ejecutar la fase de *Terminación* para indicar al GP que ya no tiene que ser monitorizada.

2. *Fase de Información*: Cada una de las hebras activas, debe informar al GP periódicamente, por ejemplo, en cada iteración del bucle, sobre la carga de trabajo realizada desde la última vez que informó. Esto permitirá al gestor calcular la carga de trabajo realizada por unidad de tiempo. El GP permanecerá en esta fase hasta que todas las hebras gestionadas hayan informado, pues debe asegurarse de que todas las hebras están ejecutando el respectivo bucle computacional. En esta fase, el GP puede además recabar otra información adicional, no disponible por las hebras a nivel de usuario aunque si a nivel del SO, que puede ser utilizada posteriormente por el criterio de decisión seleccionado.

   La fase de *Información* es la más crítica de todo el GP, dado que es la fase que más tiempo tarda en completarse, pues su tiempo de ejecución depende directamente del mecanismo utilizado por el GP para obtener la información utilizada en la fase de *Evaluación*.

3. *Fase de Evaluación*: En esta fase, el gestor debe analizar toda la información recogida en el fase anterior, y aplicar un criterio de decisión para obtener conclusiones sobre el nivel de eficiencia medido en el último intervalo de análisis. La elección de un criterio de decisión es de crucial importancia, para alcanzar niveles de eficiencia aceptables. El análisis del criterio de decisión puede desencadenar la creación de una nueva hebra, detención de alguna de las hebras activas, o simplemente no realizar nada pues se ha alcanzado un nivel óptimo de eficiencia.

   En entornos dedicados, el criterio de decisión más utilizado para aplicaciones HPC es el basado en el número de unidades de procesamiento. Sin embargo, este criterio de decisión es poco eficiente cuando trabajamos con entornos no dedicados, o las aplicaciones paralelas no tienen carga computacional suficiente, para que su ejecución en todos los procesadores del sistema sea eficiente. En los siguientes capítulos se estudiarán otros criterios de decisión tanto en entornos dedicados como no dedicados, tales como, el número de procesadores ociosos, el rendimiento de la aplicación, y los retardos de la aplicación.

4. *Fase de Restablecimiento*: Una vez tomada una decisión, el GP inicializa los contadores utilizados, como por ejemplo: la carga de trabajo realizada por cada hebra, el número de hebras activas, y el trabajo pendiente de cada hebra. Esta fase de reestablecimiento de contadores se va a ejecutar siempre que el número de hebras activas cambie, es decir, cada vez que se cree, detenga o termine una hebra. De esta forma, se garantiza que la fase de *Evaluación* valore el criterio de decisión con los parámetros recopilados en la fase de *Información*, es decir, durante el periodo de tiempo más reciente en el que no ha cambiado el número de hebras activas. La fase de *Restablecimiento* está íntimamente relacionada con la fase de *Evaluación*, dado que se debe de ejecutar una vez que el GP tome una decisión en la fase de *Evaluación*.

5. *Fase de Notificación*: En esta fase, el gestor del nivel de paralelismo notifica a cada una de las hebras activas la decisión tomada en la fase de *Evaluación*. En el supuesto de que el GP decida crear una nueva hebra, se debe seleccionar la hebra más cargada, e informarle para que se divida. Una vez que la hebra más cargada detecta que tiene que dividirse, está crea una nueva hebra y le asigna la mitad de su carga de trabajo. Si por el contrario, el gestor toma la decisión de detener o activar una hebra, entonces seleccionará una hebra activa o detenida, respectivamente. El mecanismo para seleccionar la hebra a detener es más complejo, ya que se debería detener aquella hebra que menos incida en la resolución del problema, por ejemplo, la hebra menos sobrecargada. Sin embargo, existen aplicaciones donde la hebra menos sobrecargada puede contener el trabajo pendiente que permita encontrar la solución del problema, por lo que su detención aumentaría el tiempo total de ejecución.

En cualquier gestor del nivel de paralelismo destacamos las fases *Información*, *Evaluación*, *Restablecimiento* y *Notificación*, pues influyen significativamente sobre las características *LER*, comentadas anteriormente. Como veremos más adelante, los aspectos claves de cualquier GP que permitan mejorar las características *LER* (Ligero, Eficaz y Rápido), se centraran principalmente en dos aspectos: Por un lado, utilizar distintas metodologías que permitan minimizar el retardo de las fases *Información* y *Notificación*, y por otro lado, determinar donde se van a ejecutar las fases de *Evaluación* y *Restablecimiento*.

2.2.2. Tipologías del gestor del nivel de paralelismo

Los gestores planteados en este capítulo se pueden clasificar en función de donde se van a ejecutar las distintas fases del GP. En la Figura 2.2 se muestra la estructura interna del sistema operativo Linux, donde se pueden apreciar tres niveles funcionales: nivel de usuario, nivel de kernel y nivel hardware. Las aplicaciones multihebradas se ejecutan en el nivel de usuario haciendo uso tanto de las librerías disponibles, como de llamadas al sistema, para hacer uso de los recursos hardware. El nivel de kernel se estructura en cinco grandes bloques: interfaz de llamadas al sistema, subsistema de ficheros, control de tareas, drivers de dispositivos y control hardware. Finalmente, el nivel hardware integra todos los dispositivos hardware de los que consta el sistema.

A continuación se describen nuestras propuestas en relación con las posibles alternativas de gestores del nivel de paralelismo. Todas ellas están basadas en el procedimiento

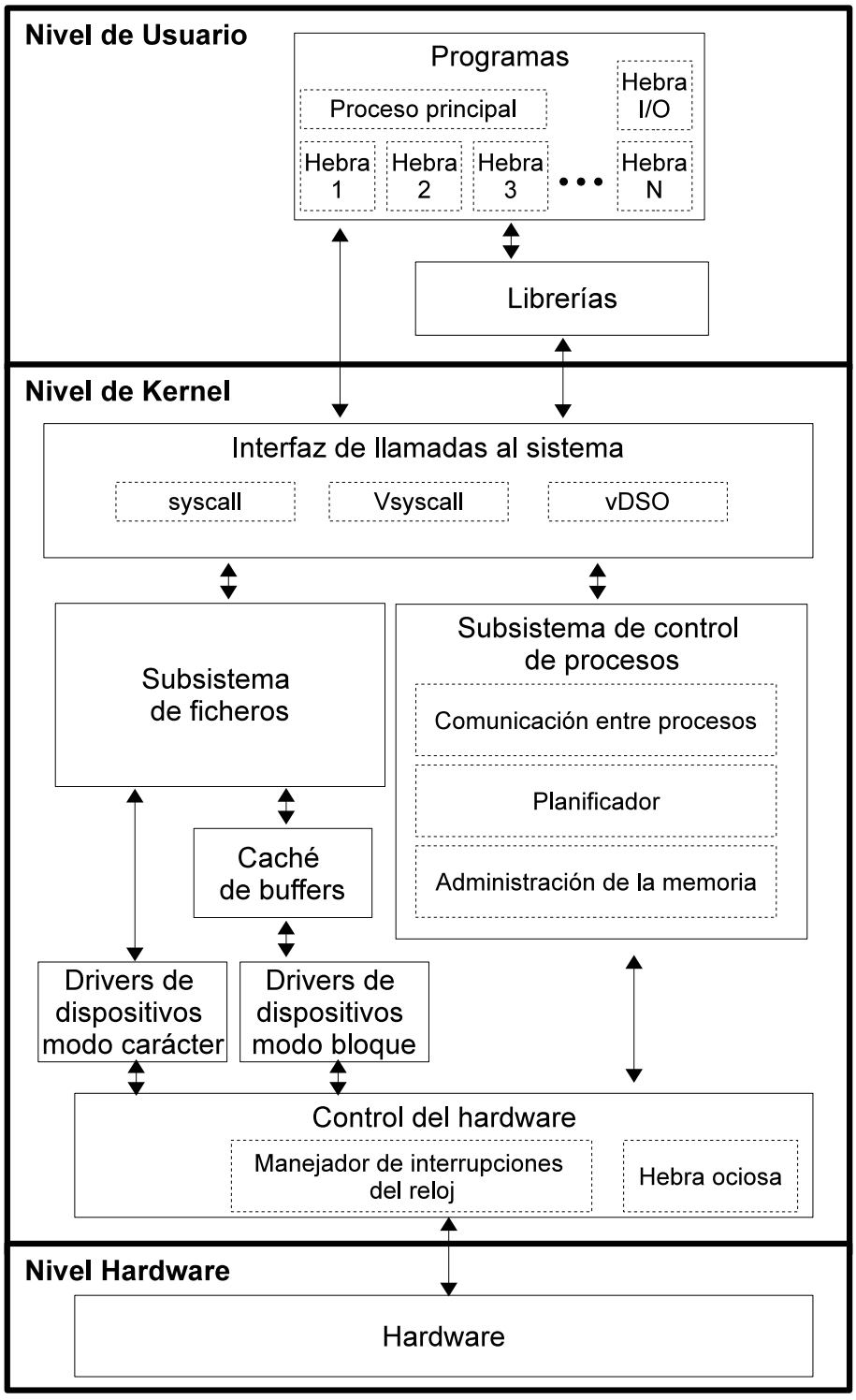


Figura 2.2: Estructura del sistema operativo Linux.

de uso de un GP como el descrito en la Sección 2.2.1. Se plantean diferentes ubicaciones para la fase de *Evaluación* de un GP: a nivel de usuario y a nivel de kernel. Adicionalmente, también se plantean distintas mejoras para acelerar las fases de *Información* y de *Notificación*. En este sentido, se realiza la siguiente clasificación:

1. Gestores de paralelismo (GP) a nivel de usuario): En este tipo de gestores la fase de *Evaluación* se ejecuta como parte de la aplicación multihebrada, o a través en una aplicación externa, pero siempre a nivel de usuario.

   a) Gestor *Aplication based on Completed Work* (*ACW*): En la fase de *Información*, las hebras y el GP intercambian información mediante el uso de memoria compartida.

   b) Gestor *Aplication based on Sleeping Threads* (*AST*): Este GP realiza lecturas en los pseudo-ficheros del SO para conocer el número de hebras que se encuentran en el estado de dormido, que será utilizado por el criterio de decisión establecido en la fase de *Evaluación* del GP.

2. Gestores a nivel de kernel: En este caso, las principales fases del GP: *Evaluación*, *Información* y *Restablecimiento* se ejecutan a nivel de SO.

   a) Gestor *Kernel decides based on Sleeping Threads* (*KST*): El gestor está integrado dentro de un módulo adicional del Kernel, de tal forma que se ejecutan las fases de *Información* y *Evaluación* a través de una llamada al sistema. La información sobre el número de hebras en el estado de dormido se obtiene directamente de las estadísticas realizadas por el SO.

   b) Gestor *Scheduler decides based on Sleeping Threads* (*SST*): En este caso, el GP realiza las mismas funciones que el modelo *KST*, pero está integrado en el núcleo del SO, de tal forma que el manejador de interrupciones del reloj ejecuta periódicamente las fases de *Información* y *Evaluación*.

   c) Gestor *Kernel Idle Thread decides based on Sleeping Threads* (*KITST*): Este gestor realiza las mismas funciones que los modelos anteriores, pero ejecuta la fase de *Evaluación* en la hebra ociosa del kernel, mientras que mantiene la ejecución de la fase de *Información* en el manejador de interrupciones del reloj.

En las siguientes secciones se detalla el funcionamiento de cada uno de los gestores propuestos.

2.3. GP a nivel de usuario

En principio, los gestores a nivel de usuario pueden usarse fácilmente en las aplicaciones multihebradas ya que su implementación es relativamente sencilla. En este caso, se utiliza un gestor ACW integrado dentro de la propia aplicación, lo que permite utilizar la memoria compartida entre las hebras activas como medio de comunicación.

Algoritmo 2.3.1 : Gestor del paralelismo a nivel de usuario

```
(1) funct IGP_get(this_thread, left_over, num_proc)
(2)   var final_decision;
(3)   update_loads_threads(this_thread, left_over);          Fase de Información
(4)   final_decision = check_decision_role(this_thread, num_proc);   Fase de Notificación
(5)   return final_decision
```

El algoritmo 2.3.1 muestra la función *IGP_get*() del gestor ACW ejecutada por cada una de las hebras de la aplicación multihebrada. Esta función requiere los tres parámetros siguientes:

1. *this_thread*: identificador de la hebra que está ejecutando el gestor en ese instante,

2. *left_over*: carga de trabajo restante de la hebra actual, y

3. *num_proc*: número de unidades de procesamiento del sistema. Como ya hemos comentado anteriormente, este parámetro es opcional, pues puede ser obtenido automáticamente por el propio GP accediendo a los pseudo-ficheros de información creados por el SO.

En este caso, el GP es ejecutado por la propia aplicación multihebrada y todas las hebras deben informar de su carga de trabajo a través de variables globales en la fase de *Información* (línea 3 del Algoritmo 2.3.1). El Algoritmo 2.3.2 muestra la función *check_decision_role*() donde se detalla la fase de *Evaluación* completa. La fase de *Información* concluye cuando todas las hebras activas han informado de la carga de trabajo realizada, lo que permitirá continuar la ejecución del GP (línea 8 del Algoritmo 2.3.2). Para evitar que varias hebras ejecuten simultáneamente el GP, se ha creado una sección crítica para las fases de *Evaluación* y *Notificación*, lo que garantiza que únicamente sean ejecutadas por la primera hebra que detecte la finalización de la fase de *Información*. Para minimizar el coste adicional producido por el GP, debemos eliminar el posible retardo que pudiera provocar esta sección crítica. Por este motivo, esta sección crítica se ejecuta con caracter no bloqueante, de tal forma que solo exista una hebra ejecutando las fases de *Evaluación* y *Notificación* del GP. Cuando una de las hebras ejecuta la fase de *Evaluación*, simplemente analizará el criterio de decisión seleccionado, que en este caso, consiste en comprobar que el número de hebras no supere el número de unidades de procesamiento del sistema (línea 11 del Algoritmo 2.3.2). Posteriormente, en el caso de detectar alguna unidad de procesamiento desocupada, se informará en la fase de *Notificación* a la hebra con mayor carga de trabajo para que cree una nueva hebra, y pueda así dividir su trabajo pendiente, generando una nueva hebra con parte de ese trabajo (línea 14 del Algoritmo 2.3.2). Sin embargo, si por el contrario existiera alguna hebra detenida, no se creará ninguna hebra nueva, sino que se reactivará la hebra que más tiempo lleve dormida (línea 16 del Algoritmo 2.3.2).

Algoritmo 2.3.2 : Fase de *Evaluación* del GP

```
(1)  var new_thread, sleep_thread, activate_thread;          Flags de las decisiones del GP
(2)  var num_threads;                                        Número total de hebras
(3)  var num_sleep_threads;                                  Número de hebras detenidas por el GP
(4)  funct check_decision_role(this_thread, num_proc)
(5)    var final_decision, num_active_threads;
(6)    final_decision = nothing;
(7)    if (!new_thread ∧ !sleep_thread ∧ !activate_thread)
(8)      if (all_threads_report())
(9)        begin_critical_section(no_blocking);              Comienza la sección crítica
(10)       num_active_threads = num_threads − num_sleep_threads;   Criterio de decisión
(11)       if (num_active_threads < num_proc)
(12)         if (!num_sleep_threads)
(13)           if (this_thread == thread_work_loadest())
(14)             final_decision = new_thread;                Crear una hebra nueva
(15)         else
(16)             final_decision = activate_thread;           Reactivar una hebra detenida
(17)       if (num_active_threads > num_proc)
(18)         final_decision = sleep_thread;                  Detener una hebra
(19)       reset_last_interval();                            Fase de Reestablecimiento
(20)       end_critical_section(no_blocking);                Termina la sección crítica
(21)   return final_decision
```

Adicionalmente, existe el riesgo de que en el siguiente intervalo de ejecución, el gestor reconsidere la creación de nuevas hebras, por ejemplo, debido a un cambio en las condiciones de ejecución, o una disminución del rendimiento instantaneo de la aplicación, o por irregularidades en la detección de las hebras, etc. En estos supuestos, el GP detectará que el número de hebras creado por la aplicación no es el más eficiente, por lo que informará a una de las hebras para que detenga su ejecución (línea 18 del Algoritmo 2.3.2).

En cualquier caso, independientemente de la decisión tomada, se ejecuta la fase de *Reestablecimiento*, reinicializando los contadores de carga de trabajo realizadas por cada hebra (línea 19 del Algoritmo 2.3.2). Durante la fase de *Notificación*, el GP informa a alguna de las hebras acerca de la decisión tomada en la fase de *Evaluación*, por ejemplo, si ha decidido crear una nueva hebra (*new_thread*), detenerla (*sleep_thread*) o reactivar una hebra detenida (*activate_thread*). Sin embargo, estas acciones no se realizan de forma inmediata. Por este motivo, es necesario impedir que el GP vuelva a ser ejecutado por otra hebra hasta que se complete la acción anteriormente indicada, comprobando que no exista ningún flag activado (línea 7 del Algoritmo 2.3.2).

Finalmente hay que resaltar que la fase de *Información* del gestor ACW no es tan crítica como en el resto de GPs, debido a que el mecanismo utilizado para informar al GP es el acceso a memoria compartida. Este mecanismo es el más rápido de todos los utilizados en esta tesis. Sin embargo, este mecanismo solo permite informar al GP con

datos procedentes de la aplicación, por lo que, el gestor ACW solo obtiene información de la aplicación.

2.3.1. Fase de *Información* basada en el pseudo-sistema de ficheros

En la fase de *Información*, el GP recopilará toda la información necesaria para analizar el criterio de decisión seleccionado en la fase de *Evaluación*. Por ejemplo, cuando el gestor estima $MaxThreads$ en función del número de unidades de procesamiento, es el usuario quien puede suministrar directamente ese número al GP. Sin embargo, se pueden seleccionar otros criterios de decisión que necesiten información suministrada por el SO. Por ejemplo, el GP puede acceder al pseudo-sistema de ficheros (*/proc*) durante la fase de *Información*, donde el sistema operativo Linux ofrece información estadística sobre todos los procesos en ejecución, así como la información del porcentaje de uso de cada unidad de procesamiento. Este pseudo-sistema de ficheros se usa principalmente como interfaz entre el nivel de usuario y las estructuras de datos que gestiona el kernel. La mayor parte de este sistema de ficheros es de sólo lectura, aunque algunos ficheros permiten modificar los valores de algunas variables del núcleo. Dentro de la jerarquía de ficheros del */proc*, existe un subdirectorio *[pid]* para cada hebra en ejecución, donde el nombre del subdirectorio es el ID del proceso o hebra. El gestor AST (Application decides based on Sleeping Threads) es un GP a nivel de usuario basado en lecturas de estos pseudo-ficheros.

De toda la información suministrada por el sistema operativo a través del fichero */proc/stat*, podemos destacar el número total de unidades de procesamiento del sistema, lo que permite establecer este valor si el usuario no lo ha especificado anteriormente. Sin embargo, otra información significativamente más relevante en sistemas no dedicados, es el número de unidades de procesamiento ociosas. Para detectar la existencia de unidades de procesamiento ociosas, desde el nivel de usuario, hay que leer la cuarta columna del fichero */proc/stat*. Esta columna en la fila i almacena el número de jiffies[1] que la unidad de procesamiento i ha estado ociosa desde el inicio del sistema. Si el GP detecta una variación en el valor de esa columna, respecto de la anterior lectura, entonces puede considerarse que la unidad de procesamiento ha estado ociosa.

Otra información interesante, que se verá con más detalle en el siguiente capítulo, es la relativa al retardo de la aplicación debido por ejemplo a las secciones críticas, aunque en las últimas versiones de Linux ninguno de los pseudo-ficheros del */proc* proporciona información explícita sobre el tiempo de espera de las distintas hebras de la aplicación. Sin embargo, se puede conocer el estado de cada una de las hebras leyendo el campo *status* del fichero $/proc/[pid_{proc}]/task/[pid_i]/stat$, donde $[pid_{proc}]$ es el pid del proceso principal de la aplicación multihebrada y $[pid_i]$ es el pid de la hebra i. No obstante, la información que se obtiene es la del estado de cada hebra en el instante en que se realizó la lectura del fichero, por lo que no se sabe con precisión qué ha ocurrido entre una lectura y otra.

Hay que tener en cuenta que el sistema operativo actualiza la información de estos ficheros cada vez que detecta una lectura del */proc* desde el nivel de usuario. Si se realizan

[1] Variable del sistema operativo que contabiliza el número de ticks de reloj transcurridos desde que se arrancó el sistema. Cada periodo de tiempo, generalmente 10 ns, se produce un tick de reloj que provoca una interrupción para ejecutar el manejador de interrupciones del reloj.

muchas lecturas consecutivas del mismo fichero, o de diferentes ficheros, el tiempo que consume el sistema operativo para actualizar la información aumenta, por lo que esta forma de obtener información sobre la aplicación y el sistema puede llegar a consumir muchos recursos computacionales. Por lo tanto, se debe concluir que la fase de *Información* del gestor *AST* permite al GP recopilar información de la aplicación utilizando memoria compartida e información del SO mediante la lectura de pseudo-ficheros. Sin embargo, la parte más crítica de esta fase es la lectura de pseudo-ficheros, ya que puede repercutir en una adaptación lenta de la aplicación multihebrada. Esta lentitud se acentúa cuando aumenta el número de hebras activas y/o las unidades de procesamiento, debido a que este gestor tiene que leer un fichero *stat* por cada hebra activa. Además, hay que leer tantas filas del fichero */proc/stat* como unidades de procesamiento existan en el sistema. Así que el gestor *AST* sería inviable en los multicore del futuro, donde se prevén sistemas con varias decenas de unidades de procesamiento [31].

La Figura 2.3 muestra donde se ejecutan las distintas fases del GP a nivel de usuario. Básicamente, las distintas hebras activas de la aplicación son responsables de ejecutar todas las fases del gestor a nivel de usuario, de tal forma, que la fase de *Evaluación* del GP es ejecutada por una cualquiera de las hebras activas en cada instante de tiempo. Sin embargo, para extender este GP a un entorno no dedicado, donde el GP debe gestionar varias aplicaciones simultaneamente, es necesario plantear diferentes alternativas:

1. *GP centralizado*: En este caso, el GP estará centralizado en una aplicación independiente, encargada de gestionar todas las hebras de las distintas aplicaciones en ejecución. Puesto que el gestor debe conocer la carga de trabajo de cada hebra, cada una de las aplicaciones debe proporcionar información de todas sus hebras al GP, a través de mensajes.

2. *GP distribuido*: En este caso, cada aplicación multihebrada ejecuta su propio GP localmente, de tal forma, que los distintos gestores estén interconectados permanentemente, a través de mensajes.

Ambos tipos de gestores, en la fase de *Información*, necesitan utilizar alguno de los mecanismos de comunicación entre procesos (IPC) disponibles en el SO [29]. Además de la memoria compartida, el SO Linux ofrece varios mecanismos de comunicación tales como señales, tuberías, paso de mensajes, sockets y semáforos. Las aplicaciones mutihebradas pueden utilizar cualquiera de estos mecanismos ejecutando llamadas al sistema, directamente desde el código fuente de la aplicación, o a través de librerías. También se ha implementado otro GP a nivel de usuario basado en sockets, pero se ha descartado su utilización en sistemas *CMP*, debido a que la comunicación usando memoria compartida es más rápida. Así que la comunicación basada en sockets o paso de mensajes son más convenientes en sistemas distribuidos. El diseño e implementación de los GPs basados en sockets, paso de mensajes o señales están fuera de los objetivos de esta tesis, dado que el intercambio de información mediante memoria compartida usado por los gestores *ACW* y *AST* ofrece un comportamiento más eficiente.

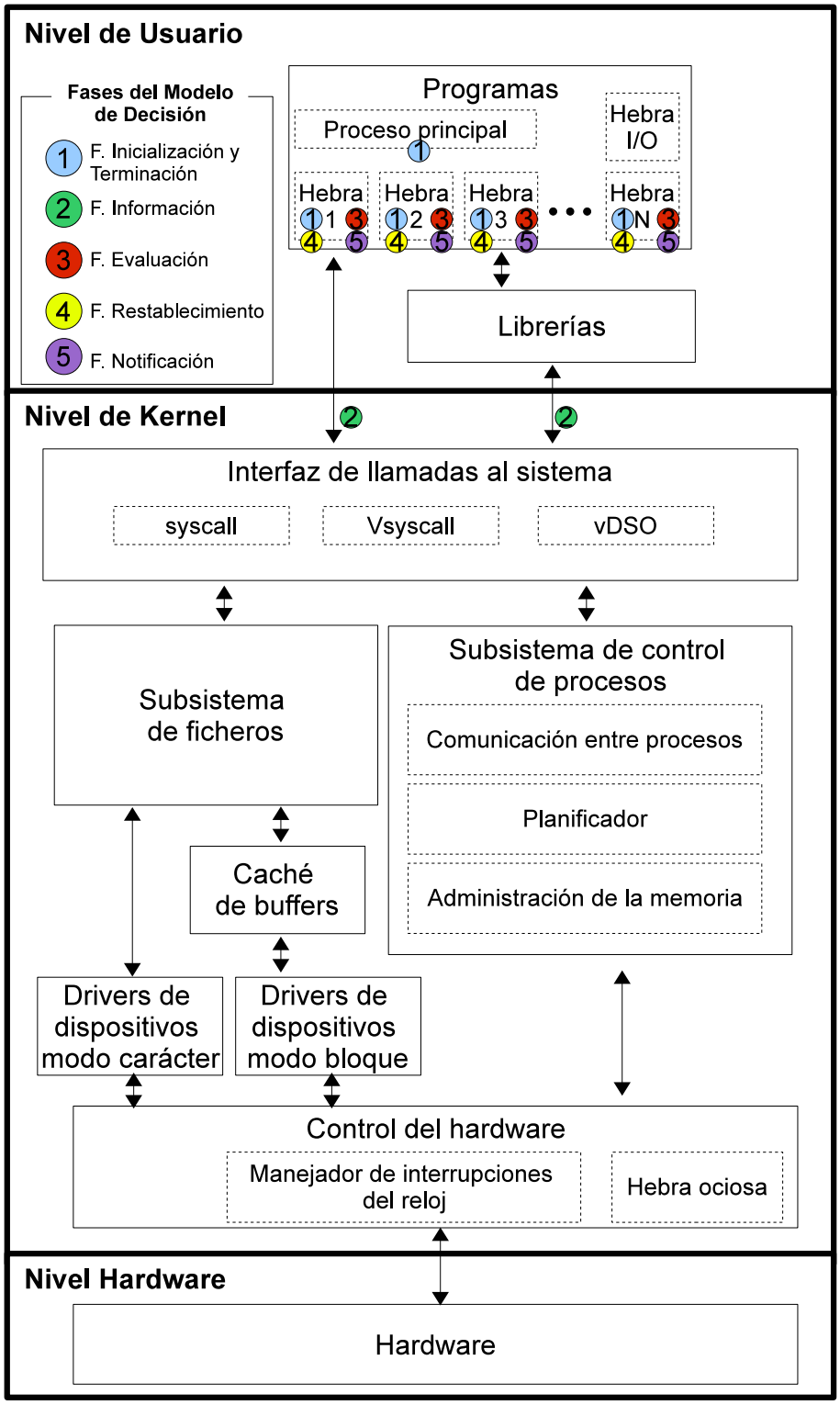


Figura 2.3: Ejecución del gestor del nivel de paralelismo a nivel de usuario.

2.4. GP a nivel de kernel

Una de las principales dificultades para diseñar un GP consiste en seleccionar un criterio de decisión que garantice un nivel de eficiencia adecuado. Esto es aún más cierto en los gestores a nivel de usuario, por la dificultad de recopilar información para implementar algunos criterios de decisión. Existen otros criterios de decisión, que se tratan en el Capítulo 3, que requieren de información no disponible a nivel de la aplicación, ni proporcionada por el sistema operativo de forma estándar a través de los pseudo-ficheros del directorio */proc*. Por este motivo, en esta sección se estudia la posibilidad de modificar el Kernel del sistema operativo. Una sencilla modificación del Kernel puede plantearse creando un único pseudo-fichero por aplicación en el que se detallen todos los parámetros utilizados por el criterio de decisión seleccionado, de tal forma, que el Kernel actualice toda la información periódicamente. De esta forma, se consigue reducir el número de ficheros que necesita leer el GP a nivel de usuario. Sin embargo, esta opción no elimina el coste asociado a la escritura/lectura del fichero.

En este sentido, se plantea el diseño de un GP distribuido entre el nivel de usuario y el nivel del kernel, donde las fases de *Información*, *Evaluación* y *Restablecimiento* se trasladan al nivel del kernel. Este tipo de GP permite obtener información sobre la eficiencia de la aplicación multihebrada de forma más rápida y precisa. En sistemas no dedicados, cada aplicación gestionada por el GP puede consultar periódicamente al SO para establecer su grado de paralelismo en el sistema. Sin embargo, este tipo de gestores requieren modificar el código fuente del SO.

En esta sección se plantean diferentes alternativas para implementar un GP a nivel de Kernel. En la subsección 2.4.1 se describe la integración del GP como un módulo del kernel, lo que requiere recompilar el módulo del kernel desarrollado y su instalación en el SO, sin necesidad de compilar todo el kernel. Posteriormente, en la Sección 2.4.2 se presenta el GP integrado completamente en el núcleo del SO, de tal forma que se ha modificado el kernel para acomodar el código del GP en el manejador de interrupciones del reloj o en la hebra ociosa del kernel, lo que requerirá la compilación e instalación de todo el kernel.

2.4.1. Módulo del Kernel

En este caso, la interacción entre la aplicación multihebrada y el GP se realiza mediante una llamada al sistema, de tal forma que, cada una de las hebras gestionadas pueda ejecutar periódicamente la llamada al sistema para informar del trabajo realizado y conocer la decisión adoptada por el GP. El gestor KST es un GP a nivel de kernel basado en llamadas al sistema. Las llamadas al sistema son rutinas que se cargan en memoria durante el arranque del SO y permiten a las aplicaciones que se ejecutan a nivel de usuario acceder a la información almacenada a nivel del kernel. Una de las ventajas de esta versión es que no requiere de una profunda modificación del kernel.

Originalmente, una llamada al sistema en Linux era un proceso costoso, pues se implementaba como una interrupción del sistema operativo (*int 0x80*). La ejecución de una llamada al sistema se realizaba vía la instrucción *int 0x80*, de tal forma, que la unidad de procesamiento (PU) pasaba el control al SO, el cual detectaba el índice de la llama-

da al sistema a ejecutar. Las interrupciones fuerzan a la PU a salvar el estado de las instrucciones en ejecución, y una vez finalizada la ejecución de la llamada al sistema, se restaura el estado anterior a la interrupción del sistema. Por este motivo, las interrupciones son relativamente costosas en tiempo de ejecución. Sin embargo, los fabricantes de procesadores, AMD e Intel, conscientes de la penalización producida por las interrupciones, han implementado en los procesadores más recientes, instrucciones propias que acelerán la ejecución de las llamadas al sistema. Las aplicaciones pueden ejecutar las instrucciones SYSCALL/SYSENTER y SYSRET/SYSEXIT que actúan de forma más rápida que la interrupción *int 0x80*. Estas instrucciones de llamada al SO están implementadas a partir de la versión 2.5 de Linux.

La Figura 2.4 muestra como se distribuye la ejecución de las distintas fases del gestor a nivel de kernel basado en llamadas al sistema, donde las fases de *Evaluación* y *Restablecimiento* se encuentran ahora a nivel de kernel (módulo del kernel asociado a la llamada al sistema). Mientras que la fase de *Información* se distribuye entre el nivel de usuario y el nivel de kernel, de tal forma que permite al GP recopilar información tanto desde la aplicación como desde el SO. Las fases de *Inicialización*, *Terminación* y *Notificación* son las únicas fases que se mantienen a nivel de usuario. Las fases de *Información* y *Notificación* se ejecutan directamente o a través de librerias, dado que la llamada al sistema se puede realizar directamente en el código de la aplicación o a través de la librería *IGP* (*Interfaz con el Gestor del nivel de Paralelismo*) descrita en la Sección 2.6.

Algoritmo 2.4.1 : GP basado en llamadas al sistema

```
(1) var current;                                              Variable declarada en el kernel
(2) funct syscall(IGP_get, work_load, num_proc)
(3)   var final_decision;
(4)   final_decision = nothing;                               Decisión por defecto
(5)   if (!check_state_process())
(6)     update_loads_threads(current, work_load);             Fase de Información
(7)     final_decision = check_decision_role(current, num_proc);   Algoritmo 2.3.1
(8)   return final_decision
```

El Algoritmo 2.4.1 muestra la implementación del GP en un módulo del kernel, de tal forma que, cada una de las hebras ejecuta, en cada iteración del bucle computacional, la llamada al sistema $IGP\_get(work\_load, num\_proc)$. Esta llamada al sistema básicamente informa al GP de la carga de trabajo $work\_load$ de cada hebra, aunque opcionalmente también se permite al usuario informar del número de procesadores num_proc del sistema. Esta función devuelve la decisión tomada por el GP $final\_decision$. Como se puede observar la decisión por defecto del gestor es no hacer nada para evitar un bloqueo del SO (línea 4), en el caso de un incorrecto funcionamiento de alguna de las hebras de la aplicación. Además, se chequea periódicamente el estado de todas las hebras (línea 5), evitando así que alguna hebra en estado zombie manipule erróneamente al GP. Una vez que se ha comprobado que todas las hebras están funcionando correctamente, se actualiza la carga de trabajo actual a partir de la carga de trabajo informada por las hebras en la llamada

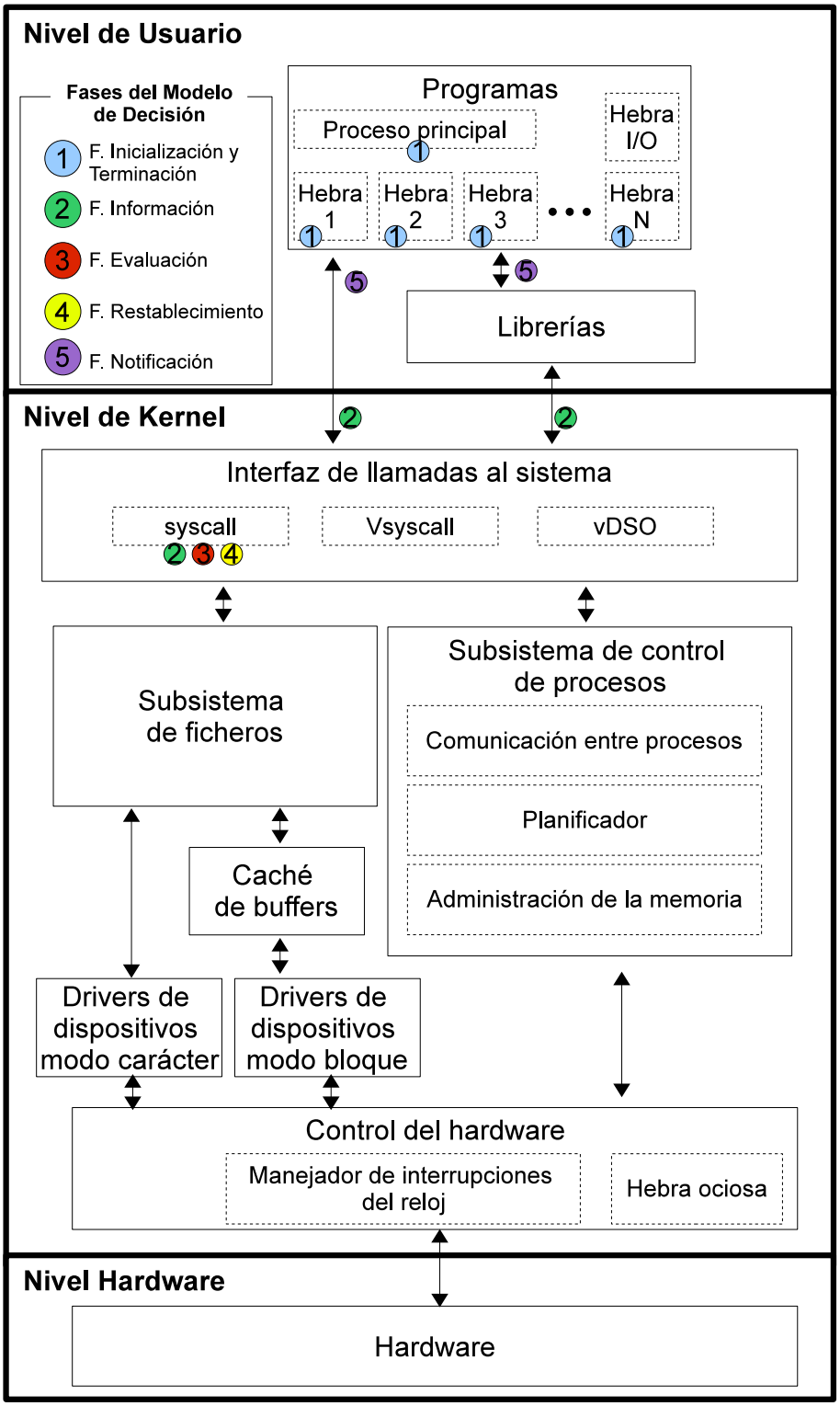


Figura 2.4: Ejecución de las fases del GP a nivel de kernel basado en llamadas al sistema.

al sistema (línea 6) y ejecuta la fase de *Evaluación* del GP, cuyo código fuente coincide con el descrito en el Algoritmo 2.3.1 del GP a nivel de usuario (línea 7). Aunque todas las hebras pueden informar de su carga de trabajo a través de una llamada al sistema en el mismo instante de tiempo, la fase de *Evaluación* se ejecuta secuencialmente por las hebras, pues está integrada en una sección crítica no bloqueante, de tal forma, que solo una hebra puede ejecutar la fase de *Evaluación* en un momento dado. La característica principal de este gestor es que esta llamada al sistema se ejecuta en un módulo a nivel de kernel, lo que le permite tener acceso a toda la información de todos los procesos, hebras y unidades de procesamiento del sistema. Por este motivo, el GP a nivel de kernel puede implementar cualquier criterio de decisión que haga uso de esta información en la fase de *Evaluación*.

2.4.2. Núcleo del Kernel

El inconveniente principal que presenta el GP basado en la llamada al sistema (KST) es el hecho de que las hebras de la aplicación ejecutan periódicamente el GP mediante llamadas al sistema, exactamente igual que cuando trabajamos con un GP a nivel de usuario, salvo que en este caso, la ejecución se realiza con el nivel de privilegios propios del Kernel. Este hecho repercute en la necesidad de detener la ejecución de la aplicación para poder ejecutar el GP, lo que se traduce en una interferencia del gestor con la ejecución de la aplicación. Aunque el retardo de las interrupciones de las llamadas al sistema se ha minimizado considerablemente con la implementación de las instrucciones SYSCALL/SYSENTRY y SYSRET/SYSEXIT en los procesadores AMD e Intel, el coste de una llamada al sistema depende tanto de la rutina que tenga asociada, como del mecanismo utilizado para el intercambio de información entre el nivel de kernel y la aplicación. Linux divide la memoria en dos segmentos: espacio de usuario y espacio de kernel. La memoria del espacio de usuario se usa para la ejecución de las aplicaciones, mientras que la memoria del espacio de kernel se usa para realizar los servicios del sistema operativo. Esta división actúa como una barrera de seguridad, de tal forma que las aplicaciones maliciosas no puedan acceder directamente a la memoria de kernel. En este sentido, las llamadas al sistema actúan de puente para pasar información entre ambos tipos de memoria. Este tránsito entre ambas memorias provoca una sobrecarga añadida, debido a que el direccionamiento de memoria requiere intercambiar algunos registros, lo que le añade más sobrecarga a todo el proceso de la llamada al sistema [70].

Manejador de interrupciones del reloj

Ahora se plantea un GP integrado completamente en el Kernel, de tal forma que algunas fases del GP no se ejecutan en la rutina asociada a la llamada al sistema, lo que permitirá reducir el tiempo de ejecución del gestor, y por lo tanto, su impacto sobre la aplicación. Esta modificación se basa en el uso del manejador de interrupciones del reloj, de la siguiente forma:

En cada tick del reloj del sistema, se ejecuta el manejador de interrupciones del reloj, implementado en Linux por la función *scheduler_tick*() que se muestra en el Algoritmo

Algoritmo 2.4.2 : GP integrado en el manejador de interrupciones de reloj.

```
(1)  var current, num_proc;                                   Variables declaradas en el kernel
(2)  funct scheduler_tick()
(3)    var cpu, rq;
(4)      cpu = smp_processor_id();                            Identifica al procesador.
(5)      rq = cpu_rq(cpu);                                    Identifica al cola de tareas.
(6)    sched_clock_tick();                                    Obtener tiempo del reloj
(7)    spin_lock(rq- > lock);                                 Bloquear cola de tareas
(8)    update_rq_clock(rq);
(9)    update_cpu_load(rq);
(10)   curr- > sched_class- > task_tick(rq, current, 0);
(11)   if (is_IGP(current))
(12)     if (!check_state_process())
(13)       copy_from_user(current- > work_load);              Fase de Información
(14)       check_decision_role(current, num_proc);            Algoritmo 2.3.1
(15)   spin_unlock(rq- > lock);                               Desbloquear cola de tareas
(16)   rq- > idle_at_tick = idle_cpu(cpu);
(17)   trigger_load_balance(rq, cpu);
```

2.4.2. Principalmente, esta función se encarga de incrementar el contador del reloj (línea 6), necesario para actualizar las estadísticas tanto de los procesos en la cola de tareas (línea 8), como de la unidad de procesamiento (línea 9), así como decrementar el quantum de tiempo asignado al proceso en ejecución (línea 10). Por otro lado, las líneas 7 y 15 permiten bloquear y desbloquear, respectivamente, el acceso a la cola de tareas asignada al procesador para mantener la coherencia entre los datos tras la gestión realizada. Finalmente, se comprueba la necesidad, o no, de replanificar la tarea actualmente en ejecución entre las distintas unidades de procesamiento (línea 17).

La función *scheduler_tick*() se ejecuta a nivel de kernel, pero tiene la peculiaridad de ejecutarse periódicamente, con un periodo por defecto de 1 ms, tras una interrupción hardware del reloj. Este tipo de interrupciones tienen la máxima prioridad, es decir, se pueden producir en cualquier momento independientemente de lo que esté haciendo la unidad de procesamiento. Esto significa que el procesador puede estar ejecutando tanto una tarea de usuario, como tareas própias del SO.

En esta versión más avanzada del GP a nivel de Kernel, también denominada SST (*Scheduler decides based on Sleeping Threads*), se propone trasladar las fases de *Información* y *Evaluación* desde el módulo de la llamada de sistema, al manejador de interrupciones del reloj del sistema. En el Algoritmo 2.4.2 se muestra, resaltado en negrita, el código del gestor integrado en el Kernel, basado principalmente en detectar si la tarea que ejecuta la función *scheduler_tick*() es una hebra gestionada por el GP (línea 11), así como en comprobar que todas las hebras de la aplicación funcionan correctamente (línea 12), lo que permite eliminar el riesgo de que caiga el SO. Una vez realizadas estas comprobaciones, la hebra ejecuta, a nivel de kernel, la fase de *Información* (línea 13) y la fase de *Evaluación*

comentadas anteriormente en el Algoritmo 2.3.1.

La fase de *Información* es la más crítica de cualquier GP debido a que es la fase que más tiempo tarda en ser completada. El gestor KST hace uso de la llamada al sistema para recibir información sobre la carga de trabajo de cada hebra. Sin embargo, el GP a nivel de kernel SST no utiliza la llamada al sistema, por lo que se debe establecer otra metodología en la fase de *Información*. La fase de *Información* del gestor SST se realiza a través del acceso directo a la memoria de la aplicación gracias a los privilegios del Kernel para leer (línea 13) y escribir información en el espacio de memoria de las aplicaciones en ejecución. Esta metodología permite al GP acceder directamente a la información almacenada en cada una de las variables (tales como, *work_load*) de las distintas hebras activas. Otra ventaja que ofrece esta versión del GP, es la alta periodicidad con que se ejecuta la fase de *Información* debido a que el manejador de interrupciones de reloj se ejecuta en cada tick, lo que se traduce en que el gestor esté permanentemente informado de las medidas de tiempos almacenadas en las estructuras de datos que gestionan las estadísticas de las colas de tareas, procesos, y unidades de procesamiento del SO.

Finalmente, aunque la fase de *Notificación* también puede ser implementada usando la metodología del acceso directo a la memoria de la aplicación, empleado en la fase de *Información*, se ha optado por usar otra metodología basada en vsyscall y vDSO (virtual Dynamic Share Object) disponibles en las últimas versiones del SO Linux [72]. Esta metodología no se puede utilizar en la fase de *Información*, pues no permite el transito de información del nivel de usuario al nivel de Kernel por motivos de seguridad, aunque si es posible el transito en sentido inverso.

Las vsyscall y vDSO son llamadas virtuales al sistema que permiten leer pero no escribir información en el espacio del kernel del proceso que realiza la llamada al sistema, por lo que se reduce la sobrecarga producida en el salto de direccionamiento de memoria entre el espacio de usuario y el espacio de kernel característico de las llamadas al sistema. Además, de esta forma, una página de memoria asignada a un proceso de usuario puede contener un subconjunto de llamadas virtuales al sistema que se ejecutan de forma segura desde el espacio de usuario, sin causar ningún agujero de seguridad en el kernel. Esta página de memoria es mapeada por el kernel durante la creación del proceso en el espacio de usuario de cada tarea en ejecución. De esta forma, cuando se realiza una llamada al sistema, no se produce ningún cambio de contexto entre las regiones de memoria del usuario y el espacio del kernel, por lo que se reduce la sobrecarga de la llamada al sistema. Esto permite reducir considerablemente el tiempo de ejecución de aquellas aplicaciones que utilizan constantemente este tipo de funciones.

Una vsyscall realmente es lo mismo que un vDSO, pues ambas están mapeadas por el Kernel, aunque vsyscall y vDSO trabajan de forma muy similar, tienen algunas diferencias. Una vsyscall está limitada a un máximo de cuatro entradas, y su posición en memoria es estática. Así que cualquier aplicación puede enlazar directamente a la dirección donde están alojadas las vsyscalls. Por otro lado, una vDSO se carga dinámicamente en el proceso del usuario, así que no es predecible la dirección de memoria asignada a la vDSO, por lo que la distribución del espacio de memoria a las distintas vDSOs es aleatoria. Por este motivo, se recomienda utilizar las vsyscalls, pero si la aplicación necesita más de 4 vsyscalls, entonces se deben utilizar las vDSOs.

Algoritmo 2.4.3 : Fase de notificación del GP a nivel de kernel a través de vDSO.

```
(1) var new_thread, sleep_thread;
(2) funct vdso_IGP_get(this_thread)
(3)   if (this_thread == new_thread)
(4)     return divide                          Crear una hebra nueva
(5)   if (this_thread == sleep_thread)
(6)     return sleep                           Dormir esta hebra
(7)   return no_new_thread
```

El Algoritmo 2.4.3 muestra como se ha integrado la fase de *Notificación* del GP a nivel de kernel declarando la vDSO llamada *vdso_IGP_get*(*this_thread*). El GP declara las variables internas del kernel *new_thread* y *sleep_thread* donde almacenará el identificador de aquella hebra, para la que en la fase de *Evaluación* del GP se había decidido repartir su carga de trabajo con una nueva hebra o parar una hebra, respectivamente. Por lo tanto, tras la fase de *Evaluación*, donde se toma la decisión de crear o no una nueva hebra, el GP identifica en *new_thread* a la hebra que debe crear una nueva hebra, y en *sleep_thread* a la hebra que debe detener su ejecución. En la llamada al sistema *vdso_IGP_get*(*this_thread*) se compara el identificador de la hebra (*this_thread*) que ejecuta la vDSO con la decisión tomada por el gestor (*new_thread* o *sleep_thread*). De tal forma que en caso de coincidencia la hebra tendrá que crear una nueva hebra o dormir.

Hebra ociosa del Kernel

La hebra ociosa del kernel es una tarea con la prioridad más baja, que siempre está en la cola de ejecución, y únicamente entra en ejecución cuando no existe ningún otro proceso pendiente de ejecución en una unidad de procesamiento. El objetivo principal de la hebra ociosa del kernel es ejecutar reiteradamente la instrucción en lenguaje ensamblador *hlt* con las interrupciones habilitadas. La instrucción *hlt* (*Halt the system*) es la única instrucción ejecutada por el procesador en estado ocioso, la cual se suele ejecutar con las interrupciones habilitadas, para permitir a dispositivos hardware, tales como el reloj, interrumpir la ejecución de la instrucción *hlt* a intervalos regulares. Estas interrupciones permiten chequear periódicamente la cola de ejecución de la unidad de procesamiento.

Esta última versión del GP a nivel de kernel, continua ejecutando la fase de *Información* en el manejador de interrupciones del reloj, sin embargo, propone que la fase de *Evaluación* sea ejecutada por la hebra ociosa del kernel, de tal forma que únicamente se ejecuta cuando alguna unidad de procesamiento está ociosa. De esta forma el GP no consume recursos cuando todas las unidades de procesamiento están a pleno rendimiento, impidiendo sobrecargar más al sistema en momentos puntuales de máxima productividad.

La Figura 2.5 muestra un esquema de la integración del gestor a nivel de kernel basado en la hebra ociosa en la estructura interna del sistema operativo Linux. Como se puede observar la Fase de *Información* recopila datos de distintas fuentes. Por un lado, los datos procedentes de la aplicación a través de la llamada al sistema, utilizando o no la librería

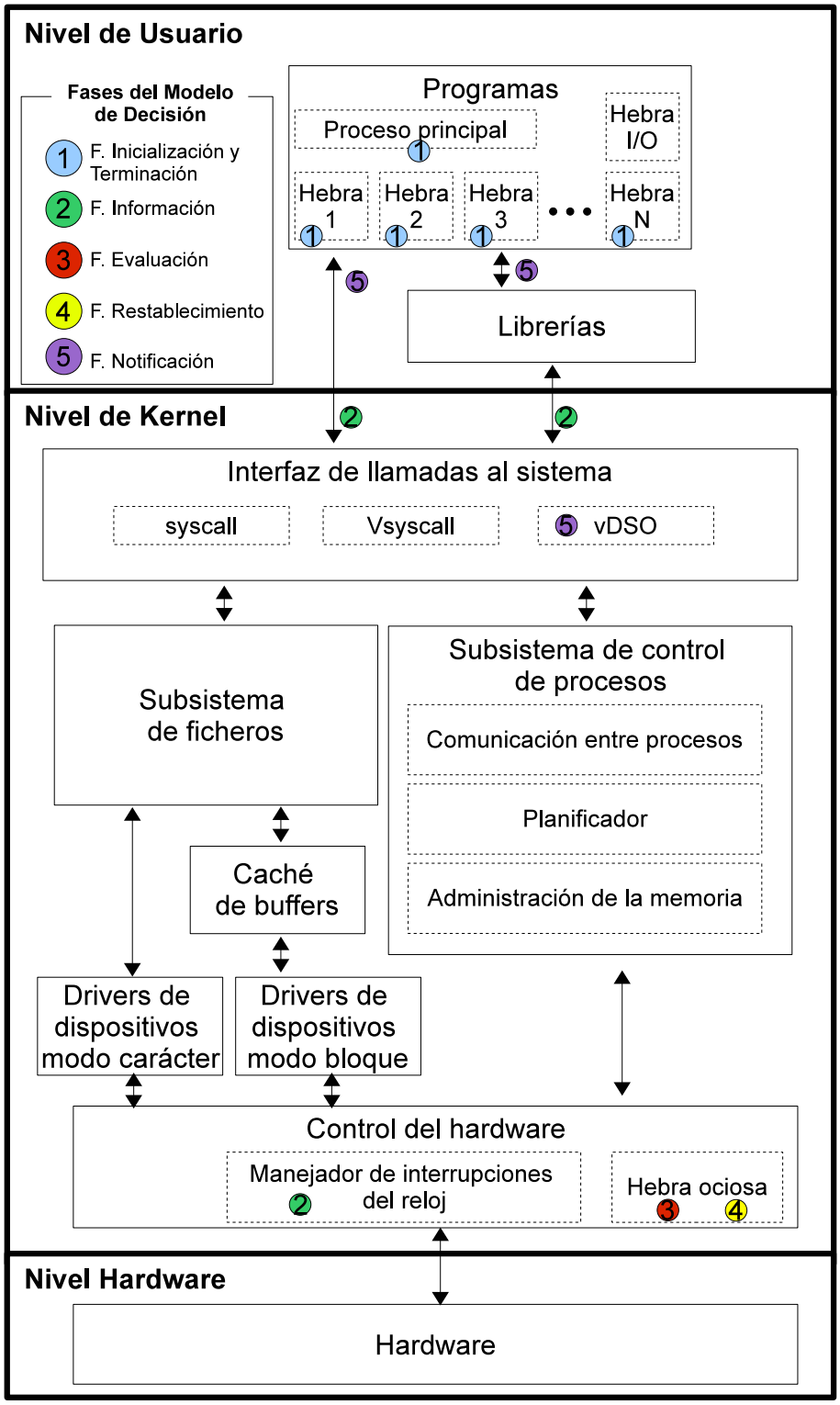


Figura 2.5: Ejecución de las fases del GP a nivel de kernel basado en la hebra ociosa.

IGP (ver Sección 2.6), y por otro lado, datos procedentes del SO a través del manejador de interrupciones del reloj. La periodicidad con la que reciben los distintos tipos de información depende del tiempo que tarda cada hebra en ejecutar la llamada al sistema y cada tick del reloj. Cada vez que se detecta una unidad de procesamiento desocupada entra en ejecución una hebra ociosa del kernel, la cual ejecuta la Fase de *Ejecución* dentro de una sección crítica no bloqueante, evitando así que otra hebra ociosa del kernel la ejecute en el mismo instante de tiempo. Finalmente, la fase de *Notificación*, en la cual se ha integrado una vDSO, se realiza a través de la llamada al sistema ejecutada por cada hebra.

2.5. Detención adaptativa de hebras

Hasta este punto nos hemos centrado en uno de los aspectos más importantes de la adaptación de aplicaciones multihebradas, basado en la creación de las hebras. Sin embargo, como ya se ha comentado anteriormente, la creación estática de hebras permite establecer el número inicial de hebras en función del número de unidades de procesamiento disponibles, de la carga de trabajo, y de la experiencia del usuario. El número de hebras en ejecución se mantiene constante hasta que Finalize la carga total de trabajo, sin tener en cuenta las variaciones de rendimiento de la aplicación provocado por la disponibilidad de los recursos y la distribución de la carga de trabajo entre las hebras. Sin embargo, estas variaciones de rendimiento se acentúan aún más cuando la aplicación se ejecuta en entornos no dedicados, donde no solo las hebras de la aplicación multihebrada, sino otras aplicaciones multihebradas o no, compiten también por los recursos del sistema.

Un inconveniente de los gestores desarrollados es que si el número actual de hebras de la aplicación es eficiente, el gestor propone la creación de una nueva hebra, lo que puede dar lugar a que la aplicación deje de ser eficiente. En entornos no dedicados, este problema se puede acentuar más, dado que pueden entrar en ejecución otras aplicaciones en cualquier momento, por lo que la diferencia del número de hebras en ejecución y el número de hebras óptimo establecido por el gestor puede ser superior a 1.

Debido a los motivos anteriores, se debe crear un mecanismo que permita a las aplicaciones multihebradas con creación estática o dinámica de hebras poder adaptarse dinámicamente, tanto a las variaciones de rendimiento producidas por las características intrínsecas de la aplicación, como a los recursos disponibles en entornos no dedicados.

En este sentido, el Gestor de paralelismo debería poder informar de la necesidad de detener alguna de las hebras de la aplicación para mejorar su eficiencia. Por lo tanto, el gestor puede notificar a cualquiera de las hebras gestionadas la posibilidad de: crear un nueva hebra, detener una hebra activa, despertar una hebra previamente dormida o simplemente continuar con su ejecución. Las hebras detenidas permanecerán durmiendo, sin consumir recursos, hasta que se modifiquen las condiciones que provocaron la detención de las hebras. Esta modificación se producirá cada vez que disminuya el número total de tareas activas, es decir, cuando termine la carga de trabajo de alguna de las hebras en ejecución, o cuando alguna aplicación termine su ejecución.

Un aspecto importante a destacar es que el GP no utilizará en su fase de *Evaluación* la información estadística relacionada con las hebras detenidas por el gestor. El GP diferencia

los motivos por los que una hebra ha estado inactiva, analizando la información de las hebras detenidas por motivos propios de la aplicación/sistema (por ejemplo, interbloqueo entre hebras, colas de tareas, unidades de procesamiento, ...), y descarta la información de aquellas hebras que han estado detenidas por el GP en actual intervalo de evaluación.

El GP permitirá la ejecución de una hebra detenida, en vez de crear una nueva hebra, siempre que el gestor confirme tal posibilidad. El principal problema que se plantea en la detención de hebras es que una hebra esté detenida durante mucho tiempo. Existen aplicaciones HPC donde la detención prolongada de la misma hebra no tiene porque repercutir negativamente en el tiempo total de ejecución. Por ejemplo, las aplicaciones basadas en el algoritmo *ray tracing*, donde cada hebra tiene asignada el procesamiento de una parte de la imagen. En este tipo de algoritmos, la detención de la misma hebra no provoca el bloqueo del resto de hebras, debido a la independencia entre las cargas de trabajo de las distintas hebras. Sin embargo, este problema es especialmente relevante en aplicaciones multihebradas basadas en algoritmos de ramificación y acotación donde el tiempo total de ejecución se puede ampliar considerablemente si se detiene la misma hebra de manera prolongada, debido a que el árbol de búsqueda se distribuye entre las distintas hebras activas. Si la solución buscada se encuentra en una rama del árbol asignada a una hebra detenida, el tiempo de computación aumenta considerablemente. Por este motivo, el GP implementa un mecanismo de detención rotativo, de tal forma, que la detención de una misma hebra se realice durante un intervalo pequeño de tiempo. Una vez que ha transcurrido ese intervalo, la hebra detenida entra en ejecución y se selecciona otra hebra para ser detenida en el siguiente intervalo de tiempo. El criterio para seleccionar la hebra a detener, de entre todas las hebras activas, establece que la hebra a detener será aquella hebra que más tiempo ha estado ejecutándose desde su última detención, garantizando así un reparto ecuánime del periodo de detención entre todas las hebras. El mecanismo utilizado por el GP para detener hebras depende del tipo de gestor implementado, es decir, GP a nivel de usuario o GP a nivel de kernel:

1. Detención a nivel de usuario: La función $decision = IGP\_get(work\_load_i)$ ejecutada por las hebras de la aplicación multihebrada en las fases de *Información* y *Notificación* del gestor de hebras, utiliza la función $Sleep(interval\_time)$ para implementar la detención de aquella hebra seleccionada por el GP, cuando así lo establezca. Dependiendo del valor asignado $interval\_time$, la función $Sleep()$ hace uso de los comandos $sleep(s)$, $usleep(us)$ o $nanosleep(ns)$, según se trate de segundos, microsegudos o nanosegundos, respectivamente. Este mecanismo propone asignar un intervalo de detención $interval\_time$ constante definido en la librería IGP.

2. Detención a nivel de kernel: El planificador del sistema operativo Linux establece varias colas de tareas: Una cola *runqueue* con tareas preparadas para su ejecución en cada una de las unidades de procesamiento disponibles, y otras colas *waitqueues* para tareas detenidas. El GP a nivel de Kernel debe ampliar su funcionalidad en la fase de *Notificación* ya que debe notificar la decisión tomada en la fase de *Evaluación* al planificador del SO. Por este motivo, el gestor integra un código adicional en la fase de *Notificación* dentro del planificador del sistema operativo. Este código se encarga de seleccionar el procesador más sobrecargado y extraer la hebra seleccionada de la

cola *runqueue* antes de que sea ejecutada por la unidad de procesamiento. El GP es el responsable de gestionar el tiempo que la hebra detenida permanece en *waitqueues* antes de pasarla a la cola *runqueue*. El intervalo de detención será igual al quantum por defecto, establecido por el planificador del sistema operativo.

Independientemente del mecanismo escogido, cuando expira el intervalo de detención de una hebra, el GP reactiva la hebra detenida para que pueda entrar en ejecución y establece un mecanismo de detención rotativo por el que se detiene una nueva hebra, siempre que se mantengan las condiciones que provocarón la detención inicial.

El inconveniente principal del mecanismo de detención rotativo a nivel de usuario aparece a la hora de despertar una hebra dormida. Este despertar no es inmediato, si el GP decide reactivar una hebra dormida, el gestor debe esperar que expire el tiempo *interval_time* asignado a la hebra. Esto es debido a que el comando $Sleep(interval\_time)$ no devolverá el control a la hebra hasta que el reloj asociado genere una interrupción. Por este motivo, para minimizar el impacto de este inconveniente, se configura la constante *interval_time* en la librería IGP, con un valor pequeño (por defecto, 10 ms). Por otro lado, el mecanismo de detención rotativo a nivel de kernel no tiene este inconveniente, puesto que el GP puede reasignar la hebra detenida de la cola *waitqueues* a la cola *runqueue* del procesador asignado cuando entre en ejecución el planificador del sistema.

Algoritmo 2.5.1 : Fase de notificación del GP en el planificador del SO.

```
(1)  funct schedule()
(2)    var cpu, rq, prev, next;
(3)    preempt_disable();                          Desactiva la posibilidad de ser interrumpido.
(4)    cpu = smp_processor_id();                   Identifica al procesador.
(5)    rq = cpu_rq(cpu);                           Identifica al cola de tareas.
(6)    prev = rq-> curr;
(7)    spin_lock_irq(rq-> lock);                   Bloquea la cola de tareas
(8)    update_rq_clock(rq);
(9)    if (signal_pending(prev)) deactivate_task(rq, prev);
(10)   else IGP_goto_sleep();                      Integra la detención de hebras en el kernel
(11)   if (unlikely(!rq-> nr_running)) idle_balance(cpu, rq);   Rebalancea la carga
(12)   next = pick_next_task(rq, prev);            Asigna al procesador la tarea con más prioridad
(13)   if (likely(prev! = next))                   ¿cambiar de tarea?
(14)     sched_info_switch(prev, next);            Gestiona el cambio de contexto
(15)     rq-> curr = next;
(16)     context_switch(rq, prev, next);
(17)   spin_unlock_irq(rq-> lock);                 Desbloquea la cola de tareas
(18)   if (unlikely(reacquire_kernel_lock(current) < 0)) goto (7)
(19)   preempt_enable_no_resched();                Activa la posibilidad de ser interrumpido
(20)   if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) goto (3)
```

Cada vez que expira el quantum de tiempo que una tarea está asignada a un pro-

cesador, o cuando la tarea se bloquea por algún motivo (secciones críticas, operaciones entrada/salida, fallos de memoria, etc), entra en ejecución el planificador del SO en ese procesador, ejecutando la función *schedule*() descrita en el Algoritmo 2.5.1. Esta función comienza deshabilitando las interrupciones (línea 3) y en las siguientes líneas, de la 4 a la 6, identifica al procesador que está ejecutando el planificador, carga un puntero a la *runqueue* asignada a ese procesador y localiza la tarea *current* como la tarea que estaba ejecutándose recientemente hasta que entró en ejecución el planificador. A continuación, se bloquea el acceso a la *runqueue* para impedir que se puedan modificar sus datos (línea 7), lo que permitirá actualizar las estadísticas relacionadas con la *runqueue*. Si la tarea *current* tiene que gestionar señales pendientes, entonces se extrae la tarea *current* de la *runqueue* (línea 9). Este es el momento más adecuado para integrar la detención adaptativa del gestor de hebras a través de la función *IGP_goto_sleep*(). Posteriormente, el planificador del SO entrará en un proceso de rebalanceo de tareas entre los distintos procesadores del sistema, si detecta que la *runqueue* del procesador está vacía (línea 11). Una vez que la *runqueue* está actualizada con las tareas asignadas a este procesador, entonces se selecciona la tarea con mayor prioridad, para ser la próxima tarea a ser ejecutada (línea 12). Si se detecta, que esta tarea no es la tarea *current*, se debe realizar un cambio de contexto salvando los datos almacenados en todos los registros del procesador sobre la tarea *current* y cargando los datos correspondientes a la nueva tarea que pasa a ser la nueva tarea *current* (línea 14 al 16). A partir de aquí, se desbloquea la *runqueue* (línea 17), y se comprueba que la nueva tarea *current* es capaz de activar el semáforo de acceso al nivel del kernel, si es incapaz, se vuelve a ejecutar el planificador del SO desde la instrucción a partir de la cual no se permitían las interrupciones (línea 18). Finalmente, se habilitan las interrupciones (línea 19) y se comprueba si se ha activado el flag *TIF_NEED_RESCHED*. En caso afirmativo, se vuelve a ejecutar el planificador del SO desde el principio (línea 20).

El Algoritmo 2.5.2 muestra la función *IGP_goto_sleep*() integrada en el planificador del SO para gestionar la detención y reactivación de hebras a nivel de Kernel. Esta función comienza verificando si la variable IGP_sleep_flag ha sido activada por la aplicación cuando configuró el GP con la posibilidad de detener las hebras a través de la función $IGP\_Initialize(M, C, P, T, Detention)$ (línea 4) [2]. Si la fase de *Evaluación* decide que se debe detener una determinada hebra (línea 5), entonces se verifica si la actual tarea de ejecución es la hebra a detener, y además se comprueba que esta hebra pueda detenerse, antes de proceder a extraer la hebra de la *runqueue* (línea 8). Posteriormente, añade la hebra en una *waitqueue* generada por el GP específicamente para este propósito (línea 10), para finalizar desmarcando la hebra como "detenible", y desactivando la orden de detener una hebra.

En el supuesto de que la decisión del GP sea despertar una hebra previamente dormida (línea 13), entonces, el primer planificador del SO que entre en ejecución en cualquier procesador, buscará una hebra en la *waitqueue* del GP (línea 14), extrae la hebra de la cola de espera (línea 16) y la inserta en la runqueue del procesador actual (línea 19). Posteriormente, se le asigna un turno rotatorio a esta hebra que permita determinar la posición

[2] El código fuente del SO Linux utiliza las funciones *likely()* y *unlikely()* en las instrucciones condicionales (por ejemplo, IF), para que el predictor de salto acelere la ejecución del SO, indicando que opción tiene más probabilidad de verificarse.

Algoritmo 2.5.2 : Implementación de la detención de hebras en un GP a nivel de kernel.

```
(1)  var IGP_sleep_flag, IGP_order_sleep, IGP_order_wakeup, current, IGP_wq;
(2)  funct IGP_goto_sleep()
(3)    var wait, task, rq, cpu;
(4)    if (unlikely(IGP_sleep_flag))                    Verificar detención de hebras habilitada.
(5)      if (unlikely(IGP_order_sleep))                 Detener hebra.
(6)        if (unlikely(current- > IGP_id = sleep_thread))
(7)          if (likely(current- > IGP_sleepable))
(8)            deactivate_task(rq, current);            Extraer hebra de la runqueue.
(9)            DECLARE_WAITQUEUE(wait, current);
(10)           add_wait_queue(IGP_wq, &wait);           Añadir la hebra a la waitqueue.
(11)           current- > IGP_sleepable = false;
(12)           IGP_order_sleep = false;
(13)     if (unlikely(IGP_order_wakeup))                Despertar hebra dormida.
(14)       task = looking_for_task(IGP_wq);
(15)       DECLARE_WAITQUEUE(wait, task);
(16)       remove_wait_queue(IGP_wq, &wait);            Extraer hebra de la waitqueue.
(17)       cpu = smp_processor_id();                    Identifica al procesador.
(18)       rq = cpu_rq(cpu);                            Identifica al cola de tareas.
(19)       activate_task(rq, task);                     Añadir hebra en la runqueue.
(20)       IGP_sleepable = round_robin(task);
(21)       IGP_order_wakeup = false;
```

que ocupa esta hebra en el mecanismo de detención rotativo (línea 20). Finalmente, se desactiva la orden de despertar una hebra especificada por el GP.

2.6. Librería IGP

Se ha diseñado la librería *IGP* (*Interfaz con el Gestor del nivel de Paralelismo*) para facilitar el diseño de aplicaciones multihebradas adaptativas, de tal forma que proporciona funciones (en lenguaje C) para comunicar los algoritmos multihebrados con el GP seleccionado. Esta librería pretende simplificar el trabajo del programador de aplicaciones multihebradas ayudandole en la configuración y manejo del gestor, de tal forma que sea transparente al tipo de gestor utilizado.

En cada una de las fases del GP se hace uso de funciones de la libreria *IGP* diseñada expresamente para establecer una comunicación entre la aplicación y el GP responsable de la gestión de las hebras. A continuación, se detallan cada una de las funciones que puede utilizar el desarrollador de las aplicaciones SPMD multihebradas:

1. Fase de *Inicialización*:

   - $IGP\_Initialize(Method, Criterion, Period, Threshold, Detention)$: Esta función le permite al proceso principal de una aplicación multihebrada solicitar al

GP ser gestionado. Los distintos argumentos de entrada de la función permiten configurar la gestión de la aplicación multihebrada:

a) *Method*: Indica el tipo de GP escogido para la gestión de las hebras. Los valores permitidos oscilan en el rango de 0 a 7, entre los que podríamos destacar el valor 0 para seleccionar el método no adaptable, los valores del 1 al 3 son para gestores a nivel de usuario, por ejemplo, *ACW* o *AST*, y finalmente el rango de 4 a 7 están reservados para gestores a nivel de kernel, tales como *KST*, *KITST* y *SST*. Se han reservado los valores 3 y 7, en el nivel de usuario y kernel respectivamente, para implementaciones futuras.

b) *Criterion*: Permite seleccionar el criterio de decisión que utilizará el GP escogido. Los valores permitidos oscilan en el rango entre 0 y 22, cuyos valores corresponden a diferentes criterios que serán analizados en el capítulo siguiente, tales como, número de procesadores ociosos, tiempo que duermen las hebras, tiempo interrumpible y no interrumpible, etc...

c) *Period*: Establece el intervalo de tiempo que debe transcurrir entre dos ejecuciones consecutivas de la fase de *Evaluación* del GP, por ejemplo, 0.1 segundos.

d) *Threshold*: Establece el umbral mínimo de eficiencia que debe obtener la aplicación multihebrada cuando es gestionada por ACW. Por ejemplo, 0.9 corresponde al 90 % de eficiencia.

e) *Detention*: Este flag permite configurar el gestor de hebras para habilitar o no la detención temporal de las hebras.

- $IGP\_Begin\_Thread(work\_load)$: Esta función permite a cada una de las hebras de la aplicación informar al gestor que se acaba de crear una hebra con la carga de trabajo ($work\_load$), y por lo tanto, está hebra tiene que ser gestionada por el GP. Esta función debe ser ejecutada al inicio de todas las hebras de la aplicación que se desean gestionar.

2. Fases de *Información* y *Notificación*:

   - $new\_thread = IGP\_get(work\_load_i)$: Esta función tiene dos finalidades. Por un lado, permite a las hebras informar al gestor sobre la carga de trabajo realizada ($work\_load_i$) desde la última comunicación (Fase de *Información*). Por otro lado, el gestor informará a la hebra de la decisión alcanzada en la fase de *Evaluación*: dividirse, detenerse o continuar ejecutándose (Fase de *Notificación*). Esta función debe ser ejecutada por cada hebra una vez completada una unidad de trabajo, por ejemplo, en cada iteración del bucle computacional.

3. Fase de *Terminación*:

   - $IGP\_Finalize()$: El proceso principal de la aplicación multihebrada informa al gestor que va a terminar su ejecución y por lo tanto el GP seleccionado dejará de gestionar esta aplicación. Esta función se debe ejecutar una vez que todas las hebras de la aplicación han completado la carga de trabajo asignada.

- *IGP_End_Thread*(): Cada una de las hebras gestionadas debe ejecutar esta función antes de terminar, para indicarle al GP que ha completado toda la carga de trabajo asignada, y por lo tanto, desea no ser gestionada, pues va a terminar su ejecución.

A continuación, se plantean dos ejemplos donde se hace uso de la librería IGP para gestionar una aplicación multihebrada. El primer ejemplo se trata de una aplicación con creación estática de hebras y el segundo ejemplo se trata de una aplicación con creación dinámica de hebras. Ambas estrategias de creación de hebras fueron estudiadas anteriormente en la Sección 1.6.

Algoritmo 2.6.1 : Configuración de un GP en el proceso principal de una aplicación multihebrada con creación estática de hebras

```
(1)  include{IGP_library}
(2)  funct main_static_process(Task, MaxThreads)
(3)      IGP_Initialize(M, C, P, T, true);                    Fase de Inicialización
(4)      global_work_load = memory_allocator(Task);           Reserva memoria
(5)      while (MaxThreads ≠ 0)
(6)          work_load_i = divide(global_work_load, MaxThreads);
(7)          create_thread(static_thread, work_load_i);
(8)          reduce(MaxThreads);
(9)      wait_results_threads();          Esperar a que todas las hebras terminen
(10)     IGP_Finalize();                                      Fase de Terminación
(11)     proccess_results();
```

Algoritmo 2.6.2 : Comunicación de la hebra con un GP

```
(1)  funct static_thread(work_load)
(2)      IGP_Begin_Thread(work_load);                         Fase de Inicialización
(3)      do
(4)          while (work_load ≠ {∅})
(5)              critical_section(global_parameters);
(6)              if (∃idle_thread)
(7)                  half_work_load = divide(work_load);      Reparto de trabajo
(8)                  send_work_load(id_idle_thread, half_work_load);
(9)                  compute_work(work_load_i);           Realizar el trabajo asignado
(10)                 IGP_get(work_load_i);                    Fases de Información
(11)             send_results();
(12)             work_load = get_work_load(busy_thread);      Rebalancear trabajo
(13)     while (global_work_load ≠ {∅})
(14)     IGP_End_Thread();                                    Fase de Terminación
```

Los Algoritmos 2.6.1 y 2.6.2 muestran la utilización de la librería *IGP* para comunicar

la aplicación multihebrada basada en la creación estática de hebras con el GP responsable de gestionar las hebras. Este esquema permite al proceso principal ejecutar la función *main_process*(*task*) encargada de repartir toda la carga de trabajo inicial (*task*) de forma equitativa entre las *num_threads* hebras creadas inicialmente (líneas 5 a 7 del Algoritmo 2.6.1). Sin embargo, antes de la creación de hebras, se configura el GP a través de los parámetros M (método), C (criterio), P (periodo), T (umbral) y detección habilitada (línea 3 del Algoritmo 2.6.1). Por otro lado, una vez comprobado que todas las hebras han terminado su ejecución, la aplicación debe informar al GP para que esta aplicación deje de ser gestionada (línea 10 del Algoritmo 2.6.1).

El Algoritmo 2.6.2 permite a las hebras computacionales inscribirse en la gestión al inicio de la ejecución (línea 2). La función *static_thread*(*work_load*) es ejecutada por cada una de las hebras activas, informando al gestor acerca de la carga de trabajo realizada en cada iteración del bucle (línea 10). Como se puede observar no aparece la Fase de *Notificación* en este esquema, debido a que el gestor no participa en la decisión de creación de hebras, pues en la creación estática de hebras tan solo se tiene en cuenta el número de hebras establecido por el usuario. Esta metodología no permite al GP gestionar la creación de hebras. Sin embargo, el gestor puede utilizar el mecanismo de detención implementado para adaptar la ejecución concurrente de las hebras activas siguiendo las directrices del GP seleccionado. Para ello se debe activar la detención de hebras en la configuración inicial del gestor (ver línea 3 del Algoritmo 2.6.1). Finalmente, al finalizar la ejecución de la hebra se debe informar al GP para que deje de gestionarla (línea 14).

Algoritmo 2.6.3 : Configuración de un GP en el proceso principal de una aplicación multihebrada con creación dinámica de hebras

```
(1) include{IGP_library}
(2) funct main_dinamic_process(Task)
(3)     IGP_Initialize(M, C, P, T, true);                    Fase de Inicialización
(4)     global_work_load = memory_allocator(Task);           Reserva memoria
(5)     create_thread(dinamic_thread, global_work_load);
(6)     wait_results_threads();                              Esperar a que todas las hebras terminen
(7)     IGP_Finalize();                                      Fase de Terminación
(8)     proccess_results();
```

Los Algoritmos 2.6.3 y 2.6.4 muestran la integración de las funciones disponibles en la librería IGP sobre una aplicación multihebrada basada en la creación dinámica de hebras. Si se comparan ambos algoritmos con los descritos en el Capítulo 1 (Algoritmos 1.6.3 y 1.6.4), se aprecia que la implementación con GP evita que el usuario establezca el valor de $MaxThreads$. Cada hebra computacional ejecuta un bucle, de tal forma, que el número de iteraciones a realizar suele ser directamente proporcional a la carga de trabajo asignada, e informa al GP en cada iteración de la carga realizada (línea 6 del Algoritmo 2.6.4). En este esquema, el gestor de hebras notifica a cada una de las hebras activas, la decisión tomada por el gestor en la fase de *Evaluación* a través de la variable *decision* (línea 6 del Algoritmo 2.6.4). Esta estrategía permite adaptar la creación de las sucesivas hebras

Algoritmo 2.6.4 : Comunicación de la hebra con un GP

```
(1)  funct dinamic_thread(work_load)
(2)    IGP_Begin_Thread(work_load);                          Fase de Inicialización
(3)    while (work_load ≠ {∅})
(4)          critical_section(global_parameters);
(5)          compute_work(work_load_i);
(6)          decision = IGP_get(work_load_i);                Fases de Información y Notificación
(7)          if (decision = new)
(8)            half_work_load = divide(work_load);
(9)            create_thread(dinamic_thread, half_work_load);
(10)         if (decision = sleep)
(11)           sleep(interval_time);                         Detención a nivel de usuario
(12)   send_results();
(13)   IGP_End_Thread();                                     Fase de Terminación
```

a los recursos disponibles y el rendimiento de la aplicación según la configuración del GP seleccionado. Como se puede observar, los algoritmos multihebrados basados en la creación dinámica de hebras permiten un mayor nivel de adaptación frente a la creación estática de hebras. Además, el balanceo dinámico de la carga de trabajo, intrínseco en los algoritmos multihebrados con creación de dinámica de hebras, permite reducir considerablemente la comunicación entre las hebras, ya que la asignación de la carga de trabajo se restringe a la creación de cada hebra. Sin embargo, las aplicaciones con creación estática de hebras necesitan implementar un balanceador dinámico de la carga de trabajo entre las hebras activas. El principal problema de la comunicación entre hebras son las detenciones que surgen en la sincronización, o la falta de trabajo, lo que provoca retardos que inciden directamente en el rendimiento de la aplicación.

2.7. Evaluación de los GPs

Para estudiar el impacto sobre las aplicaciones HPC de cada uno de los gestores propuestos en este capítulo, se ha diseñado un benchmark sintético basado en una aplicación multihebrada. La finalidad principal del benchmark es controlar los factores característicos de una aplicación multihebrada para que nos permita analizar el comportamiento de los distintos GPs. Las características de una aplicación multihebrada suelen influir directamente en su rendimiento, como por ejemplo, el acceso a memoria compartida, la comunicación entre hebras y las secciones críticas. Cada uno de estos factores pueden provocar bloqueos entre las hebras, de tal forma que influyen negativamente en el rendimiento de la aplicación. Por este motivo, se necesita diseñar un benchmark donde se pueda controlar los retardos producidos por cada uno de estos factores y evaluar el comportamiento del GP cuando aparecen dichos factores. El benchmark se ha diseñando sobre una aplicación multihebrada que implementa el algoritmo *ray tracing*, de tal forma que cada una de las hebras calcula la intersección de un elevado número de rayos sobre una esfera [56], [64] y [58]. Se

ha utilizado el algoritmo *ray tracing* porque permite realizar un reparto equitativo de la carga de trabajo entre las hebras, minimizando considerablemente la comunicación entre las hebras. Además permite incrementar fácilmente la carga de trabajo total, aumentando el número de rayos a procesar.

Algoritmo 2.7.1 : Programa principal del benchmark sintético basado en *ray tracing*

```
(1) funct Main(task)
(2)     var M, C, P, T, global_work_load, image;
(3)     IGP_Initialize(M, C, P, T, false);                    Fase de Inicialización
(4)     global_work_load = memory_allocator(task);            Reserva memoria
(5)     create_thread(create_image, global_work_load);
(6)     image = wait_results_threads();                       Esperar a que todas las hebras terminen
(7)     IGP_Finalize();                                       Fase de Terminación
(8)     proccess_results(image);                              Salvar y mostrar resultados
```

El Algoritmo 2.7.1 muestra el esqueleto del benchmark implementado, donde se ha utilizado la librería *IGP* para gestionar la generación dinámica de hebras. En el diseño de este benchmark se han eliminado las variables globales para hacer al algoritmo "*thread-safe*". Básicamente el proceso principal permite al usuario configurar y seleccionar el tipo de gestor de hebras a través de la función $IGP\_Initialize(M, C, P, T, Detention)$, se reserva la memoria (línea 4) donde almacenar la imagen total que le asignará a la primera hebra creada (línea 5) y detiene su ejecución sin consumir recursos, esperando los resultados las hebras computacionales (línea 6). Finalmente, el proceso informa al GP que desea terminar la gestión de hebras (línea 7) y muestra la imagen final por pantalla (línea 8).

El Algoritmo 2.7.2 presenta el código fuente ejecutado por cada hebra computacional. Básicamente una hebra antes de ejecutar el bucle computacional del algoritmo, solicita ser gestionada por el GP a través de la función $IGP\_Begin\_Thread(work\_load)$ y reserva memoria local. Como se puede observar, esta primera implementación carece de secciones críticas y además se ha minimizado la concurrencia en el acceso a memoria compartida, reduciendo al máximo el uso de variables globales (línea 4). Esto tiene la ventaja de reducir los tiempos de bloqueo de las hebras, provocados por la gestión interna del GP, evitando así que la ejecución del GP incida directamente en la ejecución de la aplicación multihebrada. Por otro lado, el trabajo computacional asignado a la hebra consiste en seleccionar un pixel de la imagen en cada iteración (línea 6), obtener la recta (rayo) que contiene el punto de visión y pasa por el pixel de la imagen seleccionado (línea 7), calcular la intersección del rayo con la escena (línea 8) y finalmente colorear el pixel de la imagen (línea 9). En cada iteración, la hebra informa al GP del número de pixeles calculados y chequea si puede o no crear una nueva hebra (línea 10). En el supuesto de que GP seleccione a esta hebra para crear una nueva hebra, está podrá asignar la mitad de la imagen restante a la nueva hebra creada (línea 12). Finalmente, cuando cada una de las hebras finaliza la evaluación de los pixeles asignados, envía la imagen coloreada al proceso principal (línea 14), e informa de su terminación al GP (línea 16).

La escena utilizada en los siguientes experimentos está formada por un *spheresflake*

Algoritmo 2.7.2 : Función ejecutada por cada hebra en *ray tracing*

```
(1)  funct create_image(work_load)
(2)    var ray, block, point, color, decision, image, half_image;
(3)    IGP_Begin_Thread(work_load);                    Fase de Inicialización
(4)    image = memory_allocation(work_load);           Reservar memoria para la imágen
(5)    while (work_load ≠ {∅})
(6)          point = choose_point(work_load);          Comienza el trabajo computacional
(7)          ray = create_ray(point);
(8)          color = compute_intersection_ray_sphere(ray);
(9)          image = paint_image(point, color);        Termina el trabajo computacional
(10)         decision = IGP_get(work_load_i);          Fases de Información y Notificación
(11)         if (decision = new)
(12)           half_image = divide(work_load_i);
(13)           create_thread(Thread, half_image);
(14)   send_results(image);                            Envía resultados al programa principal
(15)   free(image);                                    Liberar la reserva de memoria para la imágen
(16)   IGP_End_Thread();                               Fase de Terminación
```

(véanse las Figuras 2.6 y 2.7), que básicamente consiste en una distribución jerárquica de esferas siguiendo un determinado patrón. Así pues, esta escena utiliza como elemento básico la esfera definida por su centro $C = (c_x, c_y, c_z)$ y su radio r, de forma que los puntos (x, y, z) sobre su superficie son aquellos que verifican la siguiente ecuación:

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2 \tag{2.1}$$

Los puntos de intersección entre una esfera y un rayo $R(t) = O + D \cdot t$, con origen en $O = (o_x, o_y, o_z)$ y vector dirección normalizado $D = (d_x, d_y, d_z)$, son aquellos puntos (x, y, z) sobre la superficie de la esfera que verifican la ecuación del rayo:

$$(o_x + d_x \cdot t - c_x)^2 + (o_y + d_y \cdot t - c_y)^2 + (o_z + d_z \cdot t - c_z)^2 = r^2 \tag{2.2}$$

por lo que el cálculo de la intersección del rayo con cada una de las esferas (línea 8 del Algoritmo 2.7.2), básicamente consiste en resolver una ecuación de segundo grado:

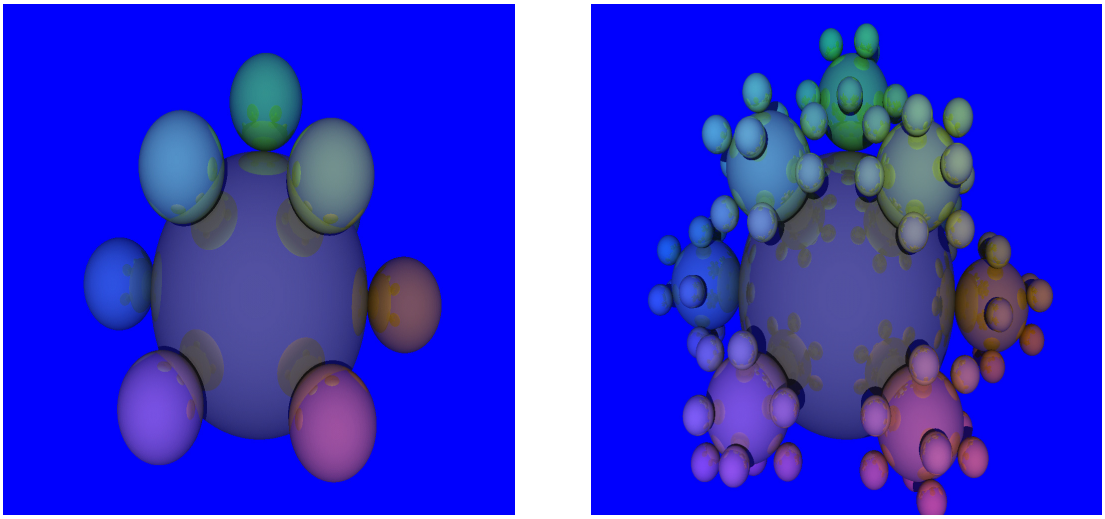
$$a \cdot t^2 + b \cdot t + c = 0 \tag{2.3}$$

donde:

$$\begin{aligned} a &= d_x^2 + d_y^2 + d_z^2 \\ b &= 2 \cdot o_x \cdot d_x - 2 \cdot c_x \cdot d_x + 2 \cdot o_y \cdot d_y - 2 \cdot c_y \cdot d_y + 2 \cdot o_z \cdot d_z - 2 \cdot c_z \cdot d_z \\ c &= o_x^2 + c_x^2 - 2 \cdot o_x \cdot c_x + o_y^2 + c_y^2 - 2 \cdot o_y \cdot c_y + o_z^2 + c_z^2 - 2 \cdot o_z \cdot c_z - r^2 \end{aligned} \tag{2.4}$$

Así que las dos posibles soluciones son:

$$t = \frac{-b \pm \sqrt{b^2 - 4 \cdot c}}{2 \cdot a} \tag{2.5}$$



(a) Spheresflake de 1º nivel, con 10 esferas

(b) Spheresflake de 2º nivel, con 91 esferas

Figura 2.6: Escenas generadas con el algoritmo *ray tracing*.

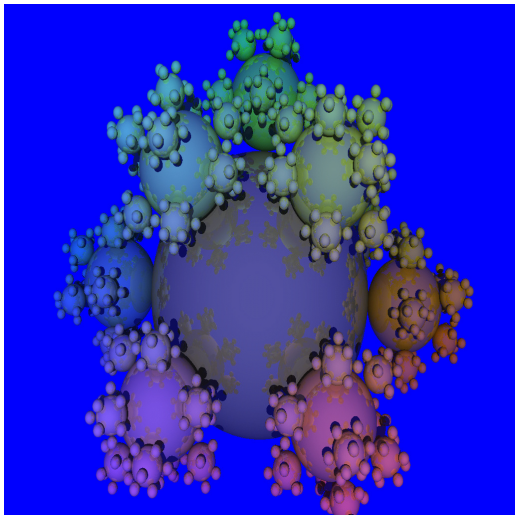
La Figura 2.6 (a) muestra la distribución de esferas generadas por el patrón *spheresflake* de 1er nivel, formada por 1 esfera en el centro y 9 esferas satélite. La Figura 2.6 (b) representa un *spheresflake* de 2do nivel, formada por 1 esfera en el centro y 9 *spheresflake* satélite de 1er nivel.

La Figura 2.7 (a) muestra una escena formada por un *spheresflake* de 3er nivel, formada por 1 esfera central, rodeada por 9 *spheresflake* satélites de 2do nivel.

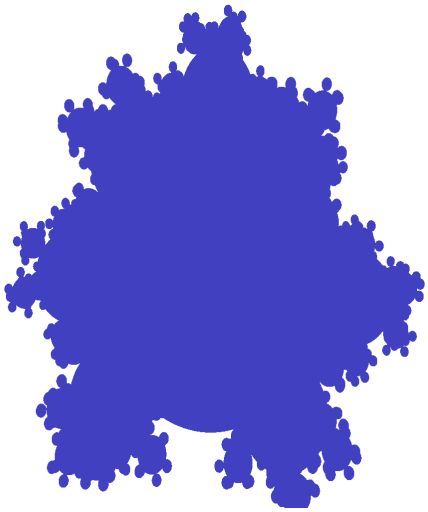
Todas las Figuras 2.6 y 2.7 (a) han sido renderizadas con el algoritmo de *ray tracing* original, permitiendo refracciones y reflexiones. Sin embargo, en el benchmark sintético utilizado no se permiten reflexiones, ni refracciones, para evitar la necesidad de acceso a memoria compartida o la existencia de secciones críticas. De esta forma, el algoritmo *ray tracing* se ha adaptado a unas condiciones que facilitan la escalabilidad. La Figura 2.7 (b) muestra la escena renderizada con el benchmark sintético que se utilizará en la experimentación.

2.7.1. GP a nivel de usuario

El primer experimento se ha realizado sobre el anterior benchmark sintético basado en una estrategia de creación dinámica de hebras, en la que se han ensayado inicialmente dos GPs a nivel de usuario: ACW y AST (descritos en la Sección 2.3). La carga de trabajo total de la aplicación se ha establecido con un número de 10^{10} rayos lanzados sobre un *spheresflake* de 3er nivel (Figura 2.7 (b)). La evaluación realizada por los GPs se ha realizado cada vez que cada hebra evaluaba al menos 10^4 rayos. El criterio de decisión utilizado en ambos GPs establece el número de hebras óptimo ($MaxThreads$) igual a 16, pues coincide con el número total de unidades de procesamiento, ya que nuestro experimento se ha realizado en un sistema dedicado sobre un multiprocesador de memoria compartida con 4 procesa-



(a) Spheresflake de 3º nivel, con 820 esferas



(b) Imagen generada con el benchmark sintético

Figura 2.7: Escena generada con el algoritmo *ray tracing*.

dores *AMD Opteron*^{TM} y cuatro núcleos por procesador. Se ha realizado inhabilitando la posibilidad de que el GP detenga hebras en ejecución.

La Figura 2.8 muestra el número de hebras en ejecución en cada instante de tiempo para los GPs planteados a nivel de usuario. Se puede observar que el tiempo total de ejecución es menor si utilizamos un GP que realiza accesos a la memoria compartida de las distintas hebras activas (shared memory), debido a que el GP basado en la lectura de pseudo-ficheros (línea continua roja) presenta unos retardos en dichas lecturas que impiden una respuesta rápida en la creación de hebras. Este detalle se aprecia con claridad en la fase inicial de la creación de hebras (ver Figura 2.9), que corresponde al primer segundo de ejecución de la aplicación multihebrada, de tal forma que el GP basado en pseudo-ficheros necesita 1/2 segundo para alcanzar el valor máximo de hebras mientras que se aprecia una buena reacción con el GP basado en memoria compartida.

2.7.2. GP a nivel de kernel

El siguiente experimento se ha realizado con un GP a nivel de kernel basado en llamadas al sistema (ver Subsección 2.4.1) sobre la misma aplicación multihebrada de *ray tracing*. Se ha realizado inhabilitando la posibilidad de que el GP detenga hebras en ejecución.

Las Figuras 2.10 y 2.11 muestran una comparativa del GP a nivel de kernel basado en llamada al sistema (KST), con el GP a nivel de usuario basado en memoria compartida (ACW). Mientras que la Figura 2.10 corresponde al tiempo total de ejecución, la Figura 2.11 tan solo muestra el primer segundo de ejecución. Observando ambas gráficas se aprecia que ambos gestores tienen un tiempo de reacción muy parecido, lo que demuestra que el transito del acceso a la memoria entre el nivel de usuario y el nivel de kernel no es un problema si utilizamos un GP basado en llamadas al sistema. En este sentido, ambas solu-

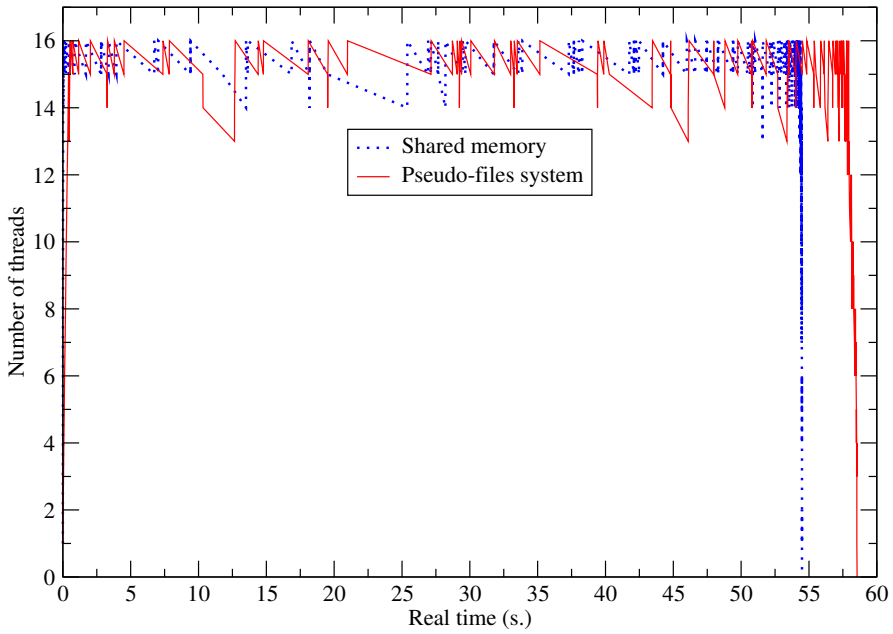


Figura 2.8: Comparativa de la evolución del número de hebras para los GPs a nivel de usuario: Shared memory (ACW: Application decides based on Completed Work) versus Pseudo-ficheros (AST: Application decides based on Sleeping Threads).

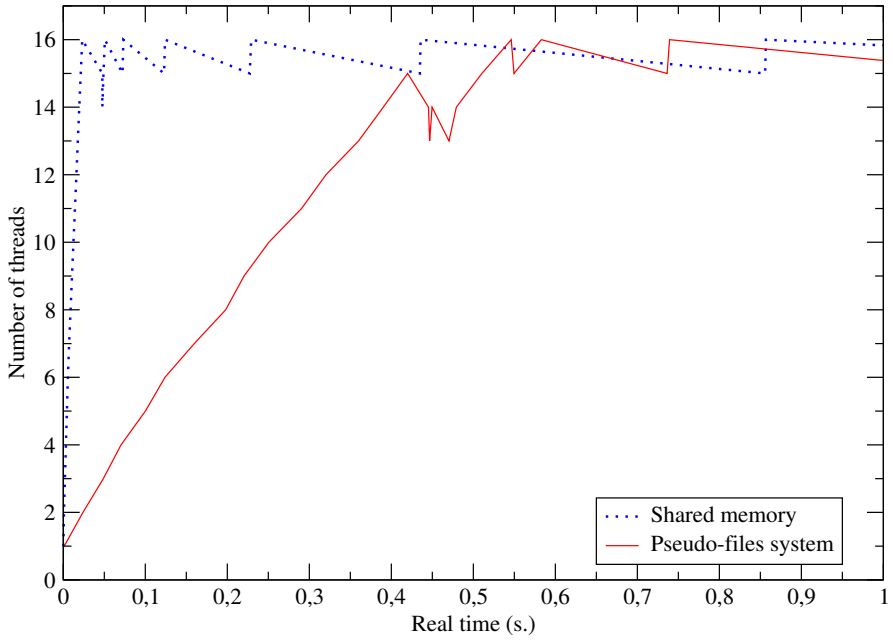


Figura 2.9: Creación inicial de hebras para los GPs a nivel de usuario: Shared memory (ACW: Application decides based on Completed Work) versus Pseudo-ficheros (AST: Application decides based on Sleeping Threads).

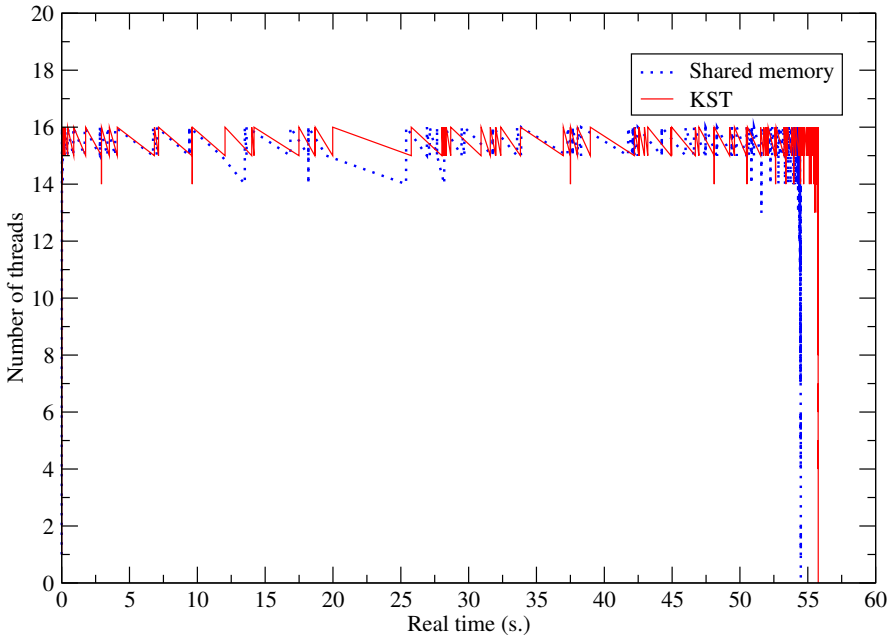


Figura 2.10: Comparativa de la evolución del número de hebras para los GPs: *KST* (Kernel decides based on Sleeping Threads) versus Shared memory (*ACW*: Application decides based on Completed Work).

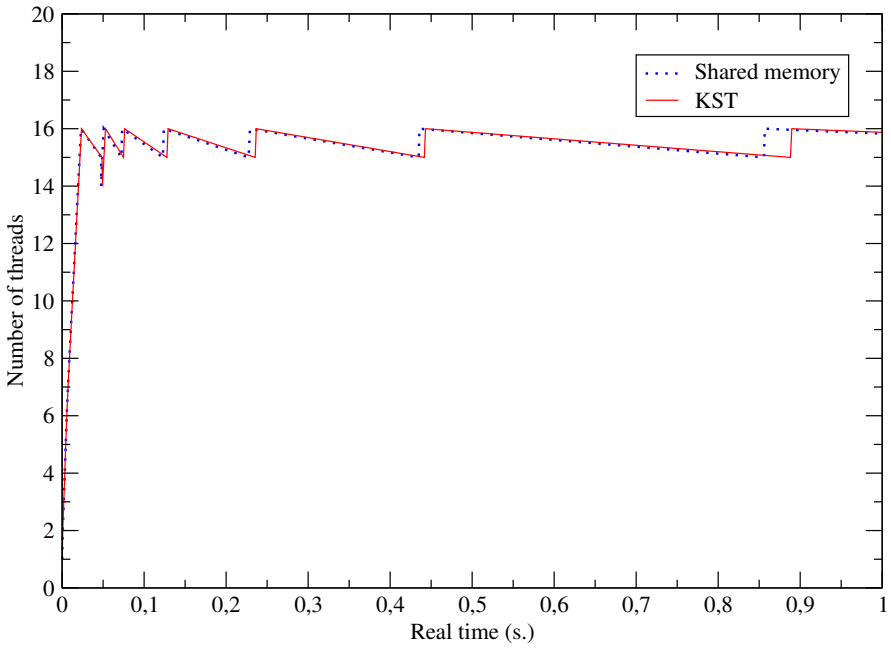


Figura 2.11: Creación inicial de hebras para los GPs: *KST* (Kernel decides based on Sleeping Threads) versus Shared memory (*ACW*: Application decides based on Completed Work).

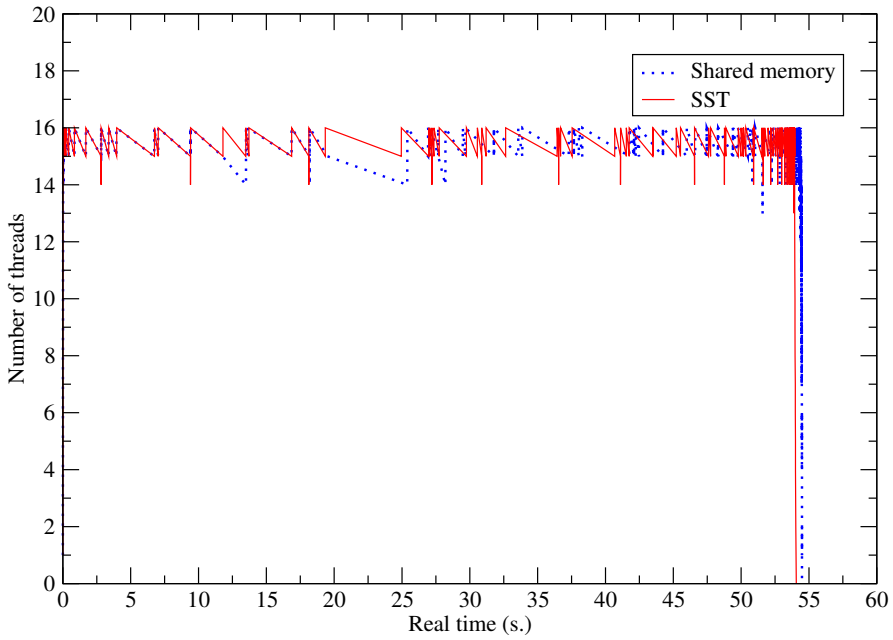


Figura 2.12: Comparativa de la evolución del número de hebras para los GPs: *SST* (Scheduler decides based on Sleeping Threads) versus Shared memory (*ACW*: Application decides based on Completed Work).

ciones son igual de válidas, por lo que la selección del GP basado en memoria compartida a nivel de usuario o un gestor basado en llamadas al sistema a nivel de kernel, dependerá del criterio de decisión que debe usar el GP. En el siguiente capítulo se analizarán distintos criterios de decisión y se observará que existen criterios de decisión que requieren datos disponibles a nivel de kernel y otros a nivel de usuario. Por ejemplo, el criterio de decisión basado en el rendimiento de la aplicación utiliza información procedente directamente del nivel de usuario (Sección 3.3), mientras que el criterio de decisión basado en el retardo de la aplicación requiere de información disponible exclusivamente a nivel de kernel (Sección 3.4).

A continuación, se han evaluado otros GPs a nivel de kernel (*SST*: Scheduler decides based on Sleeping Threads y *KITST*: Kernel Idle Thread decides based on Sleeping Threads) sobre la misma aplicación multihebrada de *ray tracing*. La diferencia entre ambos gestores radica en la entidad que ejecuta la fase de *Evaluación*: el manejador de interrupciones del reloj del sistema (descrito en la Sección 2.4.2) o la hebra ociosa del Kernel (ver Subsección 2.4.2). Tanto el criterio de decisión utilizado en estos gestores, como el número de unidades de procesamiento, intervalo de ejecución y el número total de rayos evaluados no han cambiado respecto de los experimentos anteriores.

La Figura 2.12 compara el número de hebras en cada instante de tiempo durante la ejecución de la aplicación multihebrada *ray tracing* cuando se utiliza el gestor *SST*, donde las fases de *Información* y *Evaluación* es ejecutada en cada tick del reloj, y cuando se utiliza el GP a nivel de usuario basado en memoria compartida. Sin embargo, la Figura

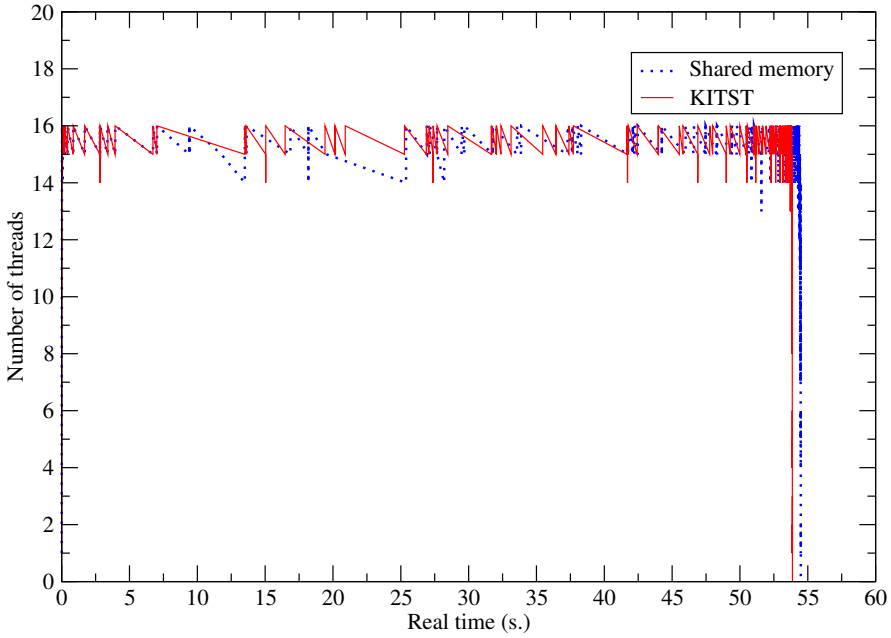


Figura 2.13: Comparativa de la evolución del número de hebras para los GPs: *KITST* (Kernel Idle Thread decides based on Sleeping Threads) versus Shared memory (*ACW*: Application decides based on Completed Work).

2.13 compara el número de hebras cuando la fase de *Evaluación* se ejecuta por la hebra ociosa del sistema (*KITST*), y el número de hebras cuando se usa el mismo GP a nivel de usuario. Se puede observar que los gestores a nivel de kernel (*SST* y *KITST*) consiguen mejorar levemente los tiempos de ejecución conseguidos con el mejor GP a nivel de usuario.

Tabla 2.1: Comparativa de los recursos consumidos por los distintos GP.

Gestores del nivel de Paralelismo (GP)	N. Ejecuciones	N. Evaluaciones	N. respuestas afirmativas	Uso PU
ACW	875.626	870.294	185	0,8 %
AST	89.444	86.446	**131**	**8,7 %**
KST	940.100	56.032	204	2,7 %
SST	798.203	49.862	165	2,5 %
KITST	888	172	168	**0,002 %**

En cada ejecución del GP, el criterio de decisión no tiene que ser evaluado ya que el GP debe verificar que todas las hebras hayan realizado parte de su carga de trabajo y recopilar toda la información necesaria antes de evaluar el criterio de decisión. En la Tabla 2.1 se observan distintas medidas relacionadas con los recursos consumidos por los distintos GPs planteados: número de veces que la aplicación ha ejecutado el gestor, número de veces que el gestor ha evaluado el criterio de decisión, número de respuestas

afirmativas realizadas por el gestor, indicando a la aplicación monitorizada que se puede crear una nueva hebra, y por último, el tanto por ciento que el gestor ha estado utilizando la unidad de procesamiento (PU) respecto del tiempo total de ejecución de la aplicación multihebrada. Entre estos parámetros destaca el GP integrado en la hebra ociosa del kernel (*KITST*) como el gestor que menos veces se ha ejecutado, lo que influye directamente en una sobrecarga en el uso de las unidades de procesamiento casi despreciable (0,002 %). Por otro lado, hay que indicar que el GP basado en peudo-ficheros es el peor de los gestores, no solo porque consume muchos recursos (8,7 % uso de PU), sino porque el número de respuestas afirmativas generadas por este gestor es muy inferior a las generadas por el resto de gestores. La justificación se encuentra en que este gestor desconoce cierta información relativa a las hebras entre dos lecturas de los pseudo-ficheros del sistema.

2.7.3. Intervalo de ejecución

Un factor importante que se debe analizar es el valor del periodo de muestreo o intervalo de ejecución con el que el gestor debe chequear el criterio de decisión establecido en la fase de *Evaluación* denominado λ. Para valores grandes de λ, se reduce la capacidad de reacción del gestor de hebras para adaptar la aplicación multihebrada a los cambios en el sistema. Esta lentitud en la adaptación suele repercutir directamente en una perdida de rendimiento de la aplicación gestionada. Teóricamente, se podría conseguir una adaptación casi inmediata para valores de λ prácticamente nulos, por lo que se requieren de GPs rápidos, especialmente durante la fase de *Información*. Además, si el chequeo del criterio de decisión se realiza muy frecuentemente, se corre el riesgo de que el GP incremente el consumo de PU. Por este motivo, se llega a un compromiso para escoger un intervalo de tiempo suficientemente grande para que la información obtenida sea representativa y suficientemente pequeño para que la reacción sea rápida. En este sentido, el mínimo intervalo de tiempo entre dos ejecuciones consecutivas de la fase de *Evaluación* del GP viene impuesto por el tiempo necesario para que todas las hebras activas ejecuten al menos una unidad de trabajo o iteración del bucle. Por otra parte, este es el mínimo intervalo de tiempo para tener información sobre como se ha comportado la aplicación cuando todas las hebras han realizado al menos una unidad de trabajo (fase de *Información*).

Se ha realizado un experimento para determinar el impacto de λ sobre la eficiencia de una aplicación multihebrada definida según la Ecuación 1.1. Se ha ensayado con el anterior benchmark sintético basado en el algoritmo *ray tracing*, utilizando el mismo algoritmo multihebrado con generación dinámica de hebras ejecutándose en un entorno dedicado con 16 procesadores. El criterio de decisión del gestor seleccionado establece $MaxThreads$ igual al número de unidades de procesamiento establecido por el usuario, y se han realizado varias ejecuciones para un rango de valores de $MaxThreads$ comprendido entre 1 y 32 hebras. Los valores de λ (intervalo de tiempo entre dos ejecuciones consecutivas del criterio de decisión) evaluados son multiplos del tiempo de ejecución de una iteración del algoritmo de ray tracing, de forma que λ ha tenido unos valores de 88.5 ns, 885 ns, 8850 ns, 0.0885 ms y 0.885 ms, para 1, 10, 100, 1.000 y 10.000 (iteraciones) rayos evaluados, respectivamente. El incremento del número de rayos en cada unidad de trabajo se traduce en un incremento directo de λ. Este experimento se ha realizado inhabilitando la posibilidad de que el GP

detenga hebras en ejecución.

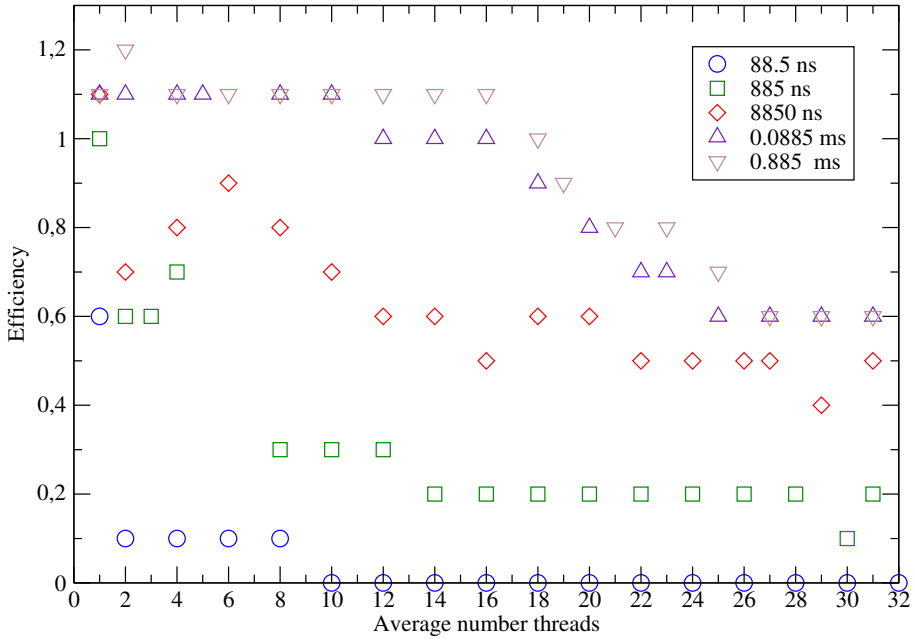


Figura 2.14: Eficiencia obtenida por ACW, cuando siempre se ejecuta el criterio de decisión.

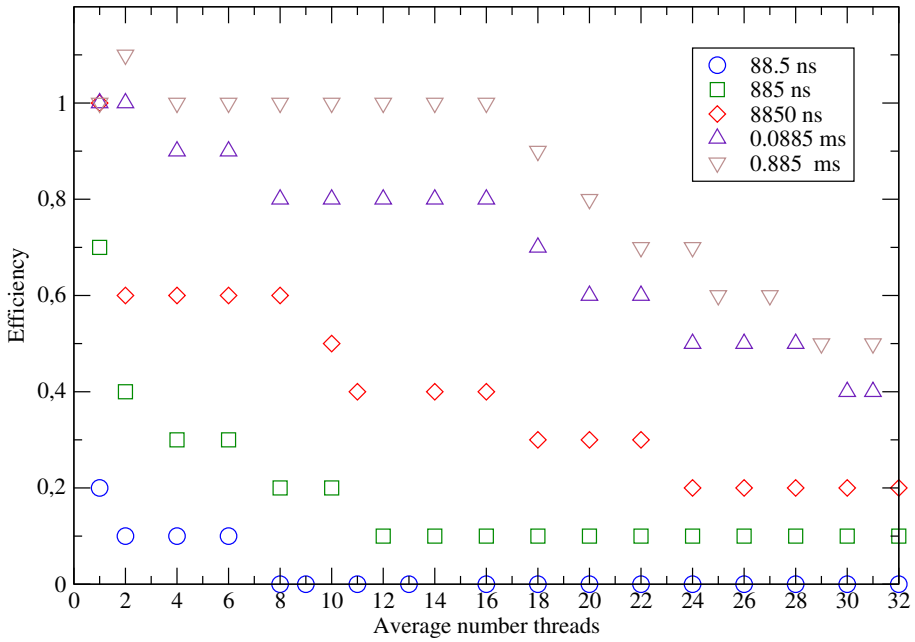


Figura 2.15: Eficiencia obtenida por KST, cuando siempre se ejecuta el criterio de decisión.

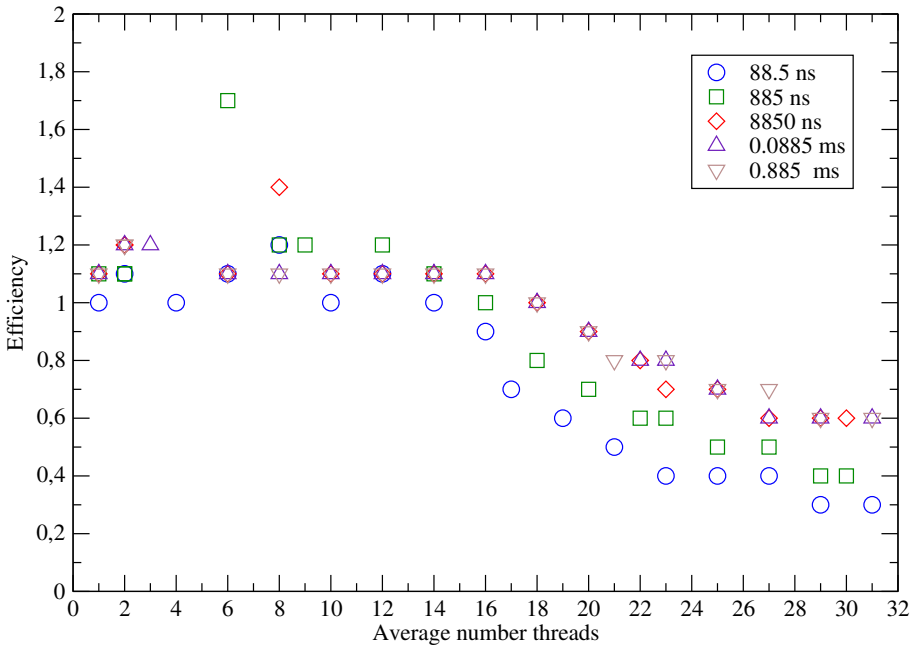


Figura 2.16: Eficiencia obtenida por ACW, ejecutando solo el criterio de decisión cuando el número de hebras activas es inferior a $MaxThreads$ o cuando cambie el número de hebras activas.

Las Figuras 2.14 y 2.15 muestran la eficiencia obtenida por la aplicación (ver Ecuación 1.1), con un GP a nivel de usuario de memoria compartida (ACW) y un GP a nivel de kernel basado en llamadas al sistema (KST), para varios valores de λ, respectivamente. Los valores de eficiencia se han obtenido respecto a la ejecución monohebrada sin utilizar ningún gestor. Se puede observar, que cuando el número medio de hebras es superior a 16, la eficiencia del gestor ACW disminuye considerablemente, debido a que ese es el número de unidades de procesamiento disponibles. Por otro lado, si se analizan los resultados obtenidos para el gestor ACW con un número medio de hebras igual, o inferior, a 16 hebras activas, solo se consiguen niveles de eficiencia aceptables (≈ 1) para valores de $\lambda \geq 0{,}0885$ ms, resaltando que las peores eficiencias se consiguen para el valor más pequeño de $\lambda = 88{,}5$ ns. Resultados parecidos se obtienen con el gestor KST, obteniendose la mejor eficiencia para $\lambda = 0{,}885$ ms, según se puede apreciar en la Figura 2.15.

Las Figuras 2.16 y 2.17 muestran la misma experimentación, pero ahora solo se permite ejecutar el criterio de decisión cuando el número de hebras activas es distinto de $MaxThreads$, y mientras no cambie el número de hebras activas. Ambos GPs permiten generar más de 16 hebras, dado que el criterio de decisión seleccionado establece $MaxThreads$ en un rango de valores comprendido entre 1 y 32 hebras. En ambos casos, se observa que la eficiencia se mantiene en buenos niveles (≈ 1) para cualquier valor de λ, cuando el número de hebras activas es inferior o igual a 16.

De este experimento se puede concluir que, por un lado evaluar mucho implica más uso de la PU y evaluar poco implica menos adaptabilidad. Sin embargo, para minimizar

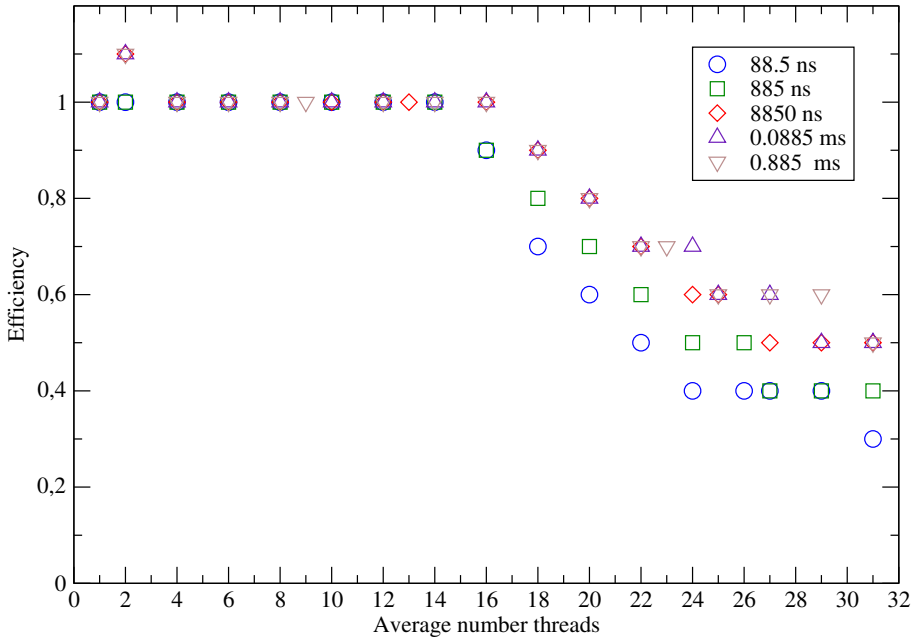


Figura 2.17: Eficiencia obtenida por KST, ejecutando solo el criterio de decisión cuando el número de hebras activas es inferior a $MaxThreads$ o cuando cambie el número de hebras activas.

el impacto de λ se debe reducir el número de veces que se ejecuta la fase de *Evaluación* de los GPs. Los objetivos de reducir el consumo de PU y mejorar la adaptabilidad de la aplicación se consiguen si se ejecuta la fase de *Evaluación* del GP sólo cuando han variado las condiciones, que en esta experimentación fueron que el número de hebras activas sea distinto a $MaxThreads$.

El GP integrado en la hebra ociosa de kernel ($KITST$) se ejecuta únicamente cuando se detecta una unidad de procesamiento ociosa, lo que podría acarrear una posible disminución de la adaptación de la aplicación. Las Figuras 2.18 y 2.19 muestran los resultados obtenidos en experimentos similares, en este caso para los GPs a nivel de kernel basados en el manejador de interrupciones del reloj (SST) y en la hebra ociosa del kernel ($KITST$), respectivamente. En ambos casos, se aprecia que ambos gestores son inmunes a valores de $\lambda \geq 885$ ns. Puede observarse que en la Figura 2.19 no aparecen valores de eficiencia cuando el número de hebras es superior al número de unidades de procesamiento, debido a que para el $KITST$ la fase de *Evaluación* del GP no se ejecuta cuando las 16 unidades de procesamiento están ocupadas.

2.8. Escalabilidad de las aplicaciones multihebradas

Todos los programadores de aplicaciones HPC centran la mayor parte de sus esfuerzos en diseñar programas con un código escalable, de tal forma que el algoritmo multihebrado obtenga niveles de eficiencia aceptables, si se aumentan ambas, la carga computacional

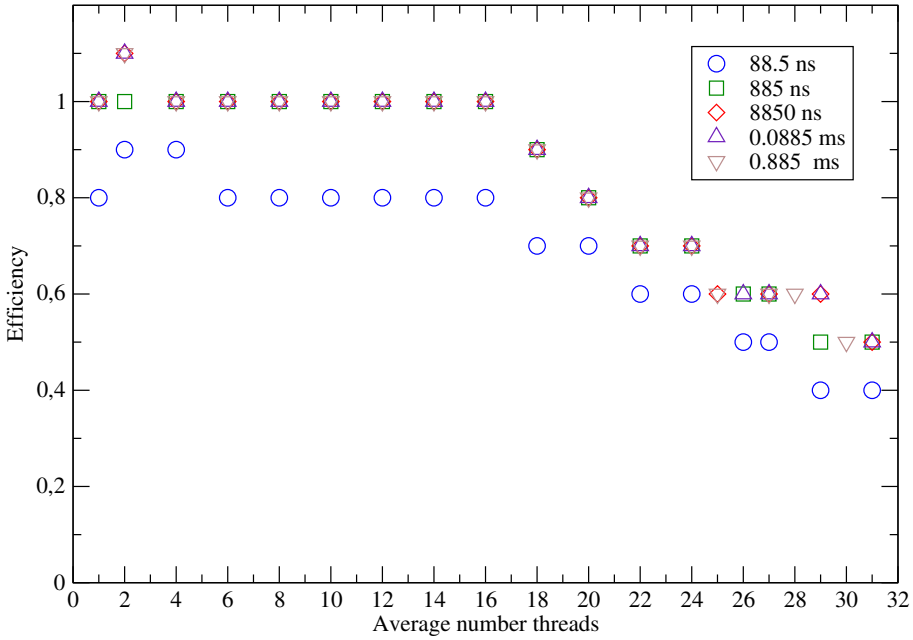


Figura 2.18: Eficiencia cuando el manejador de interrupciones de reloj ejecuta el criterio de decisión (SST) con diferentes intervalos de tiempo (λ).

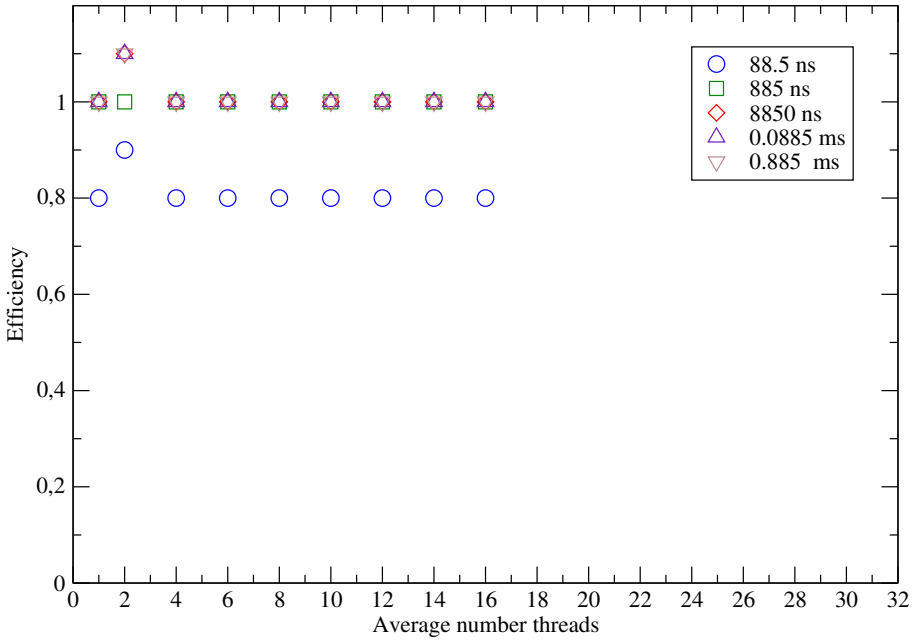


Figura 2.19: Eficiencia cuando la hebra ociosa del kernel ejecuta el criterio de decisión (KITST) con diferentes intervalos de tiempo (λ).

y el número de unidades de procesamiento utilizadas. Sin embargo, existen aplicaciones que sufren retardos propios de la implementación realizada por el programador, por lo que es muy difícil, en algunas ocasiones, reducir el impacto que dichos retardos provocan en la eficiencia de la aplicación, sobre todo cuando se intenta obtener la mayor eficiencia en sistemas que disponen de un elevado número de unidades de procesamiento. Además, este tipo de retardos suelen variar dinámicamente en tiempo de ejecución, ya que aunque cada hebra realiza el mismo trabajo, cada hebra puede estar ejecutando fases distintas del algoritmo.

En esta sección se analiza el comportamiento del GP integrado en el kernel (KITST) sobre un benchmark sintético con retardos provocados por los principales factores que pueden surgir en aplicaciones HPC: secciónes críticas y reserva de memoria. En esta tesis no se analizará el impacto que las operaciones de entrada/salida pueden producir sobre el rendimiento de la aplicación multihebrada. El tiempo dedicado a operaciones de E/S se puede reducir considerablemente haciendo uso de hebras especializadas en realizar tales operaciones. Estas hebras no serían gestionadas por el GP ya que no contribuyen a la computación de unidades de trabajo.

Para realizar las pruebas de esta sección se usará el GP basado en la hebra ociosa del kernel, dado su mayor nivel de inmunidad a λ, su reducido consumo de recursos computacionales y su rápido nivel de adaptación. Las pruebas también se han realizado inhabilitando la posibilidad de que el GP detenga hebras en ejecución.

2.8.1. Secciones críticas

Cuando una tarea (proceso o hebra) ejecuta una sección crítica bloqueante, fuerza a otras tareas a esperar a que no exista otra tarea ejecutando la sección crítica cada vez que desean entrar en dicha sección crítica. Todo recurso compartido (estructura de datos ó dispositivo) que debe ser accedido por una sola porción de código de un programa debe protegerse por una sección crítica. Se necesita de un mecanismo de sincronización en la entrada y la salida de la sección crítica para asegurar la utilización exclusiva del recurso, por ejemplo un semáforo.

La sección crítica se utiliza por lo general para resolver el problema que se plantea en un programa multihebrado cuando se accede concurrentemente a la actualización de una o múltiples variables compartidas. Los algoritmos de exclusión mutua, más conocidos como *mutex* (por *mut*ual *ex*clusion), se usan en programación concurrente para evitar que varios fragmentos de código accedan al mismo tiempo a recursos que no deben ser compartidos.

La técnica que se emplea comúnmente para conseguir la exclusión mutua es inhabilitar las interrupciones durante el conjunto de instrucciones de la sección crítica, lo que impedirá la corrupción de la estructura compartida, pues se impide que el código de la interrupción se ejecute dentro de la sección crítica. Por ejemplo, en un sistema multiprocesador de memoria compartida, se usa la operación indivisible *test-and-set* sobre una bandera para acceder a memoria compartida, de tal forma que el SO sube la bandera al entrar en la sección crítica y mantiene en espera al resto de hebras que desean acceder a la misma sección crítica hasta que el SO baje la bandera. La operación *test-and-set* realiza ambas operaciones sin liberar el bus de memoria para que pueda ser usado por otro procesador. Así, cuando la

ejecución sale de la sección crítica, se baja la bandera. Algunos sistemas operativos, por ejemplo Linux, tienen instrucciones multi-operación indivisibles (con ejecución atómica), similares a las anteriormente descritas, para manipular las listas enlazadas que se utilizan para las colas de eventos y otras estructuras de datos que los sistemas operativos usan comúnmente.

Las aplicaciones HPC intentan minimizar el uso de las secciones críticas debido a la perdida de rendimiento provocado por los interbloqueos creados en su acceso. En este sentido debemos de diferenciar dos tipos de secciones críticas: bloqueantes y no bloqueantes. En las secciones críticas bloqueantes se fuerza a detener todas aquellas hebras que intentan entrar en una sección crítica ya ocupada, lo que repercute en una reducción del rendimiento. Por otro lado, en las secciones críticas no bloqueantes se permite saltar la ejecución de la sección crítica a todas aquellas hebras que intentan acceder a una sección critica ya ocupada por otra hebra. Finalmente, indicar que la utilización de secciones críticas bloqueantes o no bloqueantes depende del problema a resolver y de la habilidad del programador.

La perdida de rendimiento es directamente proporcional al tiempo que las hebras han estado detenidas, el cual depende del número de hebras detenidas y de la duración de las secciones críticas. Por este motivo, los programadores de aplicaciones intentan reducir el número de secciones críticas en sus códigos y en caso necesario diseñar secciones críticas con un tiempo de ejecución reducido. También se recomienda analizar bien el problema por si podemos utilizar secciones críticas no bloqueantes.

Independientemente del tipo de sección crítica utilizada, el mecanismo para gestionar la sección crítica no es instantáneo, sino que requiere un cierto tiempo que, aunque pequeño, puede ser relevante si el número de secciones críticas es alto, aunque sean no bloqueantes. La mayoría de los métodos de exclusión mutua clásicos intentan reducir tanto el retardo, como la espera activa, mediante colas y cambios de contexto [30].

En esta sección se va a analizar el impacto que tienen las secciones críticas bloqueantes y no bloqueantes cuando se usa el GP integrado en la hebra ociosa del kernel. Para ello se ha añadido una sección crítica al código que ejecutan las hebras del benchmark sintético basado en el algoritmo *ray tracing* (Algoritmo 2.7.1). El Algoritmo 2.8.1 muestra (en negrita) la sección crítica integrada dentro del bucle computacional de cada hebra, lo que provoca un acceso secuencial a la sección crítica por parte de las hebras. En este experimento se ha obligado a que un porcentaje de los rayos a evaluar se hagan dentro de una sección crítica. Es decir, si la hebra i tiene una carga de trabajo de N_i rayos, el $t\,\%$ de N_i se ejecutará dentro de una sección crítica, mientras que el $(100-t)\,\%$ restante se podrá ejecutar en paralelo. La función create_image() recibe dos nuevos parámetros de entrada: t y option, donde t corresponde al tanto por ciento de carga computacional de la hebra que se ejecutará dentro de la sección crítica, mientras que option indica el tipo de sección crítica seleccionado: bloqueante o no bloqueante. En este experimento se han utilizado valores de t igual al $1\,\%$, $5\,\%$, $15\,\%$ y $25\,\%$, para evaluar como afectan a la eficiencia de la aplicación multihebrada.

La Figura 2.20 muestra la eficiencia obtenida por la aplicación multihebrada usando el GP integrado en la hebra ociosa del kernel cuando la sección crítica es bloqueante. Se puede observar que para $t = 1\,\%$ se consiguen buenos niveles de eficiencia hasta 16 hebras,

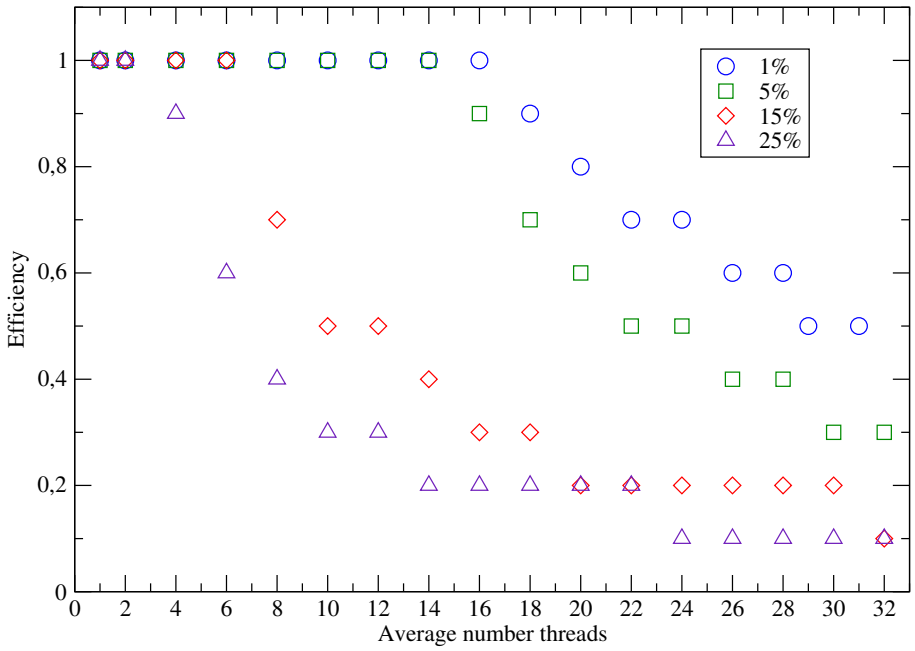


Figura 2.20: Eficiencia obtenida por el benchmark de *ray tracing* con sección crítica bloqueante utilizando el gestor KITST.

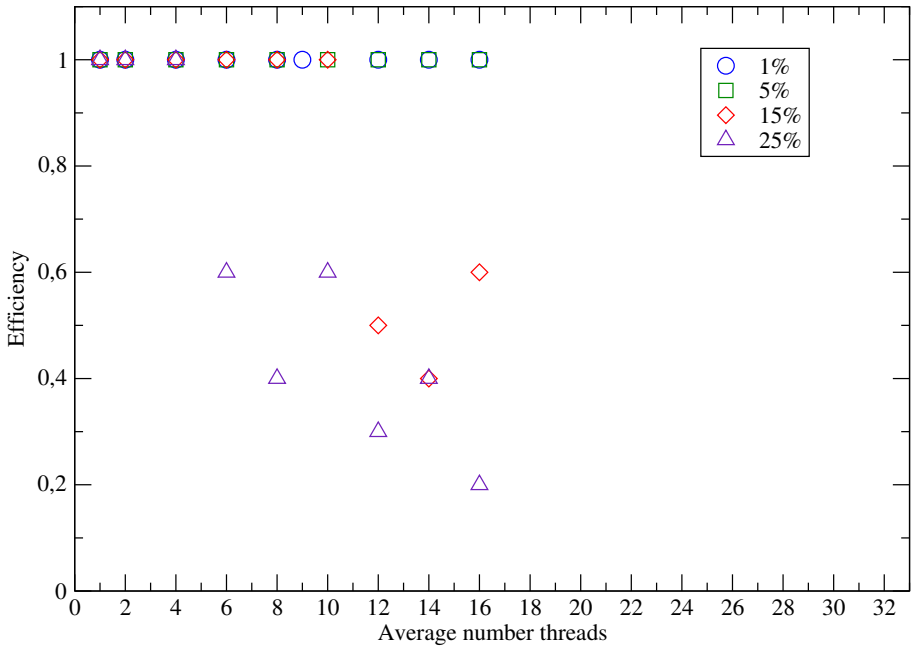


Figura 2.21: Eficiencia obtenida por el benchmark de *ray tracing* con sección crítica no bloqueante utilizando el gestor KITST.

Algoritmo 2.8.1 : Benchmark sintético basado en *ray tracing* con secciones críticas

```
(1)  funct create_image(work_load, t, option)
(2)    var ray, block, point, color, decision, image, half_image;
(3)    IGP_Begin_Thread(work_load);                          Fase de Inicialización
(4)    image = memory_allocation(work_load);                 Reservar memoria para la imágen
(5)    while (work_load ≠ {∅})
(6)          if (num_rays + + < t)
(7)            enable_critical_section(option);
(8)          point = choose_point(work_load);                Comienza el trabajo computacional
(9)          ray = create_ray(point);
(10)         color = compute_intersection_ray_sphere(ray);
(11)         image = paint_image(point, color);              Termina el trabajo computacional
(12)         if (num_rays < t)
(13)           disable_critical_section(option);
(14)         decision = IGP_get(work_load_i);                Fases de Información y Notificación
(15)         if (decision = new)
(16)           half_image = divide(work_load_i);
(17)           create_thread(Thread, half_image);
(18)   send_results(image);
(19)   free(image);                                          Liberar la reserva de memoria para la imágen
(20)   IGP_End_Thread();                                     Fase de Terminación
```

mientras que para $t = 5\,\%$ se obtienen esos mismos niveles de eficiencia pero hasta 14 hebras. Si se aumenta el valor de $t = 15\,\%$, solo se obtienen buenos niveles de eficiencia hasta 6 hebras. Sin embargo, tan solo se obtienen buenos niveles de eficiencia con 2 hebras para $t = 25\,\%$. Por otro lado, también se muestran los valores de eficiencia cuando se usan de 16 a 32 hebras, debido a que el GP integrado en la hebra ociosa del kernel ha permitido crear más de 16 hebras. Esto es debido a que la sección crítica ha bloqueado hebras en ejecución provocando así que las unidades de procesamiento estén más tiempo en estado ocioso.

Por otro lado, la Figura 2.21 muestra los resultados obtenidos con una sección crítica no bloqueante, donde se destaca como el gestor de hebras no ha creado más de 16 hebras activas, ya que la sección crítica no ha provocado bloqueos en las hebras activas. Otro aspecto esperado son los buenos niveles de eficiencia que se obtienen para valores de t iguales a $1\,\%$ y $5\,\%$, para cualquier número medio de hebras. Sin embargo, para $t = 15\,\%$ se obtienen buenos niveles de eficiencia hasta 10 hebras. Mientras que el número de hebras que mantiene una buena eficiencia se reduce a 4 hebras si utilizamos una sección crítica con una duración del $25\,\%$. La perdida de rendimiento observada para duraciones del $15\,\%$ y $25\,\%$ son provocadas principalmente por los requerimientos computacionales necesarios para gestionar la sección crítica.

2.8.2. Reserva de memoria

Otro factor, no menos importante, que incide directamente en el rendimiento de las aplicaciones multihebradas es la reserva dinámica de memoria, ya que es un recurso compartido al que se accede en exclusión mútua. Las aplicaciones usan la API *malloc*() para reservar memoria dinámicamente y la API $free()$ para liberar la memoria reservada, ambos disponibles en la librería *libc*. Antiguamente, la función *malloc*() no solamente aumentaba el espacio de direcciones asignado inicialmente en la creación del proceso por el SO, sino que también aumentaba la reserva en la memoria RAM asociada con el espacio de direcciones. Sin embargo, el diseño de nuevos SO enfocados a la demanda de páginas de memoria posibilitan que la función *malloc*() aumente el direccionamiento disponible del proceso, pero la RAM no será reservada hasta que la página sea realmente utilizada por el proceso. La función *malloc*() ejecuta internamente las llamadas al sistema *sbrk*() y *brk*() que permiten aumentar el espacio de direcciones inicialmente asignado por el SO al proceso.

Por otro lado, es común pensar que $free()$ reduce el espacio de direcciones del proceso, ya que el espacio de direcciones proporcionado por *sbrk*()/*brk*() sí puede ser decrementado pasando un valor negativo a *sbrk*(). Sin embargo, $free()$ no implementa esa técnica, debido a que $free()$ debería reducir el espacio de direcciones en el mismo orden en que fue reservado. Por lo tanto, la RAM asociada al espacio de direcciones es devuelta al SO solo cuando el proceso termina [9].

Los sistemas multicore mejoran su rendimiento al permitir la ejecución de sistemas operativos y aplicaciones multihebradas. La ejecución paralela de aplicaciones multihebradas permite que varios procesos/hebras puedan aumentar el espacio de direcciones simultaneamente, por lo que se requieren de mecanismos en el SO que permitan la sincronización eficiente de estas actividades. Inicialmente, se utilizaba un semáforo para asegurar el acceso a las regiones protegidas en *malloc*() y $free()$. El semáforo actualmente trabaja razonablemente bien para aplicaciones con pocas hebras o aplicaciones que hacen poco uso de las APIs *malloc*() y $free()$. Sin embargo, las aplicaciones HPC multihebradas que hacen mucho uso de las APIs de gestión de memoria reducen su rendimiento debido a la contención del semáforo de *malloc*() y $free()$. Por defecto, Linux emplea una estrategia de reserva de memoria optimizada para estos sistemas que no garantiza una disponibilidad real de la memoria solicitada por la función *malloc*(), aunque esta devuelva un valor diferente a NULL. En el supuesto de que el valor devuelto esté fuera del rango de memoria asignada, el sistema operativo forzará la terminación del proceso en ejecución. Normalmente, *malloc*() reserva memoria desde la pila, y ajusta el tamaño de la pila a lo que se hubiera solicitado, usando *sbrk*(). Sin embargo, cuando la reserva de los bloques de memoria supera un tamaño fijado por la variable $MMAP\_THRESHOLD$ bytes, la implementación de *malloc*(), de la librería *glibc*, reserva la memoria como una asignación anónima privada usando la llamada al sistema *mmap*(). El valor por defecto de $MMAP\_THRESHOLD$ es 128 KB, aunque puede ser modificado a través de la función *mallopt*() [69].

Los siguientes experimentos estudian el impacto que produce la reserva dinámica de memoria en la eficiencia de la aplicación haciendo uso del GP basado en la hebra ociosa del kernel. Básicamente se realizan dos experimentos, en el primero se reserva un bloque de memoria suficiente para cada grupo de rayos, denominado reserva por bloques de trabajo,

mientras que el segundo realiza una reserva de memoria por cada rayo individualmente, denominado reserva por unidad de trabajo. Todos los experimentos se han realizado inhabilitando la posibilidad de que el GP detenga hebras en ejecución.

Reserva por bloques de trabajo

La aplicación multihebrada basada en *ray tracing* (Algoritmo 2.7.1) realiza una reserva de memoria para cada una de las hebras activas. Esta reserva de memoria se realiza al inicio de la ejecución y es proporcional al tamaño de la carga de trabajo asignada a la hebra. Sin embargo, se pueden encontrar situaciones que impidan realizar esta reserva de memoria inicial en cada hebra, por ejemplo, cuando se dispone de insuficiente cantidad de memoria RAM en el sistema. En estas situaciones, es necesario que cada una de las hebras realice, dentro del bucle computacional, reservas y liberaciones de memoria por bloques.

Algoritmo 2.8.2 : Benchmark sintético basado en *ray tracing* con reserva de memoria dinámica por bloques

```
(1)  funct create_image(work_load, num_rays)
(2)    var ray, block, point, color, decision, image, half_image;
(3)    IGP_Begin_Thread(work_load);                          Fase de Inicialización
(4)    image = memory_allocation(work_load);                 Reservar memoria para la imágen
(5)    while (work_load ≠ {∅})
(6)          block = memory_allocation(num_rays);            Reservar memoria para un bloque
(7)          point = choose_point(work_load);                Comienza el trabajo computacional
(8)          block[ray] = create_ray(point);
(9)          color = compute_intersection_ray_sphere(block[ray]);
(10)         image = paint_image(point, color);              Termina el trabajo computacional
(11)         decision = IGP_get(work_load_i);                Fases de Información y Notificación
(12)         if (decision = new)
(13)           half_image = divide(work_load_i);
(14)           create_thread(Thread, half_image);
(15)         if (full(block))                                ¿Se ha utilizado todo el bloque?
(16)           free(block);                                  Liberar la reserva de memoria para el bloque
(17)   send_results(image);
(18)   free(image);                                          Liberar la reserva de memoria para la imágen
(19)   IGP_End_Thread();                                     Fase de Terminación
```

El Algoritmo 2.8.2 muestra la reserva de memoria para un bloque de rayos (línea 6) y la liberación de memoria del bloque (línea 18), una vez que se ha procesado.

El primer experimento consiste en realizar reservas de memoria en bloques con un tamaño de 64 B hasta 16 MB, de tal forma que cada bloque tenga capacidad para almacenar información relativa a 2^n rayos, donde n puede ser un número par en el rango 0 a 18. La Figura 2.22 muestra las eficiencias obtenidas cuando la reserva de memoria se realiza con la función *malloc*() para bloques de tamaño 64 B, 256 B, 1 KB, 4 KB y 16 KB, mientras

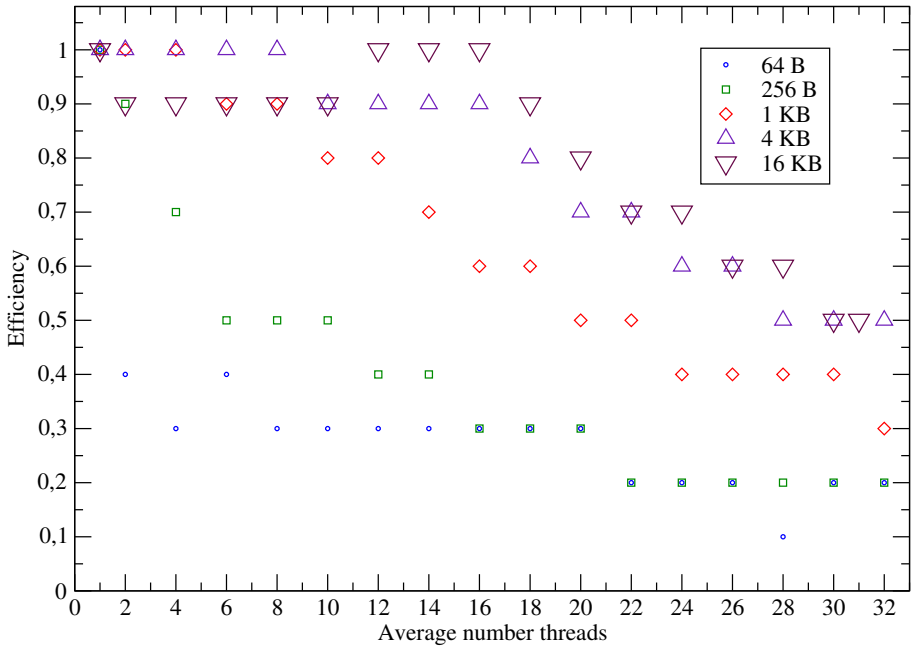


Figura 2.22: Eficiencia obtenida con reservas dinámicas de memoria para bloques desde 64 B hasta 16 KB realizadas con la función malloc().

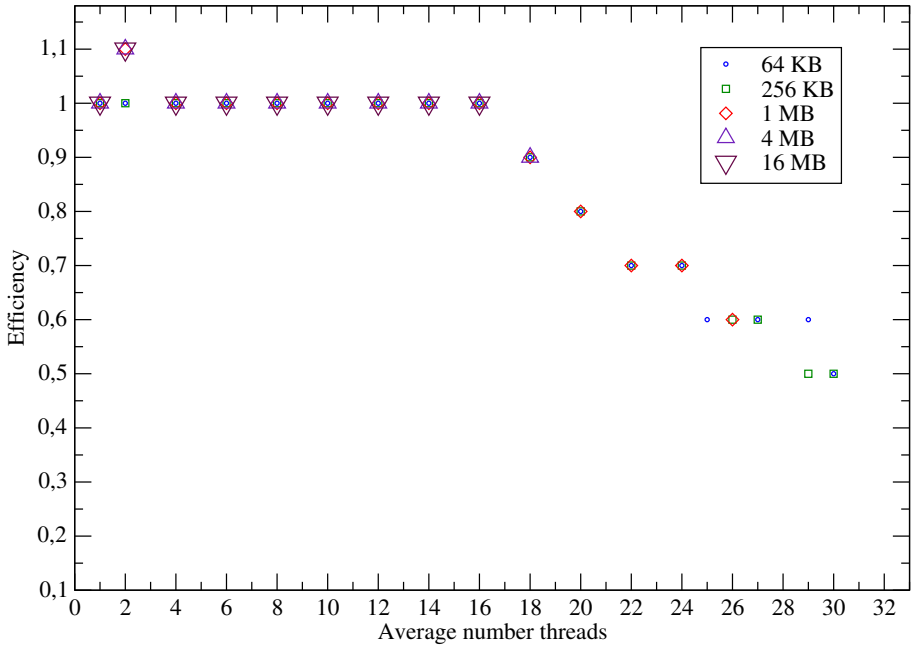


Figura 2.23: Eficiencia obtenida con reservas dinámicas de memoria para bloques desde 64 KB hasta 16 MB realizadas con la función malloc().

que la Figura 2.23 corresponde a tamaños 64 KB, 256 KB, 1 MB, 4 MB y 16 MB. En ambas figuras podemos observar que se obtienen buenos niveles de eficiencia para bloques de gran tamaño (4 KB o superior), siempre que el número de hebras no supere el número de unidades de procesamiento (en nuestro caso 16). Sin embargo, como era de esperar, se obtienen valores de eficiencia muy bajos, para tamaños de bloque inferiores a 4KB, debido a que se necesita realizar un elevado número de reservas simultáneas. Además, empeorando la eficiencia conforme aumenta el número de hebras, debido al incremento de interbloqueos entre las hebras para acceder a memoria compartida. Hay que resaltar que el gestor seleccionado ha permitido que se ejecuten simultaneamente menos de: 16 hebras para bloques de 16 MB, 18 hebras para bloques de 4 MB, 26 hebras para bloques de 1 MB, y 30 hebras para bloques de 64 KB o 256 KB (ver Figura 2.23). Esto es debido al hecho de que se minimizan los interbloqueos entre las hebras al reservar grandes bloques de memoria, lo que provoca la desaparición de estados ociosos en los procesadores, reduciéndose así el uso del GP basado en la hebra ociosa del kernel.

Reserva por unidad de trabajo

Existe otro tipo de aplicaciones HPC donde es muy difícil predecir la carga de trabajo total, ya que la carga de trabajo asignada inicialmente a cada hebra puede crecer exponencialmente en tiempo de ejecución, dependiendo de la dificultad del problema a resolver (p.e. aplicaciones de Ramificación y Acotación). Este hecho obliga a las hebras a realizar periódicamente reservas de pequeños bloques de memoria, lo que reduce la escalabilidad de la aplicación, puesto que se produciría una bajada de eficiencia al aumentar el número de hebras en ejecución, tal y como se ha apreciado en el experimento anterior. Por este motivo, este tipo de aplicaciones implementan una reserva de memoria por unidad de trabajo individual, basada en que cada hebra reserva y libera memoria en cada iteración según el trabajo a procesar en esa iteración.

Sin embargo, existen diferentes librerías disponibles en Internet que minimizan la perdida de rendimiento que experimenta la API *malloc*() cuando varias hebras gestionan memoria dinámica, simultaneamente. Por ejemplo, la libreria *ThreadAlloc* permite que cada hebra realice reservas de memoria igual al tamaño de una página de memoria (por defecto, 4 KB), y gestiona la memoria reservada de forma local y exclusiva por parte de una hebra, evitando exclusiones mutuas [32]. De esta forma, aunque la hebra realice muchas reservas de memoria de tamaño pequeño, estas son gestionadas localmente por la hebra mediante *ThreadAlloc* y realmente se realizan un número menor de reservas de tamaño 4 KB en el SO.

El Algoritmo 2.8.3 muestra la reserva de memoria por unidad de trabajo adaptada al benchmark de *ray tracing*. Como se puede observar en cada iteración, aunque se procesa un solo rayo, se reserva memoria para un bloque de 64 B (línea 6). El valor del parámetro de entrada *num_rays* establece el tamaño total del bloque, cuyos valores pueden variar entre 1, 4, 16, 64, 256 y 1024 rayos, que corresponden con tamaños de bloque en el rango 64 B hasta 64 KB. Mientras que las líneas 15 al 17 muestran la liberación del todas las reservas que componen el mismo bloque.

Las Figuras 2.24 y 2.25 muestran los valores de eficiencia obtenidos cuando se reali-

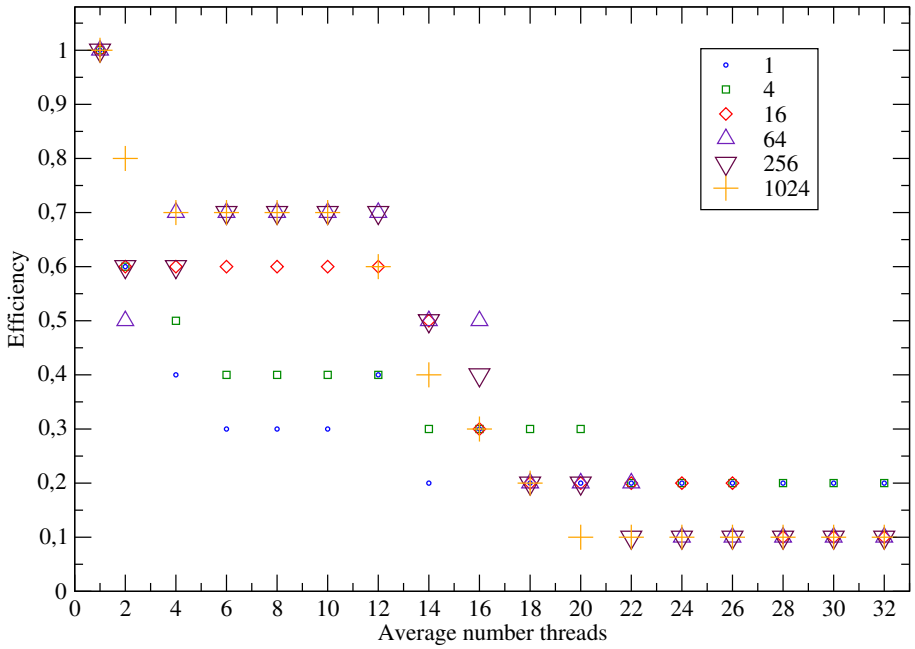


Figura 2.24: Eficiencia obtenida con *malloc*() para 1, 4, 16, 64, 256 y 1024 rayos por bloque.

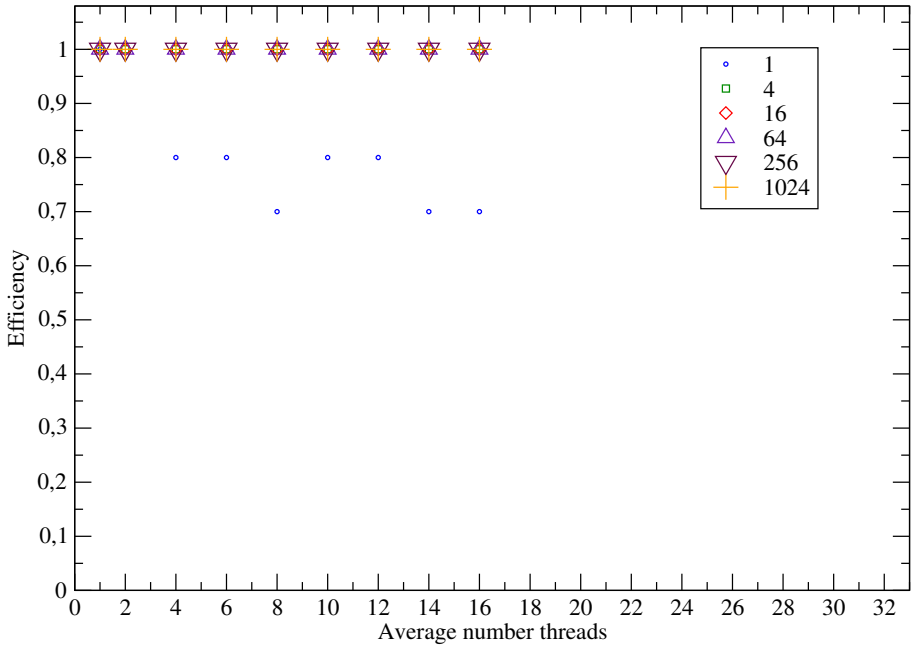


Figura 2.25: Eficiencia obtenida con *thread_alloc*() para 1, 4, 16, 64, 256 y 1024 rayos por bloque.

Algoritmo 2.8.3 : Benchmark sintético basado en *ray tracing* con reserva de memoria dinámica por rayo

```
(1)  funct create_image(work_load, num_rays)
(2)    var ray, block, point, color, decision, image, half_image;
(3)    IGP_Begin_Thread(work_load);                          Fase de Inicialización
(4)    image = memory_allocation(work_load);                 Reservar memoria para la imágen
(5)    while (work_load ≠ {∅})
(6)          block[++ray] = memory_allocation(1);            Reservar memoria para un bloque/rayo
(7)          point = choose_point(work_load);                Comienza el trabajo computacional
(8)          block[ray] = create_ray(point);
(9)          color = compute_intersection_ray_sphere(block[ray]);
(10)         image = paint_image(point, color);              Termina el trabajo computacional
(11)         decision = IGP_get(work_load_i);                Fases de Información y Notificación
(12)         if (decision = new)
(13)           half_image = divide(work_load_i);
(14)           create_thread(Thread, half_image);
(15)         if (ray = num_rays)
(16)           while (!ray − −)
(17)                 free(block[ray]);                       Liberar la reserva de memoria para el bloque
(18) send_results(image);
(19) free(image);                                            Liberar la reserva de memoria para la imágen
(20) IGP_End_Thread();                                       Fase de Terminación
```

zan n reservas dinámicas de memoria con la función *malloc*() y con la librería *ThreadAlloc*, respectivamente. La Figura 2.24 muestra que la eficiencia obtenida con la función *malloc*() es baja para cualquier número de reservas por bloque, mientras que la Figura 2.25 muestra unos buenos niveles de eficiencia de la librería *ThreadAlloc* para todos los casos. Esto es debido a que la función *malloc*() ejecuta en cada iteración la llamada al sistema *sbrk*()/*brk*(), mientras que *ThreadAlloc* solo lo ejecuta cada 4 KB. Aunque en el experimento se ha realizado con MaxThreads = 32, el GP no ha permitido generar más de 16 hebras con *thread_alloc*(). La justificación se encuentra en que el gestor de hebras se ejecuta en la hebra ociosa del kernel, y en este experimento realizado con *thread_alloc*(), los procesadores no han estado ociosos con 16 hebras, por lo que el GP no se ha podido ejecutar.

2.8.3. Detención de hebras

Todos los experimentos anteriores se han realizado inhabilitando la posibilidad de que el GP detenga hebras en ejecución. Para analizar el impacto que tiene la detención de hebras gestionadas por el GP sobre el rendimiento de una aplicación multihebrada se ha diseñado un experimento que consiste en ejecutar el algoritmo benchmark sobre una escena *spheresflake* de 3$^{\text{er}}$ nivel en un sistema dedicado con 16 procesadores. El algoritmo

multihebrado con creación dinámica de hebras comienza la ejecución pudiendo utilizar todos los procesadores disponibles, sin embargo, para apreciar mejor el comportamiento de la detención de hebras, se ha establecido mediante el cambio de la variable $MaxThreads$ que la aplicación multihebrada se ejecute en la mitad de procesadores, cuando se complete 1/3 de la carga total de trabajo. Se han implementado las dos versiones de detención de hebras, una a nivel de usuario con el gestor ACW (ver Sección 2.3) para diferentes valores de la variable *interval_time*: 10 ms, 100 ms, 1 s y 10 s, y otra a nivel de kernel usando el gestor $KITST$ (ver Sección 2.4).

Las Figuras 2.26 y 2.27 comparan el número de hebras activas durante todo el tiempo de ejecución del benchmark sintético cuando no se permite la detención de hebras (No detention) y cuando si se permite. La detención de hebras a nivel de kernel no necesita ningún parámetro de configuración, pero la detención de hebras a nivel de usuario utiliza la variable *interval_time*. En la Figura 2.26 *interval_time* es igual a 10 ms y en la Figura 2.27 se muestran resultados para 100 ms, 1 s y 10 s. Como se puede observar, la reducción del número máximo de hebras activas, de 16 a 8 hebras, se produce entorno a los 28 s de ejecución. Cuando no está permitida la detención de hebras se produce un tránsito muy lento en el número de hebras activas que se prolonga hasta los 42 s. La duración del tránsito (14 s) sin detención de hebras depende directamente de la carga de trabajo asignada a las hebras, pues la reducción de 16 hebras a 8 hebras se producirá conforme las hebras vayan terminando su trabajo. El GP sin detención de hebras consigue un menor tiempo de ejecución porque se permite que las hebras, que deberían parar su trabajo debido a las restricciones establecidas, siguen trabajando.

En ambas figuras se observa que la detención de hebras gestionada por el GP permite un tránsito de 16 a 8 hebras de forma casi inmediata, independientemente de que se trate de una detención a nivel de usuario o a nivel de kernel, incluso para distintos valores de *interval_time*. En la Figura 2.27 se aprecia un aumento en el tiempo total de ejecución cuando se utilizan valores grandes de *interval_time* (1 s y 10 s) en la detención a nivel de usuario, mientras que no se aprecia una variación significativa en el tiempo total de ejecución para valores pequeños (10 ms y 100 ms). Esto es debido a que aquellas hebras que duermen más tiempo (1 s y 10 s) tardarán más tiempo en completar su carga de trabajo, que aquellas hebras que duermen menos tiempo (10 ms y 100 ms). Finalmente, se ha observado que el comportamiento de la detención a nivel de kernel es muy parecido al de la detención a nivel de usuario con valores pequeños de *interval_time*.

2.9. Conclusiones y trabajo futuro

En este capítulo se han estudiado y establecido las características de un gestor del nivel de paralelismo, así como, cada una de las fases funcionales que deben incluirse en el diseño del GP. En función de la entidad que ejecute las principales fases del GP, se han diseñado distintos GPs clasificados en dos tipologías: gestores a nivel de usuario (ACW y AST) y gestores a nivel de kernel (KST, SST y $KITST$). Las diferencias entre los gestores ACW y AST radícan en la fase de *Información*, según el tipo de información utilizada por el criterio de decisión seleccionado. Sin embargo, los gestores KST, SST y

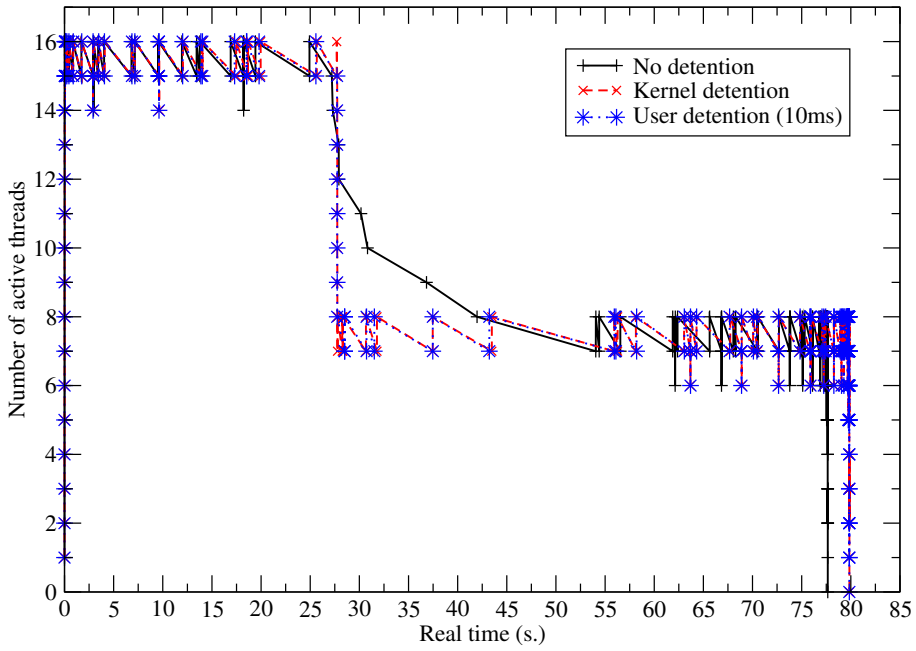


Figura 2.26: Evolución del número de hebras activas: sin detención vs con detención.

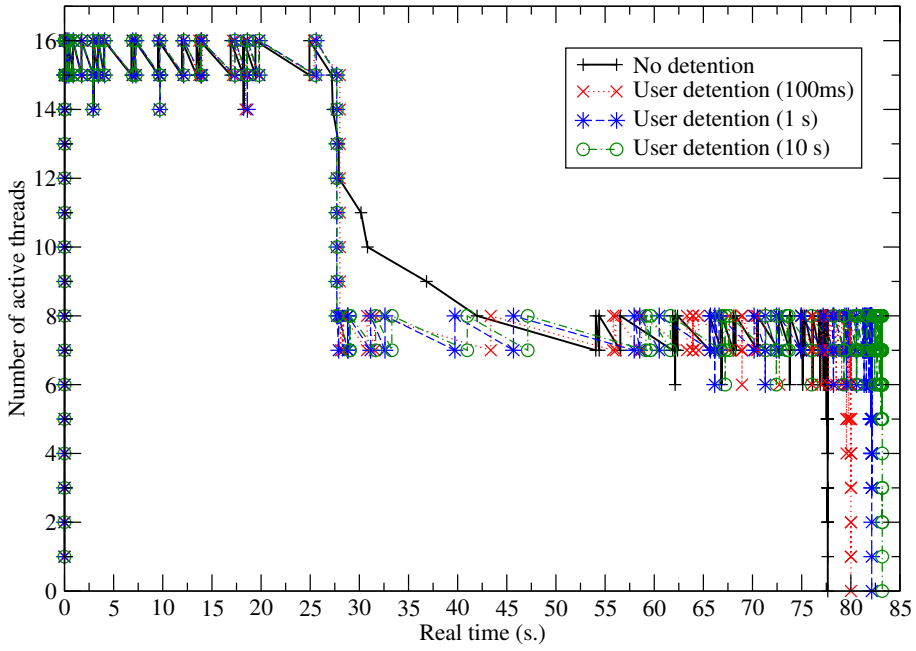


Figura 2.27: Evolución del número de hebras activas: sin detención vs detención a nivel de usuario.

KITST difieren según las entidades que ejecuten las fases de *Información* y *Evaluación*. Los programadores de aplicaciones multihebradas pueden utilizar cualquiera de los gestores implementados haciendo uso de la librería *IGP* descrita en este capítulo.

En [62] se descarta la posibilidad de utilizar gestores a nivel de usuario frente a gestores a nivel de kernel, ya que los gestores a nivel de usuario recopilan la información relativa al estado de todas las hebras activas, disponible en el directorio */proc/[pid]/tasks* del SO Linux. Sin embargo, la lentitud de este mecanismo aumenta con el número de hebras activas, debido a que el SO actualiza la información cada vez que se lee el pseudo fichero. Sin embargo, el gestor a nivel de kernel permite a la aplicación utilizar una llamada al sistema para informar al kernel sobre la carga de trabajo ejecutada por cada hebra.

A continuación, se proponen varias líneas que se podrían estudiar en un futuro:

2.9.1. Creación estática de hebras autoadaptables

Los experimentos se han ensayado sobre una aplicación multihebrada altamente escalable, basada en el algoritmo *Ray Tracing*, constatando que el gestor *AST* es el que permite una menor adaptabilidad de las aplicaciones multihebradas a las condiciones del entorno, debido principalmente al excesivo uso de los recursos computacionales. Por otro lado, todos los gestores a nivel de kernel mejoran la adaptabilidad obtenida con el gestor *ACW*. *KITST* destaca como el mejor gestor de todos gracias a su baja carga computacional.

Todos los experimentos se han realizado sobre una aplicación diseñada con creación dinámica de hebras. Sin embargo, los gestores propuestos pueden ser fácilmente utilizados en aplicaciones multihebradas basadas en la creación estática de hebras, ya que la detención de hebras puede ser beneficiosa en este tipo de algoritmos. Se propone como trabajo futuro estudiar la detención de hebras con los distintos GP sobre aplicaciones multihebradas con creación estática de hebras.

2.9.2. Automatizar el intervalo de ejecución (λ)

Uno de los parámetros de configuración en el GP establece el intervalo de tiempo (λ) entre dos análisis consecutivos del criterio de selección. La adaptabilidad del GP depende directamente de λ y los experimentos realizados han verificado que seleccionar un valor de λ lo más pequeño como sea posible permite una mayor adaptabilidad de las aplicaciones multihebradas. Sin embargo, realmente el programador es el responsable de establecer λ en función del número de iteraciones del bucle computacional. Para aumentar la adaptabilidad del GP, se ha establecido que cada una de las hebras informe al GP en cada iteración, de tal forma, que el intervalo λ depende del tiempo que tardan todas las hebras activas en ejecutar una iteración. El tiempo computacional que una hebra tarda en ejecutar una iteración depende directamente de las características intrínsecas de la aplicación multihebrada. En este sentido, nos podemos encontrar con aplicaciones con tiempos por iteración muy dispares, es decir, iteraciones que se computan muy rápidamente, frente a otras iteraciones cuyo tiempo de computo es muy superior. Esta heterogeneidad en el tiempo computacional por iteración puede influir en el valor de λ, pues el valor de λ estará limitado por el tiempo de computo de la iteración más lenta. Por lo tanto, mientras una

hebra ejecuta la iteración más lenta, el resto de hebras pueden ejecutar varias iteraciones en el mismo intervalo de tiempo.

Como trabajo futuro, sería interesante conocer el trabajo real asignado a una unidad de trabajo para que cada hebra informe al GP, ya que en nuestros experimentos se ha ensayado siempre con una iteración por hebra. Sin embargo, una hebra podría adaptar el número de iteraciones para informar al GP en función del tiempo de computación que consume cada iteración. Por ejemplo, si la computación de una iteración consume mucho tiempo, se recomienda informar al GP en cada iteración, pero si por el contrario, el tiempo de computación de una iteración es pequeño, lo razonable sería informar al GP transcurridos n iteraciones. Seria interesante diseñar algún mecanismo que automatice este hecho en la fase de Información, y evitar al programador tener que conocer este aspecto.

Capítulo 3

Criterios de decisión adaptativos

El criterio de decisión seleccionado por los distintos GPs analizados en el capítulo anterior se basa en el número de procesadores del sistema. Este capítulo comienza proponiendo una versión que se adapta mejor a la carga del sistema y que se basa en el número de procesadores ociosos (Sección 3.2). También, se describen otros criterios que han sido añadidos en los GPs diseñados, como por ejemplo, el criterio de decisión basado en analizar el rendimiento actual de la aplicación para gestionar la creación/detención de hebras (Sección 3.3). También se estudia un amplio abanico de posibles criterios de decisión basados en los retardos que sufren las hebras de la aplicación (Sección 3.4). Estos criterios de decisión son evaluados experimentalmente, haciendo uso de los distintos GPs descritos en el capítulo anterior, sobre una aplicación de ramificación y acotación (Algoritmo 3.5). Se ha estudiado el comportamiento de este tipo de aplicaciones multihebradas que se caracterizan por hacer reserva dinámica de la memoria para resolver distintas instancias de problemas de prueba estándar en Optimización Global. Más concretamente, se analiza el impacto de realizar reservas dinámicas de memoria con diferentes tamaños de bloque en la aplicación Local-PAMIGO (Sección 3.6). Finalmente, se analiza la detención de hebras gestionadas por el GP para mejorar la adaptación de la aplicación multihebrada en sistemas dedicados (Sección 3.7).

3.1. Introducción

Los distintos gestores del nivel de paralelismo (GP) planteados en el capítulo anterior permiten establecer el número de hebras activas de la aplicación multihebrada en tiempo de ejecución. La elección de un criterio de decisión adecuado influye directamente en el nivel de adaptación de la aplicación multihebrada, ya que la continua creación y detención de hebras permitirá a la aplicación multihebrada adaptarse en mayor o menor medida a los recursos disponibles en el sistema y para mantener una alta eficiencia. Los distintos factores que pueden influir en la disponibilidad de recursos y la eficiencia de la aplicación multihebrada son:

- La arquitectura del sistema, donde influyen características como: el número de procesadores disponibles, tamaño y organización de la memoria del sistema, gestión y

planificador del SO, etc.

- Las características intrinsecas de la aplicación que vienen impuestas por: las secciones críticas, reserva y acceso a memoria dínamica, tamaño y complejidad de la carga de trabajo computacional, librerías utilizadas, etc.
- La concurrencia de otras aplicaciones es especialmente relevante en sistemas no dedicados, aunque también se debe tener en cuenta en sistemas dedicados, debido a que la ejecución de tareas propias del SO puede interferir con la ejecución de la aplicación multihebrada.

En este sentido, es importante diseñar un criterio eficiente que facilite la toma de decisiones al GP, de tal forma, que la aplicación multihebrada se adapte dinámicamente a las distintas condiciones de ejecución. A continuación se procede a estudiar el criterio de decisión basado en generar tantas hebras como número de procesadores ociosos en el sistema.

3.2. Número de procesadores ociosos

En el capítulo anterior, se ha usado la variable $MaxThreads$ para adaptar el número de hebras activas al número de procesadores del sistema, independientemente de que las unidades de procesamiento estén ociosas o no (ver Sección 2.2). En [65] son las hebras las que se auto-asignan a los procesadores. Un criterio más adaptativo puede basarse en permitir a la aplicación multihebrada crear una nueva hebra cuando se detecta la existencia de un procesador ocioso. Los GPs basados en este criterio asumen que el planificador del sistema operativo tiene la capacidad de rebalancear las tareas de los procesadores (*thread-stealing*) moviendo hebras de los procesadores más cargados a los menos cargados u ociosos.

Este criterio de decisión es adecuado para aplicaciones multihebradas HPC escalables y con una carga computacional que permita tener activos a todos los procesadores del sistema. Sin embargo, la existencia de un procesador ocioso puede producirse por la ausencia de tareas a ejecutar, o también por los bloqueos que experimentan las hebras al entrar en las secciones críticas. Si se permite a la aplicación crear nuevas hebras cuando existen procesadores ociosos, se puede incrementar el número de bloqueos por secciones críticas, lo que a su vez se traduce en la existencia de más procesadores ociosos. Este incremento del número de hebras activas, muy superior al número de procesadores disponibles, suele incrementar el tiempo total de ejecución de la aplicación, por lo que el speed-up disminuye. Generalmente, el criterio de decisión basado en el número de procesadores ociosos está especialmente indicado para aplicaciones HPC altamente escalables en entornos dedicados.

El número total de procesadores disponibles en el sistema se puede obtener desde los GPs a nivel de usuario contabilizando el número de veces que aparece el campo *processor* en el fichero */proc/cpuinfo*. Mientras que los GPs a nivel de Kernel pueden obtener directamente esta información leyendo la variable *total_cpus* declarada en el fichero */drivers/base/cpu.c* del código fuente del Kernel. Sin embargo, el número de procesadores físicos es diferente del número de procesadores ociosos. El numero de procesadores ociosos es más difícil de conocer, a nivel de usuario, especialmente en sistemas no dedicados.

Por este motivo, resulta más facil para los GPs a nivel de kernel detectar periódicamente el número de procesadores ociosos a través de la función *idle_cpu*(*id_cpu*) definida en el fichero *kernel/sched/core.c*. Mientras que los GPs a nivel de usuario deben extraer esta información del fichero */proc/stat* donde se muestran periódicamente las estadísticas del sistema desde que fue reiniciado. En el fichero */proc/stat* del SO linux aparece una línea por cada procesador del sistema con la variable *cpu*, precedida por siete columnas numéricas en las que se mide el número de instantes (1/100 de segundos para sistemas x86) en los que el sistema ha estado en modo usuario, modo usuario con prioridad baja (*nice*), modo de sistema, tareas ociosas, esperas de E/S, IRQ (*hardirq* y *softirq*), respectívamente. La cuarta columna (tareas ociosas) muestra el número de instantes que el procesador ha estado ejecutando la hebra ociosa del kernel. En este sentido, el gestor de hebras puede detectar si un procesador ha estado ocioso, comparando el número de instantes que el procesador i ha estado ocioso (cuarta columna), con la medida obtenida en el chequeo anterior.

Uno de los principales inconvenientes de los GP que usan este tipo de decisión es que no son capaces de conocer el motivo de la ociosidad del procesador detectado, que puede ser debido a la ausencia de carga de trabajo en el sistema o a características intrinsecas de la aplicación. Por lo tanto, para diferentes tipos de situaciones se deberían tomar diferentes acciones. Si el motivo es la ausencia de tareas a ejecutar, se debería crear una nueva hebra, pero si el motivo es, por ejemplo, por bloqueos de las hebras al acceder a un sección crítica, no se debería crear una nueva hebra. Además, este críterio de decisión no tiene en cuenta la detención de hebras, lo que limita su nivel de adaptabilidad en sistemas no dedicados. Este criterio especifica que se debe crear una hebra cuando exista un procesador ocioso, pero no establece los motivos para detener una hebra cuando no existe un procesador ocioso.

El usuario normalmente establece el número de hebras igual al numero de unidades de proceso en el sistema. Sin embargo, existen aplicaciones multihebradas cuyo grado de escalabilidad depende de la cantidad de trabajo asignado. Este es el caso de las aplicaciones multihebras B&B, donde su nivel de escalabilidad depende del problema específico de Optimización Global a resolver [40]. En este caso puede ocurrir que el criterio de decisión basado en el número de procesadores ociosos permita obtener con la misma aplicación y arquitectura buenos niveles de eficiencia para un tipo de problema, y no tan buenos para otros problemas o funciones objetivo en algoritmos de Optimización Global que hacen uso de métodos de B&B.

3.3. Rendimiento de la aplicación

Una estrategia más centrada en la aplicación, e indirectamente relacionada con los recursos disponibles, puede basarse en realizar diferentes ejecuciones de la aplicación para cada problema a resolver con distintos números de hebras y establecer el grado de escalabilidad. De esta forma el usuario puede conocer el número óptimo de hebras que permita alcanzar el máximo rendimiento (véase Ecuación 1.2). Sin embargo, esta estrategia es poco portable, pues depende de la arquitectura del sistema, del entorno de ejecución y de

la carga de trabajo de la aplicación, y además, requiere de un preprocesamiento previo que consume recursos. Para evitar este preprocesamiento previo, se propone en esta tesis realizar medidas de rendimiento parciales ($P_i(n)$) en tiempo de ejecución, de tal forma que el rendimiento de la aplicación ($P(n)$) es un vector que almacena el promedio de los rendimientos parciales medidos en cada intervalo de tiempo en el que la ejecución se ha realizado con n hebras. Cada n podrá tener un número m de medidas de rendimientos parciales diferentes, por lo tanto el rendimiento de la aplicación se define como:

$$P(n) = \frac{\sum_{i=1}^{m} P_i(n)}{m}, \tag{3.1}$$

Basándonos en el análisis dinámico del rendimiento de la aplicación, se establecerán criterios de decisión para la creación o detención de hebras, basados en si se mantiene o no el rendimiento parcial de la aplicación multihebrada por encima de un umbral previamente establecido. Este criterio de decisión explota el hecho de que todas las hebras ejecutan el mismo bucle computacional para distintos elementos del trabajo asignado. El rendimiento parcial de la aplicación se mide en terminos del trabajo realizado por las hebras activas por unidad de tiempo. Más concretamente se define:

$$P_i(n) = \frac{\sum_{j=1}^{n} NIter_j}{\lambda_i \cdot n}, \tag{3.2}$$

como el rendimiento parcial en el intervalo de tiempo λ_i de la aplicación con n hebras activas, donde $NIter_j$ es el trabajo completado por la hebra j en el último intervalo de tiempo y λ_i es el tiempo mínimo necesario para que todas las hebras realicen al menos una unidad de trabajo (iteración). El gestor podría decidir incrementar el valor de λ_i si este es muy pequeño, pero ese estudio se deja para trabajos futuros. Una vez definido el rendimiento instantáneo de una aplicación, se procede a definir el criterio de decisión:

Definición 3.3.1 *Decisión incremental: Determina bajo que condiciones puede crearse una hebra de la aplicación. La decisión es afirmativa si y solo si se cumple la siguiente desigualdad:*

$$\frac{P(n)}{P(n-1)} \geq Threshold \tag{3.3}$$

Midiendo $P(n)$ dentro de un intervalo de tiempo (λ_i) en el que todas las hebras han realizado alguna unidad de trabajo (iteración).

En este criterio de decisión es el usuario el responsable de establecer el valor de $Threshold$ como parámetro de entrada al GP. Si se satisface la decisión para un valor de $Threshold = 1$ significa que el rendimiento se mantiene o se mejora cuando se crea una nueva hebra. Si se establece $Threshold > 1$ para n hebras, entonces el rendimiento instantáneo debe mejorar respecto al rendimiento instantáneo usando una hebra menos $(n-1)$. Si se establece un valor $Threshold < 1$, el rendimiento instantáneo de la aplicación podría disminuir. La decisión de crear nuevas hebras se toma cada vez que se verifique la Ecuación 3.3, en caso contrario, se puede decidir que se detenga alguna hebra.

El GP que implemente este criterio de decisión debe conocer el trabajo completado por cada hebra y poder determinar el intervalo de tiempo λ (Fase de *Información*). Una decisión valida resetea todos los contadores utilizados, antes de pasar a analizar la próxima decisión en el siguiente intervalo de tiempo (Fase de *Restablecimiento*). En aquellas aplicaciones donde el tiempo requerido para resolver una unidad de trabajo sea muy pequeño, el GP evaluará constantemente el criterio de decisión ya que el valor de λ es muy pequeño. En estas situaciones se recomienda aumentar el valor de λ. Para ello la aplicación tiene que informar al GP cada vez que se realizen n unidades de trabajo (iteraciones), en vez de informar en cada iteración. El programador debe establecer en su implementación el número de unidades de trabajo adecuado a realizar antes de informar al gestor, incrementando así el valor de λ.

Este criterio de decisión basado en el rendimiento de la aplicación almacena en un vector ($P(i)$), con los últimos valores del rendimiento instantáneo para $i = 1, \ldots, n$, calculados con la Ecuación 3.2. Estos valores serán utilizados en las comparaciones posteriores. El valor de n disminuye cuando una hebra finaliza su trabajo asignado o cuando es detenida por el GP. El gestor de hebras solo tiene en cuenta la información relacionada con las n hebras en ejecución, calcula su rendimiento almacenandolo en $P(n)$ y usa el almacenado en $P(n-1)$, usando una hebra menos $(n-1)$, para calcular la Ecuación 3.3.

La principal característica de este criterio de decisión es que todos los parámetros necesarios para realizar la evaluación ($NIter_i$, λ, n y $Threshold$) están disponibles a nivel de usuario, y no se necesita ningún parámetro adicional del SO. El intervalo de tiempo (λ) puede influir en la toma de decisiones al aplicar este criterio, debido principalmente a que un valor de λ grande repercute en que se toma la decisión en función del rendimiento instantáneo obtenido con un número mayor de iteraciones del bucle, como se observó en la Sección 2.7.

Finalmente, indicar que este criterio de decisión también tiene en cuenta, indirectamente, tanto los recursos disponibles, como el resto de aplicaciones ejecutadas concurrentemente en entornos no dedicados, debido a que λ aumenta no solo cuando aumenta la complejidad de la carga de trabajo, sino también cuando escasean los recursos disponibles o aumenta el número de aplicaciones.

3.4. Retardos de la aplicación

El rendimiento de la aplicación multihebrada está directamente relacionado con los retardos que experimentan las hebras durante su ejecución. A lo largo de esta sección se analizarán los aspectos más importantes que influyen en el rendimiento de la aplicación, centrándonos en los retardos más comunes experimentados por las aplicaciones multihebradas. Previamente, se necesita describir los diferentes estados por los que puede pasar una tarea (proceso o hebra) durante su ciclo de vida. En el Sistema Operativo Linux todas las tareas creadas tienen una entrada en la tabla de procesos con la infomación (control, pila, código y datos) relativa al uso y control del proceso o hebra.

Desde el instante en que una tarea es creada, por ejemplo con fork(), hasta que completa todo su trabajo y desaparece de la tabla de procesos, atraviesa diferentes estados. El estado

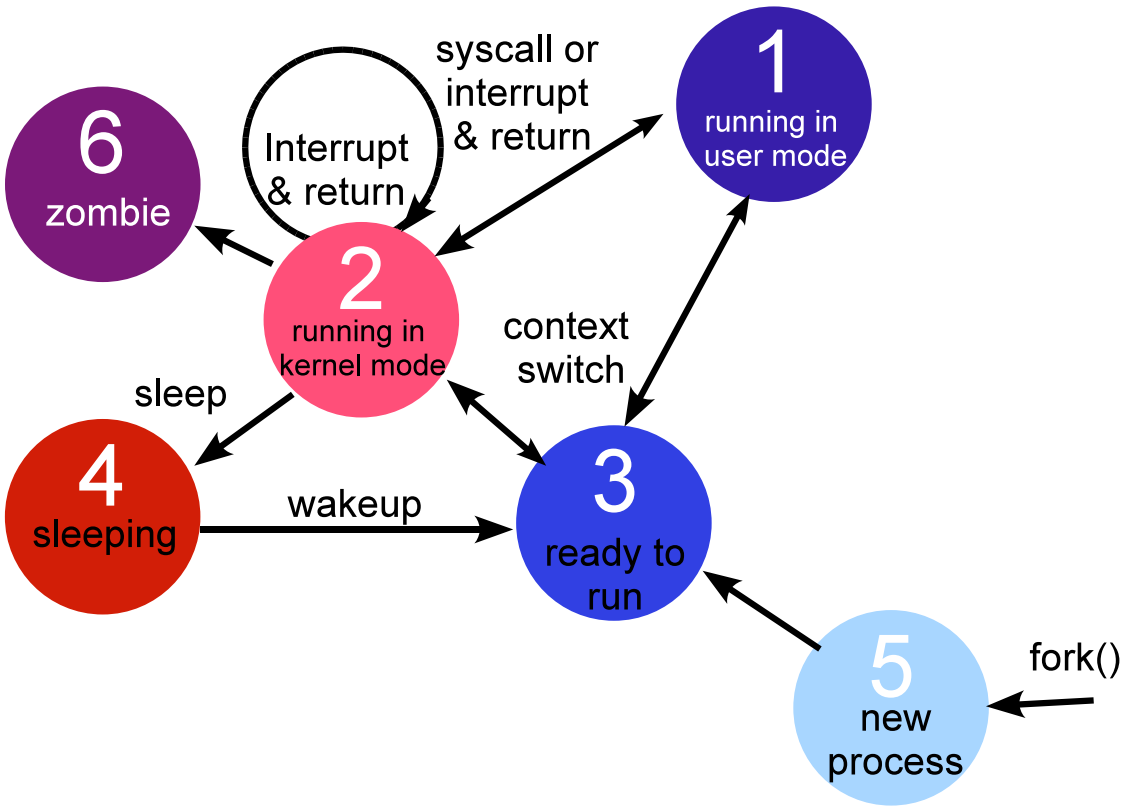


Figura 3.1: Diagrama de estados y transiciones de una tarea en Linux.

de una tarea cambia muchas veces durante su ciclo de vida. Estos cambios pueden ocurrir, por ejemplo, cuando el proceso o hebra ejecuta una llamada al sistema, se produce una interrupción o solicita recursos que actualmente no están disponibles. De hecho, en el fichero *sched.h* del código fuente del Kernel de Linux se muestran diferentes modos de operación por los que puede pasar una tarea:

1. Ejecución en modo usuario.
2. Ejecución en modo kernel.
3. Preparado para ejecutarse.
4. Durmiendo.
5. Recientemente creado, no está preparado para ejecutarse, ni está durmiendo.
6. Saliendo de una llamada al sistema (zombie).

Las transiciones entre los estados de las tareas, así como sus causas, se pueden observar en la Figura 3.1. Cualquier tarea creada entra en el sistema en el estado 5. Si la tarea es una copia de un proceso padre, es decir, ha sido creado con $fork()$, entonces comienza su ejecución en el estado en el que estaba el proceso padre, es decir, en el estado 1 ó 2. Sin embargo, si la tarea ha sido creada con $exec()$, entonces esta tarea siempre terminará en modo kernel (estado 2). Aunque es posible que el $fork()$ o $exec()$ se haya realizado en modo kernel, y la tarea pase a estado 1 durante su tiempo de vida.

Cuando un proceso está en ejecución, se puede recibir una interrupción, por lo que el proceso actualmente en ejecución se mantiene en memoria preparado para ejercutarse (estado 3).

Cuando un proceso en modo usuario (estado 1) ejecuta una llamada al sistema, pasa al estado 2 donde se ejecuta en modo kernel. Por ejemplo, si la llamada al sistema es para leer un fichero del disco, esta lectura no se realizará inmediatamente, así que el proceso pasa a dormir, cediendo voluntariamente el procesador y esperando el evento del sistema que lee el disco y prepara los datos (estado 4). Cuando el dato está disponible, el proceso se despierta. Esto no significa que el proceso se ejecute inmediatamente, sino que permanece preparado para ejecutarse en memoria principal (estado 3).

Un proceso termina explicitamente ejecutando la llamada al sistema $exit()$, que se encarga de eliminar todas las estructuras de datos utilizadas por el proceso, a excepción de la entrada en la tabla de tareas, que es responsabilidad del proceso *init*.

Estos modos de operación indican conceptualmente las actividades que está realizando una tarea, pero no indican el estado en que se encuentran. A continuación, resumimos brevemente cada uno de los estados en los que una tarea puede estar:

***RUNNING*:** tarea en ejecución. La tarea esta lista para ejecutarse, y por tanto se encuentra en la cola de ejecución asignada a un procesador. En este estado se encuentran las tareas activas, es decir, aquellas tareas que realmente estan ejecutandose en el procesador asignado, o bien pueden estar esperando su turno de ejecución. El turno de ejecución viene establecido por el planificador del SO en función de la prioridad dinámica asignada a cada tarea.

***INTERRUPTABLE*:** tarea durmiendo pero puede ser despertada. La tarea está suspendida esperando alguno de los siguientes eventos: interrupción hardware, liberación de algún recurso, a que se cumpla alguna condición, o interrupcción generada por un contador. Cuando se resuelve el evento que la bloquea, la tarea será despertada y pasará al estado *RUNNING*. Sin embargo, también puede ser despertada de forma prematura a pesar de no haber ocurrido el evento esperado, si dicha tarea recibe alguna señal (por ejemplo, SIGTERM). Un típico ejemplo son los procesos interactivos que suelen permanecer en este estado cuando están suspendidos, esperando la intervención del usuario.

***UNINTERRUPTABLE*:** tarea está durmiendo pero no puede ser interrumpida. Este estado es similar al caso anterior, exceptuando que la tarea no sería despertada si recibe alguna señal.

***ZOMBIE*:** tarea que ha completado su ejecución y no espera ningún evento, pero aún mantiene una entrada en la tabla de procesos del SO. La tarea ha finalizado su ejecución, pero su descriptor sigue en memoria para permitir que el proceso creador (padre) pueda acceder a información sobre su ejecución. Dicho descriptor será eliminado cuando el padre ejecute la llamada al sistema *wait4*(), o similares.

***TRACED*:** Estado para dar soporte a la depuración. El proceso está siendo depurado (por ejemplo, *breakpoint*).

***STOPPED*:** tarea detenida por el kernel. La ejecución del proceso ha sido detenida de forma externa (por ejemplo, al teclear Control+Z desde el shell durante la ejecución de un proceso en foreground). El proceso queda bloqueado al recibir alguna de las señales SIGSTOP, SIGSTP, SIGTTIN o SIGTTOU. Dicho proceso puede volver a un estado ejecutable cuando reciba otras señales.

***GP-STOPPED*:** tarea detenida por el GP cuando se ha habilitado la detención de hebras.

***DEAD*:** La tarea ha acabado y su descriptor se ha eliminado.

Cuando una aplicación multihebrada entra en ejecución, las hebras suelen conmutar entre los estados *RUNNING*, *INTERRUPTIBLE* y/o *UNINTERRUPTIBLE*. El rendimiento de cualquier aplicación está directamente relacionada con el tiempo que las hebras permanecen en estos estados, de tal forma que, el rendimiento disminuye rápidamente si las hebras consumen mucho tiempo en los estados *INTERRUPTIBLE* y/o *UNINTERRUPTIBLE*. Por lo tanto, se consigue el menor tiempo de ejecución si ninguna de las hebras de la aplicación multihebrada han pasado por *INTERRUPTIBLE*, ni *UNINTERRUPTIBLE*, permaneciendo todas las hebras en el estado *RUNNING*.

Diferentes factores pueden influir en la ejecución de una hebra, por ejemplo: interbloqueos entre hebras debido al acceso secuencial a secciones críticas, en operaciones de entrada/salida, reserva dínamica de memoria, fallos de cache o simplemente la espera en la cola de ejecución de un procesador. A continuación, se detallan los distintos tiempos que influyen en el retardo de la aplicación:

***BT*:** Tiempo de bloqueo, corresponde al tiempo total que una tarea permanece bloqueada, también se conoce como *sleeping time*. Este tiempo es la suma de dos tipos de bloqueo diferentes:

- ***IBT*:** Tiempo de bloqueo interrumpible, corresponde al tiempo que una tarea permanece en el estado *INTERRUPTIBLE*, es decir,
  - cuando una hebra se bloquea en secciones críticas, o
  - cuando una hebra realiza una operación de entrada/salida. En algunas aplicaciones HPC, las hebras computacionales no suelen realizar operaciones de entrada/salida significativas, por lo que el tiempo *IBT* suele ser debido exclusivamente a las secciones críticas.

***NIBT*:** Tiempo de bloqueo no interrumpible, corresponde al tiempo que una hebra permanece en el estado *UNINTERRUPTIBLE*, es decir,

- cuando se reserva memoria dinámicamente, o
- cuando una hebra accede a una dirección de memoria, y se produce un fallo de caché o de página.

***WT*:** Tiempo de espera, corresponde al tiempo que una hebra espera en la cola de ejecución (estado *RUNNING*), hasta que se le asigne la unidad de procesamiento. Este tiempo aumenta cuando existen varias tareas en la cola de ejecución de la misma unidad de procesamiento. Las tareas activas pueden ser otras aplicaciones, o incluso otras hebras de la misma aplicación, que se ejecutan concurrentemente en el mismo procesador.

El grado de escalabilidad de cada aplicación multihebrada viene establecido por el valor de BT, de tal forma que, aquellas aplicaciones cuyo BT sea cero, se pueden considerar aplicaciones altamente escalables, es decir, en teoría se consiguiría el máximo rendimiento cuando existan infinitas hebras activas, si los recursos son ilimitados. Sin embargo, la mayoria de las aplicaciones multihebradas se caracterizan por tener un valor de $BT \neq 0$ pues suelen presentar alguna o varias de las siguientes características:

1. Comunicación de datos entre las hebras activas.
2. Ejecución secuencial en secciones críticas.
3. Reserva de memoria dinámica.
4. Fallos de cache debido a la gran cantidad de datos a procesar, de tal forma que la memoria cache es insuficiente.

Una vez definidos los distintos retardos que puede experimentar una hebra en una aplicación multihebrada, se van a analizar diversos criterios de decisión que permiten minimizar alguno o varios de estos retardos.

3.4.1. Minimizar los retardos de la aplicación

El tiempo total de una aplicación incluye los retardos que surgen durante la ejecución. Diferentes factores intrínsecos a la implementación de la aplicación y al propio SO crean retardos no deseados que incrementan el tiempo total de la aplicación. Idealmente, las aplicaciones se deberían diseñar de tal forma que los retardos sean inexistentes y que por lo tanto el tiempo total de ejecución se dedique a la resolución del problema. Un primer criterio de decisión muy restrictivo basado en los retardos de la aplicación, se basa en permitir la creación de hebras siempre que no se detecte algún retardo en las distintas hebras de la aplicación. A continuación se describen los distintos criterios de decisión que se van a analizar.

Minimizar el tiempo en estados no ejecutable (*MNET*): El tiempo no ejecutable se define como la suma de tiempos que una hebra ha permanecido en estados *INTERRUPTABLE* y *UNINTERRUPTABLE*, más el tiempo que la hebra espera en estado

RUNNING para el uso de la unidad de procesamiento. Este es el criterio más restrictivo de todos, pues solo se permite crear una nueva hebra, si se comprueba que el tiempo no ejecutable en el intervalo λ de cada una de las hebras activas es cero. En este caso, la condición imprescindible para crear una nueva hebra es que todas las hebras activas deben tener un tiempo de bloqueo y tiempo de espera igual a cero, es decir, $BT = 0$ y $WT = 0$.

Minimizar el tiempo en estado dormido (*MST*): El tiempo dormido se define como la suma de tiempos que una hebra ha permanecido en estados *INTERRUPTABLE* y *UNINTERRUPTABLE*. Solo se permite crear una nueva hebra, si se comprueba que el tiempo dormido en el intervalo λ de cada una de las hebras activas es cero, es decir, $BT = 0$. Este criterio de decisión es menos restrictivo que el anterior pues no tiene en cuenta el tiempo de espera de las hebras para entrar en ejecución, por lo que se permite la existencia de varias tareas asignadas al mismo procesador.

Minimizar el tiempo de bloqueo interrumpible (*MIBT*): Solo se permite crear una nueva hebra, si se comprueba que el tiempo de bloqueo interrumpible en el intervalo λ de cada una de las hebras activas es cero, es decir, $IBT = 0$. En este caso solo se tienen en cuenta los interbloqueos entre hebras producidos por las secciones críticas y las operaciones de entrada/salida.

Minimizar el tiempo de bloqueo no interrumpible (*MNIBT*): Solo se permite crear una nueva hebra, si se comprueba que el tiempo de bloqueo no interrumpible en el intervalo λ de cada una de las hebras activas es cero, es decir, $NIBT = 0$. En este caso solo se tiene en cuenta los retardos provocados cuando se accede a memoria.

Minimizar el tiempo de espera (*MWT*): Solo se permite crear una nueva hebra, si se comprueba que el tiempo de espera en el intervalo λ de cada una de las hebras activas es cero, es decir, $WT = 0$. En este caso, se crearan nuevas hebras solo si todas las hebras existentes están en ejecución en el procesador asignado sin esperar en la cola de ejecución.

De entre los cinco criterios de decisión planteados, el criterio *MWT* está directamente relacionado con el rebalanceo de las tareas entre procesadores, gestionado por el planificador del SO, por lo que este criterio de decisión crearía hebras innecesarias en aplicaciones multihebradas con secciones críticas y/o por accesos a memoria principal, dado que aquellas hebras bloqueadas por alguno de estos motivos pasarían del estado *RUNNING* al estado *INTERRUPTIBLE* y/o *UNINTERRUPTIBLE*, sin repercutir en el tiempo de espera de las hebras. Por otro lado, tampoco podría aplicarse a sistemas no dedicados, donde varias aplicaciones multihebradas son gestionadas por el GP, de tal forma, que el gestor de hebras no sabría diferenciar si el tiempo de espera de una hebra está provocado por una u otra aplicación. Por otro lado, el criterio *MNIBT* no tiene en cuenta los retardos producidos por las secciones críticas, característica muy importante en cualquier aplicación HPC, aunque podría obtener buenos niveles de rendimiento ejecutando aplicaciones multihebradas, sin secciones críticas y en entornos dedicados.

Finalmente, los tres criterios de decisión *MNET*, *MST* y *MIBT*, que son evaluados experimentalmente en este trabajo de tesis, no garantizarán obtener un rendimiento óptimo al ejecutar una aplicación multihebrada. Esto es debido a que el número de hebras permitidas por estos criterios de decisión tan restrictivos siempre será inferior al número de hebras que permitirá obtener un buen nivel de rendimiento en el sistema. Esto se debe a que el rendimiento de una aplicación no se ve afectado hasta que los retardos que experimentan las hebras superan un umbral, por lo que permitir ciertos retardos por debajo de ese umbral puede mantener buenos niveles de rendimiento. Tan solo para aplicaciones multihebradas muy escalables, el rendimiento obtenido con estos criterios de decisión tendría niveles aceptables. La determinación de este umbral es bastante compleja, pues depende de factores relacionados con la aplicación, carga de trabajo y sobrecarga del sistema.

3.4.2. Estimación del número de hebras

Con el objetivo de diseñar otros criterios de decisión menos restrictivos, en esta sección se analizan algunas alternativas que permiten estimar el número óptimo de hebras. En este sentido, el GP basado en los retardos de la aplicación multihebrada necesita estimar un número máximo de hebras (MNT: Max. Number of Threads) que no puede ser superado por la aplicación multihebrada. Este valor variará dinámicamente en función de las condiciones de ejecución en el intervalo de tiempo λ analizado. Todas las métricas propuestas en esta sección, estiman el MNT comparando el intervalo λ con respecto al tiempo de detención de las hebras. Esta comparativa de tiempos reflejaría el número de hebras máximo a ejecutar simultáneamente, de tal forma que si el tiempo de detención de las hebras es superior a λ, MNT será cero. Por lo que esta métrica intenta establecer un compromiso razonable entre λ y el tiempo de detención de las hebras. En este sentido, se podrían plantear diferentes métricas en función de como se defina el tiempo de detención de las hebras:

1. El tiempo de detención de la hebra que menos tiempo ha estado detenida.
2. Como promedio de los tiempos de detención de todas las hebras.
3. El tiempo de detención de la hebra que más tiempo ha estado detenida.

El tiempo de detención de las hebras escogido en este tipo de métrica se basa en el mayor tiempo que una de las hebras ha estado detenida en el intervalo λ (tercera opción), pues este cálculo tiene en cuenta el retardo experimentado por la hebra con mayor tiempo de detención. Además, minimiza el tiempo de computo que requiere el GP para ejecutar el criterio de decisión, ya que se evita realizar operaciones de división (segunda opción). Se descarta la primera opción simplemente porque la hebra con menos retardo no influye directamente en el número de hebras.

A continuación, describimos diferentes métricas para calcular el número máximo de hebras, a partir de los distintos retardos de la aplicación.

- Tiempo de bloqueo (BT): El número máximo de hebras basado en el tiempo de bloqueo de la aplicación (MNT_BT) se calcula a partir del tiempo en el que las hebras

activas no se han estado ejecutando, es decir, el tiempo de bloqueo interrumpible (IBT) más el tiempo de bloqueo no interrumpible ($NIBT$). En este tiempo no se tiene en cuenta el tiempo de espera en la cola de ejecución (WT). Esta estrategia mide el tiempo que todas las hebras activas han tardado en ejecutar al menos una unidad de trabajo por iteración (λ), y lo divide entre el tiempo máximo que una hebra (i) no ha estado ejecutandose ($NIBT_i + IBT_i$), independientemente del motivo. El número máximo de hebras teniendo en cuenta BT se calcula como:

$$MNT\_BT = \frac{\lambda}{\text{máx}\{NIBT_i + IBT_i\}} \tag{3.4}$$

- Tiempo de bloqueo interrumpible (IBT): El número máximo de hebras basado en el tiempo de bloqueo interrumpible de la aplicación multihebrada (MNT_IBT) es calculado a partir del tiempo que duermen las hebras activas debido a secciones críticas y/o operaciones de entrada/salida. Esta estrategia requiere de una fase inicial donde se mide el intervalo de tiempo que una única hebra tarda en ejecutar una unidad de trabajo obtenido como la media de ejecución de N iteraciones (λ_1), dado que en esta fase inicial se garantiza un tiempo de bloqueo nulo producido por las secciones críticas. El número máximo de hebras según IBT se calcula:

$$MNT\_IBT = \frac{\lambda_1}{\text{máx}\{IBT_i\}} \tag{3.5}$$

  donde IBT_i es el tiempo medio de bloqueo de la hebra i de la aplicación por unidad de trabajo, por lo que, máx$\{IBT_i\}$ corresponde al tiempo medio de bloqueo máximo de todas las hebras activas desde el último chequeo.

- Tiempo de bloqueo no interrumpible ($NIBT$): El número máximo de hebras basado en el tiempo de bloqueo no interrumpible de la aplicación (MNT_NIBT) es calculado a partir del tiempo que las hebras activas consumen en accesos a memoria compartida y/o swapping. Esta estrategia no utiliza la fase inicial, tan solo, mide el tiempo que todas las hebras activas han tardado en ejecutar al menos una unidad de trabajo por iteración (λ), y lo divide entre el tiempo máximo que una hebra ha estado bloqueada debido a la reserva de memoria y/o swapping. El número máximo de hebras según $NIBT$ es:

$$MNT\_NIBT = \frac{\lambda}{\text{máx}\{NIBT_i\}} \tag{3.6}$$

  donde NBT_i es el tiempo de bloqueo no interrumpible de la hebra i de la aplicación.

Según estos planteamientos se podrían definir otros criterios de decisión como MNT_WT y MNT_MNET. Sin embargo, se ha descartado su implementación debido a que ambos criterios contabilizan el tiempo de espera (WT), y resultan ineficientes en aplicaciones multihebradas con secciones críticas y accesos a memoria principal, tal como se explicó en la subsección 3.4.1.

3.5. Algoritmos de ramificación y acotación adaptativos

Los algoritmos de ramificación y acotación (B&B) son comúnmente conocidos como métodos de búsqueda exhaustiva donde el problema inicial es dividido sucesivamente en subproblemas, hasta que la solución es encontrada. Estos métodos evitan analizar algunas ramificaciones del arbol de búsqueda donde se sabe que no puede encontrarse la solución. La idea de utilizar la potencia computacional de los actuales procesadores paralelos en algoritmos B&B no es nueva. Muchos investigadores se han esforzado en obtener implementaciones paralelas eficientes con este tipo de algoritmos [5], [13], [15], [16], [23], [26], [27], [40] y [49].

La experimentación realizada se ha llevado a cabo sobre un algoritmo de ramificación y acotación, que son ampliamente usados para realizar una búsqueda exhaustiva de la mejor solución en problemas de optimización. En [13] se adapta el algoritmo paralelo *PAMIGO* (Parallel Advanced Multidimensional Interval analysis Global Optimization [39]) a procesadores multicore de memoria compartida. Las dos versiones (*Global-PAMIGO* y *Local-PAMIGO*) mantienen un número de hebras igual al número de procesadores del sistema de forma estática y dinámica, respectivamente. El indicador de carga de trabajo del algoritmo *PAMIGO* se basa en el número de nodos del árbol de búsqueda almacenados y pendientes de evaluar.

En la versión *Global-PAMIGO* se usa un número estático de hebras, siguiendo un modelo Asynchronous Single Pool (*ASP*), donde hay una sola estructura de datos a la que acceden las hebras de forma asíncrona. Las hebras terminan su ejecución cuando no hay nodos del árbol en la estructura de datos global y ninguna otra hebra tiene trabajo pendiente.

En la versión *Local-PAMIGO* cada hebra tiene su estructura de datos local, por tanto se sigue un modelo Asynchronous Multiple Pool (*AMP*). En la generación dinámica de hebras, para *Local-PAMIGO*, cada hebra que detecta que el número de hebras es menor al número de cores, genera otra hebra y le asigna la mitad de su trabajo. Una hebra termina cuando no tiene más trabajo que realizar, aunque exista trabajo pendiente en otras hebras. Como puede observarse, en ambas versiones, tanto en la generación estática de hebras (*Global-PAMIGO*), como en la generación dinámica de hebras (*Local-PAMIGO*), es la aplicación la que decide el número de hebras sin interactuar con el SO, ya que es el usuario quién establece el número máximo de hebras, estáticas o dinámicas de la aplicación.

El Algoritmo 3.5.1 muestra el esqueleto del proceso principal de la aplicación *Local-PAMIGO*, que recibe varios parámetros de entrada: la función que determina la instancia del problema a resolver (*id_function*), tamaño mínimo del intervalo de búsqueda o precisión (ϵ) entre otros parámetros. Básicamente, el proceso principal de *Local-PAMIGO* se encarga de: reservar memoria para la función seleccionada (línea 4), inicializar el espacio de definición de la función seleccionada y el árbol de búsqueda de la solución (línea 5), y crear una única hebra a la que se le asigna toda la carga de trabajo (línea 6). El proceso principal se bloquea esperando la finalización del trabajo (línea 7), por lo tanto durante la mayor parte del tiempo de ejecución no consume recursos. Una vez finalizada la ejecución del trabajo realizado por las hebras dinámicas, muestra las soluciones encontradas y las estadísticas sobre el proceso de búsqueda (línea 9). Esta versión de *Local-PAMIGO* incor-

Algoritmo 3.5.1 : Proceso principal de Local-PAMIGO gestionado por el GP.

```
(1) funct local-PAMIGO(id_function, ε, other_parameters)
(2)   IGP_Initialize(M, C, P, T, true);                    Fase de Inicialización
(3)   task = choose_function(id_function);
(4)   global_work_load = memory_allocator(task);           Reserva memoria
(5)   search_tree(Initialize);                             Inicializa el intervalo de búsqueda
(6)   create_thread(global_optimice, global_work_load);
(7)   wait_results_threads();                              Esperar a que todas las hebras terminen
(8)   IGP_Finalize();                                      Fase de Terminación
(9)   show_minimum();                                      Mostrar resultado final y estadísticas
```

pora las funciones de la librería *IGP* (línea 2 y 8) para ser gestionada por el GP a nivel de Kernel (veasé la Sección 2.4).

Algoritmo 3.5.2 : Hebra de Local-PAMIGO.

```
(1) funct global_optimice(work_load)
(2)   IGP_Begin_Thread(work_load);                         Fase de Inicialización
(3)   while (work_load ≠ {∅})
(4)         evaluate_search_interval();
(5)         divide_search_interval();
(6)         update_local_search_tree();
(7)         decision = IGP_get(work_load_i);               Fases de Información y Notificación
(8)         if (decision = new)
(9)           half_work_load = divide_search_tree(tree);
(10)          create_thread(global_optimice, half_tree);
(11)  send_results();
(12)  IGP_End_Thread();                                    Fase de Terminación
```

El Algoritmo 3.5.2 detalla la función *global_optimice()* ejecutada por cada una de las hebras creadas en *Local-PAMIGO*. Las hebras comienzan informando de su carga de trabajo al GP (línea 2) y a continuación ejecutan el bucle computacional (líneas del 4 al 10), donde básicamente se evalúa si en el intervalo de búsqueda se encuentra una posible solución, para en caso contrario descartarlo. Aquel intervalo de búsqueda no descartado, se divide en dos nuevos intervalos, siempre que no se alcance la precisión (ϵ) especificada inicialmente por el usuario. El arbol de búsqueda local es actualizado para incluir los nuevos intervalos de búsqueda no eliminados y eliminar el intervalo dividido o descartado. En cada iteración del bucle computaciónal se informa al GP de la carga pendiente y del número de intervalos evaluados y se espera la decisión tomada por el gestor (línea 7). Si el GP establece que esta hebra puede crear una nueva hebra, entonces divide el arbol de búqueda local y crea una nueva hebra, a la que le asigna la mitad de la carga de trabajo pendiente (línea 10). Una vez que la hebra ha evaluado todos los intervalos del arbol de

búsqueda local, envía los resultados obtenidos, junto con información estadística, al proceso principal, a través de la memoria compartida (línea 11). Finalmente, antes de terminar la ejecución, la hebra se lo notifica al GP (línea 12).

El tiempo total de ejecución de las aplicaciones multihebradas *PAMIGO*, tanto *Local-PAMIGO* como *Global-PAMIGO*, depende directamente del tipo de función a analizar, el número de hebras en ejecución y de la precisión deseada. La generación de hebras afecta al rendimiento de la aplicación. Hay que diferenciar entre el número de hebras creadas y el número de hebras activas en cada instante, que suele ser menor. La creación dinámica de hebras de *Local-PAMIGO* ofrece una mayor complejidad en la gestión de las hebras que la creación estática de hebras característica de *Global-PAMIGO*, pero tiene la ventaja de evitar la contención en la estructura global usada en *Global-PAMIGO*. Además, la creación dinámica de hebras actua como un balanceador dinámico de la carga, necesario en algoritmos paralelos que trabajan sobre estructuras de datos irregulares. Por este motivo, todos los experimentos realizados en este capítulo se han centrado solamente en la aplicación *Local-PAMIGO*. La adaptación de la aplicación multihebrada al entorno es un factor que beneficia tanto al rendimiento de la aplicación como al del sistema. A continuación, se estudia la generación de hebras sin adaptación frente a la generación adaptativa de hebras del algoritmo *Local-PAMIGO* para diferentes tipos de GP.

3.5.1. Generación no adaptativa de hebras

En esta subsección se ha utilizado un algoritmo B&B multihebrado denomiado *Local-PAMIGO* [13], que se puede clasificar como AMP (Asynchronous Multiple Pool) [23]. El modelo de programación usado en *Local-PAMIGO* se basa en el procesamiento de hebras sobre sus propias estructuras de datos, reduciendo el número de conflictos en el acceso a datos de memoria compartida. Las hebras procesan sus respectivos arboles de búsqueda y la cooperación entre hebras consiste en la actualización de un conjunto de variables compartidas. Por otro lado, el algoritmo *Local-PAMIGO* utiliza el concepto de unidades virtuales de procesamiento (VPUs), pudiendo ocurrir que el número de VPUs sea diferente al número de unidades de procesamiento disponibles (PUs). *Local-PAMIGO* utiliza un mecanismo de generación dinámica de hebras. El usuario es quien establece el número p de VPUs para resolver el problema seleccionado. Si existen VPUs ociosas, una hebra puede generar nuevas hebras asignandole parte del arbol de búsqueda pendiente de analizar, manteniendo el número de hebras activas igual al valor de p establecido a priori. Cuando una hebra ha procesado toda la carga de trabajo que tenía asignada, termina su ejecución, disminuyendo así el número de hebras activas.

Esta metodología permite un balanceo dinámico de la carga inherente al modelo, pues una hebra que está ejecutando *Local-PAMIGO* generará una nueva hebra cuando detecta que existe una VPU ociosa, asignándole la mitad de su propia carga de trabajo. Normalmente, el usuario establece como parámetro de entrada el valor de p igual al número de PUs en los sistemas dedicados, considerandose así un mecanismo de generación de hebras dinámico pero no adaptativo. Para el usuario puede resultar difícil establecer un valor de p que permita un buen rendimiento de la aplicación paralela, principalmente cuando existen muchas PU en el sistema. Por ejemplo, para problemas pequeños, el valor de p que permite

mantener una alta eficiencia de la aplicación paralela, puede ser inferior al número de PUs disponibles. En definitiva, el valor óptimo de p, en terminos del tiempo de ejecución de la aplicación, dependerá de la arquitectura, nivel de paralelismo del algoritmo, tamaño del problema, habilidad del programador, etc...

Para mostrar como varia la eficiencia del algoritmo con la elección del valor de p se ha realizado unos experimentos preliminares, cuyos resultados se describen a continuación. La Tabla 3.1 muestra el tiempo total de ejecución en segundos (Time) y el número medio de hebras activas (*AvNRT= Average Number of Runnig Threads*) obtenidas con *Local-PAMIGO* sobre dos problemas diferentes (SHCB (*Six-Hump Camel-Back*) [10] con $\epsilon = 10^{-8}$ y Kowalik [33] con $\epsilon = 10^{-3}$), para diferentes valores de p introducidos por el usuario como parámetro de entrada. Estos resultados se han realizado en dos sistemas dedicados diferentes. Por un lado, un Quad-Core Intel Q6600, 2.40GHz y 4GB RAM (denominado *QCore*), y por otro lado, cuatro Quad-Core AMD Opteron 8356, 2.30 GHz y 64GB RAM (denominado *Frida*). Todas las modificaciones del kernel se han realizado sobre la versión 2.6.29 del sistema operativo Linux. En negrita se resaltan el número medio de hebras que permiten obtener el mínimo tiempo total de ejecución para cada una de las funciones analizadas en ambas arquitecturas.

Tabla 3.1: Número medio de hebras en ejecución (*AvNRT*) y tiempo total en segundos de *Local-PAMIGO* usando diferentes valores de p en dos sistemas con 4 (*QCore*) y 16 (*Frida*) PUs.

p		1	2	4	8	16	32	64	128
SHCB	AvNRT	0.5	1.0	**3.0**	**5.3**	4.5	4.9	4.9	**5.2**
4 PUs	Time	0.10	0.05	**0.03**	**0.03**	0.05	0.04	0.04	**0.03**
Kowalik	AvNRT	0.5	1.3	**3.0**	**6.5**	13.7	26.6	48.6	94.1
4 PUs	Time	683.73	364.64	**170.98**	**170.92**	173.16	196.2	210.79	215.22
SHCB	AvNRT	1.0	1.9	**3.5**	**5.1**	**5.0**	**6.0**	**5.9**	**6.2**
16 PUs	Time	0.08	0.05	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**
Kowalik	AvNRT	1.0	1.7	3.5	7.0	**13.8**	27.3	48	94.8
16 PUs	Time	509.11	267.18	128	64.35	**31.89**	37.89	60.46	68.85

Si el valor de p es demasiado pequeño, puede ocurrir que no se explote correctamente el paralelismo intrínseco del algoritmo, mientras que si p es demasiado grande, pueden aparecer sobrecargas debido a la concurrencia de la aplicación en la misma unidad de procesamiento y/o a la gestión del número de hebras por el planificador del SO. No hay que olvidar que puede ser dificil para el usuario establecer el valor apropiado de p antes de la ejecución de la aplicación. Por ejemplo, los tiempos mínimos de ejecución para la función *SHCB* se consiguen cuando el usuario establece p igual al número de unidades de procesadmiento, es decir $p = 4$ para *QCore* y $p = 16$ para *Frida*. Aunque dada la baja complejidad de esta función obtendríamos el mismo tiempo de ejecución para $p = 4$ en la arquitectura *Frida*. Por otro lado, la función *Kowalik* obtiene los tiempos mínimos de ejecución para $p = 16$ en *Frida*, aunque en el *QCore* se obtiene para $p = 8$. Con los datos mostrados en la Tabla 3.1, se podría concluir que si el usuario establece p igual al número de unidades de procesamiento, obtendría buenos resultados. Sin embargo, para *SHCB* en

Frida con 16 cores se obtiene un menor tiempo de ejecución y un menor uso de recursos para $p = 4$ y para la función *Kowalik*, el mejor resultado se obtiene para $p = 8$ en el *QCore*.

Por este motivo, las aplicaciones multihebradas basadas en algoritmos B&B requieren de un GP que les permita calcular dinamicamente el valor óptimo de p, sin la intervención del usuario, adaptando el número de hebras activas de la aplicación en tiempo de ejecución.

3.5.2. GPs para la generación dinámica adaptativa de hebras

A continuación, se describen cuatro GPs diferentes que permiten establecer, de forma adaptativa, el número de hebras (n) que deben ser usadas en el algoritmo B&B *Local-PAMIGO* (Algoritmos 3.5.1 y 3.5.2) para obtener una máxima eficiencia. La eficiencia de estos GPs ha sido experimentamente evaluada sobre las dos arquitecturas anteriormente mencionadas, *QCore* y *Frida*. Todos estos GPs se han ensayado deshabilitando la detención de hebras y seleccionando un criterio de decisión que permita minimizar los retardos de las hebras:

ACW (*Application decides based on Completed Work*): La aplicación decide según el trabajo completado. Corresponde a una implementación del GP a nivel de usuario basado en el rendimiento de la aplicación, integrado en la propia aplicación multihebrada que no necesita obtener ninguna información del SO. Corresponde a las distintas hebras activas informar al GP de la carga de trabajo realizada y pendiente, a través de la memoria compartida (Ver Sección 2.3).

AST (*Application decides based on Sleeping Threads*): La aplicación decide según el número de hebras dormidas, obtenido mediante la consulta de los ficheros pseudo-estáticos del SO. Consiste en un GP a nivel de usuario cuyo criterio de decisión pretende minimizar los periodos en los que las hebras están en estado *INTERRUPTIBLE* y *UNINTERRUPTIBLE*, minimizando así el tiempo no ejecutable. Para ello se analiza periodicamente el estado de todas las hebras, de forma que se permite crear una nueva hebra, si todas las hebras activas están en estado *RUNNING* en el intervalo de ejecución analizado (Ver Subsección 2.3.1).

KST-MST (*Kernel decides based on Sleeping Threads using decision rule to Minimize Sleeping Time*): Mediante llamadas al sistema, un módulo del kernel decide el número de hebras en ejecución en función del tiempo que las hebras han estado dormidas, usando el criterio de decisión que minimiza el tiempo que duermen. Es un GP a nivel de Kernel que utiliza, en cada llamada al sistema, un criterio de decisión basado en minimizar el tiempo de bloqueo de las hebras activas, de forma que se permite crear una nueva hebra cuando se obtiene un tiempo de bloqueo nulo para todas las hebras durante el último intervalo de análisis (Ver Subsección 2.4.1).

KITST-MST (*Kernel Idle Thread decides based on Sleeping Threads using decision rule to Minimize Sleeping Time*): Mediante llamadas al sistema, la hebra ociosa del Kernel decide en función del tiempo que las hebras han estado dormidas, usando el criterio de decisión que minimiza el tiempo que duermen. Este gestor se diferencia del *KST-MST* en que el criterio de decisión se analiza periodicamente por la hebra ociosa del

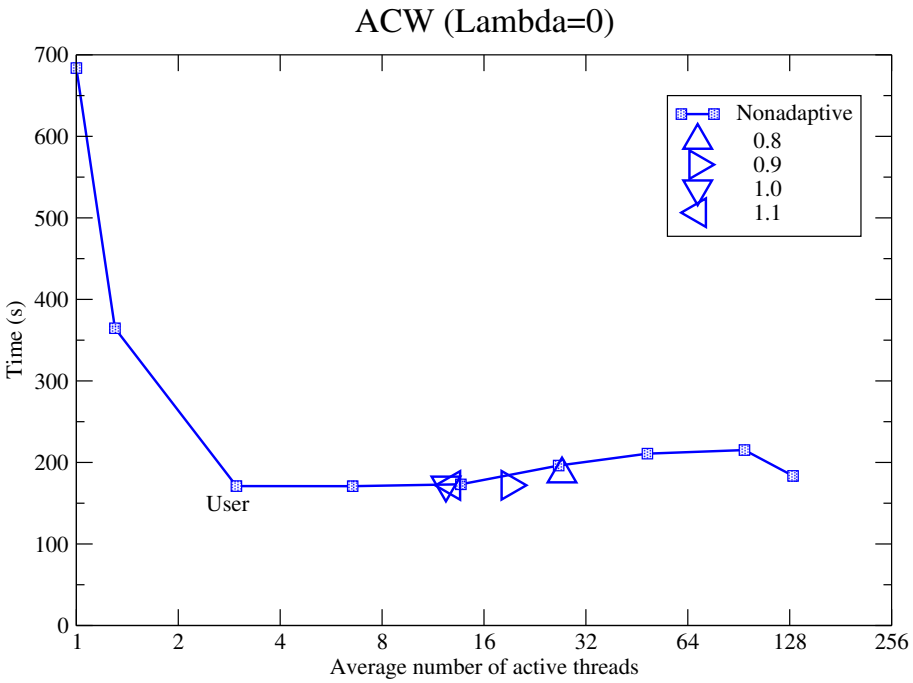


Figura 3.2: Adaptabilidad del número medio de hebras para el gestor ACW con $\lambda = 0$ para diferentes umbrales definidos según la Ecuación 3.3 en la arquitectura *QCore*.

kernel, por lo que la llamada al sistema tan solo sirve para informar de la carga de trabajo realizada por cada hebra y notificar a las hebras de la posibilidad de crear, o no, una nueva hebra, mediante el chequeo del estado de la variable *Create* (Ver Subsección 2.4.2).

A continuación se presentan los resultados de la evaluación experimental de las distintas alternativas descritas hasta ahora (ACW, AST, KST-MST y $KITST$-MST). La evaluación se ha realizado usando la función *Kowalik* como función de prueba para el algoritmo B&B (*Local-PAMIGO*). Se han utilizado las maquinas *QCore* y *Frida* descritas anteriormente.

El primer experimento se ha realizado con el GP adaptativo basado en el rendimiento de la aplicación (ACW), que requiere que el usuario seleccione el umbral que establece el nivel de rendimiento deseado. Además, se ha configurado el GP para que la fase de *Evaluación* se ejecute cada intervalo de tiempo λ, cuyo valor también es establecido por el usuario. Recuérdese que no se evalúa la decisión hasta que todas las hebras hayan realizado una unidad de trabajo, aunque el tiempo requerido sea mayor que λ.

Las Figuras 3.2 y 3.3 muestran el tiempo real de ejecución del algoritmo B&B para la función *Kowalik* frente al número medio de hebras en ejecución, obtenidos sobre las arquitecturas *QCore* y *Frida*, respectivamente. Los valores del tiempo total de ejecución para la generación de hebras no adaptativa (línea *Nonadaptive*) mostrados en ambas figuras corresponden a los descritos en la Tabla 3.1 para la función Kowalik. Nótese que, como las hebras son dinámicas, la media suele ser un poco inferior al número de unidades

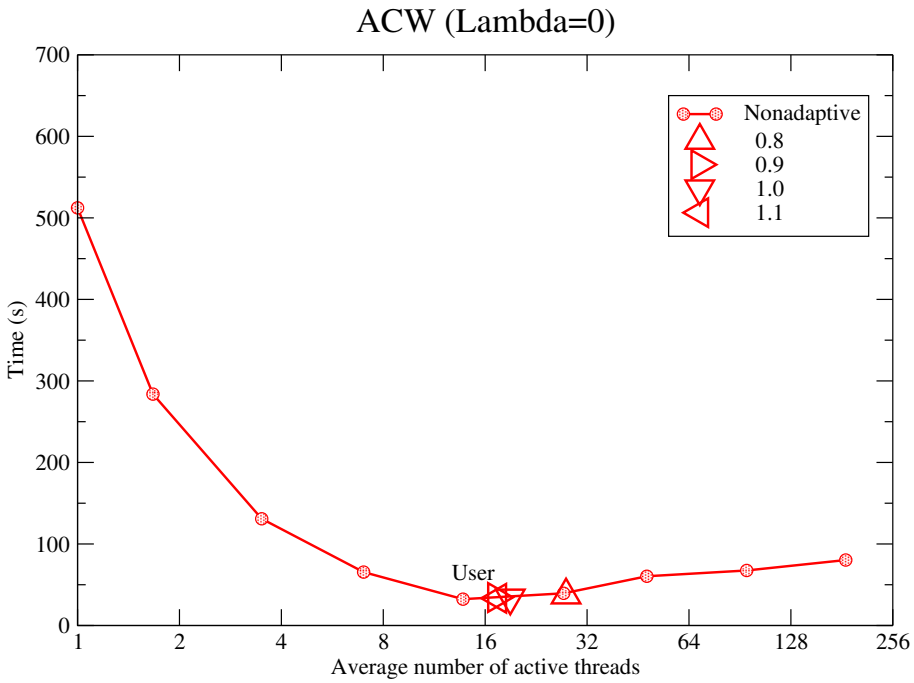


Figura 3.3: Adaptabilidad del número medio de hebras para el gestor ACW con $\lambda = 0$ para diferentes umbrales definidos según la Ecuación 3.3 en la arquitectura *Frida*.

de procesamiento (p), especificadas por el usuario. También se resalta el número medio de hebras con p igual al número de procesadores que suele ser elegido por el usuario (*User*). Además, se muestra el tiempo de ejecución del algoritmo usando el gestor ACW adaptativo con un valor de $\lambda = 0$ y diferentes valores de umbral (0.8, 0.9, 1.0 y 1.1) para el rendimiento instantáneo observado. Establecer un valor de $\lambda = 0$ significa que el GP está ejecutando ininterrumpidamente la fase de *Evaluación*, ya que tan solo debe esperar a que todas las hebras activas informen en cada iteración del bucle del trabajo completado. En la arquitectura *Frida* el número medio de hebras activas cuando se usa el GP ACW está próximo a la decisión del usuario $p = 16$, especialmente para valores del $Threshold \geq 0{,}9$. Sin embargo, en la arquitectura *QCore* no es así pues difiere mucho de la decisión del usuario $p = 4$. También, se observa, que los tiempos de ejecución obtenidos con un $Threshold$ de 1.0 están próximos a los mejores valores obtenidos en la Tabla 3.1 para *QCore* y *Frida*. En definitiva, puede deducirse que cualquier valor del $Threshold$ próximo a 1.0 es bueno. Normalmente, valores superiores al $Threshold$ generarán menos hebras, mientras que valores inferiores permitirán generar un mayor número de hebras.

Las Figuras 3.4 y 3.5 muestran los resultados obtenidos cuando el gestor ACW fija el valor de $Threshold$ a 1.0 y se establecen diferentes valores de λ: 0.0, 0.5, 1.0 y 2.0 segundos, para las arquitecturas *QCore* y *Frida*, respectivamente. Como se esperaba, un elevado valor de λ generará un menor número de hebras debido a que se ha reducido el número de decisiones. Por otro lado, para $\lambda = 0{,}0$ se producen el máximo número total de decisiones, permitiendo así obtener el número máximo de hebras, de tal forma, que se

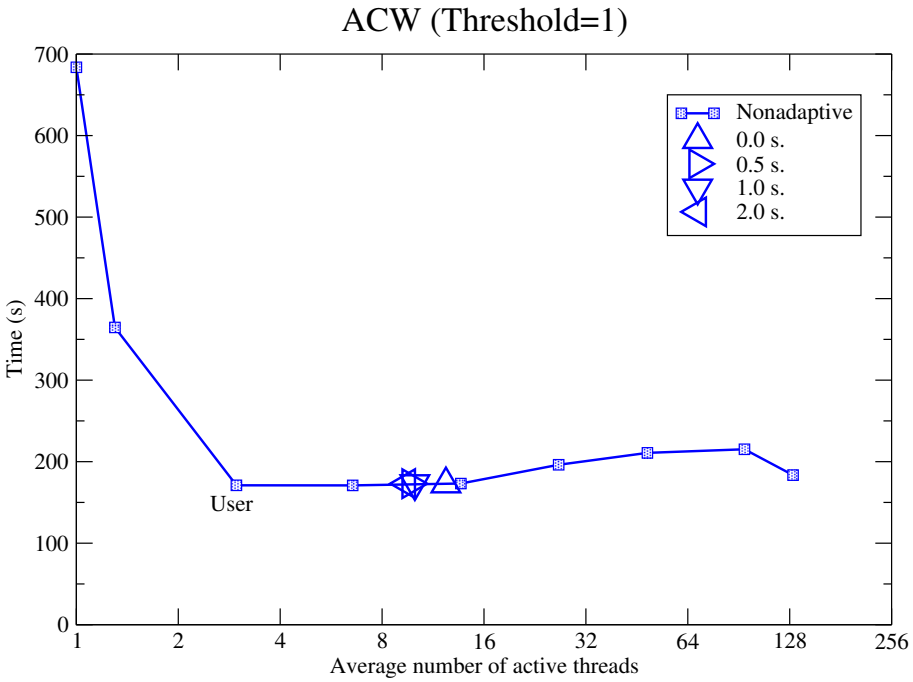


Figura 3.4: Adaptabilidad del número medio de hebras para el gestor *ACW* con $Threshold = 1{,}0$, para diferentes λ's en la arquitectura *QCore*.

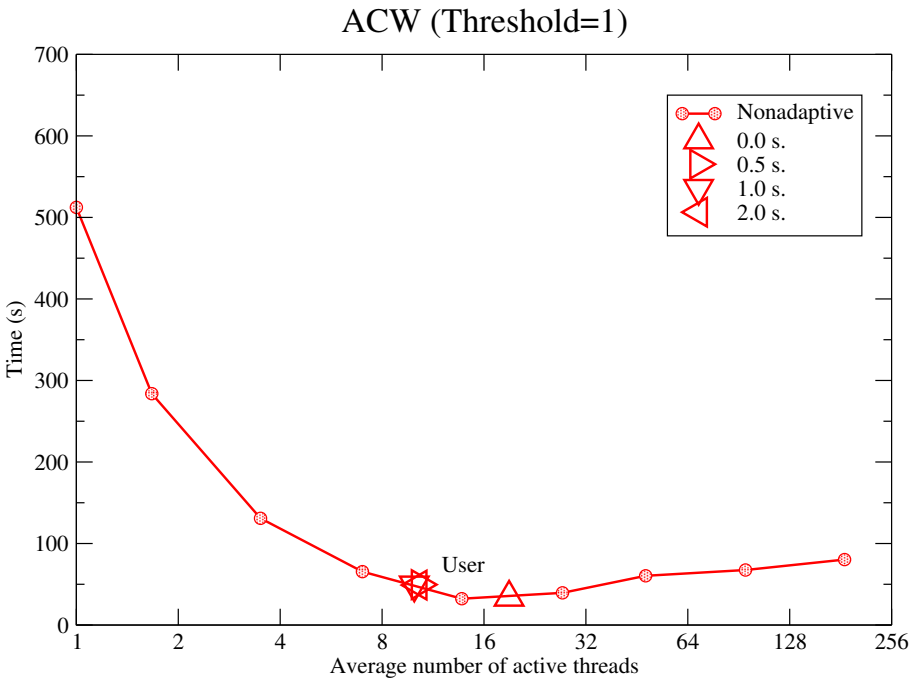


Figura 3.5: Adaptabilidad del número medio de hebras para el gestor *ACW* con $Threshold = 1{,}0$, para diferentes λ's en la arquitectura *Frida*.

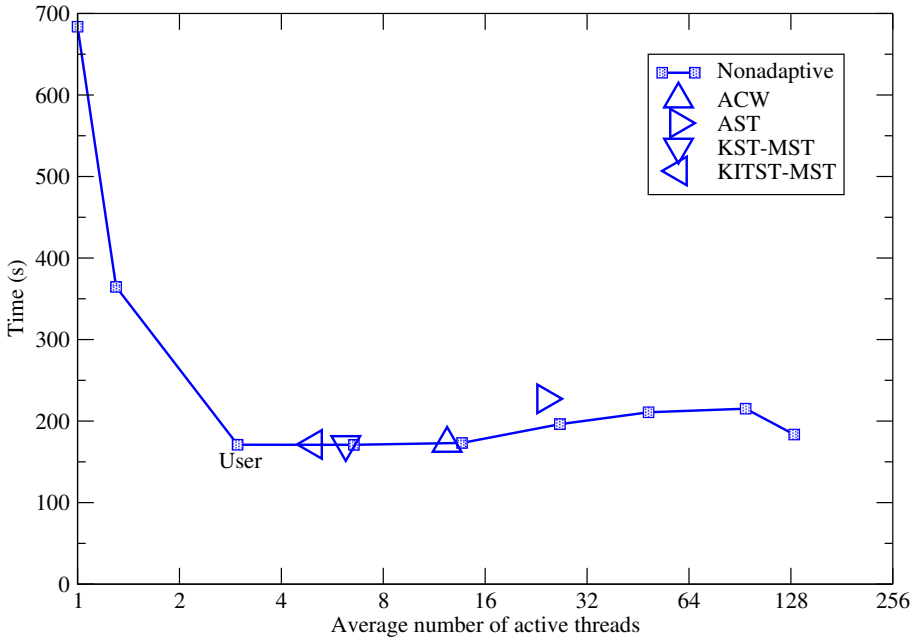


Figura 3.6: Adaptabilidad del número medio de hebras para diferentes tipos de gestores en la arquitectura *QCore*.

obtienen tiempos de ejecución próximos a los mejores para ambas arquitecturas, aunque difiera de la decisión del usuario, especialmente en la arquitectura *QCore*, ratificándose así los resultados obtenidos en la subsección 2.7.3.

Las Figuras 3.6 y 3.7 muestran una comparativa de distintos GPs propuestos a nivel de usuario (*ACW* y *AST*) y a nivel de Kernel (*KST-MST* y *KITST-MST*) sobre las dos arquitecturas *QCore* y *Frida*, respectivamente. A continuación, describimos los resultados mostrados en ambas figuras para los distintos gestores, evaluados sobre las arquitecturas *QCore* y *Frida*. En la descripción se ha seguido un orden basado en un rendimiento creciente:

AST: Este GP genera los peores resultados en términos del número medio de hebras activas y del tiempo total de ejecución. El principal inconveniente de este método es la necesidad de leer un número de ficheros igual al número de hebras en ejecución, de tal forma que en cada lectura el SO actualiza la información de los distintos ficheros. Para $\lambda = 0{,}0$ el número de comprobaciones es más elevado, siendo el número de hebras mayor pero también el tiempo dedicado a la lectura y evaluación por parte del gestor. Los resultados corroboran los obtenidos en la Sección 2.7, donde este método no obtiene toda la información relativa a los estados de bloqueo de las hebras.

ACW: En este caso, el GP basado en el rendimiento utiliza un *Threshold* igual a 1.0 y $\lambda = 0{,}0$, obteniendo un número medio de hebras activas ligeramente superior que el mejor obtenido en la Tabla 3.1 con el correspondiente aumento del tiempo total de ejecución.

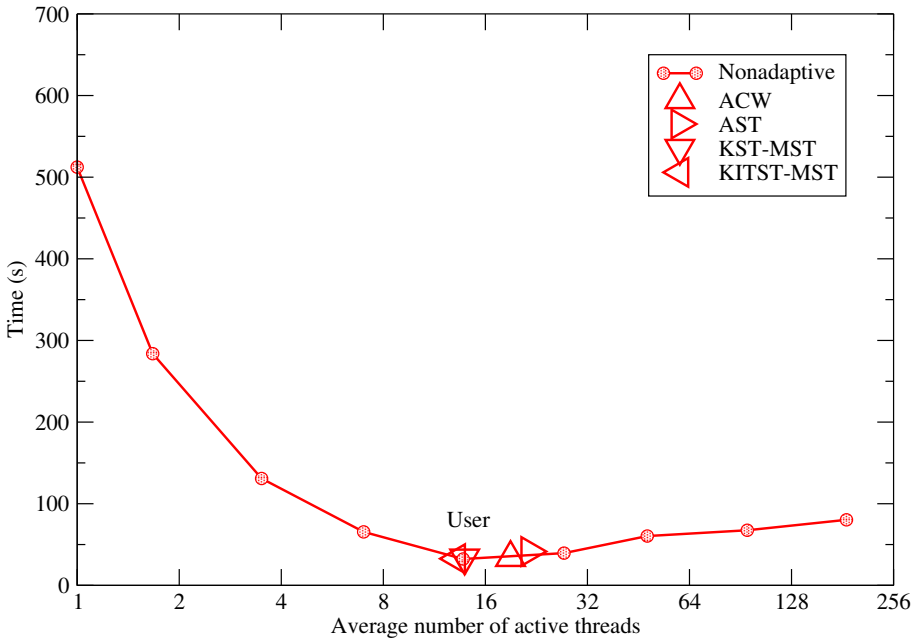


Figura 3.7: Adaptabilidad del número medio de hebras para diferentes tipos de gestores en la arquitectura *Frida*.

KST-MST: Comparando los gestores *ACW* y *AST*, podemos pensar que el criterio de decisión basado en el rendimiento de la aplicación ofrece mejores resultados que el criterio de decisión basado en el tiempo de bloqueo de las hebras realizado a nivel de usuario. Sin embargo, el GP a nivel de kernel basado en el tiempo de bloqueo de las hebras (*KST-MST*) obtiene tiempos similares a *ACW*, aunque con distinto número medio de hebras. Esto se debe a que el Kernel trabaja con información más precisa sobre las hebras bloqueadas, lo que permite tomar mejores decisiones que el gestor *AST*.

KITST-MST: Este es el mejor gestor de los cuatro en cuanto al tiempo de la aplicación y menor número de hebras utilizado. El tiempo de ejecución es similar al mejor resultado obtenido en la Tabla 3.1, usando el menor número promedio de hebras activas. Esto es debido a que las llamadas del sistema implementadas en *KITST-MST* consumen menos tiempo y las decisiones se aplican únicamente cuando algún procesador está ocioso, independientemente de λ.

Se puede concluir que la adaptabilidad de los GPs a nivel de kernel ofrece un número medio de hebras muy próximo a la mejor decisión establecida de forma estática por el usuario.

Las Figuras 3.8 y 3.9 muestran los cambios en el número de hebras en ejecución obtenidos usando los distintos GPs, en el sistema dedicado *Frida* para la función Kowalik. La Figura 3.8 abarca el tiempo de ejecución total, donde se observa como el algoritmo *Local-PAMIGO* con generación no adaptativa de hebras (línea *Nonadaptive*), establece

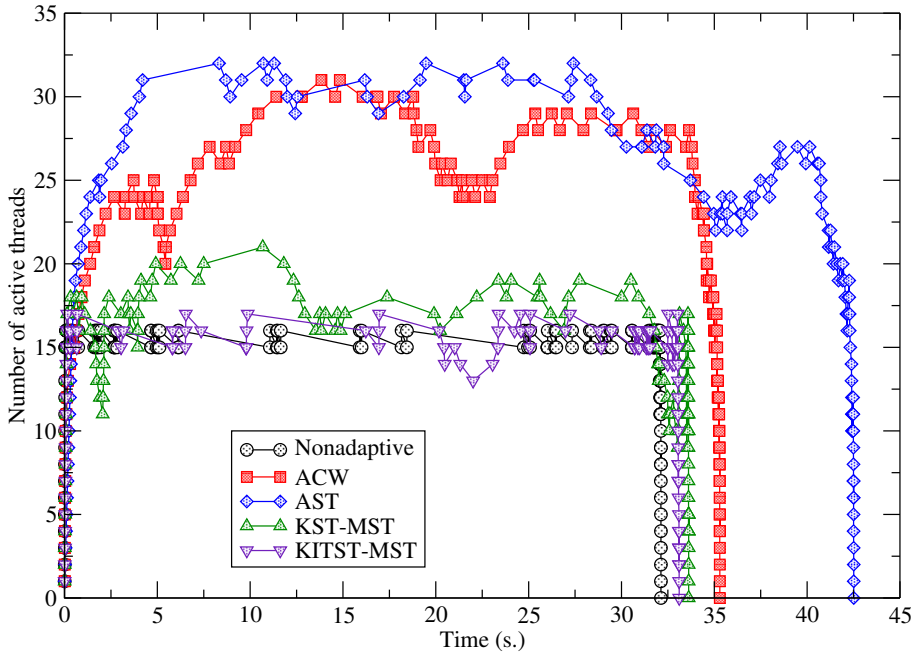


Figura 3.8: Evolución temporal del número de hebras activas para la función Kowalik.

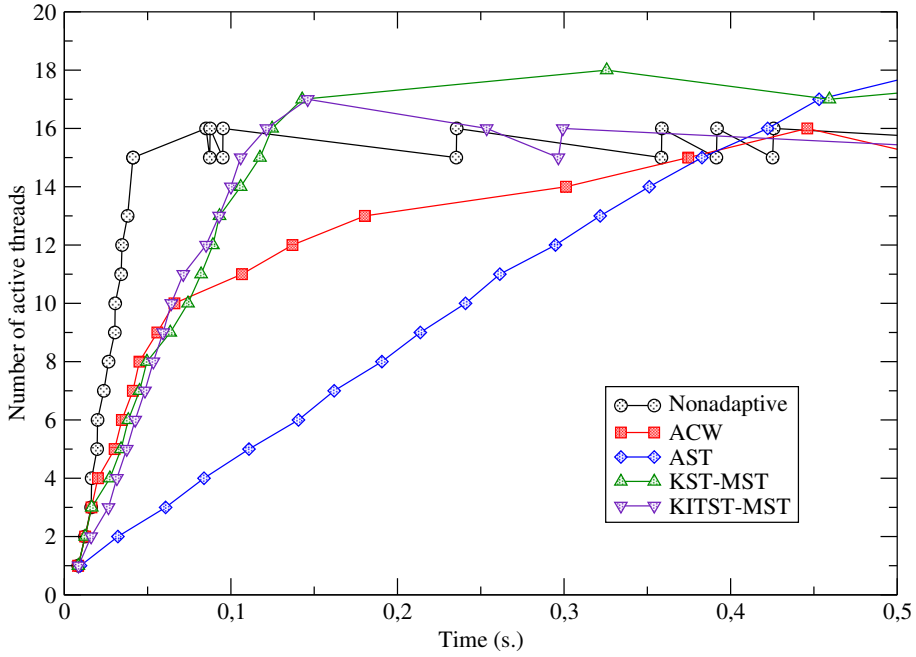


Figura 3.9: Zoom del inicio de la evolución temporal del número de hebras activas para la función Kowalik.

rápidamente el número de hebras a 16, y tan pronto como una hebra termina, otra hebra es creada siguiendo las especificaciones del usuario al establecer un valor de $p = 16$. También, se observa como los gestores a nivel de usuario *ACW* y *AST* generán un número de hebras activas muy superior a 16, superando las 20 hebras durante la mayor parte del tiempo de ejecución. Mientras, los GP a nivel de kernel *KST-MST* y *KITST-MST* mantiene un número de hebras activas entorno al número óptimo de hebras, destacando los mejores resultados del gestor *KITST-MST*. Por otro lado, la Figura 3.9 realiza un zoom de la etapa inicial de creación de hebras durante los primeros 0.5 segundos de ejecución. Se puede observar la sobrecarga de los distintos GP adaptativos. El número de hebras establecido por el gestor de hebras no adaptativo se alcanza en menos de 0.1 segundos, mientras los demás métodos necesitan más tiempo ya que deben evaluar decisiones más complejas. Las decisiones y su frecuencia, realizadas por los gestores adaptativos influyen directamente en el tiempo de ejecución, como puede observarse en la Figura 3.9. En este sentido, los gestores *ACW* y *AST* generan los peores resultados al necesitar 0.4 segundos (aprox.) para generar 16 hebras, mientras que *KST-MST* y *KITST-MST* si permiten generar el número de hebras óptimo en un tiempo similar al requerido por el gestor de hebras no adaptativo.

Este experimento ha permitido verificar que los GP a nivel de kernel, usados con las aplicaciones B&B, ofrecen un nivel de adaptación mejor que los GP a nivel de usuario, tal y como ya se corroboró sobre el algoritmo de ray tracing en el capítulo anterior. Sin embargo, el criterio de decisión basado en minimizar el tiempo que las hebras están dormidas (*MST*) le confiere al GP un mayor grado de adaptación que el criterio de decisión basado en el valor del número máximo de unidades de proceso establecido por el usuario ya que este necesita un estudio previo y no puede adaptarse a cambios de carga en el sistema, cuando no se usa la detención de hebras.

Los resultados muestran que cualquiera de los criterios de decisión *MNET*, *MST* y *MIBT* (véase la subsección 3.4.1) suele indicar un número de hebras cercano al óptimo cuando se ejecutan aplicaciones altamente escalables en sistema dedicados. Sin embargo, este tipo de criterios suele ser bastante restrictivo en el número de hebras a generar, especialmente cuando las aplicaciones tienen ciertas limitaciones que reducen su escalabilidad, o cuando se ejecutan en sistemas no dedicados. En estos casos, el número de hebras propuesto están por debajo del número óptimo por lo que si aumentamos el número de hebras se podría reducir el tiempo total de ejecución, con la consiguiente mejora de rendimiento. Por lo tanto, se puede concluir que este tipo de criterios no nos garantiza el número de hebras óptimo en todas las situaciones.

3.6. Repercusión de la gestión de la memoria

La escalabilidad de cualquier aplicación multihebrada está directamente relacionada con la gestión de memoria. En este sentido, los programadores disponen de diferentes alternativas para gestionar la memoria utilizada por cada una de las hebras activas: gestión estática y dinámica de memoria. La gestión estática de memoria reserva inicialmente toda la cantidad de memoria necesaria para almacenar, tanto la carga de trabajo inicial y

los cálculos intermedios, como los resultados, para cada hebra. Esto implica realizar una estimación de la cantidad de memoria requerida según el trabajo asignado. Sin embargo, existen aplicaciones multihebradas, por ejemplo las aplicaciones B&B, donde el uso estático de la memoria es inviable ya que los requerimientos de memoria para cada hebra son desconocidos a priori.

La gestión dinámica de memoria permite reservar y liberar periódicamente bloques de memoria según las necesidades de cada hebra. Sin embargo, el principal inconveniente de la reserva dinámica de memoria por parte de las hebras es que el acceso concurrente a la memoria principal del sistema se hace de forma exclusiva. En el capitulo anterior se observó que existen librerías, como la librería *ThreadAlloc*, que intenta evitar el acceso concurrente a memoria principal, realizando reservas de grandes bloques de memoria.

En esta sección se analiza la adaptabilidad del GP a nivel de kernel sobre distintos escenarios de gestión dinámica de memoria en algoritmos de B&B, en concreto *Local-PAMIGO*. El comportamiento de *Local-PAMIGO* difiere mucho según el problema a resolver, por ello se han seleccionado diferentes tipos de funciones a optimizar mediante *Local-PAMIGO*: Kowalik, Griewank10, Ratz5, Ratz6, Neumaier2 y EX2. En todos los casos, el algoritmo B&B obtiene el mínimo global con una precisión de $\epsilon = 10^{-5}$. Se ha seleccionado el GP basado en la hebra ociosa del kernel (*KITST*) por ser el gestor que menos recursos consume, para que nos permita detectar el impacto del criterio de decisión seleccionado. En el gestor *KITST* se han implementado los siguientes criterios de decisión para estimar el número máximo de hebras: *MNT_IBT*, *MNT_NIBT* y *MNT_BT*. Finalmente, hay que indicar que todos los experimentos se han llevado acabo sobre la arquitectura *Frida*, pues dispone de un mayor número de unidades de procesamiento. No se ha habilitado la detención de hebras.

3.6.1. Gestión no adaptativa de hebras dinámicas

El primer experimento realizado ha consistido en analizar los diferentes comportamientos de la aplicación *Local-PAMIGO* con gestión de hebras no adaptativa sobre distintos problemas de optimización global. La gestión de hebras no adaptativa se ha realizado estableciendo el máximo número de hebras (p) como parámetro de entrada, para valores de p igual a 1, 2, 4, ..., 30 y 32, respectivamente. La gestión de memoria se ha realizado con dos librerias diferentes, por un lado la librería estándar *alloc.h* de ANSI C, con las funciones *malloc*(), *realloc*() y $free()$, y por otro lado, la librería *ThreadAlloc*, con las funciones $thread\_alloc()$, $thread\_realloc()$ y $thread\_free()$. En este último caso, la librería *ThreadAlloc* se ha usado estableciendo un tamaño de bloque por defecto de 4 KB.

Las Figuras 3.10 y 3.11 muestran el tiempo de ejecución de la aplicación *Local-PAMIGO* usando reserva dinámica de memoria con la librería estandar *alloc.h* de ANSI C y *ThreadAlloc*, respectivamente. La Figura 3.10 muestra un comportamiento muy estable para todas las funciones a partir de 14 hebras. Mientras que la Figura 3.11 solo muestra esta estabilidad para los problemas *Kowalik*, *Griewank10* y *EX2*. Si se comparan ambas figuras, puede observase como en la Figura 3.11 se aprecia una reducción muy significativa del tiempo total de ejecución conforme se aumenta el número medio de hebras, para todos los problemas ensayados y para un número de hebras inferior a 16. Esta reducción es mayor

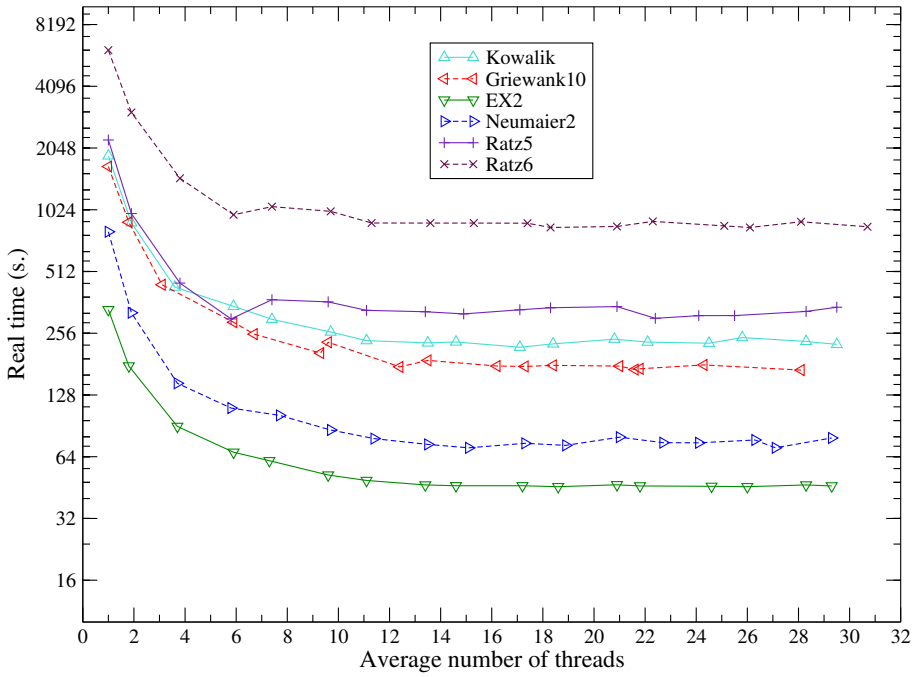


Figura 3.10: Comparativa del tiempo real de ejecución para *Local-PAMIGO* con distintos tipos de problemas, usando la librería estándar *alloc.h* de ANSI C.

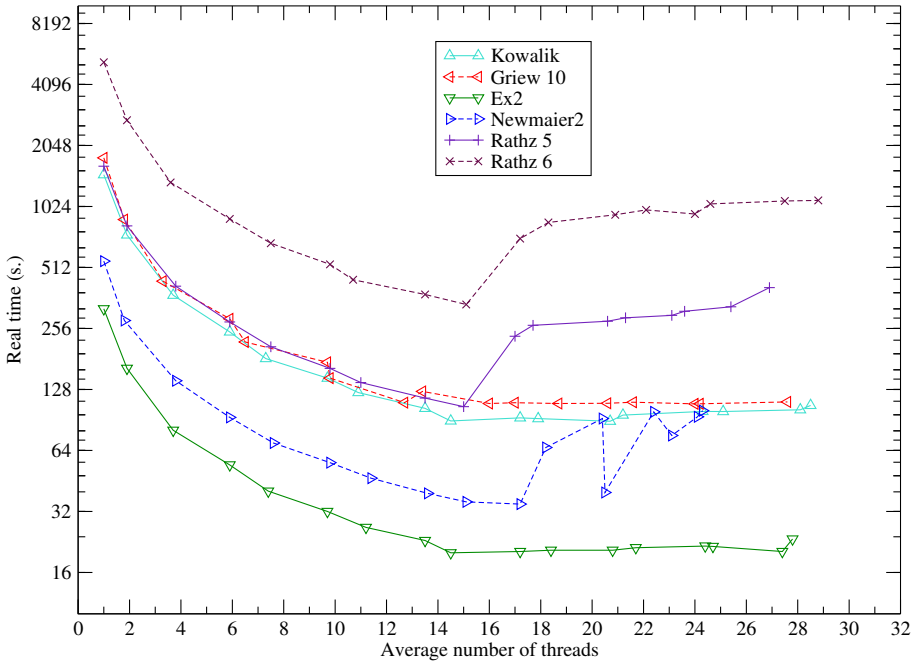


Figura 3.11: Comparativa del tiempo real de ejecución para *Local-PAMIGO* con distintos tipos de problemas, usando la librería *ThreadAlloc*.

que en la Figura 3.10. Además, se aprecia un comportamiento similar en el tiempo total de ejecución para todas las funciones tanto en la Figura 3.10 como en la Figura 3.11, salvo para las funciones *Rathz5*, *Rathz6* y *Newmaier2*. En la Figura 3.11 se observa como, concretamente estas funciones duplican el tiempo total de ejecución cuando el número de hebras supera el número de unidades de procesamiento.

Tabla 3.2: Tiempo real para *Local-PAMIGO* utilizando la librería *alloc.h* de ANSI C.

Function	η	T_{min}	$E_t(n)$	$[n_i, n_j]$	$T_{av.}$	Dev. (%)
Kowalik	17	218.8	0.5	[11, 30]	231.5	2.9
Griewank10	28	169.3	0.3	[12, 22]	175.0	2.1
EX2	**19 y 26**	45.7	0.4	[13, 29]	46.2	0.8
Neumaier2	**15 y 27**	70.7	0.7	[14, 27]	74.3	3.9
Ratz5	22	302.6	0.3	[13, 28]	324.8	4.2
Ratz6	**18 y 26**	839.2	0.4	[11, 31]	868.0	2.6

Tabla 3.3: Tiempo real para *Local-PAMIGO* utilizando la librería *ThreadAlloc*.

Function	η	T_{min}	$E_t(n)$	$[n_i, n_j]$	$T_{av.}$	δ(%)
Kowalik	21	89.2	0.8	[15, 28]	95.0	5.1
Griewank10	24	108.7	0.7	[13, 24]	109.7	0.8
EX2	**15**	20.0	1.1	[15, 27]	20.8	2.9
Neumaier2	17	34.8	0.9	**[14, 17]**	36.6	6.5
Ratz5	**15**	105.3	1.0	**[14, 15]**	110.7	6.9
Ratz6	**15**	336.1	1.0	**[14, 15]**	356.4	8.0

Las Tablas 3.2 y 3.3 muestran numéricamente la información más relevante de las Figuras 3.10 y 3.11 respectivamente. Las cabeceras indican la siguiente información:

- η: Número medio de hebras que generan el menor tiempo de ejecución (T_{min}).
- T_{min}: Mínimo tiempo de ejecución de los obtenidos usando distintos valores del máximo número de hebras permitido (p).
- $E_t(n)$: Eficiencia con una media de η hebras.
- $[n_i, n_j]$: Intervalo con el número medio de hebras que generan una pequeña diferencia respecto del menor tiempo de ejecución (T_{min}).
- $T_{av.}$: Tiempo medio de ejecución para los ensayos realizados usando un número medio de hebras en el intervalo $[n_i, n_j]$.
- δ(%): Porcentaje de desviación respecto a T_{av}.

Por un lado, la Tabla 3.2 y la Figura 3.10 muestran que el mínimo tiempo de ejecución se obtiene para la mayoria de los casos con un número medio de hebras superior al número de procesadores. Además, en los problemas *EX2*, *Neumaier2* y *Ratz6* se obtiene el menor tiempo de ejecución para dos valores de n muy diferentes. Sin embargo, se aprecian rangos muy amplios en los intervalos $[n_i, n_j]$ para todos los problemas ensayados, lo que refleja un rango de estabilidad muy extenso debido a la obtención de valores de rendimiento muy bajos.

Por otro lado, la Tabla 3.3 y la Figura 3.11 muestran que el tiempo mínimo de ejecución se obtiene para los problemas *EX2*, *Ratz5* y *Ratz6* con un número medio de hebras inferior al número de procesadores. Además, el rango de estabilidad $[n_i, n_j]$ se reduce considerablemente en todos los casos, especialmente para los problemas *Neumaier2*, *Ratz5*, y *Ratz6*.

El número óptimo de hebras será el que obtenga el mejor, o próximo al mejor, tiempo de ejecución con el menor número de hebras ya que utiliza menos recursos del sistema. Por lo tanto, el número óptimo de hebras en *Frida* para los problemas *Kowalik*, *Griewank10*, *EX2*, *Neumaier2*, *Ratz5*, *Ratz6* utilizando la librería estándar *alloc.h* de ANSI C serán 17, 28, 19, 15, 22 y 18 hebras, respectivamente, mientras que si utilizamos la librería *ThreadAlloc*, serán 21, 24, 15, 17, 15 y 15, respectivamente.

3.6.2. Generación adaptativa de hebras dinámicas

La generación adaptativa de hebras se ha implementado sobre un GP basado en la hebra ociosa del kernel encargado de estimar el número máximo de hebras activas (ver subsecciones 2.4.2 y 3.4.2). Se han realizado experimentos con tres criterios de decisión:

MNT_IBT: El número máximo de hebras basado en el tiempo de bloqueo interrumpible de la aplicación multihebrada se calcula en función del tiempo de bloqueo interrumpible producido por el acceso concurrente a las secciones críticas y/o operaciones de entrada/salida (Ecuación 3.5). En el ejemplo que nos ocupa, la aplicación multihebrada *Local-PAMIGO* carece de operaciones de entrada/salida, por lo que todo el tiempo de bloqueo de las hebras se produce en el acceso a las secciones críticas.

MNT_NIBT: El número máximo de hebras está basado en el tiempo de bloqueo no interrumpible de la aplicación multihebrada que depende directamente del tiempo de bloqueo no interrumpible producido principalmente por el acceso concurrente a memoria principal, fallos de página y/o swaping provocados por los fallos de memoria cache (Ecuación 3.6).

MNT_BT: El número máximo de hebras está basado en el tiempo de bloqueo de la aplicación multihebrada que depende del tiempo de bloqueo total, independientemente de la causa que lo origine. El tiempo de espera de las hebras en la *runqueue* no se tiene en cuenta en este criterio de decisión (Ecuación 3.4).

La experimentación se ha centrado principalmente en evaluar los distintos criterios de decisión sobre la aplicación multihebrada *Local-PAMIGO* utilizando la librería *ThreadAlloc* para gestionar la reserva dinámica de memoria. Se han ensayado distintos escenarios en función del tamaño del bloque de memoria reservada:

Bloques de memoria igual al tamaño de página por defecto: El primer escenario de gestión dinámica de memoria permite a la librería *ThreadAlloc* gestionar bloques de memoria igual al tamaño de una página de memoria establecido por defecto a 4KB.

Las Figuras 3.12 y 3.13 muestran el tiempo total de ejecución y el speed-up obtenidos para la aplicación *Local-PAMIGO* usando gestión dinámica de memoria con la librería *ThreadAlloc*. La Figura 3.12 se ha generado a partir de los datos de la Figura 3.11 obtenidos para generación de hebras no adaptativa y por el gestor *KITST* con los criterios de decisión adaptativos *MNT_IBT*, *MNT_NIBT* y *MNT_BT*. También, se resalta (en color negro el símbolo asociado a cada problema) la generación de hebras no adaptativa con el menor tiempo de ejecución para cada problema analizado.

La primera conclusión que se obtiene es que el comportamiento de los criterios de decisión depende del problema a resolver; es decir, de la función objetivo que se desea optimizar. Los tres criterios de decisión ofrecen los mismos resultados para las funciones *EX2*, *Kowalik*, y *Neumaier*. Estos resultados confirman 16 como el número medio de hebras para obtener el menor tiempo de ejecución. Sin embargo, los tres criterios de decisión presentan resultados muy diferentes para las funciones *Ratz5*, *Ratz6* y *Griewank10*. Por otro lado, la Figura 3.13 muestra los buenos resultados producidos por los distintos criterios de decisión obteniendo un speed-up próximo al lineal en todos los casos, salvo para las funciones *Ratz5* y *Ratz6*. Para estas funciones, *MNT_IBT* es el único criterio de decisión que obtiene un speed-up aceptable, lo que nos indica que el algoritmo de B&B para estas funciones experimenta interbloqueos significativos cuando el número medio de hebras pasa de 15 a 17 hebras.

Bloques de memoria inferior al tamaño de página: En este caso, se ha reducido el tamaño de bloque a 1 KB, lo que incrementará el número de solicitudes de gestión de memoria que serán resueltas de forma exclusiva por el SO. Esta experimentación se ha realizado para comprobar el impacto que tiene la reserva de pequeños bloques de memoria sobre la adaptación del gestor de hebras.

Las Figuras 3.14 y 3.15 representan el tiempo total de ejecución y speed-up obtenidos con generación de hebras adaptativa y no adaptativa, con el mismo formato que las Figuras 3.12 y 3.13, salvo que ahora los datos corresponden a bloques de memoria de 1KB.

En este caso, se observa que los tres criterios de decisión ofrecen resultados más dispersos para las funciones *EX2*, *Kowalik*, y *Neumaier*, si los comparamos con los obtenidos para bloques de memoria de 4KB. Además, ningúno de los tres criterios de decisión mejoran los resultados obtenidos con bloques igual al tamaño de página, para las funciones *Ratz5*, *Ratz6* y *Griewank10*. En general, la Figura 3.15 resalta que no se mejora ninguno de los resultados obtenidos anteriormente. Finalmente, hay que indicar que en este caso, ninguno de los tres criterios obtienen valores de speed-up aceptables, alcanzando valores de eficiencia entorno al 0.5.

Bloques de memoria superior al tamaño de página: En este experimento se reservan bloques de memoría dinámica de tamaño 16KB, reduciendo el número de interacciones con el SO para la gestión de la memoria.

Las Figuras 3.16 y 3.17 representan el tiempo total de ejecución y speed-up obtenidos

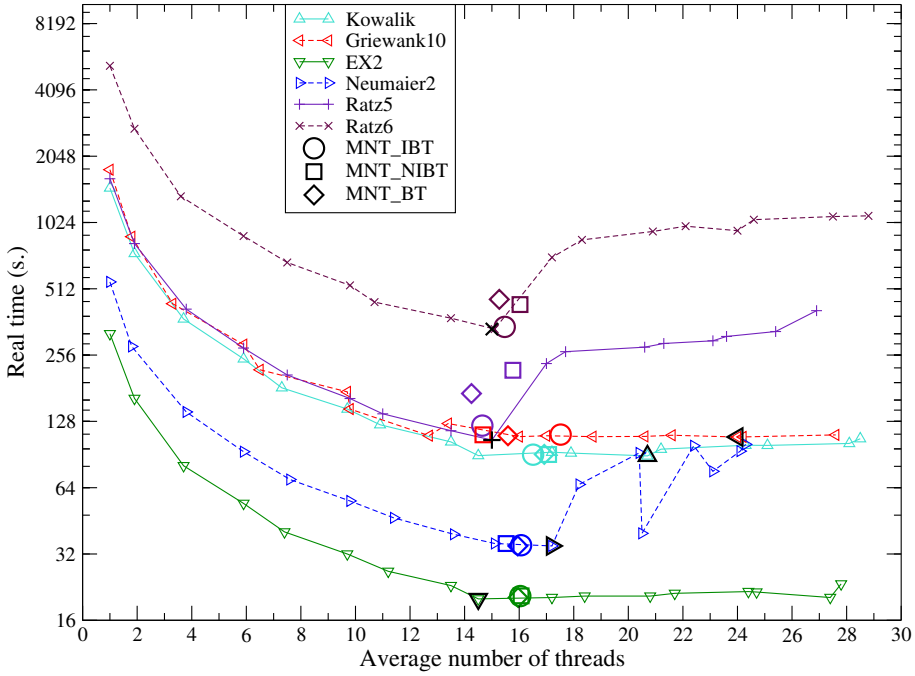


Figura 3.12: Tiempo real de ejecución usando *ThreadAlloc* con bloques de 4KB.

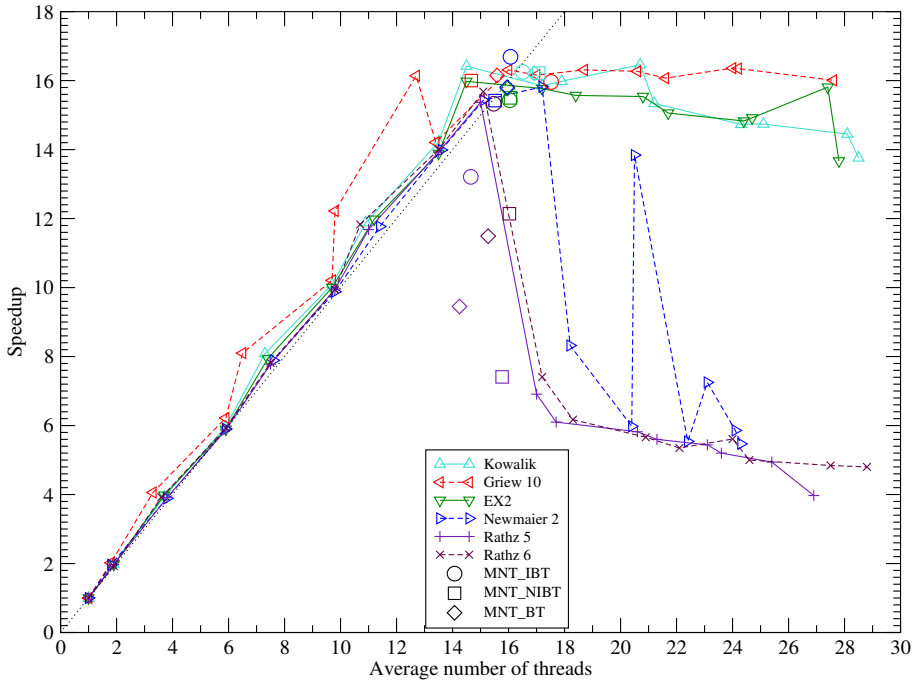


Figura 3.13: Speed-up usando *ThreadAlloc* con bloques de 4KB.

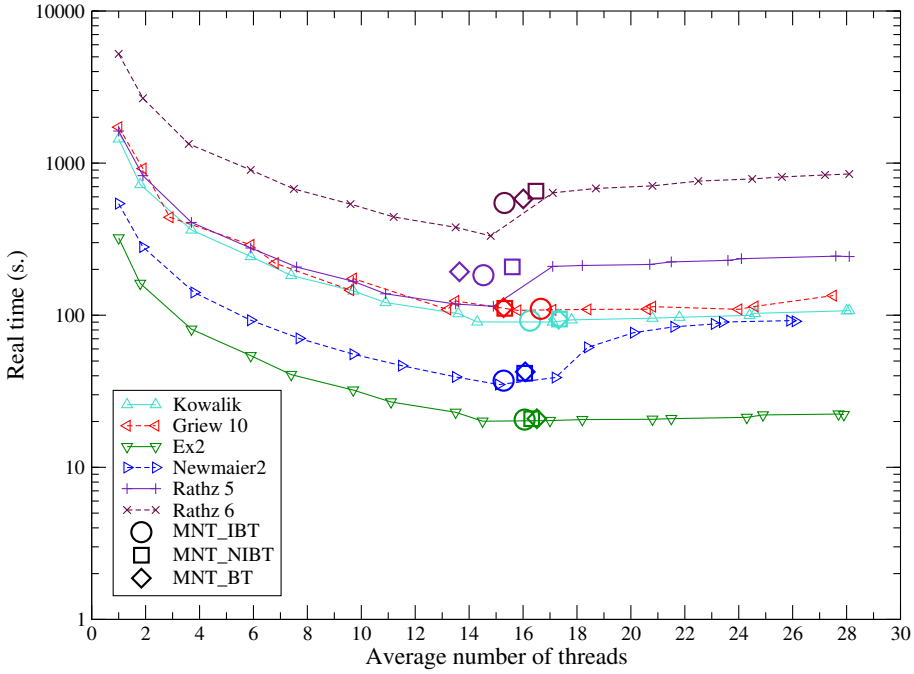


Figura 3.14: Tiempo real de ejecución usando *ThreadAlloc* con bloques de 1KB.

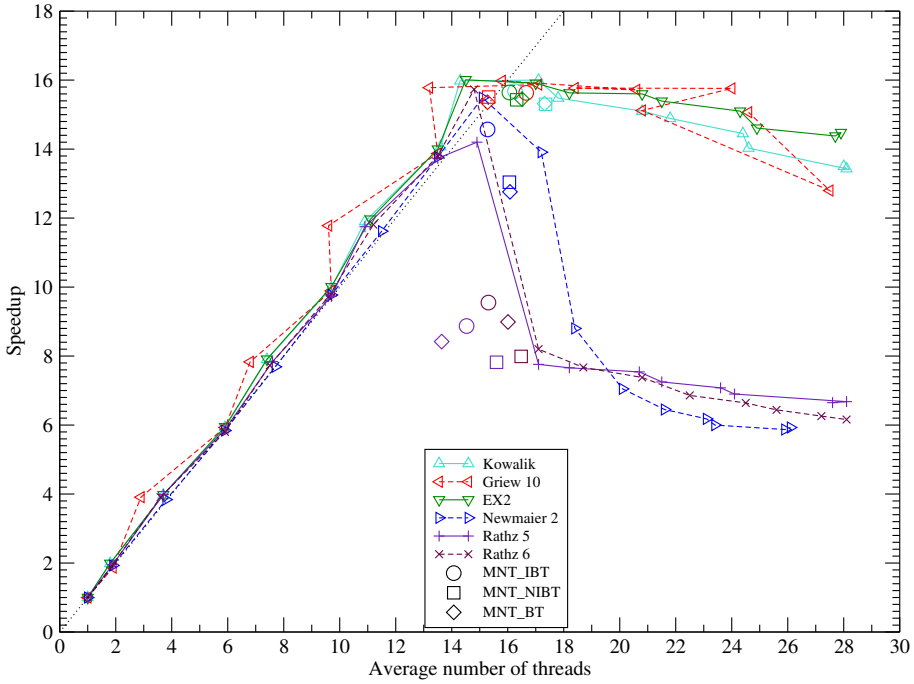


Figura 3.15: Speed-up usando *ThreadAlloc* con bloques de 1KB.

con gestión de hebras adaptativa y no adaptativa, con el mismo formato que las Figuras 3.12 y 3.13, salvo que en este caso, los datos corresponden a bloques de memoria de 16KB. Se observan comportamientos muy diferentes en la creación de hebras no adaptativa para algunos de los problemas planteados. Por ejemplo, las funciones *Ratz6* y *Newmaier2* presentan una zona estable más amplia entorno al tiempo mínimo de ejecución, lo que facilita la adaptación de los distintos criterios de decisión implementados sobre el GP a nivel de kernel, mejorando sus resultados. La Figura 3.17 confirma una mejora significativa en todas las situaciones respecto a los resultados obtenidos con bloques de tamaño de 4KB, aunque los niveles de eficiencia obtenidos para la función *Ratz5* siguén siendo bajos.

Bloques de memoria próximos al tamaño de la memoria cache: Finalmente, se ha seleccionado un tamaño de bloque lo más grande posible. En nuestro caso, se han seleccionado bloques de 1MB, pues tamaños superiores provocan un consumo de toda la memoria cache, por lo que el tiempo total de ejecución de la aplicación multihebrada crece exponencialmente al hacer uso de la memoria principal.
Las Figuras 3.18 y 3.19 representan el tiempo total de ejecución y speed-up obtenidos con generación de hebras adaptativa y no adaptativa, con el mismo formato que las Figuras 3.12 y 3.13, salvo que en este caso, los datos corresponden a bloques de memoria de 1MB. La generación de hebras no adaptativa presenta un patrón de comportamiento similar en todos los problemas de prueba. Por ejemplo, las funciones *Ratz5 Ratz6* y *Newmaier2* no presentan la inestabilidad entorno al tiempo mínimo de ejecución, característica de las situaciones anteriores. Todos los criterios de decisión utilizados en la generación adaptativa de hebras presentan un alto grado de similitud en todas las funciones, debido principalmente a que se ha minimizado el tiempo no interrumpible en el que las hebras están detenidas. La Figura 3.19 confirma unos niveles de eficiencia próximos a la unidad en todas las situaciones.

En todos los experimentos realizados sobre distintos tamaños de bloques de memoria se ha observado que los tres criterios de decisión *MNT_IBT*, *MNT_NIBT* y *MNT_BT* para la gestión adaptativa de hebras obtienen buenos niveles de eficiencia, cuando los problemas analizados presentan zonas estables entorno al mínimo tiempo de ejecución en las versiones no adaptativas. Sin embargo las funciones *Ratz5 Ratz6* y *Newmaier2* presentan inestabilidades para tamaños de bloques inferiores a 16KB. Este problema lo analizaremos con más detalle en la sección siguiente.

Otro aspecto a resaltar, es la elevada interrelación que existe entre los distintos eventos que provocan un bloqueo entre las hebras, independientemente de que influya en el tiempo interrumpible y/o el tiempo no interrumpible. Por ejemplo, cuando aumenta el tiempo en el que se bloquean las hebras debido a la reserva de memoria dinámica, el tiempo de bloqueo de las hebras en secciones críticas se reduce. Esto se debe a que la reserva de memoria secuencializa el acceso a las secciones críticas, de tal forma que, las hebras no se bloquean en las secciones críticas del programa, pues ya vienen secuencializadas de una reserva de memoria reciente.

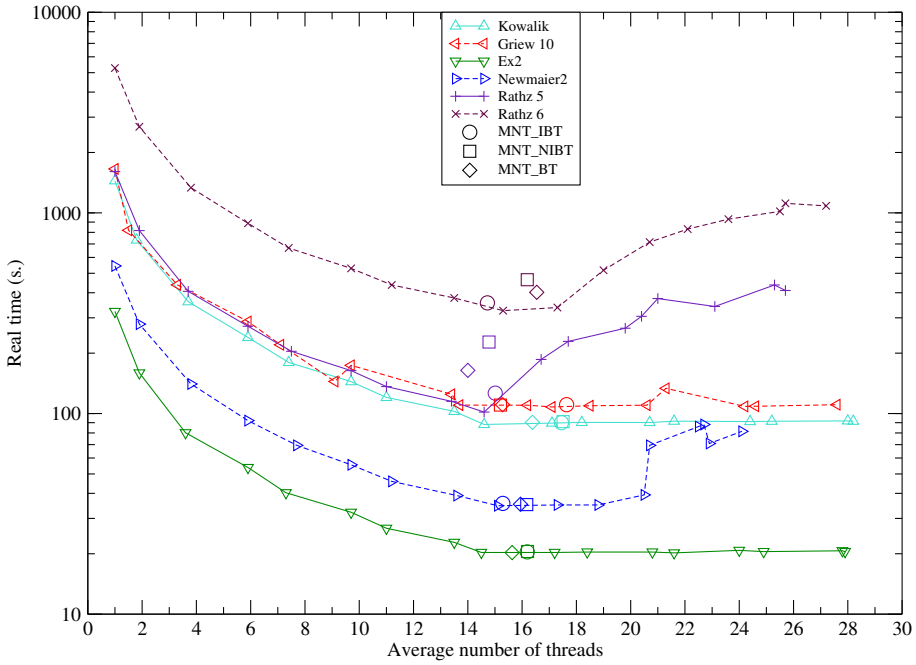


Figura 3.16: Tiempo real de ejecución usando *ThreadAlloc* con bloques de 16KB.

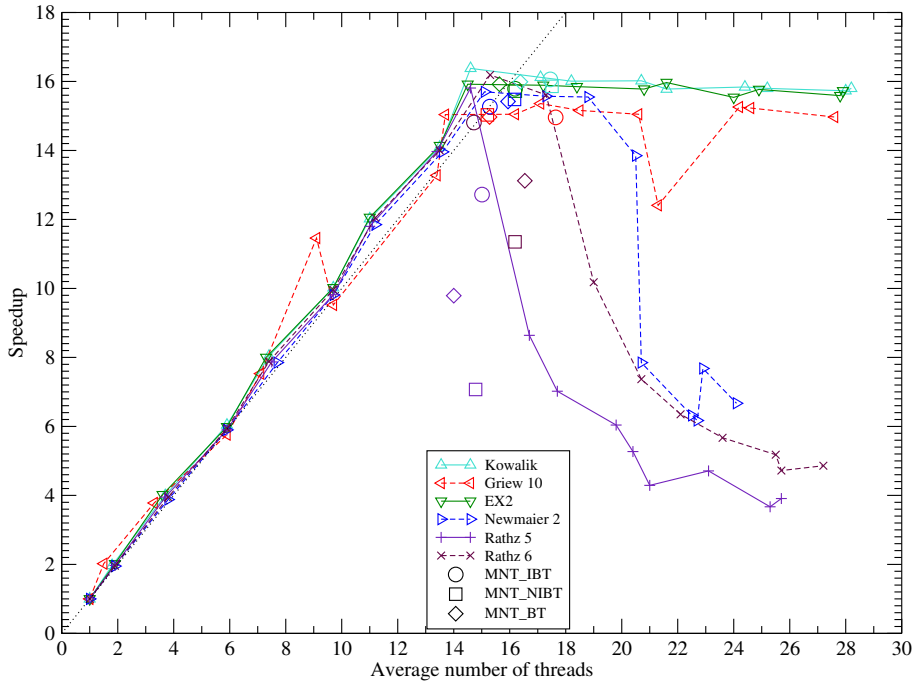


Figura 3.17: Speed-up usando *ThreadAlloc* con bloques de 16KB.

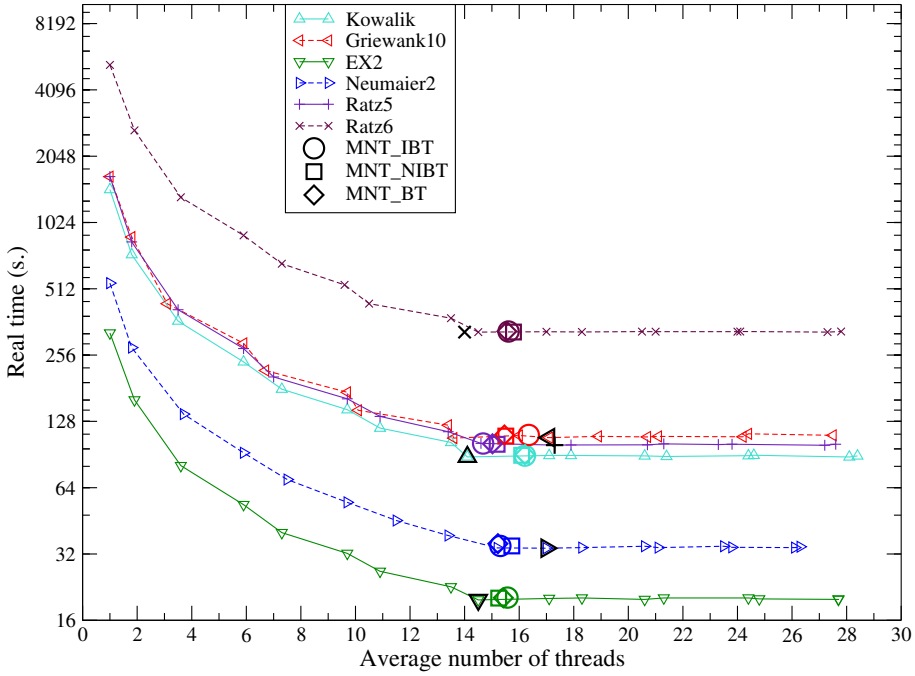


Figura 3.18: Tiempo real de ejecución usando *ThreadAlloc* con bloques de 1MB.

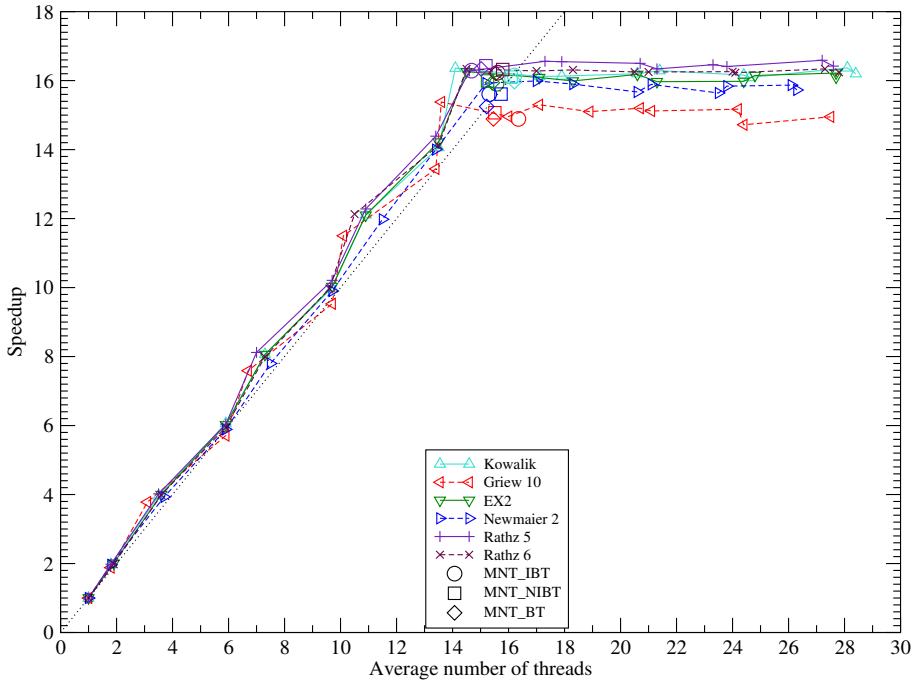


Figura 3.19: Speed-up usando *ThreadAlloc* con bloques de 1MB.

3.6.3. Gestión adaptativa de la memoria

Los experimentos anteriores demuestran que la reserva dinámica de memoria influye directamente en las decisiones realizadas por el gestor de hebras, por lo que es recomendable minimizar el número de reservas de memoria realizadas por las distintas hebras de la aplicación multihebrada. Como se ha comentado anteriormente, una posibilidad para reducir el número de reservas de memoria se centra en que cada una de las hebras reserve inicialmente un bloque de memoria suficiente para trabajar con la carga computacional asignada. Sin embargo, dadas las características peculiares de las aplicaciones B&B donde se desconoce a priori sus necesidades de memoria, se propone implementar una gestión adaptativa de la memoria del sistema.

La gestión adaptativa de memoria se basa en asignar a la aplicación multihebrada toda la memoria disponible, de tal forma, que la memoria total del sistema se reparta entre todas las hebras activas, dinámicamente. Este nuevo planteamiento implica que cuando una hebra vaya a dividir su carga de trabajo creando una nueva hebra, también divida el tamaño de memoria inicialmente reservada. Para ello, la hebra debe redimensionar su memoria dinámica, haciendo uso de la función *thread_realloc*(), y una vez creada la nueva hebra, esta reserva el bloque de memoria liberado anteriormente a través de la función *thread_alloc*(). Sin embargo, solo se puede hacer *thread_realloc*(), si la hebra no ha consumido toda la memoria, pues en el caso de que se esté usando, se le asignará la parte no usada. Este tipo de gestión adaptativa de memoria presenta una serie de posibles alternativas, que deben ser analizadas:

Tamaño máximo del bloque de memoria: En el experimento anterior, se ha comprobado que las inestabilidades desaparecían usando la librería *ThreadAlloc* con un bloque de memoria igual a 1MB. Sin embargo, debemos de seleccionar un tamaño máximo del bloque de memoria que garantice la desaparición de inestabilidades en cualquier tipo de aplicación, independientemente de la carga de trabajo. Las limitaciones respecto al máximo tamaño de bloque de memoria que se puede reservar vendrán impuestas principalmente por la organización de la memoria, los tamaños de la memoria cache y la memoria principal, y el propio sistema operativo.

Sistemas no dedicados: Este mecanismo de gestión adaptativa de la memoria se complica sustancialmente cuando entran en juego varias aplicaciones que compiten por los recursos, en este caso la memoria del sistema. El gestor de memoria debería permitir liberar memoria dinámica de una aplicación para que otra aplicación pueda hacer uso de ella.

La implementación de la gestión adaptativa de memoria puede estar integrada en el GP ampliando las funcionalidades de la librería *IGP* y realizando ciertas modificaciones a la librería *ThreadAlloc*. Sin embargo, esta propuesta de gestión adaptativa de memoria queda fuera de los objetivos planteados en esta Tesis, por lo que se ha optado por implementar la detención de hebras como solución para mejorar los resultados de las decisiones tomadas en el GP.

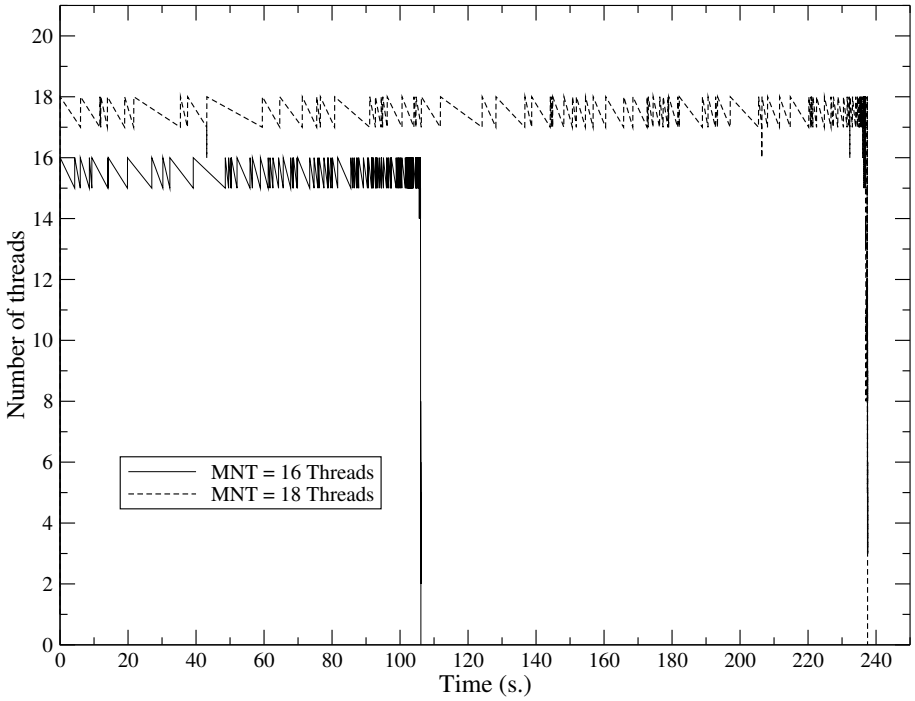


Figura 3.20: Evolución del número de hebras activas de *Local_PAMIGO* con generación no adaptativa de hebras para el problema *Ratz5* con bloques de memoria de 4KB en el sistema *Frida* (16 cores).

3.7. Detención de hebras

El principal problema en relación con la gestión de hebras adaptativa surge principalmente cuando se presentan inestabilidades próximas al tiempo mínimo de ejecución, especialmente cuando las inestabilidades se caracterizan porque se produce una gran diferencia de tiempo de ejecución para una pequeña variación en cuanto al número de hebras activas. Por ejemplo, tal y como se vio anteriormente al utilizar la librería *ThreadAlloc*, las funciones *Ratz5* y *Ratz6* presentan este tipo de inestabilidades en el intervalo comprendido entre 16 y 18 hebras activas cuando se reservan tamaños de memoria igual a 1KB (Figura 3.14) y 4KB (Figura 3.12), e incluso la función *Ratz5* con bloques de 16 KB (Figura 3.16). En todos estos casos se aprecia que el tiempo total de ejecución utilizando 18 hebras duplica al tiempo de ejecución con 16 hebras.

Los métodos adaptativos permiten la generación de una nueva hebra, si se cumple el criterio de decisión utilizado por el GP. Sin embargo, puede suceder que la nueva hebra creada disminuya considerablemente el rendimiento instantáneo de la aplicación (véase la Ecuación 3.2). En los experimentos realizados anteriormente, el número de hebras activas disminuye únicamente cuando alguna hebra finaliza el trabajo asignado, pues no se permite detener ninguna hebra en ejecución aunque disminuya el rendimiento instantáneo.

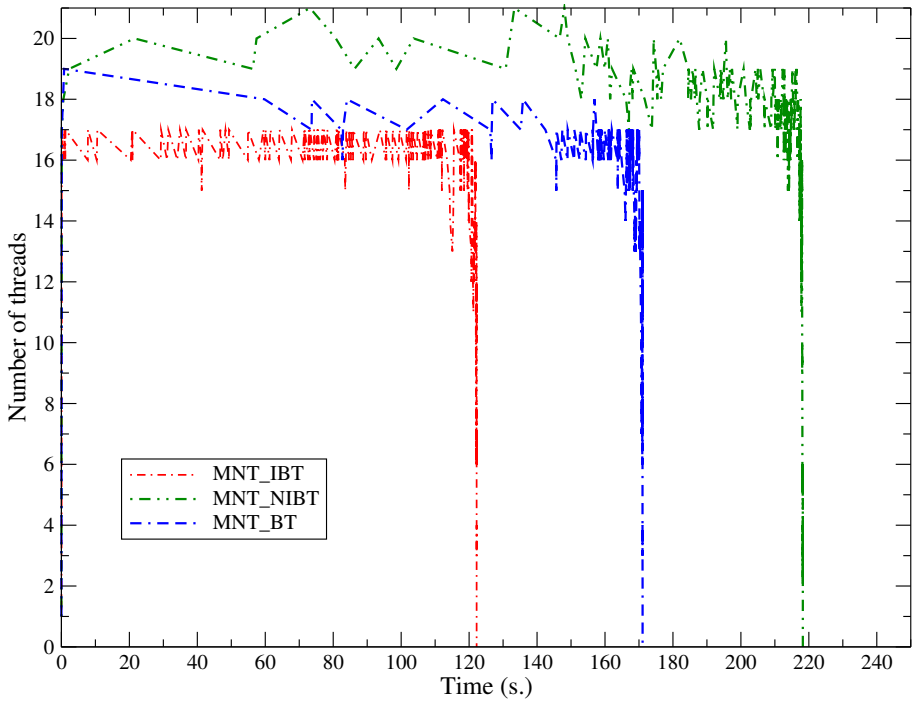


Figura 3.21: Evolución del número de hebras activas de *Local_PAMIGO* con generación adaptativa de hebras para el problema *Ratz5* con bloques de memoria de 4KB en el sistema *Frida* (16 cores).

La Figura 3.20 muestra el número de hebras activas durante la ejecución del algoritmo *Local-PAMIGO* para la función *Ratz5* utilizando generación de hebras no adaptativa. En esta figura se aprecia como el número de hebras, para $MNT = 16$, crece muy rápidamente hasta 16 y cuando una hebra finaliza su carga de trabajo, se genera una nueva hebra para alcanzar el número de hebras límite establecido por el usuario en 16. La aplicación con $MNT = 18$ tiene un comportamiento similiar, salvo que el tiempo total de ejecución es más del doble del tiempo de ejecución con $MNT = 16$. En esta figura se aprecia una mayor fluctuación en el número de hebras para $MNT = 16$ frente a la mostrada con $MNT = 18$, esto se debe a que las hebras generadas con $MNT = 18$ necesitan más tiempo para ejecutar su carga de trabajo, debido a la escasez del número de procesadores. La justificación se encuentra en el considerable aumento del tiempo *IBT* de las hebras, cuando el número de hebras supera al número de procesadores disponibles.

La Figura 3.21 muestra el comportamiento del número de hebras activas durante todo el tiempo de ejecución de la misma aplicación utilizando generación adaptativa de las hebras con los criterios de decisión *MNT_IBT*, *MNT_BT* y *MNT_NIBT*. En los primeros 80 segundos de ejecución, se aprecia claramente como la aplicación ha generado 19 hebras siguiendo las estimaciones iniciales del gestor con el criterio de decisión *MNT_BT*. Sin embargo, este número de hebras es muy superior al indicado por las nuevas estimaciones realizadas, por lo que hay que esperar la terminación de algunas hebras para que

el número de hebras concuerde con las nuevas estimaciones realizadas por el gestor. Este comportamiento también lo experimentan el resto de criterios de decisión.

De los tres gestores analizados, el gestor *MNT_IBT* ofrece el mejor rendimiento, sin embargo, no es una solución aceptable pues el tiempo total de ejecución es un 14 % superior al mínimo tiempo de ejecución obtenido con 16 hebras. Este gestor establece rápidamente 16 hebras y genera un nueva hebra para analizar su comportamiento. Sin embargo, el GP detecta que el rendimiento con 17 hebras no es tan bueno, por lo que no se generan más hebras. Hasta que no termine alguna de las hebras, el rendimiento no volverá a tener un nivel aceptable. Entonces, el gestor *MNT_IBT* generá una nueva hebra, repitiendose este bucle sucesivamente. Por otro lado, el gestor *MNT_NIBT* no tiene en cuenta el tiempo interrumpible, por lo que las estimaciones del número de hebras en la aplicación *Local-PAMIGO* oscilan entre 19 y 20 hebras durante la mayor parte del tiempo de ejecución, produciendo un incremento considerable en el tiempo de ejecución. Esto confirma que la escalabilidad de la aplicación *Local-PAMIGO* con la función *Ratz5* está más limitada por las secciones críticas que por la reserva de memoria. Finalmente, el gestor *MNT_BT* muestra un comportamiento intermedio entre *MNT_IBT* y *MNT_NIBT*.

Como se ha observado, en estas situaciones, el GP cuyo criterio de decisión se basa en *MNT_IBT*, *MNT_NIBT* o *MNT_BT* no suele aproximarse al mínimo tiempo de ejecución obtenido con 16 hebras, sino que el tiempo total de ejecución está más próximo al obtenido con 18 hebras activas. Una posible solución pasa por establecer en el GP la capacidad de detener hebras, tal y como se comentó en la Sección 2.5 del capítulo anterior. De esta forma, cada vez que el gestor estima un número de hebras inferior al número de hebras activas, no espera a que terminen las hebras en ejecución, sino que detiene la ejecución de una hebra, mediante un sistema de selección rotativo.

En esta sección, se ha repetido el experimento de la Sección 3.6, pero en este caso se ha habilitado la detención de hebras en el gestor *KITST* para las funciones *Ratz5* y *Ratz6* exclusivamente. El mecanismo de detención de hebras se ha implementado a nivel de usuario, asignando, de forma rotativa, un intervalo de detención de 10 ms a las hebras, cada vez que el gestor estime un número de hebras inferior al número de hebras activas.

Las Figuras 3.22 y 3.23 muestran el comportamiento de los gestores adaptativos habilitando la detección de hebras a nivel de usuario, utilizando distintos criterios de decisión sobre la aplicación *Local-PAMIGO* para las funciones *Ratz5* y *Ratz6*, con una reserva de bloques de memoria de tamaño 1KB y 4 KB, respectivamente. Una comparativa de estas figuras con las Figuras 3.14 y 3.12 respectivamente, confirman una mejora sustancial en la adaptabilidad del gestor para los distintos criterios de decisión, destacando *MNT_IBT* como el criterio de decisión que más se aproxima al tiempo mínimo de ejecución en ambas situaciones.

3.8. Conclusiones y trabajo futuro

Uno de los aspectos claves de cualquier gestor del nivel de paralelismo en aplicaciones multihebradas consiste en seleccionar un criterio de decisión adecuado que permita estimar correctamente el número de hebras activas en cada momento. En este capítulo se han

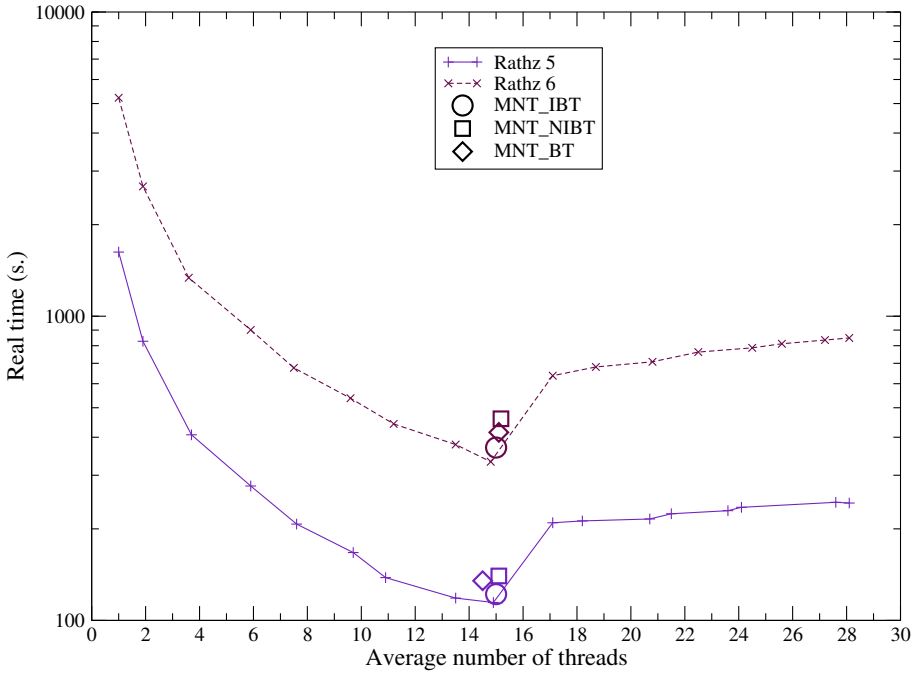


Figura 3.22: Tiempo real de ejecución usando *ThreadAlloc* con un tamaño de bloque de 1KB, habilitando la detención de hebras.

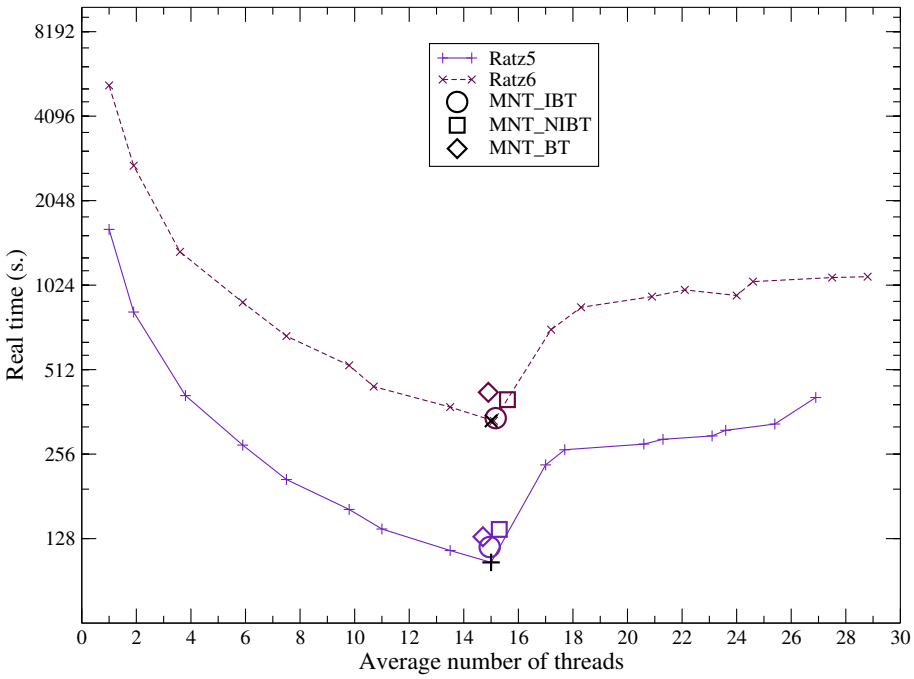


Figura 3.23: Tiempo real de ejecución usando *ThreadAlloc* con un tamaño de bloque de 4KB, habilitando la detención de hebras.

propuesto diferentes criterios de decisión. El más sencillo está basado en el número de procesadores del sistema, pero solo es adecuado para aplicaciones multihebradas muy escalables ejecutadas en entornos dedicados. Se han estudiado criterios de decisión que están basados en el análisis periódico del rendimiento de la aplicación, que pueden aplicarse fácilmente en gestores a nivel de usuario. Otros criterios usan información disponible a nivel de kernel y están basados en el análisis de los retardos que experimentan las hebras activas en tiempo de ejecución. Cada uno de los criterios de decisión desarrollados en este capítulo se han implementado en diferentes gestores tanto a nivel de usuario (*ACW* y *AST*), como a nivel de kernel (*KST* y *KITST*), y se han ensayado sobre la aplicación B&B *Local-PAMIGO* para diferentes tipos de funciones en entornos dedicados [59], [60].

Los distintos experimentos realizados sobre *Local-PAMIGO* ratifican los resultados obtenidos con *Ray Tracing* en el capítulo anterior, donde los GPs a nivel de Kernel mejoran sustancialmente las prestaciones de la aplicación multihebrada, frente a los GPs a nivel de usuario. El potencial de cualquier GP a nivel de kernel crece al disponer de un amplio abanico de criterios de decisión basados en los distintos tiempos de bloqueo que puede experimentar una hebra (*MST*, *MNET*, *MIBT*, *MNIBT*, *MWT*, *MNT_BT*, *MNT_IBT*, *MNT_NIBT* y otros), además de integrar todos los criterios de decisión disponibles a nivel de usuario (número de procesadores ociosos y rendimiento de la aplicación). Los criterios de decisión centrados en minimizar algún tiempo de bloqueo (*MST*, *MNET*, *MIBT*, *MNIBT* y *MWT*) son más restrictivos frente a los criterios basados en la estimación del número de hebras (*MNT_BT*, *MNT_IBT*, *MNT_NIBT* y otros), debido a que generan un menor número de hebras.

A continuación, se comentan varios aspectos que se podrían mejorar o implementar en los GPs diseñados en esta tesis.

3.8.1. Diseñar otros criterios de decisión

En este capítulo se han propuesto y ensayado una amplia variedad de criterios de decisión, todos ellos pensados para tener una baja carga computacional, principalmente en las Fases de *Información* y *Evaluación*. No se descarta el diseño de nuevos criterios de decisión en un futuro, pero se debe tener presente que tanto las ecuaciones que implementen ese nuevo criterio de decisión, como la recopilación de información utilizada por el criterio de decisión, no deberían aumentar considerablemente el tiempo computacional del gestor.

3.8.2. Gestión adaptativa de la memoria

Uno de los factores que afectan al rendimiento de las aplicaciones multihebradas es la gestión dinámica de la memoria, que tal y como se ha comentado en este capítulo, se produce cuando varias hebras intentan reservar y liberar memoria principal, simultáneamente. Una solución factible consiste en utilizar librerías, tales como *ThreadAlloc*, que permitan disminuir el número de peticiones de reservas de memoria solicitadas al SO, para lo cual, cada hebra reserva grandes bloques de memoria, lo que le permite gestionar la memoria localmente sin la intervención del SO. Este principio se podría integrar en el GP implementando un módulo de gestión adaptativa de la memoria.

En un futuro, se podría plantear la posibilidad de habilitar al GP para que gestione dinámicamente y de forma adaptativa toda la memoria disponible en el sistema, entre todas las hebras de las distintas aplicaciones que se están ejecutando simultáneamente. Esta mejora implicaría la creación de nuevas funciones en la librería *IGP*, de forma que el GP pueda gestionar las necesidades de memoria de las distintas hebras, minimizando el tiempo *NIBT* de las hebras.

3.8.3. Detención adaptativa de hebras

El mecanismo de detención de hebras integrado en los distintos GPs, tanto a nivel de usuario, como a nivel de kernel, permiten detener una hebra cada vez que no se verifique el criterio de decisión seleccionado. El usuario debe habilitar/deshabilitar la detención de hebras en la configuración inicial del GP antes de ejecutar la aplicación multihebrada. De esta forma, la detención de hebras permanece habilitada/deshabilitada durante la ejecución total de la aplicación. Sin embargo, la detención de hebras puede beneficiar o perjudicar el rendimiento de la aplicación, dependiendo del criterio de decisión seleccionado, las características intrinsecas de la aplicación, tipo de la carga de trabajo y otras condiciones del entorno. Existen criterios de decisión que generán mejores resultados si no se habilita la detención de hebras, como por ejemplo, el basado en el número de procesadores ociosos y los criterios basados en minimizar algún tiempo de bloqueo. Mientras que otros criterios de decisión pueden mejorar el tiempo total de ejecución al integrar la detención de hebras, tales como el basado en el rendimiento de la aplicación y los criterios basados en la estimación del número de hebras.

Se podría plantear en un futuro la implementación de un mecanismo de detención adaptativa que analice tanto las características de la aplicación multihebrada como el criterio de decisión seleccionado, y decida automáticamente si debe activar o desactivar la detención de hebras. Por ejemplo, los criterios de estimación del número de hebras, tales como *MNT_BT*, *MNT_IBT*, *MNT_NIBT* y otros, informan al GP del número de hebras óptimo según la ecuación aplicada sobre los tiempos de bloqueo de las hebras. Estos criterios permiten al GP generar una nueva hebra cuando el número de hebras estimado es igual o superior al actual. En caso contrario, si el usuario ha habilitado la detención de hebras, el GP detendrá una hebra cada vez que no se verifique el criterio de decisión. Sin embargo, la detención adaptativa podría detener tantas hebras como indique la diferencia entre el número de hebras activas y el número de hebras estimado, cada vez que no se verifique el criterio de decisión.

Capítulo 4

Adaptabilidad en sistemas no dedicados

En capítulos anteriores se han presentado diferentes gestores del nivel de paralelismo (GPs) para aplicaciones multihebradas, y se han estudiado diversos aspectos que permiten a los GPs informar sobre el nivel de paralelismo adecuado a las aplicaciones multihebradas en sistemas dedicados de tal forma que se mejore la eficiencia de la aplicación con el menor número de hebras. Las características intrínsecas de la aplicación, tales como las secciones críticas, reserva de memoria y número de hebras activas, son los factores principales que influyen en las decisiones tomadas por los GPs. En este capítulo se analizará con más detalle la capacidad de los GPs propuestos para facilitar la adaptabilidad de las aplicaciones multihebradas en sistemas no dedicados, donde además de las características de la aplicación bajo estudio, también hay que tener en cuenta la influencia de otras aplicaciones que están en ejecución. Estas otras aplicaciones pueden tener una única hebra o ser multihebradas. El GP ejecutado en sistemas no dedicados podrá gestionar las hebras de una sola aplicación, o de varias aplicaciones simultáneamente, donde los cambios en el sistema se pueden producir en cualquier momento, con la entrada en ejecución de nuevas aplicaciones o con la terminación de alguna de las aplicaciones en ejecución.

4.1. Entorno de ejecución de la aplicación

La arquitectura del sistema donde se ejecuta una aplicación es cada vez más importante, hasta el punto que los compiladores están cada vez más especializados para aprovechar al máximo los recursos del sistema [71] [31]. Sin embargo, otro aspecto a analizar es el comportamiento que experimenta la aplicación cuando se ejecuta en un sistema, donde varios factores pueden incidir directamente en el tiempo real de ejecución. Este entorno de ejecución de la aplicación debe ser analizado correctamente pues influye en gran medida sobre la disponibilidad de recursos en un sistema. Por este motivo, a continuación se realiza una clasificación del sistema según el entorno de ejecución de la aplicación:

- Sistema dedicado: la única aplicación en ejecución dispone de todos los recursos del sistema en todo momento, menos los requeridos por el SO.

- Sistema no dedicado: la aplicación del usuario comparte los recursos del sistema con otras aplicaciones de usuario, que pueden hacer uso de los recursos del sistema en cualquier momento, y con el SO.

El rendimiento de las aplicaciones multihebradas gestionadas por un GP depende directamente de la capacidad del GP para establecer el número de hebras óptimo que garantice el mejor nivel de eficiencia posible. Sin embargo, ese número de hebras se desconoce a priori, y además, puede cambiar durante la ejecución de la aplicación, según cambie la disponibilidad de los recursos del sistema, especialmente en entornos no dedicados. El rendimiento de un sistema dedicado con p procesadores coincide con el rendimiento de la aplicación multihebrada (véase Ecuación 1.2) en ejecución, cuando el número de hebras en ejecución es igual al número de procesadores ($p = n$), es decir:

$$S_{S.dedicado}(p) = S_a(n) = \frac{T(1)}{T(n)}, \tag{4.1}$$

Un sistema no dedicado se podría equiparar a un sistema dedicado en el que los recursos disponibles cambian en el tiempo, ya que el grado de disponibilidad de los recursos para una aplicación cambia en el tiempo. Alguno o todos los procesadores pueden estar ejecutando otras aplicaciones, de forma que la disponibilidad del procesador para una aplicación particular es menor que la que podría ofrecer la aplicación si el sistema fuera dedicado. A diferencia de un sistema dedicado, el rendimiento de un sistema no dedicado no coincide con el rendimiento de una aplicación, sino con el promedio de los rendimientos de todas las aplicaciones en ejecución.

$$S_{S.nodedicado}(n) = \sum_{i=1}^{m} \frac{S_{a_i}(n_i)}{m}, \tag{4.2}$$

Si se desea comparar el rendimiento de un sistema no dedicado con el rendimiento de un sistema dedicado, es importante considerar toda la información relevante sobre el rendimiento de las aplicaciones como: el tiempo real de ejecución, el tiempo de computación, el tiempo de comunicación, el tiempo de sistema, el tiempo de entrada/salida, el tiempo ocioso para cada procesador, etc.

La carga de trabajo en un sistema puede cambiar en cualquier momento y este cambio puede ser mayor en sistemas no dedicados debido a que han entrado en ejecución nuevas aplicaciones o porque ha terminado alguna de las aplicaciones en ejecución. Esto afecta a la gestión realizada por el GP y por lo tanto en la adaptabilidad de las aplicaciones en ejecución. Aunque es difícil evaluar el rendimiento de una aplicación paralela en sistemas no dedicados [3], sin embargo es muy importante hacerlo para permitir su adaptación a los cambios de la carga de trabajo del sistema, aprovechando al máximo el nivel de paralelismo que ofrece el sistema en las nuevas condiciones. Las medidas tradicionales para evaluar el rendimiento del sistema, tales como el tiempo de ejecución, el speed-up y la eficiencia, no son apropiadas para evaluar el rendimiento de sistemas no dedicados.

- Tiempo de ejecución: El rendimiento de un sistema no dedicado puede ser comparado con el rendimiento del mismo sistema dedicado, es decir, el rendimiento de una

aplicación que se obtendría en el sistema dedicado. El tiempo real de ejecución es una buena medida, e indicativo del rendimiento, pero no es suficiente. El tiempo real de ejecución se compone del tiempo de computación (T_{comp}), el tiempo de comunicación (T_{com}), el tiempo de sistema (T_{sys}) y el tiempo en el que el procesador ha estado ocioso (T_{idle}), de la siguiente forma:

$$T_{eje} = T_{comp} + T_{com} + T_{idle} + T_{sys} \tag{4.3}$$

Esta ecuación es válida para un monoprocesador, pero normalmente, la aplicación se ejecuta en un multiprocesador(n_{proc}), por lo que el tiempo real de ejecución será:

$$T_{eje} = \max_{i=0}^{n_{proc}-1} (T^i_{comp} + T^i_{com} + T^i_{idle} + T^i_{sys}) \tag{4.4}$$

El principal inconveniente de esta métrica, es que el tiempo real de ejecución de una aplicación ejecutada en un sistema no dedicado no es representativa de la aplicación, pues depende de la carga de trabajo en el sistema. Una nueva ejecución de la misma aplicación en el sistema no dedicado podría dar tiempos de ejecución muy diferentes, dependiendo de las otras aplicaciones que estén siendo ejecutadas simultáneamente en el sistema, aunque el tiempo de computación de la aplicación sería parecido [1].

- Speed-up y Eficiencia de una aplicación: Las técnicas de calcular el speed-up (véase la Ecuación 1.2) y la eficiencia (véase la Ecuación 1.1) de una aplicación paralela tienen sus inconvenientes, especialmente en sistemas no dedicados. Por ejemplo, el rendimiento de una aplicación depende de los recursos disponibles en el sistema, y la disponibilidad de estos recursos están directamente relacionados con la carga de trabajo de otras aplicaciones ya en ejecución, o que pueden entrar en ejecución en cualquier momento. Una alternativa podría ser realizar una medida del rendimiento medio de la aplicación en cada uno de los procesadores utilizados para calcular el speed-up. Sin embargo, esta alternativa no es totalmente fiable pues no garantiza que la próxima ejecución de la aplicación obtengamos las mismas medidas de rendimiento. Así que se necesita buscar una nueva forma de medir el rendimiento del sistema no dedicado.

Una posibilidad puede ser comparar el tiempo de ejecución de una aplicación en el sistema dedicado y el tiempo de ejecución de la misma aplicación ejecutada en el mismo sistema no dedicado. Esta alternativa permite tener una referencia del máximo rendimiento que podría obtener esta aplicación en el sistema no dedicado. Aunque el máximo rendimiento se mantiene como referencia, el rendimiento obtenido en sistemas no dedicados varía según la carga del sistema.

Otra alternativa, podría consistir en calcular la carga de trabajo realizada por cada procesador en un intervalo de tiempo, de tal forma, que es posible comparar la carga de trabajo realizado por unidad de tiempo con la carga potencial de trabajo que podría ser realizada por el sistema completo, que es determinado sumando la carga de trabajo realizada por cada procesador en un intervalo de tiempo. Esta forma de evaluar el rendimiento en

un sistema no dedicado es más real que el speed-up, pues ignora todas las complicaciones potenciales propias del speed-up en sistema no dedicados.

El objetivo final es dotar a las aplicaciones paralelas gestionadas por el GP del mayor nivel de paralelismo que haga un uso eficiente de los recursos disponibles en el sistema no dedicado.

4.1.1. Sistema dedicado

En capítulos anteriores, se ha comprobado que el principal objetivo de un gestor del nivel de paralelismo (GP) se basa en estimar el número óptimo de hebras que permita mantener una eficiencia alta de la aplicación multihebrada con el menor número de hebras cuando se ejecuta en un sistema dedicado.

Uno de los objetivos a la hora de paralelizar una aplicación multihebrada es el de minimizar el tiempo real de ejecución. Esto se consigue cuando la aplicación se ejecuta usando un número óptimo de hebras en cada instante de tiempo. Otro objetivo es el de obtener una aplicación eficiente que mantiene el speed-up próximo al lineal, o bien una eficiencia por encima de un umbral inferior a 1. Una aplicación puede hacer un uso eficiente de un número diferente de procesadores. Por lo tanto, se desea establecer el número de hebras que sea eficiente (nh_{Eff}) en el mayor número de procesadores posible para obtener de manera eficiente un menor tiempo de ejecución.

Se pueden obtener el menor tiempo real de ejecución de una aplicación de forma no eficiente con un número de hebras (nh_{time}) distinto de nh_{Eff}. El número de hebras que mejora el tiempo de ejecución puede ser distinto del que mantiene una buena eficiencia en el mayor número de procesadores. Normalmente $nh_{Eff} < nh_{time}$. Como estamos buscando un uso eficiente del sistema, nuestros estudios se basarán en establecer nh_{Eff}. En este sentido, el mejor rendimiento en términos de eficiencia de cualquier aplicación multihebrada se consigue cuando se obtiene un buen speed-up (S_{Best}) de la aplicación con el mayor número de procesadores, que depende directamente de establecer el número de hebras óptimo (nh_{Eff}). Así que, se puede definir el mejor rendimiento de la aplicación en un sistema dedicado ($R_{Aplication}$) como:

$$R_{Aplication} = S_{Best} = \frac{t(1)}{nh_{Eff}} \tag{4.5}$$

Una aplicación será eficiente en el sistema dedicado cuando nh_{Eff} sea igual al número de procesadores del sistema. En los capítulos anteriores se comprobó que un buen GP puede establecer el número óptimo de hebras nh_{Eff} en función del criterio de decisión seleccionado, de tal forma que obtenga una buena eficiencia en el mayor número de procesadores posible. Generalmente, la aplicación no tiene porqué ser eficiente cuando se usan todos los procesadores del sistema, y solo ocurre para algunas aplicaciones *HPC* con alta escalabilidad ejecutadas en sistemas dedicados.

4.1.2. Sistema no dedicado

El SO de un sistema no dedicado debe implementar una política de reparto que permita distribuir temporalmente los recursos del sistema entre las aplicaciones en ejecución.

La política de reparto utilizada por el planificador del SO Linux es *CFS* (*Completely Fair Scheduler*) que permite una asignación justa de las unidades de procesamiento a las aplicaciones en ejecución, según las prioridades asignadas a cada aplicación.

El máximo rendimiento de una aplicación en un sistema no dedicado suele ser muy inferior al obtenido en un sistema dedicado. Incluso en casos puntuales donde la aplicación tenga la máxima prioridad obtendrá un rendimiento inferior, dado que la característica *preemptive* del SO obliga a detener las tareas con más prioridad para que entren en ejecución las tareas con menos prioridad, y así garantizar que las tareas menos prioritarias entren alguna vez en ejecución. Sin embargo, para hacer un uso eficiente de un sistema no dedicado, se debe mantener una eficiencia alta para las aplicaciones paralelas gestionadas por el GP que se ejecuten en el sistema y además se debe conseguir que no existan procesadores ociosos. La existencia de procesadores ociosos dependerá de la escalabilidad de las aplicaciones paralelas y de la existencia de otras aplicaciones no gestionadas por el GP.

Los casos extremos son: $\sum_i nh_i^{eff} < p$ y $\sum_i nh_i^{eff} > p$ donde nh_i^{eff} es el número optimo de hebras para la aplicación i ejecutándose en el sistema dedicado. En el primer caso existirán procesadores ociosos si no se ejecutan otras aplicaciones en el sistema, lo que puede ser coherente en terminos ahorro energético. En el segundo caso, el gestor deberá usar valores menores de nh_i^{eff} para mantener el grado de eficiencia de las aplicaciones paralelas en ejecución, pero haciendo uso de todos los procesadores del sistema.

4.2. Transformación de un sistema no dedicado en varios sistemas dedicados

El SO de un multiprocesador está configurado, por defecto, de forma que todos los procesadores del sistema pueden contribuir en la computación de las aplicaciones en ejecución, ya que periódicamente el planificador del SO rebalancea la carga de trabajo entre las distintas colas de ejecución. De esta forma, en un sistema no dedicado, un mismo procesador puede tener asignadas hebras de distintas aplicaciones en su cola de ejecución. Sin embargo, existe la posibilidad de reconfigurar el SO para crear grupos aislados de procesadores, de tal forma que las aplicaciones se ejecuten exclusivamente entre los procesadores del grupo asignado. Esta característica permitiría transformar un sistema no dedicado en varios sistemas dedicados, donde cada grupo de procesadores aislados se encargue de ejecutar una única aplicación.

Por otro lado, el planificador del SO puede sobrecargarse en sistemas con muchos procesadores a la hora de realizar el rebalanceo de la carga entre todos los procesadores del sistema. Por este motivo, una forma de reducir la sobrecarga del planificador sería aislando diferentes grupos de procesadores.

Si se desea transformar un sistema no dedicado en varios sistemas dedicados, se debe tener información acerca de cuantas y que tipo de aplicaciones se ejecutan simultáneamente, creando tantos grupos de procesadores aislados como número de aplicaciones en ejecución. En el supuesto de que el número de aplicaciones supere el número de procesadores en el sistema, uno o varios grupos de procesadores aislados trabajaran como sistemas no dedicados. A continuación detallamos varias posibilidades para transformar un sistema

no dedicado en varios sistemas dedicados.

- Aislar procesadores: Aislar todos los procesadores del sistema fuerza a ejecutar el Kernel exclusivamente en el procesador 0 y todos los procesadores aislados son excluidos del balanceo de la carga realizado por el planificador del SO. Cada una de las hebras debera seleccionar el procesador, a través de la llamada al sistema $sched\_setaffinity()$. Además el kernel reduce su actividad en los procesadores aislados y solo se ejecuta cuando se realiza una llamada al sistema.

  El procedimiento para aislar procesadores consiste en:

  1. Editar el fichero */boot/grub/menu.lst*.
  2. Incluir al final de la línea del kernel, el parámetro de arranque *isolcpus=<cpu number>, ...,<cpu number>*. Por ejemplo, para aislar todos los procesadores de un sistema con 8 procesadores se debería añadir *isolcpus=1,2,3,4,5,6,7*.
  3. Reiniciar el sistema.

  Esta metodología es muy restrictiva ya que no permite aislar procesadores en caliente, por lo que se debe configurar a priori, antes del arranque del sistema.

- Crear grupos de procesadores: El SO proporciona otro mecanismo más flexible para confinar aplicaciones en un grupo de procesadores, de forma que el planificador del SO permita rebalancear las tareas de esa aplicación entre los procesadores del grupo, sin permitir su ejecución en otro procesador que no pertenece al grupo. Este mecanismo se basa en la creación de *cpuset*, que a diferencia del aislamiento de procesadores, permite al planificador del SO rebalancear la carga de trabajo entre los procesadores de distintos grupos, a no ser que se configuren grupos exclusivos, en cuyo caso, solo se permitirá el rebalanco entre los procesadores del mismo grupo. La creación de grupos de procesadores se realiza jerárquicamente, de tal forma, que el grupo principal, denominado grupo raíz, está constituido por todos los procesadores del sistema. A continuación, se muestra el procedimiento que debe realizar el administrador del sistema para crear los distintos grupos:

  1. Verificar la existencia del directorio */sys/fs/cgroup/cpuset*, y en su defecto, se debe crear.
  2. Posteriormente, se comprueba que este directorio está montado como *cpuset*, de lo contrario se debe ejecutar el siguiente comando: *mount -t cgroup -ocpuset cpuset /sys/fs/cgroup/cpuset*.
  3. En este momento, se puede generar un nuevo grupo de procesadores creando un directorio dentro de */sys/fs/cgroup/cpuset/* con el nombre del grupo, así como crear el fichero *cpuset.cpus*, asignando los procesadores que conforman el nuevo grupo. También se debe asignar memoria al grupo a través del fichero *cpuset.mems*, y crear el fichero *tasks* donde se irán almacenando los *pid* de cada tarea que se está ejecutando en el grupo de procesadores. Por ejemplo, si

deseamos crear un grupo denominado *Group1* que contenga los procesadores 2, 3, 4 y 5 se deberán ejecutar los siguientes comandos:

- *cd /sys/fs/cgroup/cpuset*
- *mkdir Grupo1*
- *cd Grupo1*
- */bin/echo 2-5 >cpuset.cpus*
- */bin/echo 1 >cpuset.mems*
- */bin/echo $$ >tasks*

4. Por defecto, todos los grupos creados tienen la misma configuración que el grupo raíz, por lo que se permite el balanceo de la carga entre los distintos grupos. Sin embargo, si se desea configurar los grupos como grupos exclusivos, se deberá activar los flags *cpuset.cpu_exclusive* y *cpuset.mem_exclusive*, creando sus respectivos ficheros.
5. Si por algún motivo se desea desactivar el rebalanceo de la carga entre los procesadores de un mismo grupo, se deberá desactivar el flag *cpuset.sched_load_balance*.
6. Finalmente, si se desea redimensionar los grupos de procesadores, se permite eliminar grupos, asignar nuevos procesadores a un grupo, e incluso eliminar algún procesador de un grupo ya existente.

Independientemente del mecanismo escogido para convertir el sistema no dedicado en varios sistemas dedicados, todas las aplicaciones comienzan ejecutándose en el mismo procesador donde se ejecuta el Kernel, es decir, en el procesador 0. Tanto si se han aislado todos los procesadores, como si se han creado grupos de procesadores exclusivos, el planificador del SO ya no realiza balanceo de la carga entre todos los procesadores, tan solo entre los procesadores del mismo grupo exclusivo. Puede dejarse al GP la responsabilidad de planificar la asignación de procesadores entre las aplicaciones en ejecución, de tal forma que el GP asignará un procesador, o grupo de procesadores, a cada una de las aplicaciones. El proceso principal de cada aplicación multihebrada gestionada por el GP solicitará al SO su ejecución en el procesador asignado por el GP antes de crear la primera hebra. La función *IGP_Initialize()* ejecutada por el proceso principal es la responsable de solicitar al SO la asignación del proceso actual al procesador seleccionado, a través de la función *sched_setaffinity(0, cpu)*, donde el valor *0* representa al proceso actual y la variable *cpu* contiene la máscara del procesador asignado por el GP.

El Algoritmo 4.2.1 describe un sencillo planificador de aplicaciones implementado en el GP a nivel de kernel. La función *Schedule_GP()* es llamada por la función *schedule()*, disponible en el fichero *sched.c* del código fuente del kernel. Esta función básicamente comprueba si se trata de un sistema dedicado o no. Para ello, básicamente se contabiliza el número de tareas en ejecución a partir de la variable *nr_running* de la cola de tareas de cada uno de los procesadores en el sistema. Si se trata de un sistema dedicado, la variable *num_process* contabilizará únicamente el proceso principal de la aplicación que solicita ser gestionada (línea 12), en cuyo caso el planificador del GP no necesita devolver ningún dato.

Algoritmo 4.2.1 : Planificador del GP en un sistema no dedicado.

```
(1)  funct Schedule_GP()
(2)    var num_process, id_cpu, mask_cpu, idle_cpu, minload_cpu;
(3)    for (each_cpu(id_cpu, mask_cpu))
(4)      rq = cpu_rq(id_cpu);                    Selecciona la cola de tareas del procesador.
(5)      num_process = num_process + rq- > nr_running;
(6)      if (idle_task(i))                       Buscar procesadores ociosos.
(7)        idle_cpu = id_cpu;
(8)      load = weighted_cpuload(id_cpu);        Buscar procesador con menor carga.
(9)      if (min_load > load)
(10)       min_load = load;
(11)       minload_cpu = id_load;
(12)   if (num_process > 1)                      Sistema no dedicado
(13)     if (empty(idle_cpu))
(14)       return (get_mask_cpu(idle_cpu))
(15)     return (get_mask_cpu(minload_cpu))
(16)   return                                    Sistema dedicado
```

En el supuesto de detectar un sistema no dedicado, la función *Schedule_GP()* busca algún procesador ocioso. Para ello el GP chequea las colas de trabajo de cada procesador (línea 6). Si todos los procesadores tienen asignado alguna tarea, entonces se busca aquel procesador que tenga la menor carga de trabajo (línea 9). Finalmente, el planificador del GP devuelve al proceso principal de la aplicación el procesador seleccionado. El resto de hebras que cree esta aplicación comenzarán a ejecutarse en el mismo procesador, sin embargo, si este procesador pertenece a un grupo exclusivo de procesadores, el planificador del SO rebalanceará las hebras creadas entre todos los procesadores del grupo.

La estrategia basada en transformar el sistema no dedicado en varios sistemas dedicados evita la concurrencia de distintas aplicaciones sobre los mismos procesadores, por lo que no suele ofrecer buenos niveles de eficiencia cuando alguna de las aplicaciones en ejecución es poco escalable, ya que el número de procesadores del grupo asignado puede exceder la escalabilidad de la aplicación. Esto también sucede cuando el número de aplicaciones en ejecución es inferior al número de grupos de procesadores definidos, dejando grupos de procesadores ociosos. En ambos casos, una posible solución es la de redefinir el tamaño de los grupos de procesadores. Además, otro aspecto que incidiría negativamente es la ejecución de aplicaciones secuenciales que no son gestionadas por el GP, por lo que el GP debería chequear periódicamente la cola de tareas de los procesadores, para redimensionar los grupos de procesadores si detecta que es necesario.

El mecanismo para redimensionar los grupos de procesadores es una solución lenta que puede plantear ciertas limitaciones en la eficiencia del sistema. Además, estos escenarios no permiten verificar la correcta adaptabilidad de la aplicación gestionada por el GP al sistema no dedicado, ya que la transformación del sistema a sistemas dedicados permitiría estudiar la adaptabilidad de la aplicación a sistemas dedicados, ya estudiados en capítulos

anteriores. Además, un GP debería evitar tener que realizar el rebalanceo de las tareas entre los procesadores, y dejar esta capacidad al planificador del SO.

4.3. Estrategias de planificación

El objetivo de los sistemas no dedicados donde varias aplicaciones se pueden ejecutar concurrentemente en todos los procesadores del sistema, se centra en mantener una eficiencia alta en la ejecución de las aplicaciones con el mayor número posible de procesadores. El comportamiento que experimenta la ejecución de una aplicación en sistemas no dedicados se vé influido directamente por las características internas de la aplicación, y por la ejecución concurrente de otras aplicaciones que compiten por los recursos del sistema. En este contexto se define:

Definición 4.3.1 *Se denomina Intervalo de Terminación (*IT*) al intervalo de tiempo que transcurre desde que ha finalizado la aplicación más rápida hasta que finaliza la aplicación más lenta. Esta definición es válida para situaciones en la que la misma aplicación es ejecutada múltiples veces, simultáneamente.*

Se pretende realizar un uso eficiente del sistema no dedicado manteniendo todos los procesadores ocupados, pero la creación de hebras de cada aplicación depende de la estrategia de planificación seleccionada, de tal forma que el GP establezca el número de hebras óptimo (nh_i^{eff}) para cada aplicación, dejando al planificador del SO la capacidad de rebalancear las tareas entre todos los procesadores del sistema. En esta situación se plantean diferentes estrategias para decidir que valor $MaxThreads_i$ es el mejor que puede ser asignado a cada aplicación. Por ejemplo, se puede plantear una estrategia de planificación basada en minimizar el tiempo de ejecución de una o varias aplicaciones, frente al resto de aplicaciones. Mientras que otra estrategia de planificación puede ser promediar el tiempo de ejecución, de tal forma que, exista un reparto equitativo de recursos entre las aplicaciones, siempre que las aplicaciones mantengan un nivel de eficiencia aceptable. A continuación, se muestran algunas posibles estrategias de planificación:

- Minimizar tiempo de ejecución de una aplicación: Esta estrategia prioriza una de las aplicaciones en ejecución frente al resto, de tal forma, que la mayoría de los recursos del sistema se asignarían a la aplicación más prioritaria, mientras que el resto de aplicaciones ejecutan un número reducido de hebras cuando su prioridad esté en un nivel intermedio, llegando incluso a estar detenidas, si su prioridad es muy baja.

  En este sentido, el GP permitirá crear un mayor número hebras a la aplicación prioritaria, frente al resto de aplicaciones. Si la aplicación prioritaria es poco escalable, el GP puede asignar más hebras a las aplicaciones menos prioritarias. Por otro lado, si aparecen aplicaciones secuenciales en ejecución, el GP debería ser capaz de adaptar la ejecución de la aplicación prioritaria a los procesadores disponibles.

- Reparto equitativo: Esta estrategia consiste en repartir equitativamente todos los recursos del sistema entre las distintas aplicaciones en ejecución. En este sentido,

cuando se ejecutan varias aplicaciones en un sistema no dedicado, el GP permitirá crear un número similar de hebras a cada aplicación dependiendo de su nivel de escalabilidad.

- Eficiencia energética: Uno de los aspectos más relevantes en la actualidad, es el impacto que la ejecución de las aplicaciones multihebradas produce sobre el consumo de energía en los procesadores multicore. Por este motivo, el número óptimo de hebras debe abarcar un rango de valores que maximice la eficiencia energética. Esta estrategia se puede plantear tanto en sistemas dedicados como en sistemas no dedicados. Para ello se propone estimar el trabajo realizado por watio en cada uno de los procesadores del sistema. A partir de esta información, el GP estimará el número de hebras en ejecución dependiendo de un umbral máximo de energía consumida. Por ejemplo, si en un sistema dedicado, una aplicación multihebrada obtiene el menor tiempo de ejecución utilizando n hebras con el 100 % de consumo energético, se puede mejorar el nivel de eficiencia energética si utiliza $n-1$ hebras, siempre que el consumo energético no supere el umbral establecido por el administrador del sistema (por ejemplo, 90 %). Aunque en este caso, aumente el tiempo real de ejecución, respecto del mínimo tiempo de ejecución obtenido con n hebras.

  Las últimas tendencias actuales se centran en reducir el consumo de energía tanto como sea posible para crear zonas de enfriamiento dentro del chip. Las investigaciones se centran en dos vías posibles, por un lado, proponen técnicas de *DVS* (*Dynamic Voltage Scaling*) que permite reducir el consumo de energía realizando un ajuste óptimo del voltaje de cada procesador. Por otro lado, se proponen técnicas de balanceo de la carga de trabajo de cada procesador migrando tareas de los procesadores ubicados en las zonas más calientes del chip a los procesadores vecinos. Por lo que el número de hebras debe coincidir con el número de procesadores activos permitiendo a los no activos el enfriamiento del chip.

  Una de las técnicas propuestas es usar un método bien conocido de particionamiento heurístico, denominado *WF* (*Worst Fit*), basado en migrar cada una de tareas entre los procesadores solo una vez, justo en el momento de entrada al sistema. Modificar la política del planificador dinámico del *WF* ofrece buenos resultados en sistemas con el *DVS* habilitado, mejorando el consumo de energía en un factor de 2,74 veces [11]. Otros establecen políticas de migración de tareas de un procesador a sus procesadores vecinos para reducir la temperatura en las zonas calientes del chip, reduciendo la sobrecarga, no realizando migraciones innecesarias [22]. Sin embargo, uno de los factores más difíciles de realizar es medir el consumo de energía por cada tarea en ejecución, cuya métrica suele generar datos erróneos en sistemas dedicados. En [36] se propone un soporte hardware para reducir el error en la medida del consumo de energía por cada tarea de un 12 % a un 4 %.

  Esta estrategia permite la ejecución concurrente de distintas aplicaciones en un sistema no dedicado, de tal forma que, el GP consigue reducir el consumo energético estableciendo un número de hebras adecuado y deteniendo hebras en ejecución. Además, el GP puede crear un grupo exclusivo de computadores que contenga un número

de procesadores cuyo consumo no supere el umbral establecido, dejando fuera del grupo aquellos procesadores que deben quedar ociosos, lo que evitará que el planificador del SO rebalancee la carga de trabajo entre todos los procesadores.

- Competir por los recursos: Esta estrategia de planificación permite al GP asignar hebras a las distintas aplicaciones en ejecución según el orden de llegada de las solicitudes. De esta forma, cuando el GP recibe simultáneamente varias peticiones para crear una hebra, ya sea de una aplicación o de distintas aplicaciones, se atenderá a aquella aplicación que primero lo solicite, descartando el resto de peticiones. Este escenario permite que el número de hebras de unas aplicaciones crezca frente a otras aplicaciones ya que las aplicaciones con un mayor número de hebras acceden al gestor más frecuentemente. Este mecanismo reduce el tiempo de computación del GP.

De todas las estrategias planteadas, la estrategia basada en la competición por los recursos en sistemas no dedicados es la que mejor permite observar la adaptabilidad de las aplicaciones implementada en los distintos GP propuestos. A continuación, se exponen diferentes experimentos sobre un sistema no dedicado con los GP diseñados en capítulos anteriores bajo la estrategia basada en la competición por los recursos.

4.4. Experimentación

La experimentación sobre el sistema no dedicado se ha realizado de forma controlada, lo que permitirá analizar los resultados con más facilidad. Para ello se ha seleccionado una única aplicación cuyo comportamiento no presenta irregularidades destacables en sistemas dedicados (véase la Sección 3.5), ejecutando varias instancias de la misma aplicación para resolver el mismo problema.

Todos los experimentos han sido realizados sobre una arquitectura multicore con cuatro Quad-Core AMD Operton 8356, a 2,30 GHz y 64 GB de memoria RAM con la versión 2.6 de Linux. La aplicación multihebrada ha sido Local-PAMIGO utilizando hebras PSOIX y librería C-XSC para implementar la aritmética de intervalos del algoritmo B&B. Este tipo de aplicaciones tienen un coste computacional muy diferente si se varía la precisión requerida para la solución, aunque se resuelva la misma instancia del problema. En todos los casos, se ha utilizado la función Kowalik con una precisión $\epsilon = 10^{-5}$ como representativa de un problema de tamaño medio.

4.4.1. Ejecución no adaptativa

Este primer experimento ha consistido en ejecutar concurrentemente un número de instancias (n) de la misma aplicación Local-PAMIGO, con el mismo número máximo de hebras activas (*MaxThreads*) para cada instancia, donde *MaxThreads* ha sido establecido por el usuario.

La Figura 4.1 muestra el tiempo consumido por la ejecución no adaptativa, obtenido para un conjunto de experimentos donde se han testeado varios valores de n y *MaxThreads*. En cada uno de los experimentos se han ejecutado (n=1, 2, 4, 8, 16 y 32) instancias

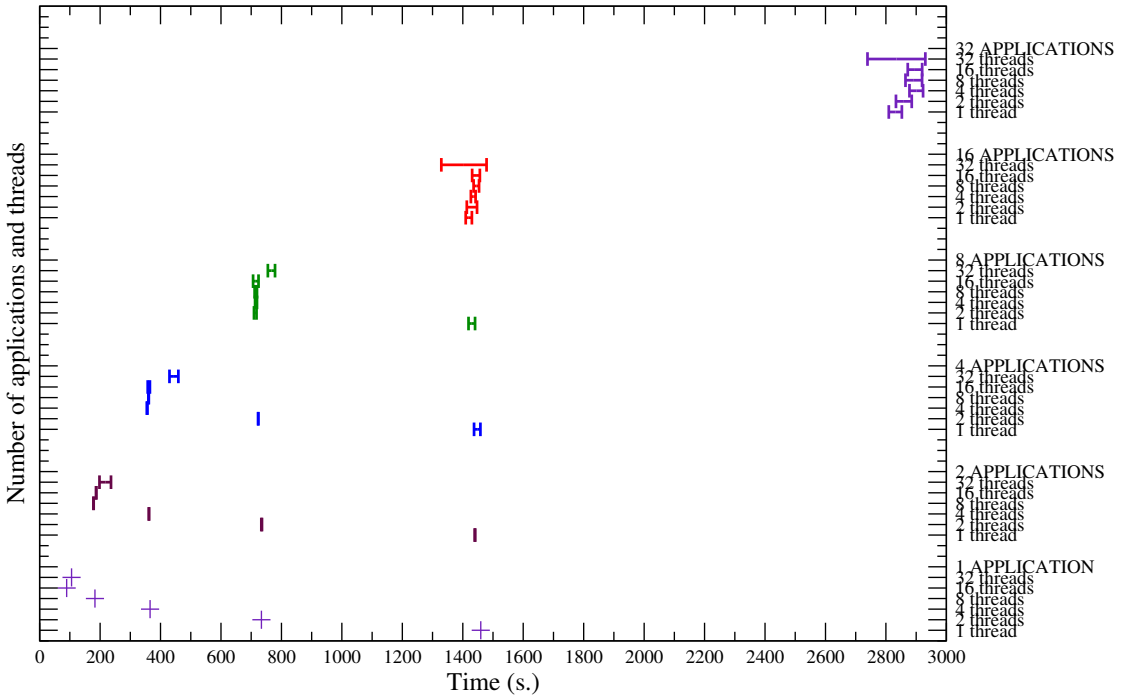


Figura 4.1: Intervalo de Terminación (IT) para varias instancias de la misma aplicación.

de la misma aplicación usando varios valores del número máximo de hebras activas por aplicación ($MaxThreads$=1, 2, 4, 8, 16 y 32 hebras). De tal forma que para $n = 1$ se trata de un sistema dedicado, mientras que para $n > 1$ sería un sistema no dedicado.

Este experimiento refleja el modo estándar utilizado por un usuario, para determinar el valor de $MaxThreads$ que minimiza el tiempo de ejecución con un n determinado. A esta metodología se denomina ejecución estática, pues es el usuario quien establece a priori el número de hebras activas de todas las aplicaciones durante el tiempo real de ejecución. Como era de esperar, el mejor resultado estático (SBr_n=*Static Best result*) se obtiene en sistemas dedicados (SBr_1 = *Static Best result para una aplicación*), con un tiempo real de 89.47 s. para $MaxThreads$=16 hebras. Por lo tanto, la heuristica común para establecer el número de hebras en ejecución igual al número de procesadores ($MaxThreads = p$) obtiene los mejores resultados en este caso.

En sistemas no dedicados ($n > 1$), los tiempos de ejecución de las n aplicaciones se muestran en la Figura 4.1 con intervalos horizontales delimitados según la Definición 4.3.1. Como se puede observar, el intervalo de ejecución para n aplicaciones aumenta conforme se incrementa el número de aplicaciones y el número de hebras por aplicación. El principal objetivo se centra en minimizar el máximo tiempo de ejecución de las aplicaciones usando el mínimo número de hebras en ejecución. Los mejores resultados de la ejecución estática para n aplicaciones (SBr_n) ejecutándose simultáneamente, para valores de n = 2, 4, 8, 16 y 32, han sido obtenidos con $MaxThreads$ igual a 8, 4, 2, 1 y 1, respectivamente. Así que, el número de hebras por aplicación ($NRTh$) debería disminuirse cuando el número

de aplicaciones aumenta, puesto que los recursos computacionales deben compartirse. Una buena práctica para establecer *NRTh* por aplicación es $MaxThreads = \text{máx}\{1, \lfloor \frac{p}{n} \rfloor\}$. Esto ocurre cuando una única aplicación realiza un trabajo computacionalmente costoso, así que esta aplicación es capaz de tener ocupados todos los procesadores del sistema. Este es el caso de la aplicación *Local-PAMIGO* cuando trabaja con la función *Kowalik* con $\epsilon = 10^{-5}$.

En un sistema dedicado donde se ejecuta una única aplicación altamente escalable, el tiempo real de ejecución disminuye considerablemente conforme se incrementa el número total de hebras a ejecutar, siempre que no se supere el número de unidades de procesamiento. Sin embargo, en sistemas no dedicados donde se ejecutan simultaneamente varias aplicaciones, el tiempo real de ejecución aumenta considerablemente respecto de su ejecución en un sistema dedicado, si se mantiene el número de hebras usado en el sistema dedicado. Para analizar la calidad de los resultados obtenidos en sistemas no dedicados (SBr_n) usaremos como referencia el tiempo de ejecución obtenido en sistemas dedicados (SBr_1), de forma que asumiremos que el mejor resultado hipotetico para n aplicaciones (HBr_n) puede ser aproximado por $n \cdot SBr_1$.

Tabla 4.1: Comparativa entre tiempo real de ejecución de HBr_n y el intervalo de terminación para SBr_n para n aplicaciones.

n	1	2	4	8	16	32
$NRTh$	16	16	16	16	16	16
HBr_n	89.47	178.94	357.88	**715.76**	1431.52	2863.04
$NRTh$	16	8	4	2	1	1
SBr_n	89.47	**178.04**	**354.58**	**709.35**	**1409.75**	**2809.48**
		178.92	**356.74**	717.53	**1430.23**	**2852.80**

La Tabla 4.1 muestra una comparativa del tiempo real de ejecución en segundos entre el mejor resultado hipotético (HBr_n) y el mejor resultado estático (SBr_n) para n aplicaciones. Se muestran dos filas para SBr_n, la fila superior de SBr_n muestra el tiempo real de ejecución de la aplicación más rápida, y la fila inferior de SBr_n corresponde a la aplicación más lenta en ejecutarse. El tiempo real de ejecución para el resto de aplicaciones se encuentra entre ambos límites. En negrita se resaltan los mínimos tiempo de ejecución para n aplicaciones, donde se puede destacar que prácticamente en todas las situaciones las aplicaciones ejecutadas concurrentemente (SBr_n) terminan antes que si se realiza una ejecución secuencial (HBr_n). Esto demuestra que un correcto reparto de los recursos mejora la eficiencia del sistema. Además, cuando el sistema se encuentra sobrecargado con 32 aplicaciones, la ejecución concurrente es más rápida que la ejecución secuencial.

4.4.2. Ejecución adaptativa

El segundo experimento se basa en ejecutar las mismas n instancias de la aplicación *Local-PAMIGO* utilizando los gestores de adaptación dinámica basados en GP a nivel de usuario (ACW) y GP a nivel de kernel, tales como *KST* y *KITST*. El criterio de decisión utilizado en los gestores a nivel de kernel está basado en la estimación del número de hebras según

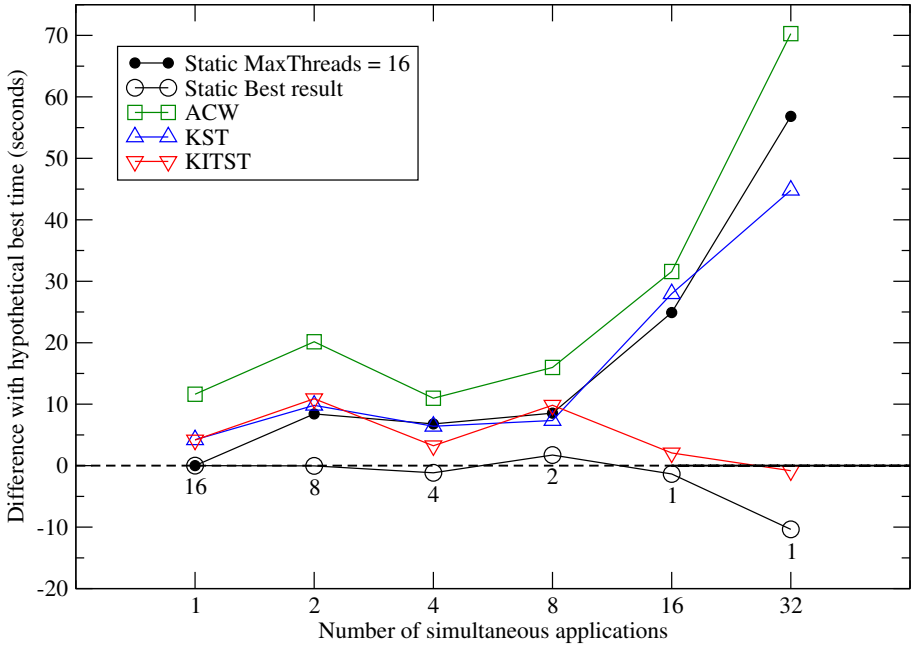


Figura 4.2: Máximo tiempo de ejecución respecto del mejor tiempo hipotético tanto para la ejecución adaptativa y no adaptativa de n aplicaciones concurrentemente.

el tiempo interrumpible que las hebras han estado bloqueadas. Además, en ninguno de los gestores se ha habilitado la detención de hebras.

La Figura 4.2 muestra las diferencias entre HBr_n (linea discontinua con valor cero), el máximo tiempo de ejecución por aplicación usando ejecución estática Static MaxThreads = 16, la mejor ejecución estática (SBr_n) y los GP dinámicos (ACW, KST y $KITST$) para $n = 1$, 2, 4, 8, 16 y 32 aplicaciones ejecutandose simultaneamente. La ejecución estática Static MaxThreads = 16 corresponde a la ejecución realizada por el usuario, quien ejecuta todas las aplicaciones con $MaxThreads = p$, independientemente de n. Sin embargo, la mejor ejecución estática (SBr_n) muestra los mejores valores de $MaxThreads$ para los diferentes valores de n. Como se puede observar en la Figura 4.2, los resultados de ACW, Static MaxThreads = 16 y KST son peores que sus correspondientes HBr_n, principalmente para $n > 8$. Sin embargo, el gestor $KITST$ y la mejor ejecución estática (SBr_n) obtienen resultados próximos o mejores que los propuestos por HBr_n. El principal inconveniente de SBr_n es que se deben realizar varias ejecuciones con diferentes valores de $MaxThreads$ para determinar el mejor valor de $MaxThreads$ para cada valor de n. Sin embargo, los GP dinámicos establecen $NRTh$ de forma adaptativa, y por lo tanto, no necesitan ejecutar experimentaciones adicionales.

La Figura 4.3 muestra el número medio de hebras en ejecución ($AvNRTh$) utilizado para n aplicaciones, en cada uno de los GP. Si se observan las Figuras 4.2 y 4.3 se puede concluir que es mejor usar un valor $NRTh$ tan pequeño como sea posible. El problema consiste en determinar el valor de ese número en cada caso. Por ejemplo, el mejor $NRTh$

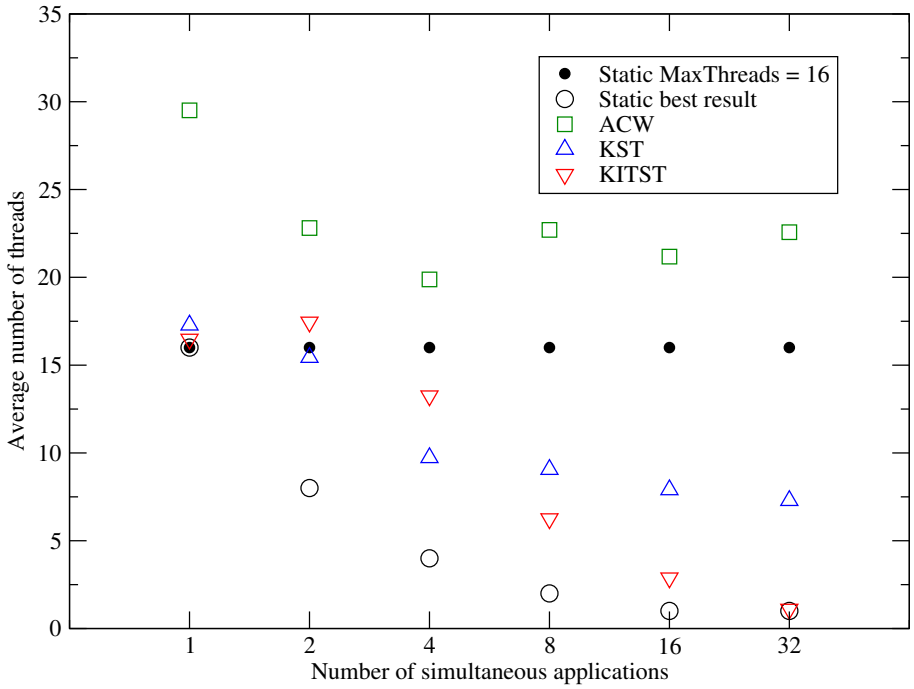


Figura 4.3: Número promedio de hebras para la ejecución adaptativa y no adaptativa de n aplicaciones simultáneamente.

será diferente de $MaxThreads = \text{máx}\{1, \lfloor \frac{p}{n} \rfloor\}$ cuando la aplicación no puede abastecer de suficiente trabajo a los p procesadores. Además, *NRTh* debería variar durante la ejecución de una aplicación irregular. Todas estas consideraciones hacen de este problema un desafio a resolver.

El escenario más desfavorable se produce cuando se ejecutan 32 aplicaciones simultáneamente debido a la sobrecarga que experimenta el sistema. Las Figuras 4.4 y 4.5 muestran la evolución temporal del *NRTh* de cada una de las 32 aplicaciones ejecutadas simultáneamente, para la ejecución estática estableciendo el parámetro *MaxThreads* en 16 hebras, y el gestor adaptativo a nivel de usuario *ACW*, respectivamente. Como se puede observar en la Figura 4.4 la ejecución estática tiene un comportamiento predecible pues mantiene el *NRTh* igual o próximo a *MaxThreads* durante la mayor parte del tiempo de ejecución, obteniendo así pequeñas diferencias en el tiempo real de ejecución de todas las aplicaciones.

Por otro lado, la Figura 4.5 muestra los resultados obtenidos cuando se usa el gestor adaptativo *ACW*. Este GP genera hebras dependiendo del valor asignado al parámetro *Umbral* (véase la Ecuación 3.3). En estos experimentos se ha establecido un mismo valor de *Umbral*=1.0, para todas las aplicaciones. Como se puede observar, este GP genera un elevado número de hebras en las aplicaciones existentes, y una mayor fluctuación en la generación de nuevas hebras. Una ventaja del gestor *ACW* es que un elevado número de las 32 aplicaciones terminan mucho antes con el gestor *ACW* frente a la ejecución estática.

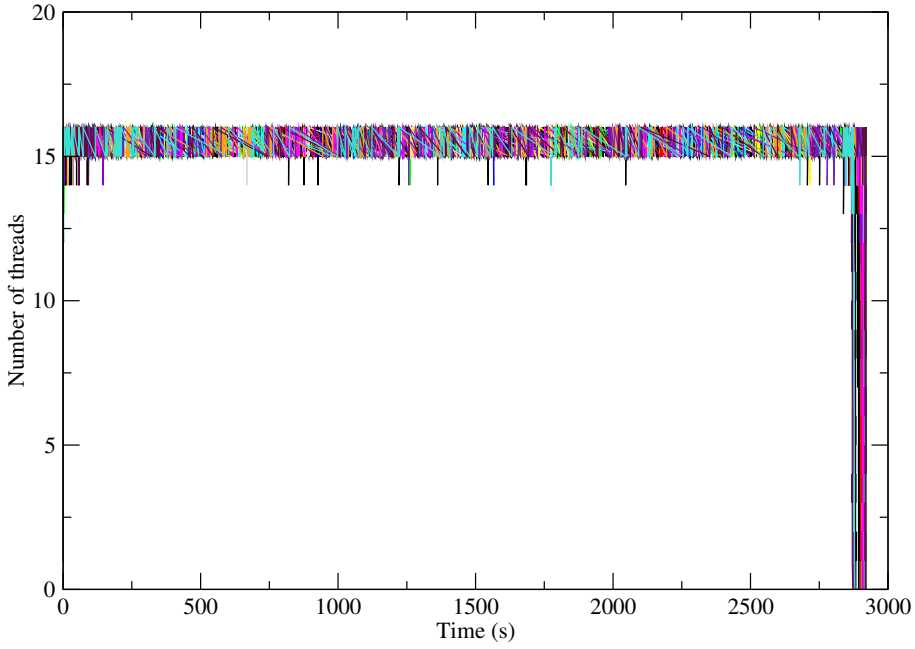


Figura 4.4: Evolución temporal del número de hebras para la ejecución no adaptativa de 32 aplicaciones con *MaxThreads*=16 para cada una.

Si se suman los tiempos de ejecución de todas las aplicaciones, se obtiene que la ejecución estática tarda un 11 % más que utilizando el gestor ACW. Otro detalle a resaltar es que el intervalo de terminación es considerablemente muy superior para el gestor ACW, frente a la ejecución estática, rondando el 25 % del tiempo de terminación máximo, debido a que el número de hebras creado para cada aplicación durante la ejecución es muy diferente. La justificación se encuentra en que aquellas aplicaciones con más hebras en ejecución reducirán su tiempo de ejecución, frente al resto de aplicaciones.

Las Figuras 4.6 y 4.7 muestran la evolución temporal del número de hebras usando los gestores dinámicamente adaptativos KST y $KITST$, respectivamente. Como se puede observar, el gestor KST genera fluctuaciones del número de hebras similares al gestor ACW, salvo que el intervalo de terminación es menor. Además, KST produce un valor de $AvNRTh$ considerablemente menor que el generado por el gestor ACW, aunque es mayor que el generado por SBr_n (ver Figura 4.3). Por otro lado, el gestor KST produce peores valores del máximo tiempo de ejecución si lo comparamos con SBr_n (ver Figura 4.2). También se aprecia, que aquellas aplicaciones que obtienen un menor número de hebras durante la competición por los recursos, corresponde con las aplicaciones que tardan más en terminar su ejecución.

Comparando los gestores KST y $KITST$, los valores de $AvNRTh$ para ambos gestores disminuye conforme aumenta el número de aplicaciones (ver Figura 4.3). Esta disminución es mayor en el gestor $KITST$ debido a que este gestor chequea la regla de decisión únicamente cuando se ejecutan las hebras ociosas del SO. Cuando el número de aplica-

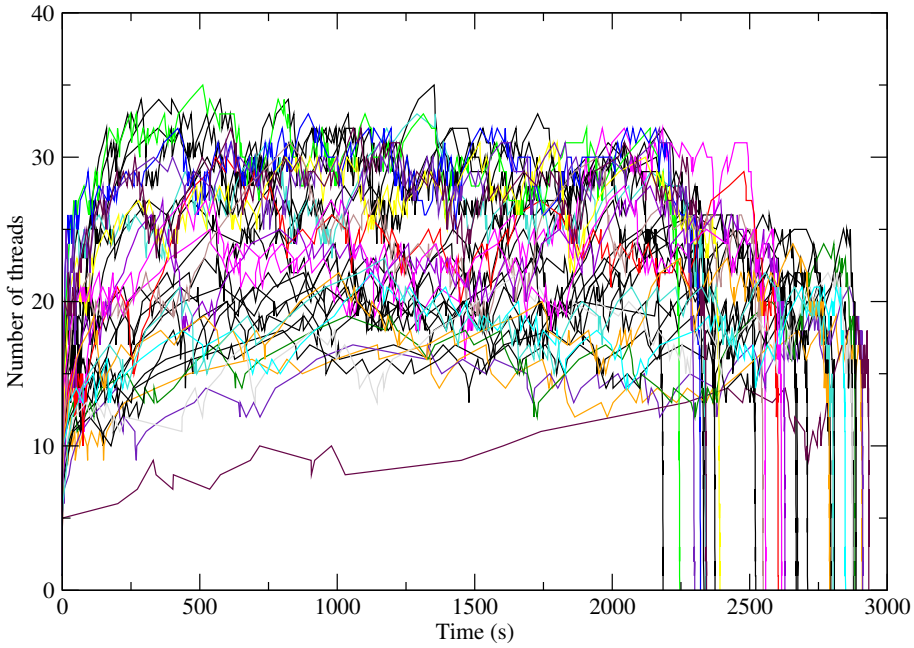


Figura 4.5: Evolución temporal del número de hebras para la ejecución adaptativa de 32 instancias de la misma aplicación con el gestor *ACW*.

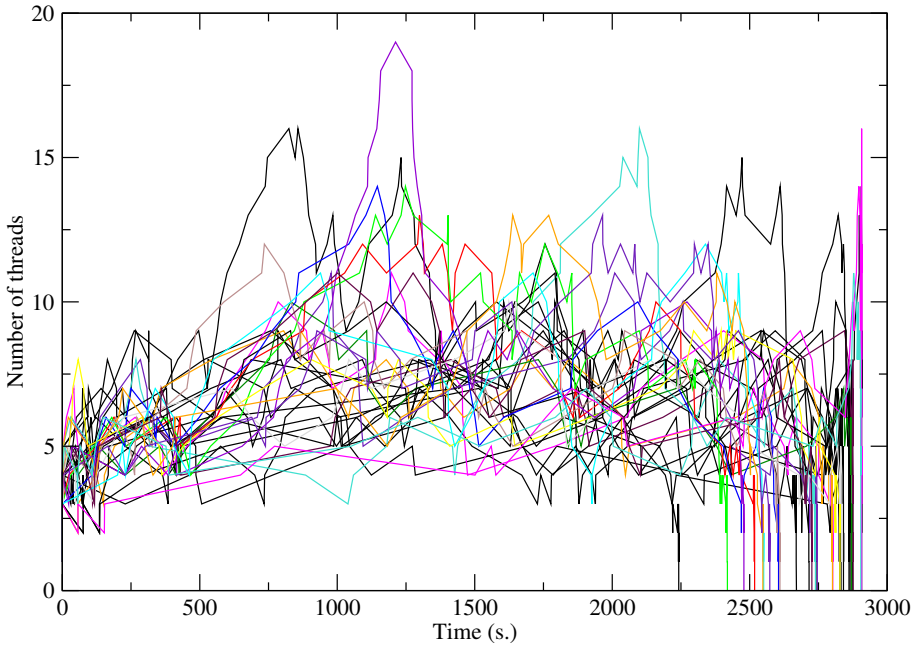


Figura 4.6: Evolución temporal del número de hebras para la ejecución adaptativa de 32 instancias de la misma aplicación con el gestor *KST*.

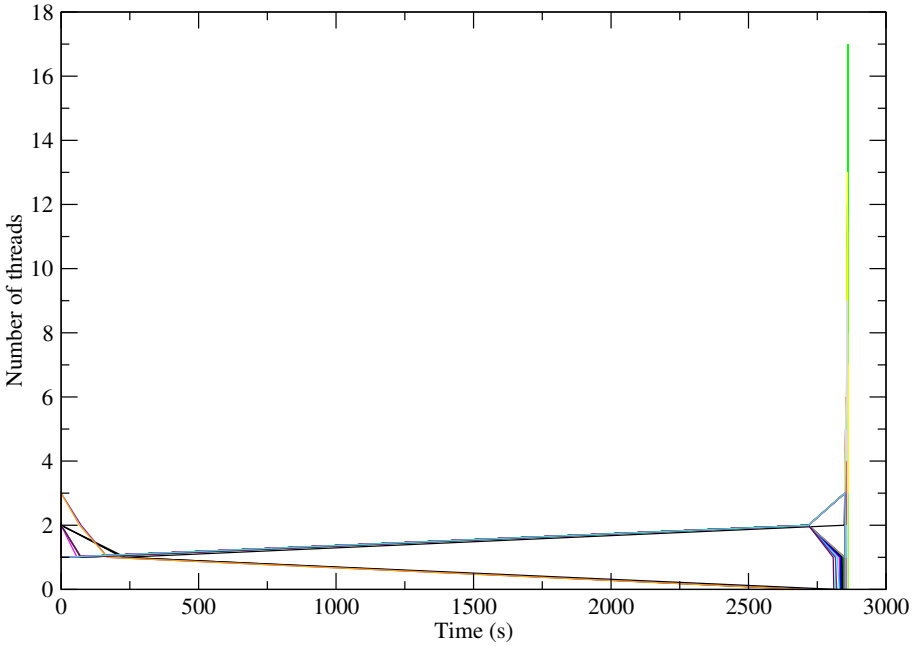


Figura 4.7: Evolución temporal del número de hebras para la ejecución adaptativa de 32 instancias de la misma aplicación con el gestor *KITST*.

ciones aumenta, el número de estados ociosos de algún procesador disminuye, reduciendo así la posibilidad de generar nuevas hebras. Además, los gestores *ACW* y *KST* generan intervalos de terminación demasiado grandes, lo que no sucede en el gestor *KITST*.

Por otro lado, hay que resaltar que con ambos gestores *KST* y *KITST*, el número de hebras de las últimas aplicaciones en ejecución tienden a incrementarse hacia p, lo que se puede apreciar mejor justo al final de su ejecución en las Figuras 4.6 y 4.7. La Figura 4.8 muestra el zoom del intervalo de terminación correspondiente a la Figura 4.7, donde se observa como a partir de 2850 s. el número de hebras de las aplicaciones va creciendo conforme van terminando otras aplicaciones, liberando así los recursos. Este hecho confirma que estos GP poseen un alto grado de adaptación a la carga computacional del sistema. Los resultados mostrados en la Figura 4.7 indican que el gestor *KITST* mejora a los otros GP en terminos de tiempo de ejecución y capacidad para adaptarse a la carga del sistema, gestionando de manera eficiente el número de hebras para cada aplicación en ejecución.

4.5. Conclusiones y trabajo futuro

En este capítulo se han planteado diferentes estrategias para determinar el nivel de paralelismo de varias aplicaciones en el sistema no dedicado, destacando la competición por los recursos como la estrategia más adecuada para evaluar el nivel de adaptabilidad que se consigue con las decisiones tomadas por los distintos GPs. En nuestro trabajo se ha descartado la transformación del sistema no dedicado en varios sistemas dedicados, debido

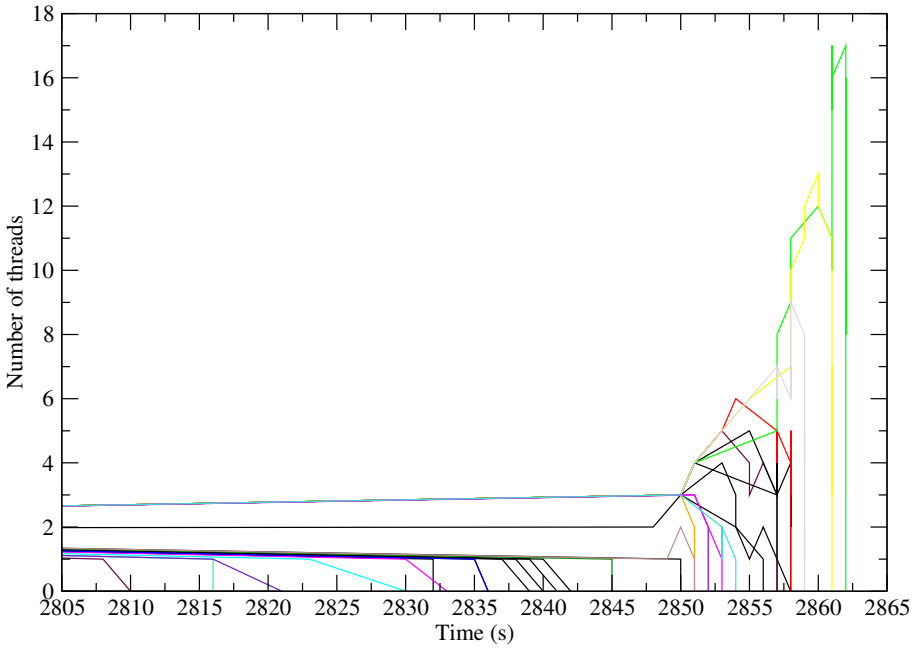


Figura 4.8: Intervalo de terminación para la ejecución adaptativa de 32 instancias de la misma aplicación con el gestor *KITST*.

a que esta alternativa no permitiría evaluar correctamente la adaptabilidad de los GPs en sistemas no dedicados.

También se han realizado experimentos con distintos modelos de GP en un sistema no dedicado basado en la ejecución simultánea de varias instancias de la misma aplicación que compiten por los recursos, destacando el gestor *KITST* como el que consigue una mejor adaptabilidad de las aplicaciones especialmente cuando el sistema no dedicado está sobrecargado. Por otro lado, se puede concluir que el gestor a nivel de usuario *ACW* mejora el tiempo total de ejecución de las n aplicaciones, frente a su correspondiente ejecución no adaptativa, en sistemas no dedicados [61] y [60].

Un posible trabajo futuro consiste en la implementación en los gestores a nivel de kernel (*KST* y *KITST*) de la estrategia de planificación basada en la eficiencia energética, donde el administrador del sistema pueda establecer los límites en el consumo energético, sin reducir considerablemente el tiempo real de ejecución de las aplicaciones multihebradas, tanto en sistemas dedicados como no dedicados.

Conclusiones y principales aportaciones

En esta tesis se ha realizado un estudio sobre la adaptabilidad de las aplicaciones multihebradas en entornos reales de ejecución sobre el SO Linux, tanto en sistemas dedicados como en sistemas no dedicados, centrándose en una aplicación B&B que presenta diferentes irregularidades según el tipo de problema a resolver.

El mejor rendimiento de las aplicaciones se consigue a partir de una buena adaptabilidad para lo que se necesita establecer el número óptimo de hebras que no suele ser siempre el mismo, pues varía en función de las características intrínsecas de la aplicación, la carga de trabajo a resolver y resto de aplicaciones en ejecución. Además, la estimación del número óptimo de hebras debe realizarse en cada instante de tiempo, especialmente en sistemas no dedicados, donde nuevas aplicaciones pueden entrar en ejecución en cualquier momento. En este sentido, se han diseñado varias tipologías de Gestores de nivel de paralelismo (GPs) cuya finalidad principal es ayudar a las aplicaciones multihebradas informándoles periódicamente del número óptimo de hebras, para lo cual debe de monitorizar a todas las hebras activas de la aplicación, según el criterio de decisión configurado.

El GP informa a la aplicación monitorizada la creación o detención de hebras, para que la aplicación actúe en consecuencia a nivel de usuario. Sin embargo, el GP también tiene capacidad para detener y activar hebras a nivel de kernel si así se ha configurado.

En este trabajo de tesis se han propuesto varias tipologías de GPs, unas a nivel de usuario y otras a nivel de kernel. Los primeros GPs diseñados fueron los GPs a nivel de usuario (ACW y AST) que consiguen una adaptabilidad aceptable de las aplicaciones a partir de una sencilla implementación. Sin embargo, estos GPs están limitados en cuanto a la selección del criterio de decisión, pues tan solo pueden configurar criterios de decisión que requieran información disponible a nivel de usuario, tales como, el número de procesadores ociosos y el rendimiento parcial de la aplicación. Para conseguir un mayor nivel de adaptabilidad es necesario utilizar otros criterios de decisión que analicen información accesible exclusivamente a nivel de kernel. Esta peculiaridad forzó al diseño de GPs a nivel de kernel (KST, SST y $KITST$). Las distintas versiones del GP a nivel de kernel difieren en función del nivel de integración del GP en el SO. Por este motivo, la integración de cualquier GP a nivel de kernel ofrece al SO Linux un valor añadido al permitir informar a las aplicaciones multihebradas información útil para mejorar su nivel de adaptabilidad. En este sentido, cualquiera de los GPs a nivel de kernel propuestos pueden utilizar cualquier criterio de decisión, incluidos los utilizados por los GPs a nivel de usuario, tales como:

1. Número de procesadores ociosos.

2. Rendimiento parcial de la aplicación.

3. Minimizar los tiempos de bloqueo de las hebras (*MST*, *MNET*, *MIBT*, *MNIBT* y *MWT*).

4. Estimar el número de hebras a partir de los tiempo de bloqueo de las hebras (*MNT_BT*, *MNT_IBT* y *MNT_NIBT*).

Desde el punto de vista del usuario, cuya finalidad es obtener el mejor rendimiento de la aplicación multihebrada, se aconseja un criterio de decisión basado en la estimación del número de hebras. Aunque la elección de un criterio concreto depende de las características intrinsecas de la aplicación, tales como secciones críticas y acceso a memoria, el criterio de decisión *MNT_IBT* suele generar los mejores resultados en aplicaciones sin fallos de memoria.

Desde el punto de vista de un administrador del sistema, cuya finalidad es obtener la mejor eficiencia del sistema, se aconseja un criterio de decisión que minimice el tiempo de bloqueo. El criterio de decisión *MST* es el más restrictivo de todos los planteados, pues minimiza el tiempo de bloqueo de cualquier hebra independientemente de la causa que lo origine.

4.6. GPs automatizados

En el Capítulo 2 se establecieron las características que debe tener un buen gestor del nivel de paralelismo y se detallaron las etapas o fases de funcionamiento que componen el GP. Las distintas versiones de GP implementadas dependen básicamente de donde se ejecutan las principales fases. Los programadores de aplicaciones multihebradas pueden utilizar cualquiera de los GPs implementados haciendo uso de la librería *IGP* descrita en ese capítulo, independientemente de que la estrategia de creación de hebras sea estática o dinámica.

Uno de los parámetros de configuración en el GP es establecer el intervalo de tiempo (λ) entre dos análisis consecutivos del criterio de selección. El programador es el responsable de establecer λ en función del número de iteraciones del bucle computacional. En la aplicación B&B utilizada se ha establecido que cada una de las hebras informe al GP en cada iteración, de tal forma, que el intervalo λ depende del tiempo que tardan todas las hebras activas en ejecutar una iteración. El tiempo computacional que una hebra tarda en ejecutar una iteración depende directamente de las características intrínsecas de la aplicación multihebrada, entre las que distinguimos la complejidad en los cálculos computacionales y el tipo de carga de trabajo. En este sentido, existen otras aplicaciones con tiempos por iteración muy dispares, es decir, iteraciones que se computan muy rápidamente, frente a otras iteraciones cuyo tiempo de computo es muy superior. Esta heterogeneidad en el tiempo computacional por iteración puede influir en el valor de λ, pues el valor de λ estará limitado por el tiempo de computo de la iteración más lenta. Por lo tanto, mientras una

hebra ejecuta la iteración más lenta, el resto de hebras pueden ejecutar varias iteraciones en el mismo intervalo de tiempo.

Como trabajo futuro, se propone conocer el número de iteraciones adecuado para que cada hebra informe al GP, puesto que aqui se ha ensayado con una iteración por hebra. En realidad el número de iteraciones que cada hebra informa debería también ser adaptativo en función de la complejidad de la carga de trabajo a resolver en cada iteración. Seria interesante diseñar un mecanismo que automatice este hecho en la fase de *Información* del GP, y eximir al programador de tener que implementar este aspecto en su código.

Otro aspecto mejorable, sería la posibilidad de que el GP adquiera la funcionalidad de un *profiler* [1], de tal forma que proporcione información estadística a la aplicación, o directamente al usuario, una vez finalizada la ejecución de la aplicación. Esta información podría incluir datos relativos al tiempo real/core, tiempo de usuario/core, tiempo de sistema/core de la aplicación multihebrada, tiempo de bloqueo interrumpible y no interrumpible, etc. Esta información puede ser muy interesante y útil al programador para detectar errores en su implementación, pues se evitaría el coste y la perturbación del *profiler* sobre la aplicación.

4.7. Criterios de decisión especializados

El criterio de decisión seleccionado es uno de los aspectos claves en cualquier GP para estimar correctamente el número de hebras activas en cada momento. En el Capítulo 3 se han propuesto diferentes criterios de decisión, de tal forma, que los distintos experimentos realizados sobre *Local-PAMIGO* ratifican los resultados obtenidos con *Ray Tracing* en el Capítulo 2, donde los GPs a nivel de Kernel mejoran sustancialmente las prestaciones de la aplicación multihebrada, frente a los GPs a nivel de usuario. La amplia variedad de criterios de decisión ensayados van encaminados a diseñar el criterio de decisión más eficiente, pero además que permita minimizar el tiempo computacional del GP, especialmente en las fases de *Información* y *Evaluación*.

En este sentido, se deja una puerta abierta para proponer en un futuro nuevos criterios de decisión que estimen el número óptimo de hebras a partir de nuevas ecuaciones que operen con los tiempos de bloqueo de las hebras, o con otro tipo de información. Además, se podría plantear la posibilidad de diseñar criterios de decisión especializados para cada tipo de aplicación, de tal forma, que el GP aplicara un criterio de decisión para cada tipo de aplicación. Estos nuevos criterios de decisión no pueden aumentar considerablemente el tiempo computacional del GP, pero podrían establecer un mecanismo de detención adaptativa, que analice tanto las características de la aplicación multihebrada como el criterio de decisión seleccionado, y decida automáticamente si debe activar o desactivar la detención de hebras.

La detención de hebras es otro parámetro que debe configurar el programador de la aplicación, por lo tanto, un paso más avanzado en la automatización sería analizar cuando el mecanismo de detención de hebras beneficia la adaptabilidad de la aplicación y cuando

[1] Herramienta que se ejecuta en paralelo a la aplicación multihebrada, permitiendo recopilar información y detectar puntos problemáticos donde se produce una perdida de rendimiento.

no. La detención de hebras implementada en este trabajo de tesis, permite detener una hebra cada vez que lo indique el criterio de decisión. Por ejemplo, los criterios de estimación del número de hebras, tales como *MNT_BT*, *MNT_IBT*, *MNT_NIBT* y otros, informan al GP del número de hebras óptimo según la ecuación aplicada sobre los tiempos de bloqueo de las hebras. Estos criterios permiten al GP generar una nueva hebra si el número de hebras estimado es igual o inferior al real. En caso contrario, si el usuario ha habilitado la detención de hebras, el GP detendrá una hebra cada vez que no se verifique el criterio de decisión. Sin embargo, la detención adaptativa podría detener un número mayor de hebras cada vez que no se verifique el criterio de decisión, por ejemplo, tantas hebras como indique la diferencia entre el número de hebras activas y el número de hebras estimado.

4.8. Analizar otras estrategias de planificación

En el Capitulo 4 se realizaron experimentos donde la ejecución concurrente de varias instancias de la misma aplicación competían por los recursos en un sistema no dedicado. Sin embargo, se pueden ampliar estos estudios en trabajos futuros donde se analicen los comportamientos de los GPs con distintas aplicaciones ejecutándose en un sistema no dedicado, donde la estrategia de planificación sea la competición por los recursos.

Otro aspecto interesante pasa por evaluar la estrategia de planificación basada en la eficiencia energética, donde el administrador del sistema pueda reducir el consumo energético, sin reducir considerablemente el tiempo total de ejecución, tanto en sistemas dedicados como no dedicados.

Bibliografía

[1] S. Ali, H.J. Siegel, M. Maheswaran, S. Ali, and D. Hensgen. Task execution time modeling for heterogeneous computing systems. In *Proceedings of the 9th Heterogeneous Computing Workshop*, HCW '00, pages 185–, Washington, DC, USA, 2000. IEEE Computer Society.

[2] J.A. Álvarez, J. Roca, and J.J. Fernández. Multithreaded tomographic reconstruction. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 81–88. Springer, 2007.

[3] C. Anglano. Predicting parallel applications performance on non-dedicated cluster platforms. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 172–179, New York, NY, USA, 1998. ACM.

[4] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52:56–67, October 2009.

[5] D.A. Bader, W.E. Hart, and C.A. Phillips. Parallel algorithm design for branch and bound. In H.J. Greenberg, editor, *Tutorials on Emerging Methodologies and Applications in Operations Research*, chapter 5, pages 1–44. Kluwer Academic Press, 2004.

[6] J.L. Berenguel, L.G. Casado, I. García, and E.M.T. Hendrix. On estimating workload in interval branch-and-bound global optimization algorithms. In *Journal of Global Optimization*, volume 56, pages 821–844. Springer, 2013.

[7] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[8] S. Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[9] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[10] F.H. Branin. Widely convergent method for finding multiple solutions of simultaneous nonlinear equations. *IBM J. Res. Dev.*, 16(5):504–522, September 1972.

[11] J.L.M. Cabrelles. *A Dynamic Power-Aware Partitioner with Real-Time Task Migration for Embedded Multicore Processors.* PhD thesis, Universidad Politécnica de Valencia, https://riunet.upv.es/, 2012.

[12] O. Caprani, B. Godthaab, and K. Madsen. Use of a real-valued local minimum in parallel interval global optimization. *Interval Computations*, 2:71–82, 1993.

[13] L.G. Casado, J.A. Martínez, I. García, and E.M.T. Hendrix. Branch-and-bound interval global optimization on shared memory multiprocessors. *Optimization Methods and Software*, 23(3):689–701, 2008.

[14] J. Corbalan, X. Martorell, and J. Labarta. Performance-driven processor allocation. *IEEE Trans. Parallel Distrib. Syst.*, 16(7):599–611, 2005.

[15] R. Corrêa and A. Ferreira. Parallel best-first branch and bound in discrete optimization: a framework. In R. Corrêa and P. M. Pardalos, editors, *IRREGULAR '95, Solving Combinatorial Optimization Problems in Parallel Methods and Techniques*, pages 145–170. Springer, 1996.

[16] T. Crainic, B. Le Cun, and C. Roucairol. Parallel branch and bound algorithms. In El-Ghazali Talbi, editor, *Parallel Combinatorial Optimization*, chapter 1, pages 1–28. Willey, 2006.

[17] A. de Bruin, G. Kindervater, and H. Trienekens. Asynchronous parallel branch and bound and anomalies. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *Lecture Notes in Computer Science*, pages 363–377. Springer Berlin / Heidelberg, 1995.

[18] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[19] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, volume 36 of *SC '08*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[20] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *38th International Conference on Parallel Processing (ICPP 09)*, pages 124–131, Vienna, Austria, September 2009. IEEE Computer Society.

[21] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[22] Y. Ge, P. Malani, and Q. Qiu. Distributed task migration for thermal management in many-core systems. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 579–584, New York, NY, USA, 2010. ACM.

[23] B. Gendron and T.G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[24] J.L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[25] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 99:1–99:10, New York, NY, USA, 2013. ACM.

[26] T. Henriksen and K. Madsen. Parallel algorithms for global optimization. *Interval Computations*, 3(5):88–95, 1992.

[27] T. Henriksen and K. Madsen. Use of a depth-first strategy in parallel global optimization. Technical Report 92-10, Institute for Numerical Analysis, Technical University of Denmark, 1992.

[28] J.F.R. Herrera, L.G. Casado, E.M.T. Hendrix, and I. García. Pareto optimality and robustness in bi-blending problems. In *TOP*, pages 254–273. Springer, 2014.

[29] P.K. Immich, R.S. Bhagavatula, and R. Pendse. Performance analysis of five interprocess communication mechanisms across UNIX operating systems. *J. Syst. Softw.*, 68(1):27–43, oct 2003.

[30] M. Isard and A. Birrell. Automatic mutual exclusion. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 3:1–3:6, Berkeley, CA, USA, 2007. USENIX Association.

[31] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Elsevier Waltham (Mass.), Amsterdam, Boston (Mass.), Heidelberg..., et al., 2013.

[32] K. Knizhnik. Fast memory allocation library for multithreaded applications. http://www.garret.ru/threadalloc/readme.html, 2006.

[33] J.S. Kowalik and E.R. Kamgnia. An exponential function as a model for a conjugate gradient optimization method. *Journal of Mathematical Analysis and Applications*, 67:476–482, 1979.

[34] J. Lee, J.H. Park, H. Kim, C. Jung, D. Lim, and S.Y. Han. Adaptive execution techniques of parallel programs for multiprocessors. *J. Parallel Distrib. Comput.*, 70(5):467–480, 2010.

[35] J. Lee, J.H. Park, H. Kim, C. Jung, D. Lim, and S.Y. Han. Adaptive execution techniques of parallel programs for multiprocessors. *Journal of Parallel and Distributed Computing*, 70(5):467–480, 2010.

[36] Qixiao Liu, Miquel Moreto, Victor Jimenez, Jaume Abella, Francisco J. Cazorla, and Mateo Valero. Hardware support for accurate per-task energy metering in multicore systems. *ACM Trans. Archit. Code Optim.*, 10(4):34:1–34:27, December 2013.

[37] X. Liu, J. Mellor-Crummey, and M. Fagan. A New Approach for Performance Analysis of OpenMP Programs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 69–80, New York, NY, USA, 2013. ACM.

[38] C. Luk. Intel software autotuning tool. https://software.intel.com/en-us/articles/intel-software-autotuning-tool, 2010.

[39] J.A. Martínez, L.G. Casado, J.A. Alvarez, and I. García. Interval parallel global optimization with Charm++. In *Applied Parallel Computing: State of the Art in Scientific Computing Series*, volume 3732 of *Lecture Notes in Computer Science*, pages 161–168. Springer, 2006.

[40] J.A. Martínez, L.G. Casado, I. García, and B. Tóth. AMIGO: Advanced multidimensional interval analysis global optimization algorithm. In C.A. Floudas and P.M. Pardalos, editors, *Frontiers in Global Optimization*, volume 74 of *Nonconvex Optimization and Applications*, pages 313–326. Kluwer Academic PUBLISHERs, 2004. http://kapis.www.wkap.nl/prod/b/1-4020-7699-1.

[41] M.P. Matijkiw and M.M.K. Martin. Exploring coordination of threads in multicore libraries. http://www.seas.upenn.edu/~cse400/CSE400_2009_2010/final_report/Matijkiw.pdf, 2010. Dept. of CIS - Senior Design 2009-2010.

[42] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *First ACM SIGOPS EuroSys Conference*, pages 18–21, Leuven, Belgium, apr 2006.

[43] R. Miceli, G. Civario, A. Sikora, and et al. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. *Applied Parallel and Scientific Computing*, 7782:328–342, 2013.

[44] B. Nichols, D. Buttlar, and J. Proulx Farrell. *PThreads Programming: A POSIX Standard for Better Multiprocessing.* O'Reilly, 1998.

[45] S.L. Olivier and J.F. Prins. Evaluating OpenMP 3.0 run time systems on unbalanced task graphs. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 63–78, Berlin, Heidelberg, 2009. Springer-Verlag.

[46] S.L. Olivier and J.F. Prins. Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*, 38:341–360, 2010.

[47] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):2–11, September 1996.

[48] OpenMP Architecture Review Board. *OpenMP Application Program Interface, version 3.0.* OpenMP, 2008.

[49] P.M. Pardalos, G. Xue, and P.D. Panagiotopoulos. Parallel algorithms for global optimization problems. In *Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques*, pages 232–247. Springer-Verlag, 1996.

[50] D.A. Patterson. Software knows best: portable parallelism requires standardized measurements of transparent hardware. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 1–2, New York, NY, USA, 2010. ACM.

[51] D.A. Penry. Multicore diversity: a software developer's nightmare. *SIGOPS Oper. Syst. Rev.*, 43(2):100–101, 2009.

[52] K.K. Pusukuri, R. Gupta, and L.N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *Proceedings of the 2011 International Symposium on Workload Characterization*, pages 116–125, Austin, TX, USA, October 2011.

[53] R. Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers.* Apress, Berkely, CA, USA, 1st edition, 2013.

[54] J. Reinders. *Intel Threading Building Blocks.* O'Reilly, 2007.

[55] J.F. Sanjuan-Estrada. Procesamiento paralelo de ray tracing. In *XIII Jornadas de Paralelismo*, pages 153–158, Lleida, 2002.

[56] J.F. Sanjuan-Estrada, L.G. Casado, and I. García. Reliable algorithms for ray intersection in computer graphics based on interval arithmetic. In *Proceedings of XVI Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI 2003)*, pages 35–42, Sao Carlos, Brasil, June 2003. IEEE Computer Society.

[57] J.F. Sanjuan-Estrada, L.G. Casado, and I. García. Interval arithmetic algorithms for rendering isosurfaces. In *Proceedings of 17'th IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation*, pages 93–99, Paris, Francia, July 2005.

[58] J.F. Sanjuan-Estrada, L.G. Casado, and I. García. Reliable computation of roots to render real polynomials in complex space. In *Proceedings of the First International Conference on Computer Graphics Theory and Applications (GRAPP06)*, pages 25–28, Setubal, Portugal, February 2006.

[59] J.F. Sanjuan-Estrada, L.G. Casado, and I. García. Dynamic number of threads based on application performance and computational resources at run-time for interval branch and bound algorithms. In *Proceedings of CMMSE'10*, volume 3, pages 828–839, Almería, June 2010.

[60] J.F. Sanjuan-Estrada, L.G. Casado, and I. García. Adaptive parallel interval branch and bound algorithms based on their performance for multicore architectures. *The Journal of Supercomputing*, pages 376–384, 2011.

[61] J.F. Sanjuan-Estrada, L.G. Casado, and I. García. Adaptive parallel interval global optimization algorithms based on their performance for non-dedicated multicore architectures. In *Proceedings of PDP 2011 - The 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, pages 252–256, Cyprus, February 2011. IEEE.

[62] J.F. Sanjuan-Estrada, L.G. Casado, I. García, and E.M.T. Hendrix. Performance driven cooperation between kernel and auto-tuning multi-threaded interval B&B applications. In *Proceedings of ICCSA'12*, volume 7333, pages 57–70, Salvador de Bahia, June 2012. LNCS, Springer.

[63] J.F. Sanjuan-Estrada, L.G. Casado, J.A. Martínez, M. Soler, and I. García. Concurrencia-paralelismo auto-gestionada por procesos multihebrados en sistemas multicore. In *XX Jornadas de Paralelismo*, pages 7–11, A Coruña, 2009.

[64] J.F. Sanjuan-Estrada and I. García L.G. Casado. Rendering (complex) algebraic surfaces. In *Proceedings of the First International Conference on Computer Vision Theory and Applications (VISAPP06)*, pages 139–146, Setubal, Portugal, February 2006.

[65] C. Severance, R. Enbody, and P. Petersen. Managing the overall balance of operating system threads on a multiprocessor using automatic self-allocating threads (ASAT). *Journal of Parallel and Distributed Computing*, 37(1):106–112, aug 1996.

[66] D. Shelepov, J.C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z.F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, 2009.

[67] R. Strong, J. Mudigonda, M.C. Jeffrey., N. Binkert, and D. Tullsen. Fast switching of threads between cores. *SIGOPS Oper. Syst. Rev.*, 43(2):35–45, 2009.

[68] M.A. Suleman, M.K. Qureshi, and Y.N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. *SIGARCH Comput. Archit. News*, 36:277–286, March 2008.

[69] S. Valat, M. Pérache, and W. Jalby. Introducing kernel-level page reuse for high performance computing. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '13, pages 3:1–3:9, New York, NY, USA, 2013. ACM.

[70] E. Vicente, R. Matias, L. Borges, and A. Macêdo. Evaluation of compound system calls in the linux kernel. *SIGOPS Oper. Syst. Rev.*, 46(1):53–63, feb 2012.

[71] S. WANG, P.C. YEW, and A. ZHAI. Code transformations for enhancing the performance of speculatively parallel threads. *Journal of Circuits, Systems and Computers*, 21(02):1240008, 2012.

[72] S. Yamada, S. Kusakabe, and H. Taniguchi. Impact of wrapped system call mechanism on commodity processors. In Joaquim Filipe, Boris Shishkov, and Markus Helfert, editors, *ICSOFT (1)*, pages 308–315. INSTICC Press, 2006.

[73] C. Yu and P. Petrov. Adaptive multi-threading for dynamic workloads in embedded multiprocessors. In *Proceedings of the 23rd symposium on Integrated circuits and system design*, SBCCI '10, pages 67–72, New York, NY, USA, 2010. ACM.

[74] F. Zabatta and K. Ying. Dynamic thread creation: An asynchronous load balancing scheme for parallel searches. In *Proceedings of 10th International Conference on Parallel and Distributed Computing and Systems*, pages 20–24. IASTED ACTA Press, Las Vegas, NV, 1998.

Apéndice A

Índice de abreviaturas

ACW *Application decides based on Completed Work*: GP integrado en la aplicación basado en el trabajo completado.

AMP *Asynchronous Multiple Pool*: Programación con múltiples estructuras de datos de acceso asíncronos.

API *Application Programming Interface*: Conjunto de funciones y procedimientos utilizados generalmente en las librerías de programación.

ASP *Asynchronous Single Pool*: Programación con una única estructura de datos de acceso asíncrono.

AST *Application decides based on Sleeping Threads*: GP integrado en la aplicación basado el tiempo que duermen las hebras.

AvNRT *Average Number of Running Threads*: Número medio de hebras de una aplicación en ejecución.

B&B *Branch and Bound*: Metodología algorítmica para la resolución de problemas de optimización combinatoria basado en la ramificación y acotación.

BT *Blocked Time*: Tiempo que una hebra permanece en los estados *INTERRUPTIBLE* y *UNINTERRUPTIBLE*.

CFS *Completely Fair Scheduler*: Planificador del sistema operativo que realiza un reparto equitativo del uso del procesador entre los procesos.

CMP *Chip-Level Multiprocessing*: Arquitectura de multiprocesadores integrados en el mismo chip.

GP *manaGer of level of Parallelism*: Gestor del nivel de paralelismo.

HBr *Hypothetical Best result*: Mejor resultado hipotético.

HPC *High-Performance Computing*: Alto rendimiento computacional.

IBT *Interruptible Blocked Time*: Tiempo que un proceso o hebra permanece en estado *INTERRUPTIBLE*.

IGP *Interface of the manaGer of level of Parallelism*: Interfaz del Gestor del nivel de paralelismo.

IPC *InterProcess Communication*: Mecanismo de comunicación entre procesos.

KITST *Kernel Idle Thread decides based on Sleeping Threads*: GP integrado en la hebra ociosa del kernel basado el tiempo que duermen las hebras.

KITST-MST *Kernel Idle Thread decides based on Sleeping Threads using decision rule to Minimize Sleeping Time*: GP integrado en la hebra ociosa del kernel basado en minimizar el tiempo que duermen las hebras.

KST *Kernel decides based on Sleeping Threads*: GP integrado en el kernel basado el tiempo que duermen las hebras.

KST-MST *Kernel decides based on Sleeping Threads using decision rule to Minimize Sleeping Time*: GP integrado en el kernel basado en minimizar el tiempo que duermen las hebras.

LER *Light, Efficient and fast Response*: Características que todo GP debe ser: Ligero, Eficaz y de respuesta Rápida.

MIBT *Minimize Interruptible Blocked Time*: Criterio de decisión que minimiza el tiempo que las hebras permanecen en el estado *INTERRUPTIBLE*.

MIC *Many Integrated Core architecture*: Arquitectura con muchos procesadores integrados en el mismo chip.

MNET *Minimize Non-Executable Time*: Criterio de decisión que minimiza el tiempo que las hebras permanecen en los estados *INTERRUPTIBLE* y *UNINTERRUPTIBLE* más el tiempo de espera (WT) en la cola de ejecución.

MNIBT *Minimize Non-Interruptible Blocked Time*: Criterio de decisión que minimiza el tiempo que las hebras permanecen en el estado *UNINTERRUPTIBLE*.

MNT *Maximum Number of Threads*: Máximo número de hebras que una aplicación multihebrada puede tener en ejecución.

MNT_BT *Maximum Number of Threads based on Blocked Time*: Criterio de decisión que calcula el MNT a partir del tiempo de bloqueo de las hebras.

MNT_IBT *Maximum Number of Threads based on Interruptible Blocked Time*: Criterio de decisión que calcula el MNT a partir del tiempo de bloqueo interrumpible de las hebras.

MNT_MNET *Maximum Number of Threads based on Minimize Non-Executable Time*: Criterio de decisión que calcula el MNT minimizando el tiempo que las hebras permanecen en estados no ejecutable.

MNT_NIBT *Maximum Number of Threads based on Non-Interruptible Blocked Time*: Criterio de decisión que calcula el MNT a partir del tiempo de bloqueo no interrumpible.

MNT_WT *Maximum Number of Threads based on Waiting Time*: Máximo número de hebras basado en el tiempo de espera.

MST *Minimize Sleeping Time*: Criterio de decisión que minimiza el tiempo que las hebras permanecen en los estados *INTERRUPTIBLE* y *UNINTERRUPTIBLE*.

MWT *Minimize Waiting Time*: Criterio de decisión que minimiza el tiempo que espera de las hebras en la cola de ejecución.

NIBT *UnInterruptible Blocked Time*: Tiempo que un hebra permanece en estado *UNINTERRUPTIBLE*.

SO *Operating System*: Sistema Operativo.

PAMIGO *Parallel Advanced Multidimensional Interval analysis Global Optimization*: Algoritmo paralelo de análisis de intervalos multidimensional avanzado para optimización global.

PARSEC *Princeton Application Repository for Shared-Memory Computers*: Repositorio de aplicaciones para computadores con memoria compartida de la Universidad de Princeton.

PU *Processing Unit*: Unidad de procesamiento.

SHCB *Six-Hump Camel-Back*: Función bidimensional con seis mínimos locales, dos de los cuales son mínimos globales.

SBr *Static Best result*: Mejor resultado estático.

SPMD *Single Program, Multiple Data*: El mismo código del programa que se ejecuta sobre múltiples datos.

SST *Scheduler decides based on Sleeping Threads*: GP integrado en manejador del reloj del sistema basado el tiempo que duermen las hebras.

TFLOP *Tera FLoat Operation Per second*: 10^12 operaciones en punto flotante por segundo.

TBB *Threading Building Blocks*: Librería de C++, desarrollada por Intel, para la programación de aplicaciones multihebradas.

vDSO *virtual Dynamically linked Shared Objects*: Igual que las vSYSCALL, salvo que tiene una ubicación dinámica en la memoria asignada al proceso.

VPU *Virtual Processing Unit*: Unidad de procesamiento virtual.

vSYSCALL *virtual SYStem CALL*: Llamada al sistema que evita cruzar la frontera entre el espacio de usuario y el kernel.

WT *Waiting Time*: Tiempo que una hebra espera en la cola antes de ejecutarse en el procesador.