**Universitat Autònoma de Barcelona**

Departament d'Arquitectura de Computadors i Sistemes Operatius

Màster en Computació d'altes prestacions

# FACTORES DE RENDIMIENTO EN ENTORNOS MULTICORE.

Memòria del trabajo de "Iniciación a la Investigación. Trabajo de Fin de Máster" del "Máster en Computacio d'altes prestacions", realizada por *César Allande Álvarez*, bajo la dirección de *Eduardo César Galobardes* Presentada en la Escuela de Ingeniería (Departamento de Arquitectura de Computadores y Sistemas Operativos)

5 de Julio de 2010

# Resumen

Este documento refleja el estudio de investigación para la detección de factores que afectan al rendimiento en entornos multicore. Debido a la gran diversidad de arquitecturas multicore se ha definido un marco de trabajo, que consiste en la adopción de una arquitectura específica, un modelo de programación basado en paralelismo de datos, y aplicaciones del tipo Single Program Multiple Data. Una vez definido el marco de trabajo, se han evaluado los factores de rendimiento con especial atención al modelo de programación. Por este motivo, se ha analizado la librería de threads y la API OpenMP para detectar aquellas funciones sensibles de ser sintonizadas al permitir un comportamiento adaptativo de la aplicación al entorno, y que dependiendo de su adecuada utilización han de mejorar el rendimiento de la aplicación.

# Resum

Aquest document reflexa l'estudi d'investigación per a la detecció de factores que afecten al rendiment en entorns multicore. Deguat a la gran quantitat d'arquitectures multicore s'ha definit un marc de treball acotat, que consisteix en la adopció d'una arquitectura específica, un model de programación basat en paral·lelisme de dates, i aplicacions del tipus Single Program Multiple Data. Una vegada definit el marc de treball, s'ha evaluat els factors de rendiment amb especial atenció al model de programació. Per aquest motiu, s'ha analitzat la llibreria de thread i la API OpenMP per a detectar aquelles funcions sensibles de ser sintonitzades, al permetre un comportament adaptatiu de l'aplicació a l'entorn, i que, depenent de la seva adequada utilització s'aconsegueix una millora en el rendiment de la aplicació.

# Abstract

This work reflects research studies for the detection of factors that affect performance in multicore environments. Due to the wide variety of multicore architectures we have defined a framework, consisting of a specific architecture, a programming model based on data parallelism, and Single Program Multiple Data applications. Having defined the framework, we evaluate the performance factors with special attention to programming model. For this reason, we have analyzed threaad libreary and OpenMP API to detect thos candidates functions to be tuned, allowin applications to beave adaptively to the computing environment, and based on their propper use will improve performance.

# Índice general

# Índice de figuras

# Índice de tablas

# Capítulo 1

# Introducción

Este documento está centrado en la detección de parámetros que afectan al rendimiento en entornos multicore dentro del ámbito de HPC. Existe gran variedad de sistemas multicore o CMP (Chip Multiprocessors) con diferencias en su arquitectura y modelos de programación que explotan sus capacidades. La implantación de sistemas multicore en HPC actualmente está ampliamente extendida, y estos nuevos recursos deben ser eficientemente utilizadados. Para conseguir este objetivo se ha definido un marco de trabajo acotado, que ha de permitir sentar unas bases de conocimiento de estos entornos. El marco de trabajo ha definido una arquitectura real, un modelo de programación basado en el paralelismo de datos y aplicaciones del tipo SPMD (Single Program Multiple Data). Se ha elegido el modelo de programación OpenMP en su versión 2.5, debido a su extensa utilización, su sencillez para expresar paralelismo y por ser un modelo que permite expresar el paralelismo de datos.

Una vez definido el marco se ha realizado un análisis de la implementación OpenMP de GNU. Esta implementación consiste en tres módulos, que son las directivas de precompilación, la librería en tiempo de ejecución (libgomp) y la API de bajo nivel de POSIX threads (pthread de GNU). Como resultado del análisis se han detectado varios factores OpenMP parametrizables que afectan al rendimiento, como son la gestión de la planificación en el reparto de cargas, el acceso concurrente a memoria y la influencia de localidad de los datos sobre una estructura de memoria jerárquica. Para evaluar estos factores se han seleccionados aplicaciones, adecuadas para cada uno de los factores, que reflejan la influencia en la ejecución de cada uno de ellos.

## 1.1. Contexto

El mundo de los microprocesadores ha sufrido una nueva revolución por causa de los procesadores multicore. Se ha diseñado un nuevo tipo de procesadores basados en la integración de dos o más unidades de proceso dentro de un mismo chip, el cual basa su aumento de prestaciones en la utilización de estrategias de paralelismo.

En el modelo anterior, los procesadores de un solo núcleo conseguían la mejora de prestaciones mediante el aumento de la frecuencia de reloj y aportes funcionales basados en estrategias como pipeline, cachés multinivel, branch prediction, unidades de cómputo superescalares, planificación estática (en orden), planificación dinámica (fuera de orden), etc.

Si una aplicación obtiene un tiempo de ejecución determinado en un entornos monocore, al aumentar la frecuencia de reloj de este procesador, por norma general el tiempo de ejecución se reduce de forma transparente al programador. Esto supone una gran ventaja ya que la aplicación es considerada una caja negra. Sin embargo, por razones técnicas ya no es posible seguir aumentando la frecuencia de reloj. La principal razón es el consumo, que se incrementa más que linealmente al aumentar la frecuencia y genera además problemas de disipación de calor; pero también existen límites físicos en los transistores MOS para frecuencias elevadas, que tienen un umbral máximo en la velocidad de conmutación.

Por otra parte la capacidad de integración de transistores dentro del chip ha seguido aumentando, y estos nuevos recursos se han utilizado para crear copias de procesadores completos dentro de un mismo chip. Antes de llegar al modelo multicore han surgido algunos modelos que intentaban responder a la pregunta de ¿Cómo utilizar la alta capacidad de integración de transistores? Algunas de ellas:

- Incrementar el tamaño de la cache de segundo nivel, para mejorar los aciertos de cache. Sin embargo esta aproximación esta limitada por la cantidad de fallos de cache. Si la mitad de los ciclos son utilizados en esperas debidas a fallos de cache de segundo nivel, doblar la capacidad de esta memoria no puede conseguir más que una disminución del tiempo de ejecución a la mitad. Por otra parte, aumentar la cache de nivel 1 es inviable debido a que fuerza una reducción de la frecuencia de reloj.

- Explotar el paralelismo a nivel de instrucción, mediante la adición de las unidades funcionales. Sin embargo la limitación de este modelo reside en la cantidad limitada de instrucciones sensibles de ser paralelizables.

- Diseños de core capaces de ejecutar múltiples instrucciones de diferentes hilos de ejecución, mientras por otra parte se explota el ILP de cada thread,

- Explotación de recursos para la integración de múltiples procesadores dentro de un mismo chip (multicore).

A partir de esta última opción surgen las tecnologías multicore, y de esta forma los desarrolladores de hardware pretenden conseguir la meta de cumplir las expectativas de la Ley de Moore [1], basada en doblar las prestaciones cada 18 meses. No obstante, la percepción del usuario final es que no siempre se consigue un aumento en las prestaciones, ya que las aplicaciones no están respondiendo a este patrón de mejora.

Debido al aumento en el número de cores por chip, se dispone de más recursos que han de permitir aumentar las prestaciones, y para explotar estos recursos se deben utilizar estrategias de paralelismo y concurrencia. Sin embargo al utilizar estrategias de paralelismo, el rendimiento puede verse afectado por la fracción no paralelizable de la aplicación, tal como indica la ley de Amdahl [2]. El campo del paralelismo ha sido ampliamente estudiado en el ámbito de Computación de Altas Prestaciones (HPC-High Performance Computing), por lo que las estrategias utilizadas en HPC pueden trasladarse ahora a nivel de un procesador con múltiples cores. La primera consecuencia de este hecho es que las aplicaciones ya no pueden ser consideradas como una caja negra, en un entorno paralelo es necesario conocer la aplicación para poder sintonizarla y obtener el mejor rendimiento.

No obstante, aunque la ley de Amdahl especifica el límite de paralelización, normalmente será relevante sólo si la fracción serie es proporcional al tamaño del problema, esto raramente es así en aplicaciones HPC, afortunadamente como define la ley de Gustafson [3] la proporción de los cómputos secuenciales (no paralelos) normalmente decrece según el tamaño del problema se incrementa. Por tanto, sí podemos utilizar paralelismo en entornos HPC aprovechando los recursos multicore.

Actualmente los procesadores multicore están ampliamente implantados, desde equipos de sobremesa hasta en sistemas HPC (High Performance Computing). En la gráfica 1.1 se refleja el grado de implantación en los 500 supercomputadores más potentes de la actualidad para la lista del Top500 de Junio de 2010 [4], donde no únicamente los mejores supercomputadores disponen de sistemas multicore: Jaguar (Opteron 6 cores),[5], Nebulae (Intel Xeon 6 cores and Nvidia Tesla GPU)[6], Road Runner (PowerXCell 8i, 8+1 cores) [7], Kraken (Opteron 6 cores) [8], sino que el 96 % de estos computadores disponen de procesadores multicore. Por tanto únicamente 20 su-

percomputadores disponen de otras arquitecturas de procesador como single core o procesdores vectoriales, como por ejemplo el Earth Simulator de Japón (procesador vectorial NEC @ 3200 MHz) [9].

La utilización de estos procesadores plantea varias preguntas al respecto, ¿están siendo eficientemente utilizados?, ¿existe un aumento lineal en el rendimiento? o ¿cómo afecta al consumo del sistema?.

Las métricas utilizadas en entornos multicore tienen diferentes motivaciones: el consumo es un factor a tener en cuenta en sistemas de HPC cuando pretendemos optimizar la potencia consumida respecto al tiempo de ejecución, existe un campo centrado en este problema, el GreenComputing; La eficiencia nos indica si los recursos están siendo utilizados, es decir si para el coste de inversión de una máquina, sus recursos están siendo eficientemente utilizados; el rendimiento de una aplicación entendido como el tiempo de ejecución de la misma.

Normalmente al utilizar paralelismo se debe gestionar la concurrencia de los medios compartidos. es una necesidad el aprovechamiento de los recursos multicore, pero ¿Deben ser los programadores de aplicaciones, los diseñadores de sistemas operativos o los arquitectos hardware? Por eso se debe marcar una ruptura en el mundo de la computación y entrar plenamente en la era del reto multicore.

Hay que destacar que existe una gran brecha entre el estado de desarrollo hardware y software, en general el hardware está más avanzado que las herramientas software que los explotan. Sin embargo, el campo del paralelismo no empieza desde cero, se han heredado modelos y herramientas existentes en HPC. En particular, el modelo multicore actualmente se enmarca dentro del modelo de memoria compartida, debido a la arquitectura de los procesadores actuales, y correspondiendo a la taxonomía de Flynn [10] corresponde a un modelo de MIMD (Multiple-Instruction Multiple-Data), aunque en revisiones más recientes de la taxonomía se ha creado una extensión llamada SMP (Symmetric Multi-Processing), que corresponde a un modelo donde múltiples procesadores trabajan de forma conjunta. No obstante este modelo de procesadores multicore es dependiente de la tecnología actual, y no sabemos cómo van a evolucionar los procesadores, o si éstos van a continuar con este modelo.

## 1.2. Motivación

En la figura 1.1 se muestra el grado de implantación de procesadores multicore en los mejores sistemas de HPC. Se ha obtenido la información de la lista Top500 de Junio de 2010. Podemos

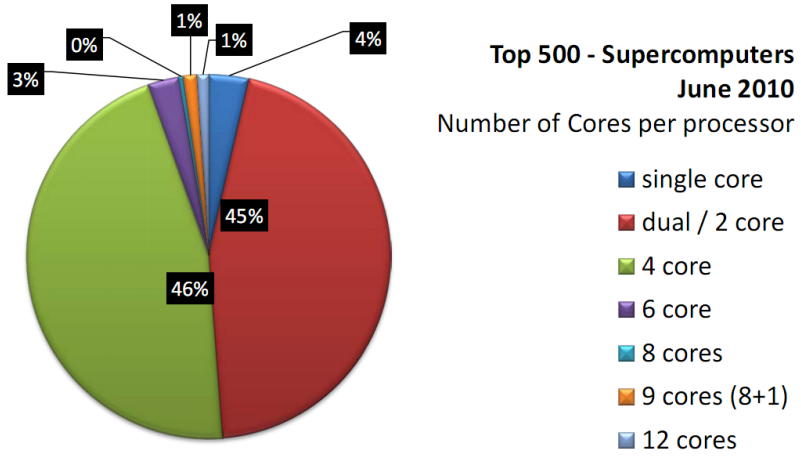


Figura 1.1: *Utilización de tecnologías Multicore en sistemas HPC. Lista Top500 Junio 2010*

observar como únicamente un 4 % de los Supercomputadores mantienen procesadores de mononúcleo y en cambio el resto utilizan procesadores de multicore.

Como los entornos de HPC integran procesadores multicore, es necesario que las aplicaciones HPC tengan conciencia de la arquitectura hardware del procesador. El tiempo de ejecución dependerá de la arquitectura hardware y de cuán eficientemente se utilicen los recursos. Además el comportamiento de la aplicación normalmente depende de los datos de entrada. Por este motivo, una aplicación debe adaptarse a la arquitectura y a los datos, es decir, ser sintonizada de acuerdo a sus características y a las del entorno en el que se ejecuta. Dicha sintonización puede realizarse antes de la ejecución, durante la ejecución (dinámicamente) o después de la ejecución, dependiendo de las características de la propia aplicación.

La definición de un modelo de rendimiento para entornos multicore supone una predicción del comportamiento de una aplicación. Para la creación de esta predicción resulta necesario caracterizar el comportamiento de la aplicación y la arquitectura hardware. Conociendo estos parámetros se puede inferir en puntos específicos de la aplicación, del hardware o en la capa que interconecta y gestiona estos dos elementos, el modelo de programación. Este proceso de sintonización necesita por tanto la definición de un modelo de rendimiento y debe aportar una metodología que permita adaptar la aplicación basada en la predicción del modelo de rendimiento.

## 1.3. Objetivos

El principal objetivo de esta tesina consiste en sentar las bases para la definición de un modelo de rendimiento en entornos multicore. Para la definición de un modelo de rendimiento se necesita

caracterizar la aplicación y la arquitectura. En el ámbito de esta tesina se pretenden caracterizar factores que afectan al rendimiento en entornos multicore, que han de permitir obtener indicadores de cómo se adapta la aplicación a un entorno multicore en un momento determinado de su ejecución.

Debido a la diversidad de elementos que intervienen en las aplicaciones multicore se ha definido un marco de trabajo acotado. De esta forma se pueden caracterizar tanto la aplicación como la arquitectura hardware. En este marco se definen los parámetros sintonizables de un modelo de programación que permita adaptar de la mejor forma posible una aplicación en un entorno multicore.

Una vez detectados los factores que afectan al rendimiento se evalúan en tiempo de ejecución para cada uno de los parámetros sintonizable del modelo de programación.

Situados en el contexto de los entornos multicore se describen a continuación los actuales objetivos de estudio desarrollados en este documento.

El objetivo de esta tesina es el estudio de los factores que afectan al rendimiento en entornos multicore. Descritos los factores se analizarán los parámetros que permitan la sintonización de aplicaciones paralelas. Los actores que intervienen en el rendimiento son la aplicación paralela, la arquitectura hardware y el paradigma de programación paralela.

- Como paradigma de programación paralela se ha elegido OpenMP por ser el estándar de facto en entornos de memoria compartida, por su capacidad de expresar el paralelismo de forma sencilla.

- Se han desarrollado dos toy benchmarks (Multiplicación de Matrices) y una benchmark de aplicación real basada en la segmentación de imágenes de resonancia magnética. Los tres benchmarks implementados corresponden a una aplicación con paradigma SPMD (Single Program Multiple Data).

- Las arquitectura disponible para la experimentación es una estación de trabajo con procesador Intel Core2Duo

En resumen, el contexto descrito indica la existencia de múltiples ámbitos en los que es necesario investigar para hacer frente a los retos planteados.

## 1.4. Trabajos Relacionados

Existen en la literatura gran cantidad de grupos investigando sobre el rendimiento de diferentes modelos de programación. En esta sección se introducen algunos de los trabajos más relevantes relacionados con esta Tesina.

La unidad de ejecución en entornos de memoria compartida, y como consecuencia en entornos multicore, son los threads. En los sistemas operativos existen dos claros niveles de implementación, los threads nivel de usuario y los threads del kernel. Los primeros tienen la ventaja de ser hilos muy ligeros y gestionados por un planificador a nivel de usuario, por el contrario los threads de kernel se gestionan por el planificador de sistema operativo. A la hora de utilizar unos u otros se debe tener en cuenta el rendimiento. En [11] se evalúa el rendimiento entre threads a nivel de kernel y threads a nivel de usuario. En el documento se resuelve que los threads a nivel de kernel tienen un rendimiento significativamente peor. Como conclusión se refleja cuán esencial es en HPC la utilización de threads a nivel de usuario. Existen en diversos grupos de investigación diferentes implementaciones propias de librerías que implementan threads a nivel de usuario. El nanos group del BSC (Barcelona Supercomputing Center) tiene su propia librería de threads, originalmente nanothreads, Nanos4 [12], una implementación de ligthweight threads a nivel de usuario que dan soporte a diferentes lenguajes de programación. En el LaBRI (Laboratoire Bordelais de Recherche en Informatique) del INRIA han desarrollado otro modelo de librería de threads de nivel de usuario llamada Marcel [13], con un interfaz compatible con POSIX y un modelo de threads M:N (M threads de usuario asignados a N threads de kernel). En la librería Marcel hay que destacar el planificador BubbleSched que permite gestionar dinámicamente la asignación de threads sobre procesadores heterogéneos. El framework de BubbleSched permite que los desarrolladores creen su propia política de planificación así como de depuración ya que proporciona herramientas de rastreo (tracing tools).

No obstante, se considera que la programación con threads en aplicaciones paralelas es una programación a bajo nivel. Existen otros modelos de programación que abstraen la creación, gestión y eliminación de threads. Uno de los modelos más extendidos es OpenMP, que permite definir el paralelismo de una aplicación de forma bastante sencilla. En la literatura existen muchos estudios sobre el rendimiento obtenido con este modelo de programación paralela en diferentes arquitecturas de memoria compartida. En [14] se han evaluado diferentes benchmarks de OpenMP como el paquete de microbenchmarks OpenMP del EPCC (Edimburgh Parallel Computing Center) [15], NAS Parallel Benchmarks (NASA Advanced Supercomputing) [16] y SPEC (SPEC

OMPL2001) [17]. En este estudio se ha evaluado la ejecución de estos benchmarks sobre una máquina Sun Fire 15K SMP, con soporte de 72 a 102 procesadores y procesadores UltraSPARC III a 900MHz. En el estudio se utilizan los EPCC benchmarks para evaluar los overheads en la utilización de diferentes directivas OpenMP sobre un diferente número de threads (6,12,24,48,64 y 70) ejecutados en la máquina Sun Fire 15K. Sobre los benchmarks de aplicación de NAS y SPEC se evalúa la escalabilidad del sistema Sun Fire 15K. En este documento se concluye que el sistema escala bien en 5 de los 7 benchmarks evaluados del SPEC OMPL2001. Sobre el conjunto de NAS, se ha evaluado la clase B de los benchmarks CG con escalabilidad superlineal, LU con una escalabilidad perfecta (menos para 70 threads), los BT, SP i MG muestran un relativo buen rendimiento, y el rendimiento conseguido en FT resulta muy pobre. Con el paquete de microbenchmarks de EPCC han proporcionado una visión de escalabilidad de las directivas individuales de OpenMP y se ha medido el overhead de utilización de las directivas, para estimar el sobrecoste en benchmarks de aplicación como los SPEC OMPL2001, comprobando que el sobrecoste de utilización es muy pequeño, apenas representando el 1 % del tiempo de ejecución, con la excepción del CG del NAS, dónde el overhead por la utilización de OpenMP alcanza un sobrecoste estimado del 12 %.

El modelo de programación OpenMP corresponde a arquitecturas de memoria compartida, y estas arquitecturas normalmente permiten poca escalabilidad, por tanto son muy comunes los estudios que evalúan arquitecturas con diferentes niveles de programación. Los nodos con modelos SMP (Symmetric Multi Processing) y de colaboración entre nodos en una arquitectura distribuida utilizando el modelo de programación MPI (Message Passing Interface) [18]. Por otro lado se puede simular un modelo de memoria compartida dentro de un sistema distribuido a través de una capa de software. El modelo de Software Distributed Shared Memory es evaluado en [19] sobre un modelo de programación OpenMP. Para la evaluación del modelo SDSM se han utilizado dos benchmarks NAS (EP y CG) y el Ocean Kernel del Splash2 benchmark suite [20]. En este estudio se proponen técnicas para mejorar el prefetching de páginas de memoria en regiones iterativas paralelas, detectando a qué páginas accede cada iteración y replanificando las iteraciones para favorecer la localidad espacial y temporal. Como conclusiones del estudio, se indica que algunas aplicaciones de benchmark consiguen un buen speedup, mientras que las que no consiguen un buen speedup tampoco consiguen mejoras en otros sistemas DSM. Por tanto, se concluye la necesidad de replantear algunos conceptos del sistema para mejorar el balanceo de carga, que resulta ser el factor que más está afectando al rendimiento.

Tal como se ha reflejado anteriormente OpenMP ha sido evaluado en numerosas arquitecturas. OpenMP es un modelo de programación que permite la definición de regiones de trabajo que se

van a ejecutar de forma paralela. Esta sección paralela es ejecutada con un patrón de tipo fork-join. En las nuevas revisiones de OpenMP se han creado nuevas funcionalidades como la definición de regiones paralelas anidadas o, ya más recientemente, con la inclusión de un modelo de programación funcional mediante la definición de tareas (Tasks) ejecutadas sobre un modelo DAGs (Grafos Acíclicos Dirigidos). En [21] se ha realizado un estudio de los sobrecostes obtenidos en la utilización de OpenMP con paralelismo anidado de regiones paralelas. La máquina utilizada para la evaluación ha sido un servidor Compaq Prolian ML570 modelo SMP con 4 procesadores Intel Xeon III single-core. En este estudio se ha evaluado la implementación de diferentes compiladores sobre la funcionalidad de paralelismo anidado. Los compiladores evaluados son algunos de los más extensamente utilizados como Intel C Compiler (ICC), Sun C Compiler (SUNCC), GCC (GNU C Compiler) y también los compiladores source-to-source Omni Compiler y OMPi. Este último permite la utilización de diferentes librerías de threads, por tanto se ha evaluado para diferentes librerías de threads, la primera es la librería por defecto basada en POSIX threads y la segunda un librería de threads de alto rendimiento basada en POSIX threads gestionados a nivel de usuario. El benchmark utilizado ha sido el EPCC. En cuanto a las conclusiones, se han evidenciado problemas de escalabilidad en la utilización del paralelismo anidado, si el número de threads creados aumenta respecto al número de procesadores disponibles. Se sugiere, que el modelo de threads a nivel de kernel utilizado en la mayoría de implementaciones, está generando una sobrecarga en la librería de tiempo de ejecución. Cuando el número de threads compitiendo por los recursos hardware excede significativamente el número de procesadores, el sistema se sobrecarga y los overheads de paralelización sobrepasan cualquier beneficio en el rendimiento esperado. La experimentación se ha limitado a dos niveles de anidamiento, quedando claro que añadir subniveles de anidamiento únicamente empeoraría la situación.

Cada implementación del modelo OpenMP tiene ciertas singularidades, ya que para la misma aplicación utilizando diferentes compiladores el tiempo de ejecución puede ser significativamente diferente. En [22] se evalúa el compilador de investigación OpenUH [23]. Este compilador es open source e implementa el modelo OpenMP v2.5, consiste en un compilador source-to-source que genera código C o Fortran77 portable entre plataformas. En este documento se evalúan las diferentes primitivas OpenMP para la sincronización de threads mediante el microbenchmark syncbench del paquete EPCC. Se han desarrollado diferentes implementaciones de las primitivas de barrera utilizando modelos clásicos [24] de la literatura como son las barreras centralizadas bloqueantes, barreras centralizadas, barreras diseminadas, barreras en árbol y barreras de torneo. Por tanto en este documento se refleja la afectación al rendimiento de las diferentes interpretaciones del estándar

OpenMP. Estos modelos se han evaluado en dos benchmarks kernel, el Additive Schwarz Preconditioned Conjugate Gradient (ASPCG), y el Generalized Incompressible Direct and Large-Eddy Simulations of Turbulence (GenIDLEST), ambos ejecutados con hasta 128 threads.

Si hasta ahora hemos visto estudios comparativos entre las diferentes implementaciones software respecto a una arquitectura común, existen estudios que intentan explotar las capacidades de la utilización de librerías que tienen en cuenta las arquitecturas multicore. En este grupo tenemos los modelos del BSC que consiguen explotar diferentes modelos de multiprocesadores, los Star Superscalar (*Ss). Están basados en el compilador Mercurium que es un compilador source-to-source desarrollado por el NANOS group del BSC. Se ha generado un entorno para diferentes tipos de arquitecturas multicore. SMP SuperScalar está orientado a la optimización en entornos SMP [25], CellSs para los procesadores CellBE de IBM [26], GPUSs para entornos híbridos con CPU + GPU (Graphical Processing Units)[27]. El modelo StarSs se define en un modelo de paralelismo de tareas mediante DAGs y se ha planteado como unas extensiones del estándar OpenMP que son interpretadas por el compilador mercurium. Cada modelo de StartSs tiene en cuenta las características de la arquitectura hardware subyacente.

En estos trabajos se puede observar la importancia de caracterizar la arquitectura hardware. Las arquitecturas multicore son muy diversas y están en constante evolución, hemos pasado de arquitecturas dual core, modelos con poca complejidad de interconexión basada en un medio compartido y jerarquías de memoria, modelos sencillos de implementar pero que no proporcionan mucha escalabilidad, hasta modelos más elaborados como los procesadores Cell de IBM que integran hasta 9 cores en un chip, para los cuales se ha diseñado una red de interconexión en anillo que proporciona una mayor escalabilidad. Por tanto si hablamos de multicore hay que hablar de escalabilidad de los cores, y los problemas problemas de rendimiento son diferentes en procesadores con unidades de cores y procesadores con cientos o miles de cores. Se debe modelar una teoría o modelo de rendimiento sobre los sistemas concurrentes que considere las propiedades esenciales de los nuevos modelos de procesadores en la era de la revolución multicore [28], [29].

## 1.5. Estructura del Documento

En cuanto a la estructura del documento, en la **sección 2** se expone el contexto de los *Entornos Paralelos de HPC*, que es el ámbito dónde se desarrolla esta investigación, explicando las diferentes arquitecturas, modelos de programación y aplicaciones paralelas HPC.

En la **sección 3** se entra en detalle en las *Entornos Multicore*. Se reflejan las diferentes características de los modelos actuales de procesadores multicore.

En la **sección 4** entramos a definir un *Marco de Trabajo* acotado sobre el cuál se van a desarrollar las características más importantes de arquitectura, modelo de programación y aplicación.

La **sección 5** consiste en un *Análisis de los Factores de Rendimiento* detectados en el marco de trabajo y sobre los cuales se ha desarrollado la experimentación que ha permitido evaluar el impacto de cada uno de ellos.

Para finalizar, la **sección 6** refleja las *Conclusiones* del trabajo de investigación sobre los factores de rendimiento en entornos multicore, que ha de sentar las bases para el desarrollo de Tesis para la definición de un modelo de rendimiento en entornos multicore.

# Capítulo 2

# Entornos Paralelos de HPC

En este capítulo vamos a fijar el marco general de la investigación, que se ubica en un entorno de High Performance Computing. El ámbito de HPC esta centrado en la resolución de grandes problemas de cálculo científico que requieren una gran cantidad de cómputo y que utilizan estrategias de paralelismo para su resolución. Por tanto, para aquellos problemas que no pueden ser resueltos secuencialmente en un tiempo razonable, se utilizan múltiples recursos trabajando en paralelo de forma coordinada. Aplicaciones de múltiples campos científicos (biología, física, criptografía, inteligencia artificial, astronomía, astrofísica, etc.) tienen grandes necesidades de cómputo para la resolución de problemas de simulación y predicción de modelos climáticos, dinámica molecular, cromodinámica cuántica, simulaciones para la predicción de incendios forestales, etc.

En el entorno de HPC se deben tener en cuenta tres factores que actúan de forma combinada, estos son: la arquitectura del computador paralelo, el modelo de programación utilizado que permite explotar los recursos del computador y las características de la aplicación ejecutada en este entorno. A continuación se exponen diferentes sistemas HPC, explicando sus principales características, con el objetivo de situar los sistemas multicore dentro de las arquitecturas HPC, explicar los modelos de programación existentes en HPC para identificar un modelo adecuado para entornos multicore y los paradigmas de las aplicaciones HPC. Por último, se detallan metodologías y herramientas dedicadas a mejorar el rendimiento de aplicaciones paralelas, de ellas se identificar aquellas herramientas que han de permitir medir factores de rendimiento en entornos multicore, y se expone la dificultad y necesidad en la definición un modelo de rendimiento específico para entornos multicore.

## 2.1. Arquitecturas paralelas

Se debe tener en cuenta la arquitectura del computador ya que su naturaleza determina el tipo de cálculos, la cantidad de recursos de cómputo o la capacidad de paralelismo. Si tenemos que clasificar las arquitecturas podemos referir una visión general mediante la taxonomía de Flynn [10]. Esta taxonomía considera cuatro tipos de arquitecturas.

- SISD (Single Instruction, Single Data stream), este modelo no explota el paralelismo. Corresponde a un modelo de monoprocesador.
- MISD (Multiple Instruction, Single Data stream), este modelo se utiliza en situaciones de paralelismo redundante, por ejemplo en sistemas duplicados para soportar Tolerancia a Fallos.
- SIMD (Single Instruction, Multiple Data stream), un procesador explota varios flujos de datos dentro de un único flujo de instrucciones para realizar operaciones que pueden ser paralelizadas de manera natural, por ejemplo, un procesador vectorial, como por ejemplo ILLIAC IV o hasta los actuales modelos de programación de propósito general que explotan los recursos vectoriales de las GPUs (Graphical Processing Units).
- MIMD (Multiple Instruction, Multiple Data stream), varios procesadores autónomos ejecutan simultáneamente instrucciones diferentes sobre conjuntos de datos diferentes.

De esta clasificación podemos relacionar las arquitecturas tipo con las arquitecturas reales existentes en la actualidad. Las arquitecturas SISD equivalen a un procesador mononúcleo y las MIMD con procesadores paralelos. Si bien el modelo MIMD encaja con los procesadores multicore, esta es una afirmación muy bastante general, así, las arquitecturas MIMD podemos clasificarlas en dos subtipos.

- Sistemas de Multicomputadoras de memoria distribuida. Estos sistemas están compuestos por nodos de procesamiento independientes, cada nodo con su propio espacio de memoria (NO Remote Memory Access), que trabajan de forma conjunta intercambiando información mediante una red dedicada de altas prestaciones como en los modelos MPP (Massive Parallel Processor), una red de comunicaciones de bajas prestaciones y alcance limitado (Clúster) o de jerarquía de redes con alcance de larga distancia (Grid). Estos modelos utilizan modelos de programación basados en el paso de mensajes como por ejemplo MPI o PVM.

- Sistemas de Multiprocesador de memoria compartida, en estos sistemas los procesadores tienen una visión global del espacio de memoria, estos sistemas pueden clasificarse a su vez como:

  - UMA (Uniform Memory Access), en estas arquitecturas el tiempo de acceso a cualquier posición de la memoria tendrá la misma latencia. Corresponden a este modelo las arquitecturas PVP (Parallel Vector Processors) y SMP (Symmetric Multi Processors)
  - NUMA (Non Uniform Memory Access), donde el tiempo de acceso dependerá de la localidad de los datos. Utilizan una capa DSM (Distributed Shared Memory) que puede estar implementada mediante hardware o software. Corresponden a este modelo COMA (Cache Only Memory Architecture), CC-NUMA (Cache Coherent) y NCC-NUMA (Non Cache Coherent)

## 2.2. Modelos de programación paralelos

Los modelos de programación paralelos permiten al usuario definir la utilización de los recursos paralelos y expresar el paralelismo de la aplicación. Para que un modelo de programación tenga éxito el modelo debería cumplir las siguientes características [30]:

1. **Fácil de programar**. Debe permitir al programador representar fácilmente la descomposición del problema, la asociación de threads a los procesadores, la comunicación y la sincronización entre threads.

2. **Proporcionar una metodología de desarrollo**, para incluir el paralelismo en la semántica del programa, así como técnicas de transformación.

3. **Independencia de la arquitectura**, para permitir la portabilidad del programa entre máquinas.

4. **Fácil de entender**, para facilitar la reutilización de código.

5. **Garantizar rendimiento**. Garantizar una buena ejecución sobre diferentes arquitecturas

La forma en que los modelos de programación paralela expresan el paralelismo se pueden clasificar según la granularidad de la región paralela:

### 2.2.1. Paralelismo de aplicación

Modelo de alta granularidad. El paralelismo a nivel de aplicación tiene independencia de datos entre aplicaciones. Éste ámbito está más orientado a sistemas de HTC (High Throughput Computers), donde la métrica de referencia es el throughput, es decir la cantidad media de trabajos servidos por unidad de tiempo para todo el sistema.

### 2.2.2. Paralelismo funcional

De granularidad media, el paralelismo de funcional (o de tareas) se centra en la distribución de procesos de ejecución entre los diferentes nodos de cómputo. En cuanto a los lenguajes de programación de tareas existen Cilk, Cilk++, y OpenMP en la versión 3.0.

### 2.2.3. Paralelismo de datos

Este paralelismo es de granularidad fina, las regiones paralelas dependen de los datos de entrada, como por ejemplo en regiones de bucles iterativos. El paralelismo de datos se centra en la distribución de los datos entre los diferentes nodos de cómputo y puede permitir anidamiento de regiones paralelas. Este modelo supone tres etapas de ejecución, el primero consiste en la descomposición y repartición de datos, el segundo es la replicación de tareas y ejecución paralela sobre el subconjunto de datos, y el tercero es la sincronización y actualización de datos entre los diferentes procesos. Existe gran diversidad de lenguajes de programación paralelos que explotan este tipo de paralelismo, los lenguajes explícitos de threads como POSIX Threads o JAVA threads, o implícitos como OpenMP v2.5,

## 2.3. Aplicaciones Paralelas

En la programación de aplicaciones paralelas se debe caracterizar el comportamiento de la aplicación, es decir, identificar las necesidades de paralelismo en el algoritmo. Afortunadamente la mayoría de las aplicaciones científicas presentan un comportamiento que puede clasificarse dentro de un conjunto reducido de patrones de comportamiento. Cada uno de estos patrones de comportamiento se puede identificar con un paradigma de programación paralela como por ejemplo: Master-Worker, SPMD (single program multiple data), pipeline y divide and Conquer.

### 2.3.1. Paradigmas de Programación Paralela

La clasificación de patrones de aplicaciones paralelas engloba a la mayoría de aplicaciones paralelas existentes. Entre los paradigmas de programación paralela más conocidos tenemos:

#### 2.3.1.1. Master-Worker

Este paradigma se compone de dos elementos, el máster y los workers. El máster se encarga de dividir las tareas y distribuir las cargas de trabajo. Los workers son entidades que reciben la información, la procesan y devuelven los resultados al máster. En este paradigma la repartición de carga se puede realizar de forma estática o dinámica. La repartición estática distribuye la totalidad de la carga sobre el conjunto de workers. En la repartición dinámica se asignan divisiones de los trozos de carga a los workers y cuando éstos finalizan el cómputo, el Máster asigna más carga disponible hasta completar toda la carga de trabajo.

Existe un overhead debido a la gestión de comunicación entre el máster y los workers, y en el caso de ser un sistema de memoria distribuida un overhead debido a la comunicación de las cargas de trabajo.

#### 2.3.1.2. SPMD

Este patrón Single Program Múltiple Data tiene dos características a evaluar, el programa y los datos. Cada nodo del sistema paralelo va a ejecutar el mismo programa sobre una descomposición del conjunto de datos de entrada. Sin embargo, puede existir una dependencia de datos entre los diferentes nodos, que hará necesaria la existencia de comunicaciones entre nodos para resolver las dependencias. Por lo tanto, existe una necesidad de sincronismo entre los procesos que puede ser resuelta mediante paso de mensajes, en entornos distribuídos, o mediante la definición de secciones críticas, en entornos de memoria compartida.

Por tanto la barrera de sincronización viene determinada por el nodo más lento y puede existir un desbalanceo de carga. Se pueden aplicar estrategias de planificación para minimizar este desbalanceo.

#### 2.3.1.3. Pipeline

El paralelismo de este tipo de paradigma está basado en la descomposición funcional del problema, por tanto se deben identificar las subtareas del algoritmo, que pueden ser ejecutadas de forma paralela. Cada una de estas etapas funcionales tiene dependencia de datos con la etapa an-

terior. Cada etapa alimenta a la siguiente, y los patrones de comunicación pueden ser síncronos o asíncronos. La eficiencia de ejecución de éste paradigma depende del balanceo de carga en cada una de las etapas del pipeline, ya que la etapa más lenta acostumbra a determinar el tiempo de ejecución de la aplicación.

#### 2.3.1.4. Divide and Conquer

En este paradigma, los problemas son subdivididos en subproblemas, lo cual permite que cada una de esta subdivisiones pueda ser resuelta independientemente. Al finalizar la ejecución se integran todos los resultados que corresponderán al resultado final de la aplicación. En el proceso de partición, distribución y reensamblado se utilizan los recursos paralelos. La estructura de ejecución del programa tiene forma de árbol, ya que el proceso raíz subdivide el trabajo para los N recursos y asigna a cada una de las N ramas una carga de trabajo, donde cada una de estas ramas subdividirá el trabajo de forma recursiva.

## 2.4. Monitorización, Análisis y Sintonización

Los modelos de rendimiento son modelos matemáticos que permiten evaluar costes de alternativas de diseño de las aplicaciones, que deben permitir explicar observaciones de comportamiento, predecir circunstancias futuras o abstraer un sistema. Los factores que intervienen en un modelo de rendimiento definen la precisión del modelo y deben incluir características de la aplicación tales como el paradigma de programación utilizado, la carga de trabajo o precisión del modelo de programación tales como la granularidad, y de la arquitectura tales como factores de latencias de memoria, latencias de cálculo, etc. Debido a la dificultad de generar un modelo de rendimiento general, es posible crear un modelo de rendimiento para un entornos específicos, como por ejemplo para un paradigma de programación paralela sobre un tipo de arquitectura determinada. Por eso, en [31] se define un modelo de rendimiento para aplicaciones de tipo Master-Worker.

Teniendo definido un modelo de rendimiento se puede adaptar el sistema a las necesidades específicas de la aplicación. Esta solución puede resultar económicamente costosa si se requiere la adquisición de nuevo hardware, o al revés, el modelo puede permitir adaptar la aplicación a los recursos del sistema disponible, solución costosa en cuanto a recursos humanos por la necesidad de reprogramación de la aplicación para adaptarla al sistema. Sin embargo, es posible utilizar técnicas de sintonización de aplicaciones, que permiten que una aplicación sea portable entre sistemas, ya que la sintonización ha de permitir adaptar la aplicación a los recursos disponibles.

Para generar una estimación de comportamiento de la aplicación en un momento determinado, el modelo de rendimiento necesita conocer los parámetros de rendimiento de la aplicación. Para obtener esta información se pueden utilizar diferentes herramientas de análisis de rendimiento.

En general, la mejora de rendimiento de aplicaciones paralelas requiere tres fases principales: la monitorización, el análisis y la sintonización.

Existen herramientas para la monitorización, que insertan código de instrumentación en la aplicación, que generará un fichero con información sobre el tiempo de ejecución, contadores, muestreo o eventos. Los aplicaciones de monitorización pueden o bien mostrar trazas de ejecución, o como los profilers, ofrecer resúmenes de los valores de muestreo. Existen diferentes herramientas de monitorización, generales como el profiler gprof [32], más específicas como POMP [33] o ompp [34], que son profilers para aplicaciones OpenMP. La herramienta ompp permite obtener información para cada thread de ejecución, como tiempo de ejecución, el grafo de llamadas. También permite la integración del interface PAPI (Performance Application Program Interface) [35] que permite monitorizar contadores hardware reales del procesador, como fallos/aciertos de cache, operaciones en punto flotante,etc.

El análisis del comportamiento de la aplicación puede realizarse antes de su ejecución, por ejemplo durante la compilación; después de la ejecución, basada en trazas generadas durante la monitorización; o en tiempo de ejecución, adaptando dinámicamente la aplicación a las condiciones actuales de ejecución.

Las herramientas de análisis clásico (post-mortem), utilizan las trazas para, mediante una base de conocimiento, mostrar los problemas de rendimiento. Las herramientas MPICL [36], ParaGraph [37], Pablo [38], Vampir, VampirServer o VampirTrace [39] [40] son ejemplos de herramientas de análisis post-mortem.

Las herramientas de análisis automático, detectan los problemas de rendimiento y ofrecen sugerencias al usuario. Las herramientas Scalasca [41], Periscope [42], Tau [43], Paraver [44] y Dimemas [45] o KappaPi [46] son ejemplos de herramientas que generan un análisis automático post-mortem basado en información de monitorización.

Las herramientas de análisis dinámico (en tiempo de ejecución) permiten instrumentar la aplicación de forma dinámica obteniendo información específica en tiempo de ejecución. La herramienta Paradyn [47] es la única que genera un análisis dinámico.

En cuanto a herramientas de sintonización, éstas son capaces de monitorizar y analizar dinámicamente la aplicación para adaptarla al sistema. Las herramientas Autopilot [48], Active Harmony [49], Perco [50] y MATE [51]permiten sintonizar aplicaciones.

# Capítulo 3

# Entornos Multicore

En este capítulo se describen las características de los sistemas multicore existentes, y se plantean algunas tendencias de desarrollo futuro de estos sistemas.

En la actualidad, podemos encontrar procesadores multicore en estaciones de trabajo, portátiles, sistemas de HPC (ya que utilizan componentes genéricos para abaratar el coste), y también en sistemas empotrados. Los procesdores multicore representan el nuevo modelo tecnológico que debe permitir escalar las prestaciones en el futuro. Actualmente, en la denominada era multicore, disponen de un número pequeño de cores y la escalabilidad se consigue utilizando estrategias ad hoc. Sin embargo, al aumentar cada vez más el número de cores, gracias a las capacidades de integración de los transistores, se plantea un problema acerca del diseño de la arquitectura de los cores. Los chips monocore aplicaban estrategias de pipelining, branch prediction, etc, y estas estrategias eran implementadas físicamente utilizando circuitos lógicos que consumen transistores y espacio en el chip. Los primeros sistemas multicore básicamente duplicaban un sistema monoprocesador dentro de un mismo chip agregando una pequeña lógica de control y obteniendo así un sistema dual. Si pretendemos escalar el número de cores, cientos o miles, el espacio dentro del chip supone un problema y la arquitectura interna de los cores debe ser replanteada ¿que beneficio obtenemos de los complejos módulos de microarquitectura respecto al aumento de la escalabilidad de cores más simples? Los noveles modelos de sistemas multicore que integran decenas de cores utilizan la segunda estrategia, cores menos potentes pero con un número mayor de ellos.

Como consecuencia podemos intentar incluir el máximo número de cores posibles dentro de un chip, y suponer que las prestaciones van a aumentar de forma lineal. Sin embargo, esto no es así, existe un problema a la hora de escalar el número de cores, debido a unidades compartidas, como buses, memorias cache o interfícies compartidas para el acceso a memoria principal. En [52] Anant Agarwal plantea la KILL Rule, que sugiere que los recursos de un sistema multicore deben

ser incrementados sólo si por cada 1 % de incremento del área del core se obtiene al menos un 1 % de aumento de rendimiento, por tanto al pretender aumentar el recurso debemos mejorar las prestaciones, de lo contrario *"KIll if Less than Linear"*.

A continuación mostramos una clasificación de algunos componentes de los procesadores multicore, como diferentes planteamientos de diseño, que han de permitir aumentar la escalabilidad de los sistemas CMP. Gracias a avances tecnológicos como las arquitecturas de procesadores y memorias 3D se consigue un mayor integración y cercanía entre componentes que reducen la latencia en las comunicaciones. Por tanto, gracias a nuevos diseños que permiten mayor escalabilidad y la alta capacidad de integración, podemos afrontar la próxima era tecnológica de los procesadores, manycore.

## 3.1. Arquitectura de los cores

Una forma de clasificar los procesadores multicore depende de la arquitectura interna de los cores, ya que la funcionalidad de estos módulos puede ser homogénea o heterogénea. La mayoría de los procesadores actuales integra cores idénticos, como los procesadores Intel Core 2 o Tilera 64 [53]. Existen otros modelos que, aunque implementan el mismo repertorio de instrucciones, tienen diferentes características no funcionales, como por ejemplo los procesadores Sun Niagara T1 [54], cuyos cores comparten una unidad de punto flotante entre los 8 cores de forma transparente. Modelos heterogéneos con cores que ejecutan diferentes repertorios de instrucciones como los procesadores Cell de IBM, donde un core realiza la función de frontend, con un repertorios de instrucciones de tipo PowerPC, y los otros 8 cores, llamados *synergistic processing elements*, interpretan un repertorio de instrucciones RISC diferente.

Sin embargo, los sistemas heterogéneos no se basan únicamente en el repertorio de instrucciones, ya que un sistema heterogéneo podría consistir en un procesador con cores trabajando a diferentes frecuencias, que permitirían diferenciar unidades de paralelismo limitado y unidades de alto paralelismo. Existe una arquitectura que se correspone de forma aproximada a este modelo, los sistemas de aceleración por hardware GPU [55] utilizan los procesadores vectoriales de las tarjetas gráficas como unidades de soporte para la ejecución de alto cómputo numérico, donde las frecuencias de los cores difieren. Sin embargo este último ejemplo no siempre encaja dentro de un sistema multicore ortodoxo, ya que la comunicación entre el procesador interno y el procesador GPU (Graphical Processing Unit) no acostumbra a ser intra-core. Aunque cabe destacar la microarquitectura Larrabee de Intel [56], actualmente un proyecto cancelado, concebido como un

sistema híbrido CPU y GPU,

Los sistemas homogéneos tienen la ventaja de ser más simples a la hora de gestionar los recursos. Los modelos con unidades especializadas tienen la ventaja de ser más eficientes. Los enteramente heterogéneos prometen una alta escalabilidad siempre que la aplicación se puede adaptar de forma natural al hardware, por ejemplo, el procesador Cell [57] va a proporcionar buenas prestaciones sobre aplicaciones paralelas que utilicen un modelo masterizado.

La era manycore que basa la escalabilidad en el número de cores, debe tener en cuenta que existen unas implicaciones en el diseño de los CMP respecto a la capacidad de integración en el chip, por tanto las propuestas actuales van en el sentido de diseñar grupos de cores funcionales heterogéneos, de propósito cada vez más específico que como unidades independientes tienen una arquitectura cada vez más sencilla.

## 3.2. Jerarquías de memoria

Las primeras generaciones de sistemas multicore heredaron la jerarquía de cache multinivel de los sistema single core. Estos sistemas jerárquicos proporcionan un protocolo transparente de coherencia de memoria y permiten la comunicación entre cores mediante los niveles de cache compartida. Si bien desde entonces la evolución ha consistido mayormente en pequeñas modificaciones en cuanto a tamaños, alcance de la compartición y niveles de cache, podemos resumir este tópico en:

- Los primeros procesadores duales disponían de una cache de nivel 2 privada, y esencialmente eran un doble single core en un mismo chip con una mínima lógica de interconexión y un mecanismo de coherencia basado en el protocolo Snoopy. Éste, consiste en monitorizar el bus de acceso a memoria para comparar las direcciones de escritura de las operaciones de escritura con los bloques de memoria locales, y si detecta una escritura sobre un bloque de memoria local, entonces puede invalidar la operación. El modelo de compartición de caché de nivel 2 también existe en los procesadores más modernos, como el Tilera 64 aunque el protocolo de coherencia está basado en la gestión de un directorio de la interconexión en malla.

- Caches de segundo nivel compartida entre cores, como en los modelos Sun Niagara (3 MB L2 cache) [54], así como los Intel Core 2 (entre 2 y 6 MB) [58].

- Caches de segundo nivel separadas y una cache L3 compartida. Los procesadores AMD

Phenom (512 KB L2 per core y 8 MB compartidos de L3) [59] o los más recientes Intel Core i7 (256 KB L2 per core, compartida L3 de 8 MB) [60].

- Compartición por subconjuntos. Así como en los modelos iniciales, donde se duplican procesadores para integrarlos en el mismo chip, los procesadores Intel Core 2 Quad consisten en dos unidades Core 2 Duo en un mismo chip. Por tanto, tienen una caché de L2 compartida por conjuntos de 2 cores.

Por otro lado, los procesadores Cell implementan otro modelo, donde los cores especializados, synergistic processing elements, tienen una memoria local de 256 KB que se comunica con el sistema memoria mediante transferencias DMA.

Las jerarquías de memoria permiten particionar el conjunto de datos, los diferentes niveles de compartición con niveles de latencias más bajas según la cercanía de la memoria con el procesador, permiten mejorar el rendimiento gracias a la propiedad de localidad de datos. Sin embargo esta característica depende de la carga de trabajo y de la aplicación, ya que algunas aplicaciones tienen un alto grado de compartición y otras no comparten en absoluto, por tanto la jerarquía de memoria debe ser lo suficientemente inteligente para adaptar su comportamiento a muy diferentes condiciones. La tendencia es que los sistemas CMP es que éstos tienen cada vez procesadores, y para estas arquitecturas las capas de jerarquía de memoria son cada vez más fragmentadas, se proponen modelos capaces de adaptarse a las necesidades pudiendo cambiar su estado entre privado y compartido. En [61] se evalúan arquitecturas de memoria cache adaptativas. El modelo ESP-NUCA (Enhanced Shared-Private Non-Uniform Cache Architecture) evalúa diferentes políticas de gestión de estos recursos.

## 3.3. Redes de interconexión

La interconexión entre los nodos inicialmente ha consistido en el acceso a un medio compartido mediante una jerarquía de memoria, sin embargo este modelo presenta poca escalabilidad, y cuando hablamos de decenas de cores se precisa una interconexión más elaborada. Los procesadores Cell de IBM interconectan los 9 cores mediante una red en anillo de alta velocidad. Este modelo presenta ciertas ventajas respecto a los buses ya que requiere un menor consumo y permite frecuencias más elevadas debido al menor número de líneas de comunicación y a una simple política de arbitraje de acceso al medio. Existen modelos que proporcionan baja latencia y un alto ancho de banda, como las redes crossbar dedicadas de los procesadores Sun Niagara.

Si queremos entrar en la era manycore, donde el número de cores aumenta de decenas a cientos o miles de cores, entonces se debe aplicar un modelo de red altamente escalable. Por las necesidades de interconexión de estos sistemas multicore y la similitud con sistemas HPC se pueden utilizar estrategias utilizadas en HPC aplicadas en entornos multicore. Éstas redes de interconexión se llaman Network-On-Chip (NOC).

Por este motivo los procesadores Tilera utilizados en sistemas empotrados que disponen de hasta 100 cores implementan una Network-On-Chip basada en una red conmutada con una topología de malla 2D.

Como modelos teóricos para sistemas manycore tenemos los modelos de fat trees, que son una solución para obtener mayor escalabilidad. Y como hemos hablado de sistemas multicore con cores heterogéneos, entonces la interconexión entre estas diferentes unidades funcionales podrían ser de forma diferente entre módulos y con una conexión global jerarquizada.

Al aumentar el número de cores en una Network-On-Chip se deben implementar protocolos de comunicación y encaminamiento dentro de la red, conformando un modelo intracore distribuido. Por este motivo en rMPI [62], existe una propuesta para integrar dentro del repertorio de instrucciones (ISA) primitivas del modelo de paso de mensajes que permitan la comunicación intracore de baja latencia. Para favorecer la compatibilidad, utiliza primitivas del modelo de programación MPI, que es el estándar *de facto* en entornos distribuidos. El repertorio de instrucciones del modelo rMPI ha de permitir al programador controlar todos los recursos del chip, incluyendo puertas, cables y pines. Es conocido que los sistemas distribuidos son altamente escalables, utilizando estas arquitecturas de interconexión, se consigue alta escalabilidad en sistemas CMP.

## 3.4. Interfaz de acceso a memoria

El los sistemas multicore el acceso a memoria principal es un cuello de botella, por causa de ser un medio compartido concurrentemente. El rendimiento puede verse afectado por la latencia de acceso a memoria. Para mejorar este cuello de botella se proponen diseños de memoria en 3D [63]. Estas memorias tienen menor latencia por estar más próximas al procesador, ya que se sitúan sobre la CPU, y utilizan canales de transmisión de alto ancho de banda. En la construcción de estas memorias, se superponen capas 2D de células DRAM encima del procesador. Los buses de comunicación, llamados TSV (Through Silicon Vias) son grabadas en el silicio durante el proceso de creación. Este tipo de arquitectura está siendo muy estudiada, gracias a que la tecnología permite crear estos diseños tridimensionales que consiguen integrar más elementos y mas cercanos entre

ellos.

# Capítulo 4

# Marco de trabajo

Debido a la diversidad de arquitecturas, modelos de programación y paradigmas de aplicación, para estudiar los factores que afectan al rendimiento en entornos multicore, se ha definido un marco de trabajo acotado para iniciar la investigación. En este capítulo se desarrollan los elementos del marco de trabajo evaluado, que consisten en la definición de una arquitectura multicore y el desarrollo de los fundamentos teóricos del modelo de programación utilizado para expresar el paralelismo de las aplicaciones.

## 4.1. Nivel de Arquitectura

Para la selección de la arquitectura de estudio se ha utilizado un procesador multicore. En entornos multiprocesador, como los sistemas SMP (Symmetric Multi Processor) de memoria compartida, tienen estructuras similares, sin embargo contienen elementos en la arquitectura que pueden afectar a la medición de los factores de rendimiento. El procesador corresponde a un modelo MIMD multiprocesador de memoria compartida con acceso a memoria de tipo UMA (Uniform Memory Access):

- Entorno 2 cores:
  - Core2Duo E4700 @ 2.6GHz.
  - Caché L1 Datos 32 KBytes. Tamaño de línea 64 Bytes. Número de líneas 512. Asociatividad de 8 vías.
  - Caché L1 Instrucciones 32 KBytes. Tamaño de línea 64 Bytes. Número de líneas 512. Asociatividad de 8 vías.

- Caché L2 Unificada 2048 KBytes. Tamaño de línea 64 Bytes. Número de líneas 32768. Asociatividad de 8 vías.
- Memoria RAM 2 GBytes.

## 4.2. Nivel de Modelo de programación

Existen diferentes modelos de programación; orientados al paralelismo funcional o al paralelismo de datos. Como cada uno de ellos tiene diferente problemática, el marco de trabajo ha elegido un modelo de programación basado en el paralelismo de datos.

Los lenguajes de programación dentro de este modelo pueden ser explícitos, como por ejemplo las librerías de threads, o explícitos como OpenMP. El estudio se ha centrado en OpenMP, por causa de múltiples ventajas: es un estándar muy extendido en entornos de memoria compartida, es un modelo ampliamente estudiado en HPC, existen implementaciones para diversos lenguajes de programación (C, C++, Fortran), además es un modelo que está en constante revisión y actualización, desde su aparición en 1997, ha evolucionado hasta la actual versión 3.0, en que el modelo permite expresar tanto el modelo de paralelismo de datos como el modelo de paralelismo funcional. Para ajustar el marco de trabajo, el estudio del modelo de programación se ha basado en la definición OpenMP en su versión 2.5 que corresponde a un modelo de paralelismo de datos.

Para poder evaluar los factores de rendimiento en aplicaciones que utilizan el modelo de programación OpenMP, se ha seguido la siguiente metodología:

- Estudio de la API de threads subyacente a OpenMP, que ha de permitir evaluar el alcance de la implementación OpenMP. Al ser los threads la unidad básica de ejecución en entornos paralelos de memoria compartida, se ha estudiado la API de los POSIX threads, para comprobar el nivel de abstracción de OpenMP sobre la librería de implementa los threads. Además el rendimiento de OpenMP vendrá determinado por su traducción a la librería de threads y el rendimiento obtenido con dicha librería.

- Estudio de la API OpenMP, esta fase de análisis ha permitido clasificar los elementos de la definición del estándar OpenMP para una implementación concreta (GCC libgomp). Se ha evaluado el alcance y capacidad de abstracción de la librería de threads.

- Análisis de factores de rendimiento, una vez analizados los componentes de la API de threads y de la API OpenMP, se han evaluado aquellos elementos de la API, que han de

permitir al programador expresar el paralelismo, y que dependiendo de su utilización pueden afectar al rendimiento de la aplicación. Este apartado está desarrollado en el capítulo siguiente.

En los siguientes apartados se desarrolla el contenido de los elementos explicados en la metodología anterior, y que han de permitir el análisis de los factores de rendimiento.

### 4.2.1. Threads

Las unidades mínimas de ejecución en sistemas multicore son los threads. Existen dos tipos de threads, los threads creados a nivel de usuario, y los kernel threads o threads a nivel de sistema operativo.

Los threads creados a nivel de usuarios se asocia en unidades planificables por el sistema operativo. Existe una capa que divide el espacio de memoria del sistema operativo entre espacio de usuario y espacio de kernel o núcleo. Los modelos de programación trabajan a nivel de usuario, por tanto los threads se asocia a procesos llamados LWP (Ligth Weight Processes) que proporcionan la interfície entre el contexto de la capa de usuario y la capa de núcleo. Los LWP son unidades planificadas por el sistema operativo, ya que éstos, están asociados a los kernel threads de la capa del kernel. Existen tres tipos de modelos de gestión de threads: los gestionados enteramente por el sistema operativo (modelo 1:1), donde cada thread asociado a un LWP se relaciona con un único kernel thread, los gestionados a nivel de usuario (N:1), que a través de un planificador de threads externo al kernel, son capaces de compartir una unidad planificable de LWP. Los últimos son los modelos híbridos (N:M) mostrado en 4.1, que permiten combinar la gestión de threads entre los niveles de usuario y kernel.

Los threads gestionados a nivel de usuario (modelos N:1 y N:M) tienen una menor latencia en la creación de threads y cambio de contexto, sin embargo, tienen un overhead en la gestión del planificador de threads de usuario. En el ámbito de HPC, existe diversidad de modelos de threads. Existen modelos de programación que utilizan threads gestionados a nivel de usuario (librería nanos4 del Barcelona Supercomputing Center, librería Marcel del centro LaBRI del INRIA) que como se ha comentado, proporcionan un menor coste de creación, y eliminación. Están librerías están específicamente diseñadas para modelos de paralelismo funcional, que requieren constantes replicaciones de tareas, por tanto, se benefician de un modelo de threads gestionado a nivel de usuario con menor overhead en la creación y cambio de contexto de los threads. Actualmente los sistemas operativos de forma general implementan por defecto el modelo (1:1) delegando la

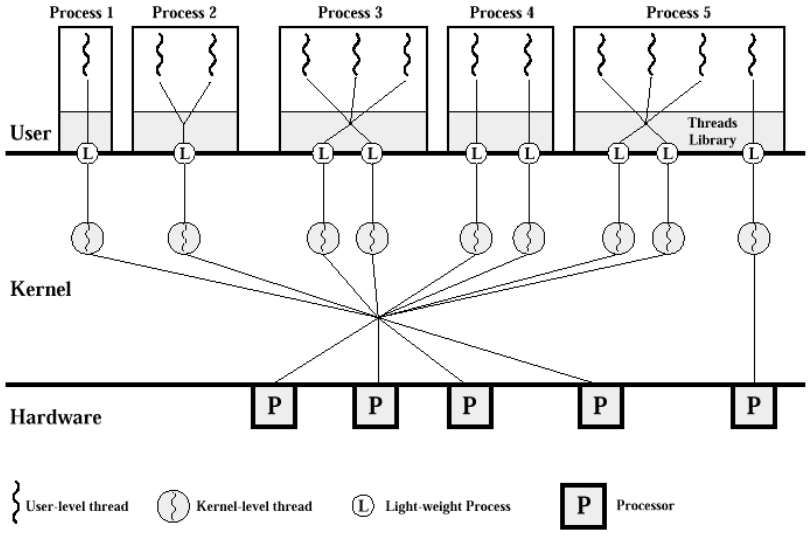


Figura 4.1: *Modelo de thread M:N*

planificación de los threads al sistema operativo, como las Native POSIX Threads Library o Solaris Threads, que en aplicaciones de propósito general obtienen un mejor rendimiento del sistema.

Las liberías de threads más utilizadas son aquellas que están integrados por defecto en los diferentes sistemas operativos. En sistemas Linux se utiliza la implementación NPTL con threads estándar POSIX, implementada con un modelo (1:1). En sistemas Solaris, hasta la aparición de Solaris 8 el modelo de threads era multinivel, modelo N:M, pero a partir de esta versión se cambió por un modelo 1:1. El sistema Solaris incluye dos API de threads, la librería de Solaris threads y la alternativa libposix que permite portabilidad entre sistemas Unix Like.

Históricamente ha habido problemas de compatibilidad entre aplicaciones que utilizan threads, debido a que cada sistema operativo ha utilizado implementaciones de librerías de threads propias. En 1988 surgió el estándar POSIX (Portable Operating System Interface for Unix) [64], que define un modelo de interfície común para entornos Unix-Like, como consecuencia, permite la portabilidad de código entre diferentes sistemas operativos. Sin embargo, aunque se ha definido una interfície común, cada implementación tiene sus propias interpretaciones y particularidades que pueden afectar de diferente forma al rendimiento de una misma aplicación. Existen algunos estudios que reflejan diferencias significativas en tiempo de ejecución sobre diferentes implementaciones de threads [65]. Las librerías evaluadas en este estudio son Solaris threads, Provenzano threads, FSU_threads, PCthreads, CLthreads, LinuxThreads. Las métricas que se han evaluado son:

- *Gestión de threads*, creación, sincronización y finalización de threads, tiempo de ejecución, y comparativas entre la utilización de threads y procesos, comparando los overhead de crea-

ción y overheads entre funciones de sincronización de threads y la función *wait* para la sincronización de procesos.

- *Gestión de la sincronización*, evaluación de las herramientas de sincronización y de exclusión mutua, como las funciones de lock/unlock, mutex lock/unlock sin contención, sobre variables condicionales en la generación de signal y broadcast, y tiempo de respuesta de variables condicionales.

Existe una gran diversidad de liberías que implementan threads, aquellas que cumplen con una definición estándar para ser más portables o aquellas que implementan funcionalidades específicas, íntimamente relacionados con el sistema operativo o gestionados a nivel de usuario, etc. Sin embargo, habiendo considerado los threads como la unidad de planificació básica en sistemas multicore, o de memoria compartida existen otras unidades de ejecución en los sistemas operativos como son los procesos,¿qué motivos hay para la utilización de threads y no de procesos?, si decidimos utilizar threads existen diferentes modelos, de usuario o de kernel ¿qué modelo de threads es conveniente utilizar?. Una vez definido el modelo de threads, debemos tener en cuenta las características de la implementación ¿qué características implementa mi librería de threads? En los siguientes apartados se van a tratar de desarrollar alguno de estos contenidos, desde las preguntas surgidas en las definición del marco de trabajo de la tesina.

#### 4.2.1.1. Threads vs. Process

Existen diferencias entre los dos tipos de unidades ejecutables en los sistemas operativos, los hilos de ejecución (threads) y los procesos. Ambos permiten paralelizar aplicaciones, sin embargo, los threads se han convertido en estándar de facto. ¿Qué ventajas ofrecen respecto a los procesos?

Los threads no dejan de ser más que unidades de ejecución asociadas a un proceso, al crear un grupo de threads estos comparten el espacio de direccionamiento de usuario y bloque de control de proceso. De forma general cada thread tendrá su propio puntero a pila, registros de estado, propiedades de planificación (política y prioridad), conjuntos de señales pendientes y bloqueadas y los datos específicos de thread (Thread Specific Data). A diferencia de los threads, en la creación de procesos cada nuevo proceso tendrá que duplicar los recursos de memoria.

- Menor tiempo de creación para un hilo que para un proceso, debido a que la creación de un nuevo thread utiliza el espacio de memoria del actual proceso.

- Menor tiempo en la finalización de un thread

- Menor sobrecarga en la comunicación entre hilos que entre procesos, debido a la compartición del mismo espacio de memoria.

#### 4.2.1.2. User level threads vs. Kernel level threads

Como se ha comentado anteriormente los threads pueden ser gestionados en la capa de usuario o de núcleo del sistema operativo. Cada uno de estos modelos tienen características diferentes que se detallan a continuación y sus ventajas e inconvenientes:

***User level threads***

- El kernel del sistema no gestiona la actividad de los threads, aunque si la del Ligth Weight process asociado
- Cuando un threads realiza una llamada al sistema, todo el proceso se bloquea, pero para la librería de threads el thread continúa en el estado de ejecución.
- Los estados del thread son independientes de los estados del proceso
- Ventajas:
  - El cambio de contexto entre threads no implica al kernel.
  - La planificación de threads puede ser adaptada específicamente al tipo de aplicación.
  - Los User level threads pueden ejecutarse en cualquier sistema operativo, ya que dependen de la librería.
- Inconvenientes
  - La mayoría de las llamadas al sistema bloquean el proceso, y por tanto todos los threads asociados al LWP.
  - Normalmente el kernel solo puede asociar procesos a procesadores, esto significa que dos threads en un mismo proceso se ejecutan simultáneamente sobre el mismo procesador

***Kernel level threads***

- La gestión de kernel threads es responsabilidad del kernel del sistema operativo.
- El kernel mantiene la información del contexto de threads y procesos, cambiar el contexto entre threads implica utilizar el planificador del kernel.

- Ventajas:
  - El kernel puede planificar simultáneamente múltiples threads del mismo proceso en múltiples procesadores.
  - El bloqueo se realiza a nivel de thread.
  - Las rutinas del kernel pueden ser multithread
- Inconvenientes:
  - El cambio de contexto entre threads involucra al kernel, y puede afectar al rendimiento.

#### 4.2.1.3. POSIX threads - implementación NPTL

Una de las librerías más extendidas es la NPTL (Native POSIX Threads Library), librería ha sido desarrollada por RedHat. Actualmente esta librería está integrada en la GNU C Library (glibc) distribuida por GNU con licencia GNU Lesser General Public License de la Free Software Foundation. Al proporcionarse con el sistema operativo Linux, está integrada en un gran número de sistemas. Esta implementación es compatible con el estándar POSIX threads (IEEE Std 1003.1c-1995). NPTL se convirtió en el sustituto de la anterior librería LinuxThreads por resultar incompatible con el estándar POSIX, y debido a las dificultades de modificar el diseño de LinuxThreads se adoptó la librería NPTL del proyecto RedHat. Algunas decisiones de implementación están explicadas en el documento de diseño de NPTL [66], donde se explican algunos detalles importantes respecto a la implementación, como la adopción del modelo de gestión de threads (1:1) en detrimento de otros, debido a la simplicidad relativa en la implementación respecto a un modelo (N:M) o que el modelo (1:1) simplifica la gestión de señales en el núcleo. Sin embargo, pese a esta limitación en el diseño, el NPTL se ha sido adoptado en los sistemas Linux principalmente por soportar POSIX.

Como se ha comentado anteriormente, la anterior librería LinuxThreads no cumple la definición del estándar POSIX. De la necesidad de utilizar una librería compatible, surgen las propuesta de RedHat con su NPTL, y NGPT (Next Generation POSIX Thread) de IBM, éste modelo de tipo (N:M). Éste último resultó complejo de implementar y finalmente el proyecto fue abandonado. También ayudó en la adopción de NPTL, la incorporación del soporte en el kernel linux de primitivas como *futex()* (Fast Userspace Locking), una nueva implementación de *Mutex* adaptada para threads, la adaptación de la llamada al sistema *clone()* para la creación de threads, y mejoras en las primitivas de gestión de señales y herencia en threads. La implementación NPTL ha formado

Cuadro 4.1: Resumen de primitivas de la librería POSIX

| | |
|---|---|
| pthread_ | threads y subrutinas |
| pthread_attr_ | atributos de thread |
| pthread_mutex_ | mútex |
| pthread_mutexattr_ | atributos de mútex |
| pthread_cond_ | variables condicionales |
| pthread_condattr_ | atributos de variables condicionales |
| pthread_key_ | datos locates a los threads |
| pthread_rwlock_ | bloqueos de lectura/escritura |
| pthread_barrier_ | barreras de sincronización |

parte del kernel linux desde la versión 2.5 y actualmente está incluida en la librería GNU C Library (libgc). En 4.1 se muestra un resumen de las primitivas definidas en API de POSIX threads.

Analizando la librería, vemos que la misma proporciona funciones con soporte para programas multithread, como funciones de sincronización, Thread Local Storage, etc. También están incluídos en esta implementación, los semáforos definidos en el estándar POSIX 1003.1b, que define extensiones realtime. A continuación se detallan algunas primitivas, y su uso en entornos multithread, del estándar POSIX threads incluídas en la librería NPTL.

***Thread - Creación y Finalización***

Primitivas para la creación de un thread, inicialización de una estructura de atributos del thread, y funciones para la finalización de un thread y destrucción de los atributos. Los atributos pueden ser `detachstate, schedpolicy, schedparam, inheritsched, scope, stackaddr, stacksize, stack i guardsize`

```
pthread_create (thread,attr,start_routine,arg)
pthread_exit (status)
pthread_attr_init (attr)
pthread_attr_destroy (attr)
```

***Threads - Barreras de Sincronización y Atributo de Separación***

La primitiva join indica un punto de sincronismo, esperando la finalización de un thread determinado. Por el contrario si se define un thread como *detach* (desasociado), el thread no va a generar una señal al finalizar, y por tanto este thread no va a ser *joinable*. El thread se ejecutará de forma

desasistida. Para la definición de un thread desasociado, se podrá llamar a la función de primitiva o durante la creación, configurando la estructura de parámetros de atributos de creación.

```
pthread_join (threadid,status)
pthread_detach (threadid)
pthread_attr_setdetachstate (attr,detachstate)
pthread_attr_getdetachstate (attr,detachstate)
```

***Threads - Gestión de pila***

Las siguientes primitivas permiten acceder al puntero de pila del thread o modificar el tamaño de pila.

```
pthread_attr_getstacksize (attr, stacksize)
pthread_attr_setstacksize (attr, stacksize)
pthread_attr_getstackaddr (attr, stackaddr)
pthread_attr_setstackaddr (attr, stackaddr)
```

***Thread - Planificación y alcance***

Estas definiciones de alcance de thread y definición de la política de planificación solo tienen sentido en implementaciones del tipo N:1 o N:M, ya que en los sistemas 1:1 (como el englobado en el marco de trabajo) la asociación del thread al LWP es directa y no modificable. Definición de las políticas de planificación del thread, que puede ser mediante una política First Input First Output (SCHED_FIFO) ejecutándose hasta la finalización el primer thread creado, mediante un política Round Robin (SCHED_RR) en la cual se ejecutará durante un quanto de tiempo . La política SCHED_OTHER consiste en la política de planificación por defecto del sistema operativo, en el entorno corresponde a la política time-sharing del planificador de Linux. La definición del alcance del thread permite la ejecución en Real Time definiendo el thread PTHREAD_SCOPE_SYSTEM o la ejecución en tiempo compartido con PTHREAD_SCOPE_PROCESS

```
pthread_getschedparam(thread, policy, param)
pthread_setschedparam(thread, policy, param)
pthread_attr_getscope(attr, contentionscope)
pthread_attr_setscope(attr, contentionscope)
```

***Threads - Rutinas diversas***

Las funciones pthread_self y pthread_equal trabajan con información de los threads, pthread_self

obtiene el identificador propio del thread, y pthread_equal permite comparar el identificador de dos thread. La función pthread_once es utilizado para la inicialización de valores de los threads, garantiza que se va a inicializar una única vez.

```
pthread_self ()
pthread_equal (thread1,thread2)
pthread_once (once_control, init_routine)
```

***Mutex - Creación y Terminación***

Los mútex son dispositivos de exclusión mutua. Estos dispositivos se utilizan para la protección de datos en el acceso a memoria compartida en modificaciones concurrentes, y la implementación de secciones críticas. Las siguientes primitivas permiten la creación y eliminación de un mútex. Así como la definición de atributos del mútex, como mútex rápidos mediante *PTHREAD_MUTEX_ADAPTIVE_NP*, mútex recursivos con *PTHREAD_MUTEX_RECURSIVE_NP*, mútex temporales *PTHREAD_MUTEX_TIMED_NP*,
mútex de comprobación de error *PTHREAD_MUTEX_ERRORCHECK_NP*. La extensión NP de estos modelos indica que es una funcionalidad añadida de implementación y no definida en POSIX.

```
pthread_mutex_init (mutex,attr)
pthread_mutex_destroy (mutex)
pthread_mutexattr_init (attr)
pthread_mutexattr_destroy (attr)
```

***Mutex - Bloqueo y desbloqueo***

Las siguientes primitivas permiten el bloqueo y desbloqueo de un mútex definido. La primitiva *pthread_mutex_trylock* intenta bloquear el mútex y si no lo consigue continúa la ejecución retornando, a diferencia de *pthread_mutex_lock* que bloquea la ejecución esperando que el mútex quede liberado. Para conocer el estado *pthread_mutex_trylock* retorna el código de estado indicando la consecución o no del mútex.

```
pthread_mutex_lock (mutex)
pthread_mutex_trylock (mutex)
pthread_mutex_unlock (mutex)
```

***Variables condicionales - Creación y terminación***

Las variables condicionales son herramientas de sincronización que permiten la suspensión de un thread hasta que se cumpla el predicado de condición sobre una región de memoria compartida, controlando los cambios de estado mediante señales.

```
pthread_cond_init (condition,attr)
pthread_cond_destroy (condition)
pthread_condattr_init (attr)
pthread_condattr_destroy (attr)
```

***Variables condicionales - Espera y señalización***

Las siguientes primitivas realizan la gestión de señales sobre variables condicionales. La función *pthread_cond_wait* automáticamente bloquea el mútex y espera la señalización de la condición, el thread entra en estado suspendido sin consumir CPU, mientras no se produzca una señal sobre la variable condicional. Para generar una señal se utiliza *pthread_cond_signal* que envía una señal para un único thread, a diferencia de *pthread_cond_broadcast* que reinicia todos los threads en espera de la variable condicional.

```
pthread_cond_wait (condition,mutex)
pthread_cond_signal (condition)
pthread_cond_broadcast (condition)
```

***Thread Local Store - Variables globales específicas de thread***

Los thread pueden requerir de variables globales o estáticas, con diferentes valores para diferentes grupos de threads. Los Thread-Specific data de POSIX responden a esta necesidad. Cada thread dispone de una región privada de memoria o thread Specific data (TSD), en esta región se definen los índices de acceso a TLS (Thread Local Store) mediante claves. Estas claves son comunes a todos los threads, pero solo algunos threads las tendrán indexadas en su TSD, y por tanto solamente accesible por ellas. Mediante *pthread_key_create* creamos un TLS asociándolo a una clave y la función que se ejecutará en la destrucción del TLS, llamada con *pthread_key_delete* realizará la liberación de los recursos. Para obtener o definir los datos del TLS, utilizaremos *getspecific y setspecific* respectivamente. Para definir los datos asociaremos la clave a un puntero, como por ejemplo, si se quiere crear un TLS de 100 elementos, *pthread_setspecific(buffer_key, malloc(100);*

```
pthread_key_create (key, destr_function)
pthread_key_delete (key)
pthread_setspecific (key, pointer)
```

```
pthread_getspecific (key)
```

### 4.2.2. OpenMP

La API de OpenMP expresa el paralelismo de las regiones paralelas utilizando un modelo fork-join. Las regiones paralelizables del código son ejecutadas por un grupo de trabajo, donde los recursos de ejecución son los threads. El modelo fork-join que consiste en la definición de un thread máster, que es el encargado de: crear al conjunto de threads del grupo de trabajo, gestionar la carga de trabajo de cada thread y además incorpora elementos de control para definir puntos de sincronismo entre las diferentes unidades. Existen barreras de sincronismo implícitas y explícitas, por ejemplo al final de cada región paralela, donde el thread máster vuelve a tomar el control del flujo de ejecución para continuar con la ejecución serie. El modelo está detallado en la interfaz de programación de aplicaciones (API) definida por el consorcio OpenGroup [67],[68].

OpenMP permite abstraer la utilización de la librería de threads subyacente, esto le permite al programador, centrarse en aspectos relacionados con el cómputo paralelo. Para expresar el paralelismo, el programador puede definir variables de entorno, o realizar definiciones explícitas en el código mediante la instrumentación de sentencias "#pragma" y llamadas a funciones de la librería de tiempo de ejecución, la aplicación debe estar enlazada con la librería de tiempo de ejecución encargada de gestionar la creación, el flujo y la finalización de los threads.

Existen múltiples implementaciones del estándar OpenMP integradas en compiladores comerciales y de libre distribución 4.2, e implementaciones diseñadas para la investigación en entornos específicos. De estos últimas cabe destacar:

- **INTONE project** Versión prototipo de los compiladores Fortran y C, asimismo como soporte OpenMP con una librería en tiempo de ejecución.

- **Nanos++ y Mercurium** Librería de tiempo de ejecución y compilador source-to-source con soporte a los modelos de programación OpenMP y StarSs.

- **OdinMP** Implementación gratuita y portable. Desarrollada en el departamento de Tecnología de la Información de la Lund University, Suecia.

- **OdinMP2** una mejora de la distribución opensource OdinMP con soporte OpenMP 2.5 (excepto threadprivate). La distribución incluye un test de conformidad para OpenMP.

Cuadro 4.2: Compiladores con soporte OpenMP

| **Compiler Family** | **Supported languages** | **Enhancements** |
|---|---|---|
| SGI MIPSpro | Fortran, C, C++ | - |
| IBM XL C/C++ IBM XL Fortran | Fortran C, C++ | nested parallelism,static and runtime error checking |
| Compaq Fortran | Fortran | - |
| Sun Studio | Fortran 95, C, C++ | nested parallelism, static and runtime error checking |
| Portland Group Compilers and Tools | Fortran, C, and C++ | - |
| Absoft Pro FortranMP | Fortran, C, and C++ | - |
| Lahey/Fujitsu Fortran 95 | Fortran | nested parallelism, dynamic thread adjustment, |
| Fujitsu-Siemens Fortran 95 (Solaris) | Fortran | - |
| Intel Software Development Products | Fortran, C, and C++ | interoperability with POSIX and Windows threads, Cluster OpenMP |
| PathScale | Fortran, C and C++ | - |
| GNU | Fortran, C and C++ | Soporte OpenMP v3.0, nested parallelism |
| HP | Fortran, C and C++ | - |

- **Omni** Colección de programas y librerías (en C y Java). Omni convierte programas C y Fortran77 con pragmas OpenMP en código C, adecuado para compilar con el compilador nativo, linkado con la librería Omni OpenMP de tiempo de ejecución.

- **OMPi** OMPi, es un compilador ligero source-to-source para C.

- **OpenUH** compilador open source de OpenMP 2.5 en conjunto con C, C++, and Fortran 77/90, inicialmente para la arquitectura IA-64, y recientemente con soporte para arquitecturas x86 y x86_64. Desarrollado por la universidad de Houston. Implementa Linux ABI (Application Binary Interface) y API (Application Program Interface).

- **SLIRP** wrapper orientado a la creación de módulos para el interprete S-Lang.

Siguiendo la metodología de análisis de marco de trabajo, se exponen a continuación los elementos más relevantes del modelo OpenMP, que han de permitir evaluar una implementación específica en el marco de trabajo definido. Los componentes del modelo OpenMP que permiten expresar el paralelismo de la aplicación y la definición de los recursos que intervienen en una ejecución paralela son: variables de entorno, funciones de llamadas a la librería de gestión en tiempo de ejecución y la definición explícita, en el código fuente, del paralelismo y elementos concurrentes.

La definición explícita en el código fuente es convertida, en tiempo de precompilación, a estructuras de control y llamadas a la librería en tiempo de ejecución. A su vez, la librería en tiempo de ejecución requiere la inicialización de ciertos valores (número de threads, tipo de gestión de planificación de los threads, etc.). Éstos pueden ser definidos a través de variables de entorno. La estructura de datos ICV (internal control variables) contiene la información de los atributos de las regiones paralelas, éstas variables son inicializadas en el contexto de la aplicación, y que en caso de no estar definidos por el usuario son inicializados por defecto con valores dependientes a la implementación. En la tabla 4.3 se detalla una relación de prioridad en la asignación de ICV sobre las regiones paralelas.

A continuación se detallan los elementos que definen el modelo OpenMP:

***OpenMP - Variables de entorno***

Las variables de entorno permiten inicializar las ICV de forma implícita al programador, ya que son explícitas del sistema.

- OMP_DYNAMIC (true|false), permite definir la ICV *dyn-var*, encargada de des/habilitar el ajuste dinámico del número de threads en las regiones paralelas.
- OMP_MAX_ACTIVE_LEVELS (#), asigna el número máximo de niveles del anidamiento de regiones paralelas. (ICV:*max-active-levels-var*).
- OMP_NESTED (true|false), des/activa el anidamiento de regiones paralelas.(ICV:*nest-var*).
- OMP_NUM_THREADS (#), especifica el número de threads por defecto para una región paralela. (ICV:*nthreads-var*)
- OMP_SCHEDULE type[,chunk], que define el tipo de planificación aplicada a las regiones paralelas. (ICV:*run-sched-var*)

static (chunksize), divide las iteraciones en trozos del tamaño definido por *chunksize*, y son asignados a los thread aplicando un modelo Round Robin. Si no se define ningún tamaño se dividirá la carga total de iteraciones en partes aproximadamente iguales.

dynamic (chunksize), la planificación dinámica dividirá las iteraciones en trozos definidos por *chunksize*, y asignará estas cargas a los threads disponibles que no estén ejecutando, o a aquellos que habiendo acabado una su trabajo soliciten nueva carga.

guided (chunksize), esta planificación evalúa el trabajo pendiente y lo divide entre el número de recursos disponibles, hasta llegar a cargas del tamaño definido por el *chunksize*, en su defecto será 1.

auto (chunksize), al definir esta planificación, se delega la gestión al compilador o a la librería de tiempo de ejecución. En la implementación GNU esta planificación equivale a una *dynamic*.

- OMP_STACKSIZE (#[B|K|M|G]), define el tamaño en KBytes de la pila de cada thread creado. (ICV:stacksize-var).
- OMP_THREAD_LIMIT (#), define el número máximo de threads para todo el programa. (ICV:thread-limit-var).
- OMP_WAIT_POLICY [ACTIVE|PASSIVE], indica si en tiempo de espera los threads deben consumir tiempo de CPU o no. (ICV:wait-policy-var)
- GOMP_CPU_AFFINITY (#[-#|:#]*) Asocia el thread a un core/CPU. Permite definir una lista de asignación para cada nuevo thread, Esta variable de entorno es específica de la implementación GNU en su compilador GCC.

***OpenMP - Funciones de tiempo de ejecución***

Las funciones en tiempo de ejecución permiten controlar y consultar el entorno de ejecución paralela, las funciones de bloqueo permiten sincronizar el acceso a los datos compartidos y ofrecen rutinas de temporización

- *omp_get_active_level*; permite consultar el número de regiones anidadas activas.
- *omp_get_ancestor_thread_num*; obtiene el identificador del thread del nivel superior de anidamiento.

Cuadro 4.3: Relaciones de sobreescritura de las Internal Control Variables

| **construct clause, if used** | **overrides call to API routine variable** | **overrides setting of environment** | **overrides initial value of** |
|---|---|---|---|
| (none) | omp_set_dynamic() | OMP_DYNAMIC | dyn-var |
| (none) | omp_set_nested() | OMP_NESTED | nest-var |
| num_threads | omp_set_num_threads() | OMP_NUM_THREADS | nthreads-var |
| schedule | omp_set_schedule() | OMP_SCHEDULE | run-sched-var |
| schedule | (none) | (none) | def-sched-var |
| (none) | (none) | OMP_STACKSIZE | stacksize-var |
| (none) | (none) | OMP_WAIT_POLICY | wait-policy-var |
| (none) | (none) | OMP_THREAD_LIMIT | thread-limit-var |
| (none) | omp_set_max_ active_levels() | OMP_MAX_ ACTIVE_LEVELS | max-active-levels-var |

- *omp_(get|set)_dynamic*; habilita o deshabilita el ajuste dinámico de threads. Si se deshabilita el número de threads deberá ser explícitamente definido
- *omp_get_level*; informa del nivel de regiones anidadas para la región paralela actual. l
- *omp_(get|set)_max_active_levels*; definir/obtener el número máximo de niveles activos anidados.
- *omp_get_max_threads*; límite máximo de threads que van a ser utilizados en una región que no incluya una definición explícita.
- *omp_(get|set)_nested*; definen el número de nivel de anidamiento.
- *omp_get_num_procs*; retorna el número de procesadores.
- *omp_(get|set)_num_threads*; afecta al número de threads que van a ser utilizados en la región paralela.
- *omp_(get|set)_schedule*; permite definir el tipo de planificación por defecto.
- *omp_get_team_size*; retorna el número de threads de la región paralela activa.
- *omp_get_thread_limit*; retorna el número máximo de threads disponibles.
- *omp_get_thread_num*; retorna el número de threads involucrados en la actual región paralela.

- *omp_in_parallel*; retorna *true* si la llamada a la rutina se encuentra en una región paralela.
- *omp_(init|destroy)_lock*; inicializa un mútex para la definición de una sección crítica.
- *omp_(set|unset)_lock*; si se habilita, la región paralela suspende su ejecución hasta que se habilite el cerrojo.
- *omp_test_lock*; intenta obtener un mútex, si no obtiene el mútex no detiene su ejecución.
- *omp_(init|destroy)_nest_lock*; inicializa/destruye un mútex para secciones anidadas.
- *omp_(set|unset)_nest_lock*; habilita o deshabilita el anidamiento de regiones paralelas.
- *omp_test_nest_lock*; intenta obtener un mútex, si no obtiene el mútex no detiene su ejecución.
- *omp_get_wtick*; retorna la precisión del timer utilizado
- *omp_get_wtime*; permite obtener el tiempo transcurrido en segundos.

***OpenMP - Directivas***

En este apartado se introduce el comportamiento de las directivas OpenMP. Estas directivas se transforman en tiempo de precompilación a estructuras lógicas que gestionan las regiones paralelas, haciendo uso de las funciones internas de la librería de tiempo de ejecución.

- Construcciones paralelas

  *PARALLEL* cuando un thread encuentra una región paralela, éste se convierte en el máster de un nuevo grupo de threads, que ejecutan la sección en paralelo, incluido el máster thread. Existe una barrera implícita al final de la región paralela que permite la sincronización del grupo de trabajo.

- Construcciones colaborativas

  *FOR*; esta construcción indica que el bucle definido inmediatamente después de la definición, va a ser ejecutado entre un nuevo grupo de trabajo. El thread máster va a repartir el número de iteraciones a ejecutar entre los componentes del grupo de trabajo. El gestor de planificaciones utiliza por defecto una planificación dinámica, pero ésta puede ser definida de forma explícita. La definición del bucle debe limitarse a una forma canónica detallada en la especificación.

  *SECTIONS*; la definición sections corresponde a un trabajo colaborativo no iterativo, dónde los thread ejecutarán en paralelo los bloques estructurados dentro de la región paralela.

- *SINGLE*; garantiza que únicamente un thread va a ejecutar el bloque estructurado definido por la directiva *SINGLE*:

- Construcciones colaborativas combinadas; se permite la definición combinada de construcciones paralelas en una única directiva

  *PARALLEL LOOP*

  *PARALLEL SECTIONS*

- Construcciones de sincronización y ejecución única

  *MASTER*; define un bloque estructurado que será ejecutado por el máster thread del grupo de trabajo.

  *CRITICAL*; define un bloque estructurado que será ejecutado únicamente por un thread a la vez.

  *ATOMIC*; garantiza la actualización atómica de la variable que interviene de la operación aritmética definida inmediatamente después. La expresión debe utilizar una forma canónica definida en la especificación.

  *FLUSH*; realiza una operación para actualizar la vista de los valores de memoria.

  *BARRIER*; define una barrera explícita de sincronización, en el punto en que aparece la directiva.

  *ORDERED*; garantiza que el bloque estructurado dentro del bucle paralelizado definido inmediatamente después de la directiva, será ejecutado en orden.

- Construcciones para la compartición de datos

  *Atributos de variables definidas en una construcción*

  *THREADPRIVATE*; realiza una copia local a cada thread colaborativo, de las variables definidas en la directiva.

  *Atributos de variables no definidas en una construcción*

  *PRIVATE*; define una variable como local al thread.

  *FIRSTPRIVATE*; define una variable como local al thread y la inicializa al valor previo a la región paralela.

  *LASTPRIVATE*; define una variable externa a la región paralela como privada al thread, y al finalizar la región paralela actualiza el valor de la variable original.

*REDUCTION*; especifica un operador y una variable, sobre la que cada región paralela crea una copia, al salir de la región paralela se actualizará la variable original realizando la operación definida mediante el operador sobre cada uno de los valores parciales de las copias locales a los threads.

- Construcciones de copia de datos

  *COPYIN*; válido para una construcción PARALLEL, copia el valor privado del thread máster en el valor privado de cada miembro del grupo de trabajo.

  *COPYPRIVATE*; válido para una construcción SINGLE, permite generar una difusión broadcast del valor generado en la construcción SINGLE sobre los threads del grupo de trabajo.

#### 4.2.2.1. GNU C Compiler + libgomp

En este apartado se detallan algunas características de la implementación OpenMP de GNU. Esta implementación está integrada dentro del compilador GCC, que es capaz de compilar código fuente escrito en C, C++, Objective-C, Fortran, Java i Ada. Este compilador ha sido desarrollado para convertirse en el compilador del sistema operativo GNU. Además del gran número de lenguajes de programación que soporta, también proporciona soporte para el modelo de programación OpenMP, para ello, el compilador es capaz de realizar las conversiones de las definiciones de directivas en tiempo de precompilación y mediante la librería *libgomp* proporciona soporte en tiempo de ejecución. Por defecto utiliza la librería pthreads para la creación, gestión y terminación de los threads que intervienen en los grupos de trabajo paralelo.

El compilador GCC tiene un diseño modular de componentes, como se puede aprecian en 4.2 está basado en un Front End, Middle End y un Back End. El primero se encarga de realizar el escaneo y parseo del código fuente que genera árboles AST (Abstract Syntaxt Trees), éstos árboles son analizados por el módulo GENERIC, para generar una representación en árbol, independiente del lenguaje, además es capaz de detectar deficiencias y realizar optimizaciones simples. El siguiente módulo, llamado GIMPLE, realiza una conversión del árbol a tuplas de no más de tres operandos, utilizada para realizar posteriores optimizaciones del código. Durante este proceso se genera la conversión de sentencias OpenMP a estructuras de control y llamadas a funciones de la librería *libgomp*. El módulo RTL (Register Transfer Languaje) genera una representación intermedia muy próxima al código ensamblador.

Para evaluar estas representaciones intermedias, transformaciones y expansiones, el compila-

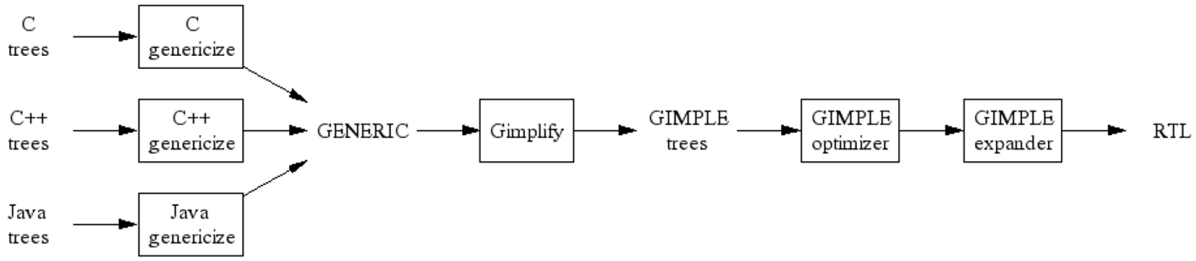


Figura 4.2: *Estructura modular del compilador GCC*

dor GCC permite la extracción de códigos intermedios durante la compilación de un programa, de esta forma podemos evaluar el código generado en la transformación de las directivas OpenMP. Algunos de los parámetros del compilador para extraer las representaciones intermedias son:

- *gcc -fopenmp code.c*; genera código ejecutable con la inclusión del modelo OpenMP.
- *gcc -fopenmp code.c -fdump-tree-gimple*; genera código intermedio, sin transformaciones de las directiva OpenMP.
- *gcc -fopenmp code.c -fdump-tree-omplower*; realiza la conversión al formato de tuplas, e inicializa variables y estructuras de control
- *gcc -fopenmp code.c -fdump-tree-ompexp*; expansiona el código, incluyen las transformación de las directivas OpenMP a llamadas a la librería libgomp.

Los detalles de la implementación están detallados en [69]. A continuación mostramos un ejemplo del código generado en los pasos intermedios.

***Código fuente con definición de directivas OpenMP*** Programa paralelo, implementado por el programador. Contiene la definición de una región paralela mediante la directiva *Parallel*. La sentencia de compilación es: `gcc -fopenmp code.c`

```
#include <stdio.h>
#include <omp.h>
int main (int argc, char *argv[]){
  int i;
```

```
  printf ("Hola mundo\n");
  omp_set_num_threads (4);
  #pragma omp parallel
  {
    #pragma critical
    {
  i = i + 8;
    }
  }
}
```

***Código intermedio con inicialización de estructuras*** El programa descrito anteriormente ha sido compilado con los flags que permiten extraer etapas intermedia de compilación. En este ejemplo se ha generado el paso previo a la transformación del código OpenMP. En esta etapa se ha generado una estructura de tuplas de las instrucciones, y se han inicializado estructuras de control y variables. La sentencia de compilación es: `gcc -fopenmp code.c -fdump-tree-omplower`

```
;; Function main (main)
main (argc, argv)
{
  int D.1643;
  int D.1644;
  struct .omp_data_s.0 .omp_data_o.1;
  __builtin_puts (&"Hola mundo"[0]);
  omp_set_num_threads (4);
  {
    .omp_data_o.1.i = i;
    #pragma omp parallel shared(i) [child fn: main.omp_fn.0 (.omp_data_o.1)]
      {
       .omp_data_i = &.omp_data_o.1;
       D.1643 = .omp_data_i->i;
       D.1644 = D.1643 + 8;
       .omp_data_i->i = D.1644;
       OMP_RETURN
      }
    i = .omp_data_o.1.i;
  }
}
```

***Código intermedio con expansión de las directivas OpenMP*** Este código genera una etapa más avanzada respecto a la representación anterior. Expande el código para la generación de la sección paralela, que utiliza llamadas a la librería libgomp, encargada de inicializar el grupo de trabajo y gestionar su ejecución. La sentencia de compilación es: `gcc -fopenmp code.c -fdump-tree-ompexp`

```
;; Function main (main)
OMP region tree
bb 2: omp_parallel
bb 3: OMP_RETURN
Merging blocks 2 and 6
Merging blocks 2 and 4
main (argc, argv)
{
  int i;
  int D.1643;
  int D.1644;
  struct .omp_data_s.0 .omp_data_o.1;
<bb 2>:
  __builtin_puts (&"Hola mundo"[0]);
  omp_set_num_threads (4);
  .omp_data_o.1.i = i;
  __builtin_GOMP_parallel_start (main.omp_fn.0, &.omp_data_o.1, 0);
  main.omp_fn.0 (&.omp_data_o.1);
  __builtin_GOMP_parallel_end ();
  i = .omp_data_o.1.i;
  return;
}
;; Function main.omp_fn.0 (main.omp_fn.0)
main.omp_fn.0 (.omp_data_i)
{
  int D.1647;
  int D.1646;
  int i [value-expr: .omp_data_i->i];
<bb 2>:
  D.1646 = .omp_data_i->i;
  D.1647 = D.1646 + 8;
  .omp_data_i->i = D.1647;
```

```
  return;
}
```

# Capítulo 5

# Análisis de los Factores de Rendimiento

Después de reflejar en el apartado anterior las características del marco de trabajo para una arquitectura y modelo de programación determinado, se ha realizado el análisis de aquellas primitivas de threads y OpenMP que permiten ser sintonizadas para una aplicación. En este capítulo se presentan los factores de rendimiento detectados, y se presentan los experimentos de realizados, que han de permitir observar cómo éstos factores afectan al rendimiento de aplicaciones.

## 5.1. Factores de Rendimiento

La librería POSIX threads permite una definición explícita para la creación, gestión y terminación de los threads. Como se comentó en el capítulo anterior, los threads son las unidades de ejecución mínimas planificables. Existe una gran interrelación entre los threads y el sistema operativo, ya que es necesario el soporte del sistema operativo al modelo de thread. En nuestro marco de trabajo el modelo soportado consiste en un modelo de threads de tipo (1:1), donde cada thread tiene asociado un LWP gestionado por el planificador del sistema operativo.

Para obtener un buen rendimiento en sistemas multicore es necesario poder especificar la asignación de threads a cores. Los sistemas multicore son sistemas altamente integrados, y en su arquitectura existen múltiples recursos compartidos. Resulta necesario poder controlar la asignación de threads a cores, ya que dependiendo de las características del programa a ejecutar y la distribución de los datos, se utilizarán unos recursos compartidos u otros. Utilizando el modelo explícito de threads es posible definir esta relación.

En el ámbito de threads y procesadores se utiliza el concepto de afinidad, que sugiere semejanza o parentesco. Los threads afines son aquellos que tienen semejanza en cuanto a su funcionalidad, o respecto a los datos con los que trabajan. Si hablamos de semejanza en cuanto a los datos co-

munes, en una arquitectura con diferentes elementos de memoria (por ejemplo la memoria cache), resulta beneficioso para la aplicación, agrupar threads afines en un mismo core para evitar overheads causados por fallos de cache, por ejemplo, si están accediendo a los mismo datos. En el caso contrario, si no existe afinidad cuando un thread está altamente acoplado a un core, si interfiere otro thread en la compartición del recurso de cómputo, el rendimiento resultará afectado.

Para poder definir afinidad entre threads es necesario que el sistema operativo soporte esta capacidad. En sistemas linux existe la librería *sched*, que integra la función *sched_setaffinity*. Ésta función permite especificar la ejecución de un thread sobre un procesador. Como se ha comentado anteriormente la programación explícita de threads se puede abstraer con el modelo de programación OpenMP. En la definición del estándar OpenMP no se incluye ninguna funcionalidad que permita la definición de afinidad. No obstante, la implementación de GNU incluye una variable de entorno que permite definir la afinidad y por tanto permite esta funcionalidad. La variable de entorno de OpenMP es *GOMP_CPU_AFFINITY*. Al definir la afinidad con esta variable, se debe especificar el orden de asignación de procesadores a los nuevos threads creados, mediante una lista cíclica. Por tanto, se ha de tener en cuenta este factor a la hora de medir el rendimiento de las aplicaciones en sistemas CMP.

Una vez los threads son asignados a los diferentes procesadores se pueden iniciar la ejecución serie o paralela. Sin embargo en los sistemas de memoria compartida, los threads iniciados pueden necesitar acceder a elementos compartidos de forma concurrente. Esto puede generar problemas en la lectura y escritura de datos i race conditions. Existen elementos de sincronización que controlan el acceso a los medios compartidos, pero, estos elementos de sincronización pueden generar problemas de interbloqueo e inanición. Aún cuando se controle el acceso a memoria, los elementos de sincronización son computacionalmente costosos, ya que pueden dormir un proceso o monopolizar el procesador y siempre generan un overhead debido a la gestión de la concurrencia. La librería de POSIX threads es capaz de gestionar la concurrencia de memoria compartida y la sincronización de threads mediante mútex, variables condicionales, y las extensiones de semáforos. En OpenMP también se proporcionan herramientas para el acceso concurrente de las unidades de ejecución, mediante la definición de regiones críticas, operaciones atómicas, reducción y funciones de la librería de tiempo de ejecución para la creación de elementos de exclusión mutua. Las diferentes operaciones de acceso a memoria compartida tienen diferentes latencias y flexibilidad. En nuestro caso, nos interesa evaluar el rendimiento, y las diferentes latencias de las operaciones de acceso a memoria. Existen múltiples estudios sobre la sincronización de acceso a memoria [70] [71]. Por tanto, los elementos de sincronización, aunque proporcionan herramientas para una eje-

cución correcta, siempre genen un overhead por la gestión de la concurrencia, por tanto este es un efecto que se debe minimizar y ha de ser un factor a tener en cuenta para evaluar el rendimiento de una aplicación.

La ejecución concurrente surge de los sistemas multitarea. En estos entornos, el número de threads es muy superior al número de procesadores disponibles. Para garantizar que algunos procesos no monopolicen el uso del procesador, el sistema operativo proporciona concurrencia a los threads, planificando quantos de tiempo de ejecución a cada proceso o thread. Las unidad planificables en un modelo de threads (1:1) son gestionadas por el sistema operativo, por tanto, si la librería de threads define diferentes tipos de planificaciones sobre threads, el sistema operativo deberá soportarlo, por otro lado, si el sistema operativo no soporta la definición de planificación para nuestra aplicación, entonces, perdemos esta funcionalidad. Así ha sucedido dentro del marco de trabajo, al no poder sintonizar las políticas de planificación de la librería de threads. No obstante, el modelo OpenMP implementa su propia gestión de planificación de threads a nivel de usuario, y aporta primitivas que permiten definir el tipo de planificación a utilizar en la gestión de los threads. Por tanto, este también es un factor de rendimiento que puede ser evaluado.

En este análisis hemos caracterizado algunos problemas de rendimiento para nuestro marco de trabajo. En la parte de experimentación se han planteado experimentos que han de permitir reflejar cómo afecta la parametrización de cada uno de ellos. Se han diseñado aplicaciones, que por sus características, permiten el estudio de estos factores de rendimiento dentro del marco de trabajo definido. En los siguientes apartados se explican en detalle las aplicaciones diseñadas y los factores que éstas van a permitir evaluar. Finalmente se desarrollan y explican los resultados obtenidos para estos experimentos.

## 5.2. Caracterización de Aplicaciones Paralelas

Dentro del marco de trabajo se han diseñado diferentes aplicaciones de tipo benchmark que han de permitir expresar problemas de rendimiento en el entorno de OpenMP. Las aplicaciones corresponden a un modelo SPMD (Single Program Multiple Data) en un entorno de memoria compartida. Por tanto estas aplicaciones tienen ciertas características comunes como son la descomposición del conjunto de datos de entrada, la repartición sobre los recursos disponibles y la ejecución de un programa común sobre cada uno de los recursos.

### 5.2.1. Segmentación de Imágenes de Resonancia Magnética

El benchmark de segmentación de IRM (Imágenes de Resonancia Magnética) se corresponde a un benchmark de tipo real, ya que pertenece al núcleo de una aplicación utilizada para el estudio de diferencias estructurales entre imágenes IRM correspondientes a la región cerebral del conjunto de imágenes de pacientes de estudio. Esta aplicación es utilizada en el campo de la neuroimagen para desarrollar estudios de VBM (Vóxel Based Morphometry). El núcleo ha sido implementado a partir del documento que detalla el proceso de segmentación según el modelo correspondiente a la Tesis Doctoral de John Ashburner sobre *Computational Neuroanatomy* [72]. El modelo implementado está detallado en el capítulo 5, [73], en el cual se explica el método de segmentación de IRM en las diferentes clases de tejido de las que se compone el cerebro, materia blanca, materia gris y líquido cefalorraquídeo. Utiliza un modelo probabilístico. Conociendo a priori las probabilidades de pertenencia de cada Vóxel (píxel volumétrico) para cada tejido, gracias a una plantilla generada, se permite obtener la clasificación de cada vóxel. La parte relacionada con la corrección de la no uniformidad en la intensidad de la imagen, no ha sido implementada, ya que las imágenes disponibles para la evaluación del modelo no estan afectadas por este artefacto de distorsión.

El modelo completo de segmentación de IRM está incluido en el paquete SPM (Statistical Parametric Mapping). Ha sido desarrollado en el lenguaje interpretado de Matlab. Las datos de entrada corresponden a un conjunto de imágenes de resonancia magnética, debidamente anonimizadas y convertidas del formato nativo DICOM a un conjunto de ficheros de texto plano que contienen los valores de imagen. Todas las imágenes tienen la misma dimensionalidad que corresponde a 91 x 109 x 109 píxeles volumétricos, y por tanto, cada píxel corresponde a un punto del espacio tridimensional. El valor de cada elemento corresponde a un valor de intensidad obtenido durante la resonancia, para cada punto del espacio. El valor de intensidad indican la densidad de tejido adiposo en ese punto, ya que siendo diferente la densidad del tejido adiposo para cada uno de los tejidos, es posible hacer la clasificación y segmentar la imagen.

El núcleo perteneciente a la segmentación de IRM tiene la característica de ser un algoritmo de altas necesidades de cómputo aritmético. El tipo de operaciones corresponden a sucesivas multiplicaciones de matrices punto a punto, cálculo de probabilidades y cómputo de una estimación de calidad. Con esta estimación, se compara el valor de mejora obtenida para cada iteración respecto al valor de mejora anterior y se calcula la diferencia entre ellas, este valor es comparado con un umbral de precisión, que determina si la mejora obtenida no es significativa, y por tanto no es posible mejorar la calidad de la imagen en siguientes iteraciones. El algoritmo realiza la segmentación para cada corte bidimensional de la imagen de entrada, y las operaciones de multiplicación

de matrices tienen una carga de trabajo de 91 x 109 operaciones flotantes.

El paralelismo del benchmark se puede aplicar a tres niveles, mediante un paralelismo de datos de granularidad fina, expresando el paralelismo en las operaciones de multiplicación de matrices, o bien mediante un paralelismo de datos de alta granularidad, expresando el paralelismo mediante un paradigma SPMD para cada corte de la imagen tridimensional. El último nivel de paralelismo corresponde a un paralelismo de aplicación, debido a que el proceso normalmente evalúa grupos de estudio, y por tanto dispone de un gran numero de imágenes tridimensionales de resonancia magnética que requieren ser segmentadas. Este proceso también se puede paralelizar a este nivel.

La segmentación de IRM corresponde a una aplicación embarazosamente paralela, sin dependencia entre niveles de imagen o corte, y que debido a la naturaleza de los datos (estructura cerebral), el cómputo tiene cierto desbalanceo de carga en las regiones internas respecto a los cortes más externos que apenas contienen información. Por tanto, esta aplicación se considera adecuada para evaluar diferentes técnicas de planificación, que permitan balancear la carga entre las regiones internas y externas de la imagen.

### 5.2.2. Multiplicación de Matrices

Esta aplicación corresponde a un toy Benchmark, el algoritmo calcula un multiplicación de matrices. Las multiplicación de matrices desarrollada corresponde a un paradigma SPMD, donde existe una descomposición del espacio de datos de la matriz resultante, donde cada posición de la matriz, de tamaño N x N, debe realizar N operaciones de multiplicación y acumulación del valor. La complejidad del algoritmo es de O($N^3$). El prograna ha sido diseñado para ser paralelizado en un entorno de memoria compartida mediante la definición de un paralelismo de datos a nivel de iteración, utilizando OpenMP sobre el bucle más externo. Para el cálculo de los valores de la matriz resultante se debe definir una política de acceso a la variable compartida. Se pueden evaluar las diferentes herramientas de exclusión mutua para esta variable.

### 5.2.3. Multiplicación de Matrices por Bloques

Este toy Benchmark implementa una multiplicación de matrices por bloques, donde cada nodo de computo realiza el cálculo sobre el bloque asignado de la matriz resultante. El tamaño del bloque es parametrizable, pudiéndose definir un tamaño de bloque que permita ajustar la cantidad de datos necesarios para el cómputo de un bloque sobre la arquitectura caché del procesadores. De esta forma, se favorece la localidad de los datos y por lo tanto se minimizará el overhead debido a la alta latencia de acceso a la memoria principal. La complejidad del algoritmo es de

O($N^3$) y el acceso a la variable de actualización de la matriz de salida se realiza mediante la primitiva de reducción, proporcionada por OpenMP para el acceso concurrente a memoria. Debido a que la implementación se ha realizado en ANSI C, éste tiene la caracteristica de almacenar secuencialmente en memoria los datos de una matriz por filas, por tanto, para alinear los datos, la multiplicación de matrices se ha realizado transponiendo la matriz B. De esta forma, se han almacenado los datos en memoria de forma contigua, para favorecer la localidad espacial, ya que conviene que el acceso a la lectura de la matriz no genere innecesarios fallos de caché, al no estar contiguos los datos físicos respecto a los datos lógicos.

Para la evaluación de este experimento se han utilizado herramientas de profiling como PAPI i ompp, instrumentando la aplicación en la región paralela. Con PAPI se han obtenido medidas sobre los contadores reales del procesador, tales como fallos de caché de los dos niveles existentes en la arquitectura del marco de trabajo y ciclos de ejecución, el tiempo se ha medido medido con el profiler ompp, que también permite obtener información a nivel de thread y sobre el uso de primitivas OpenMP.

## 5.3. Experimentación

En esta sección explicamos los objetivos y mostramos los resultados de los experimentos planteados para cada uno de los factores de rendimiento que a priori, deben afectar al rendimiento de las aplicaciones. Los factores de rendimiento evaluados dentro del marco de trabajo son las políticas de planificación de iteraciones en OpenMP, diferentes elementos de sincronismo que proporciona OpenMP, y un estudio sobre cómo afecta la localidad de los datos en un entorno paralelo, utilizando el modelo de programación.

### 5.3.1. Gestores de planificación

En este experimento, se pretende demostrar cómo los diferentes tipos de planificaciones afectan al rendimiento de la aplicación. Para este experimento se ha utilizado el benchmark de aplicación para la segmentación de imágenes de resonancia magnética. Los tipos de planificaciones que proporciona OpenMP son las planificaciones *static*, *dynamic* y *guided*. Estas planificaciones se aplican sobre el conjunto de iteraciones de una sección paralela definida en la aplicación de segmentación de imágenes IRM mediante *# pragma omp parallel for*. Esta sección se encarga de segmentar cada corte de la imagen, y por tanto el bucle iterativo recorre cada corte a segmentar de la imagen. La directiva OpenMP, sobre esta región a paralelizar, distribuye los cortes sobre el

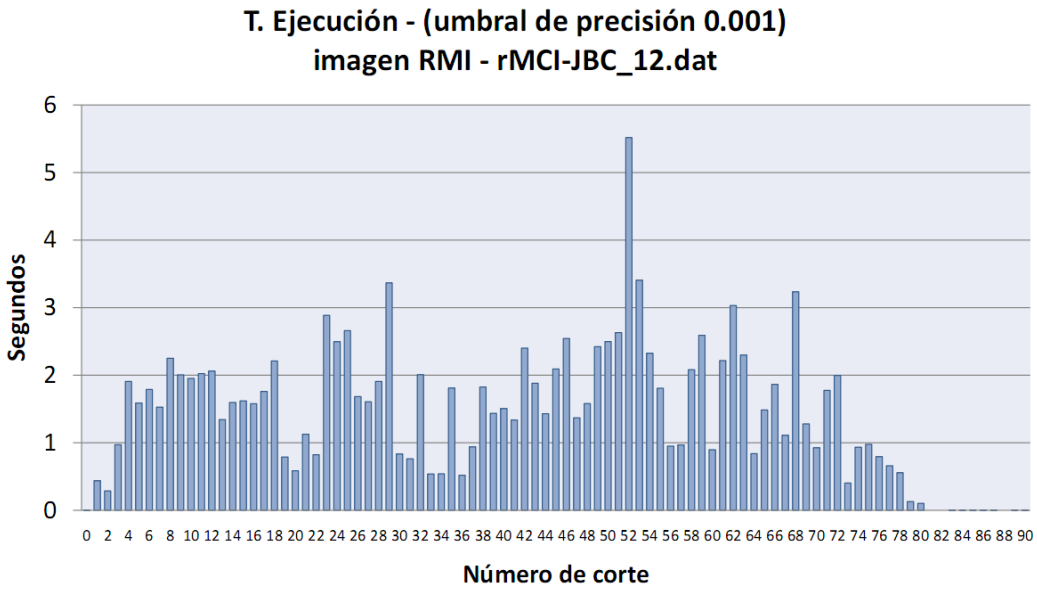


Figura 5.1: *Tiempo de ejecución del proceso de segmentación de imágenes RMI, realizado por cortes de una imagen 3D*

conjunto de procesadores.

En la experimentación, primero se ha querido caracterizar el comportamiento del algoritmo para los datos de entrada. Se ha ejecutado la aplicación en serie para cada imagen. Se ha observado que los tiempos de ejecución eran muy similares para todo el conjunto de imágenes. Se ha analizado el tiempo de ejecución de cada iteración del conjunto imágenes, en la gráfica 5.1 se presenta el comportamiento de la imagen etiquetada *rMCI-JBC_12.dat*. Se puede observar que los tiempos de ejecución están relativamente desbalanceados. Los cortes situados a los extremos representan poco tiempos de ejecución, que corresponde a la poca cantidad de información existente en ellas. En estos cortes laterales, el algoritmo finaliza rápidamente la segmentación, ya que al calcular la calidad durante el proceso de segmentación, converge rápidamente, al no alcanzar el umbral calidad necesaria considera que no puede obtener una mejora en la segmentación.

Una vez se ha comprobado que las cargas de trabajo para cada iteración presentan una ejecución desbalanceada, se ha evaluado el comportamiento de la aplicación aplicando paralelismo. Aunque por defecto, en la implementació de OpenMP, se gestiona una planificación dinámica, se ha definido el tipo de planificación de forma explícita. En la gráfica 5.2 se presentan los tiempos de ejecución al aplicar diferentes políticas de planificación sobre la imagen *rMCI-JBC_12.dat*.

Lo primero que se puede observar es que el tiempo de ejecución se reduce a la mitad respecto a la ejecución para 1 thread (ejecución serie), y la ejecución en 2 threads (ejecución paralela),

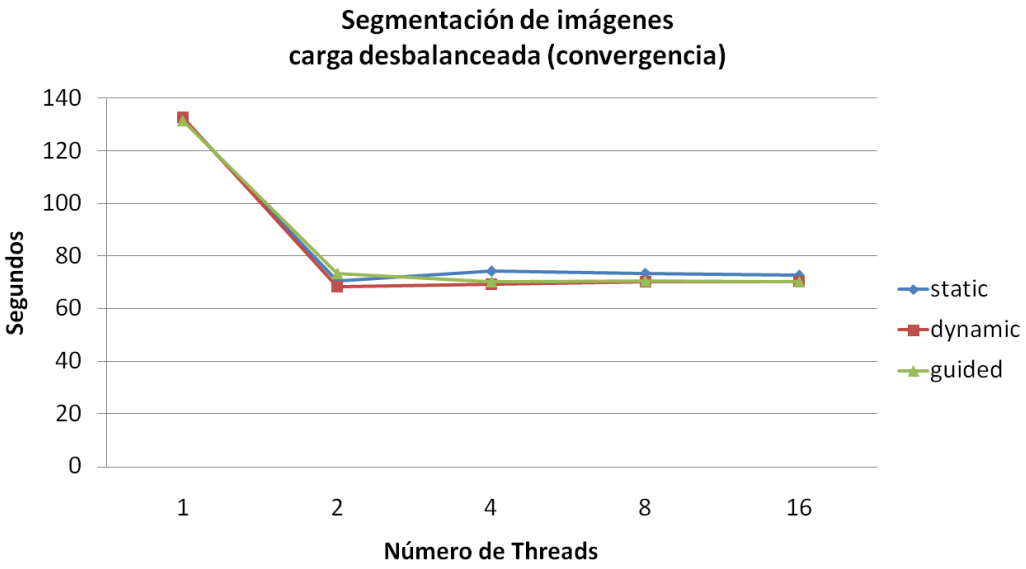


Figura 5.2: *Tiempo de ejecución del proceso de segmentación de imágenes RMI, para diferentes políticas de planificación y número de threads*

debido a que se están utilizando el doble de recursos, de 1 cores a 2 cores. Se ha ejecutado el experimento con un número variable de threads para determinar si existe una sobrecarga debida a la gestión de los threads. Al aumentar el número de threads los tiempos de ejecución, puede parecer que prácticamente se mantienen igual para las diferentes planificaciones.

No obstante para determinar la sobrecarga, en la gráfica 5.3 se muestra el comportamiento paralalelo, desestimando el tiempo serie. La planificación que ha obtenido el mayor rendimiento es la planificación *dynamic*. El resto de planificadores presentan una sobrecarga respecto a ésta. Esta sobrecarga puede ser debida a la gestión de los threads para cada planificación, como por ejemplo, un desbalanceo de datos debido a un específico reparto de carga. En la gráfica, también se muestra el porcentaje de sobrecoste para las políticas *guided* y *static* respecto a la ejecución *dynamic*. Siendo destacable los sobrecostes de la planificación guiada para 2 threads, y guiada para 4 threads, con un 7,4 % y 7,2 % respectivamente. Por tanto, no se refleja ninguna tendencia que demuestre una clara mejoría para el aumento de número de threads, y la mejor ejecución se corresponde a aquella que relaciona el mismo número de threads que cores. Esto no se aplica a la gestión estática, ya que esta política tiene una lenta adaptación en el balanceo de carga, ya que únicamente consigue minimizar el desbalanceo cuando aumenta la granularidad de la carga de trabajo.

Teniendo en cuenta que la ejecución con diferentes planificaciones, en este experimento, no

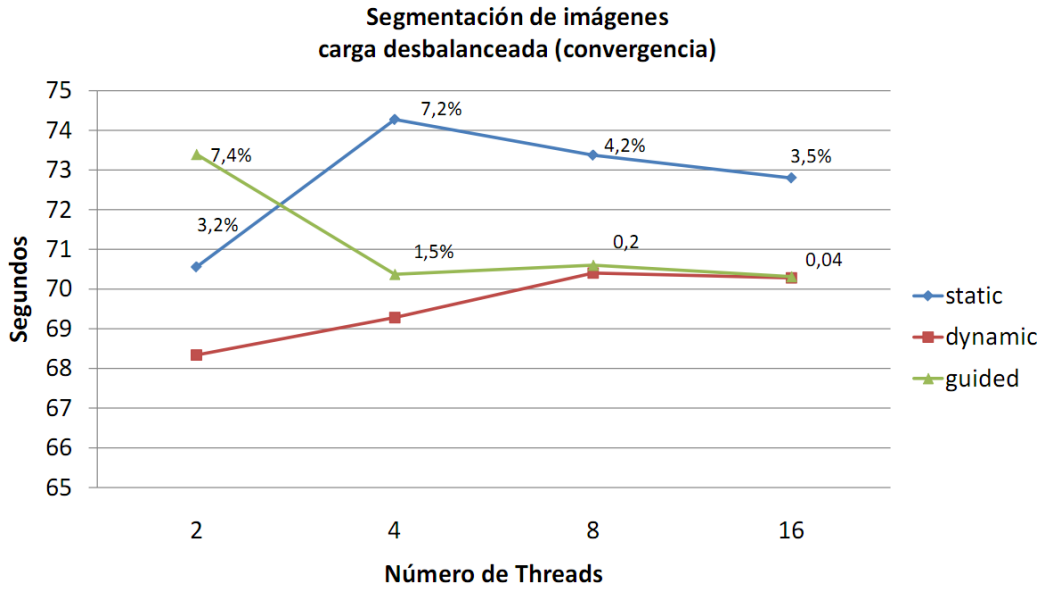


Figura 5.3: *Detalle de los tiempos de ejecución paralelos de la segmentación de IRM. Se muestran los overheads de las políticas guided y static, respecto a la planificación dinámica.*

se ha demostrado que la flexibilidad en la definición de políticas de planificación de OpenMP permita beneficiar a la aplicación, se ha evaluado para el resto de imágenes, obteniendo resultados similares. Por este motivo se ha desarrollado una carga de trabajo sintética. Para obtener esta carga se ha modificado el algoritmo de segmentación de imágenes para eliminar el control de calidad, y forzar un mismo número de iteraciones para cada corte. El objetivo es calcular el overhead neto generado en la utilización de las diferentes planificaciones. El resultado esperado consiste en una carga balanceada, donde cada corte ha de tardar el mismo tiempo en ejecutarse. El resultado obtenido se muestra en la gráfica 5.4.

En esta gráfica, se puede observar que tampoco se ha obtenido el resultado esperado, una carga sintética balanceada. El tiempo de ejecución de los cortes de imagen ha aumentado en los extremos. Esto es debido a latencias en la generación de una excepción provocada por una operación inválida, la división por cero. Sin embargo, esta carga sintética desbalanceada es sensible de ser evaluada para las políticas de planificación. En 5.5 tenemos los tiempos de ejecución para las diferentes políticas de planificación.

En la gráfica se puede apreciar que la planificación estática no consigue adaptarse a la carga de trabajo para 2, 4 y 8 threads. Esto es debido al funcionamiento de la planificación. Ésta divide la carga entre el número de threads al inicio de la ejecución de la región paralela. Por este motivo las cargas asignadas a los threads no están compensadas. Los primeros y últimos threads se

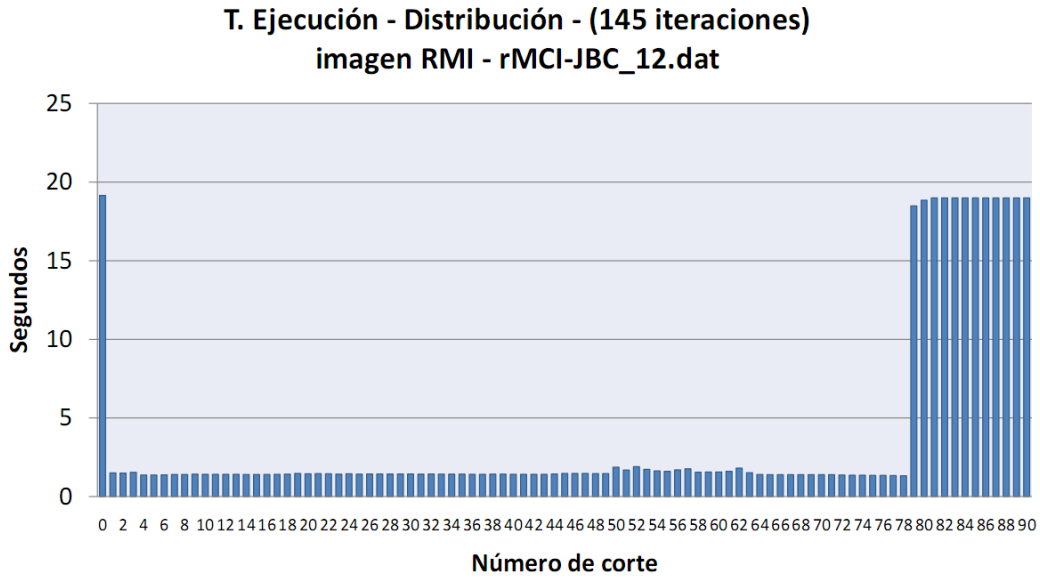


Figura 5.4: *Esta gráfica muestra el desbalanceo de carga existente, para una imagen de entrada, para el proceso de segmentación modificado, sin evaluación de calidad.*

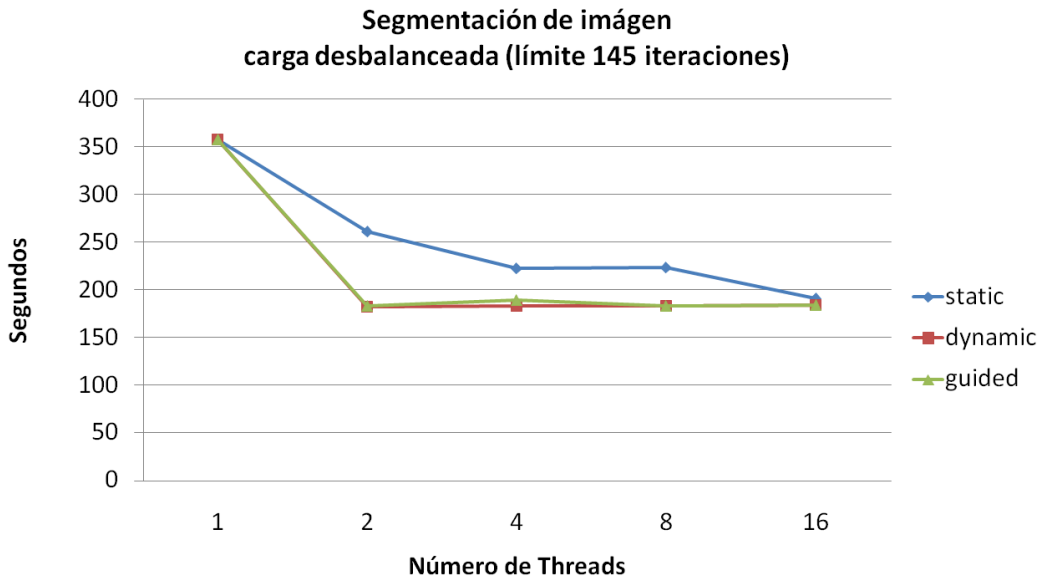


Figura 5.5: *Tiempos de ejecución para el proceso de segmentación modificado. Se han eliminado el control de calidad, forzando un número de iteraciones constante para cada corte.*

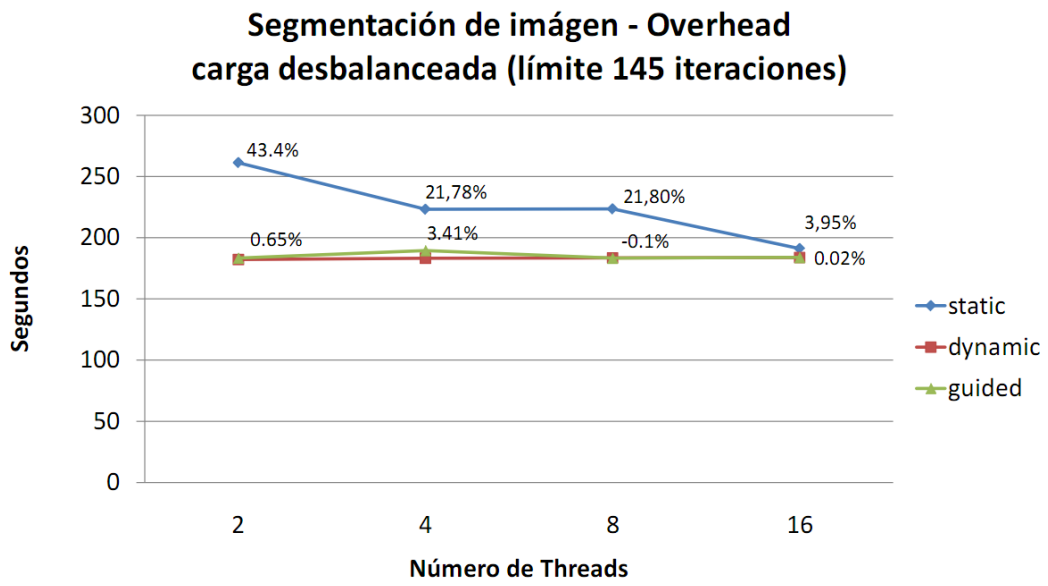


Figura 5.6: *Detalle de los tiempos de ejecución paralelos de la segmentación de IRM modificado. Se muestran los overhead de las políticas guided y static, respecto a la planificación dinámica.*

encargan de ejecutar las iteraciones más pesadas, afectando al rendimiento final de la aplicación. Sin embargo, cuando el número de threads aumenta hasta 8, la granularidad de la carga consigue repartir entre diferentes threads las cargas pesadas, obteniendo una adaptación y aproximando la ejecución al tiempo ideal.

Si mostramos el detalle de la ejecución paralela, descartando el tiempo serie y evaluando los overheads, obtenemos la gráfica 5.6. En esta gráfica podemos observar que el overhead de la planificación estática para 2 threads es de un 43 %, siendo también significativa para 4 y 8 threads, con 21,78 % y 21,80 % respectivamente.

La planificación dinámica obtiene una ejecución prácticamente constante al aumentar el número de threads, minimizando el overhead debido a la utilización del gestor de planificaciones, como nos recuerda la ley de Gustafson, el cómputo secuencial decrece cuando el tamaño del problema aumenta.

La planificación guiada no obtiene una mejora significativa respecto a la planificación dinámica. Únicamente para 8 thread obtiene un mejor tiempo de ejecución.

En este punto se ha observado cómo la gestión de la planificación es un factor a tener en cuenta, ya que afecta al rendimiento de la aplicación, pese a que la planificación dinámica, utilizada como planificación por defecto en la implementación de GNU, obtiene la mejor adaptación para las diferentes cargas evaluadas. Por tanto, aunque no se ha encontrado una carga de trabajo que jus-

tifique la necesidad de sintonizar la política de planificación, consideramos que es un factor a tener en cuenta a la hora de desarrollar una aplicación, ya que éstas u otras políticas de planificación pueden determinar el rendimiento de la aplicación.

### 5.3.2. Acceso a memoria

Este experimento evalúa las primitivas OpenMP para el control de acceso a memoria. Éstas afectan al rendimiento de las aplicaciones, debido a que generan una sobrecarga en el acceso concurrente a memoria compartida. Los threads que intentan acceder a datos compartidos deben tener garantizado el acceso exclusivo, mediante técnicas de exclusión mutua, para evitar problemas en la integridad de los datos. Sin embargo, la exclusión mutua puede generar problemas de interbloqueo e inanición, que el programador debe ser capaz de evitar. Las primitivas OpenMP para el acceso a memoria se basan en la definición de secciones críticas mediante la directiva *critical*, la sentencia *atomic* que garantiza la atomicidad de la operación, y *reduction*, que protege la escritura en memoria al finalizar la región paralela.

Las diferencia entre estas primitivas residen en la flexibilidad de las directivas para la definición de las secciones críticas, y la sobrecarga que cada una de ellas aporta. Esta relación determina la utilización de una primitiva u otra, aunque la base de todas ellas es la misma, garantizar el acceso exclusivo de un thread a una región de memoria.

La primitiva *critical* permite la definición de una sección de código que va a ser exclusiva, por tanto, favorece la flexibilidad de la región crítica. Esto significa que cuando un thread entra en una sección crítica, el resto de threads que intenten entrar deberán cambiar su estado de ejecución, esperando la liberación del recurso, por tanto, estos threads permanecerán ociosos. La primitiva *atomic* permite definir la sección critica a nivel de operación. Por tanto, la flexibilidad de esta operación es menor, ya que únicamente definirá la operación de atomicidad sobre la operación situada inmediatamente después de la definición *atomic*. Por último, el elemento *reduction* define una variable compartida y el tipo de operación que se va a ejecutar con ella. Por tanto, conociendo la variable y la operación que se va a realizar sobre ésta, el precompilador es capaz de crear una copia local en cada thread, con el valor inicial de la variable anterior a la definición de la sección crítica, y únicamente al finalizar la región paralela, la posición de memoria original modificará su contenido, recolectando los datos parciales calculados por cada thread, y aplicando la operación de actualización sobre la variable original. El comportamiento de estas herramientas de sincronización en el acceso a memoria, están representadas en el esquema 5.7

Este experimento ha evaluado el acceso concurrente a memoria compartida con las primitivas

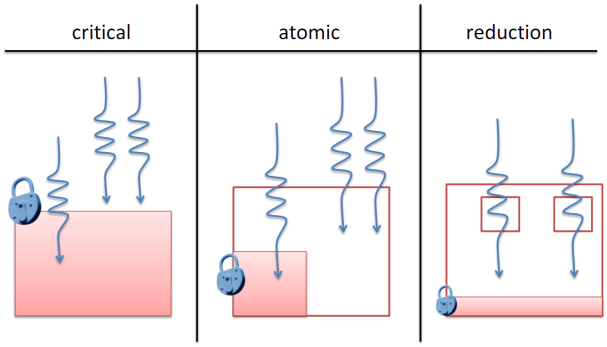


Figura 5.7: *Representación esquemática de los tipos de herramientas de sincronización en el acceso definidos en OpenMP.*

de sincronización de acceso a memoria definidas en OpenMP. Se ha evaluado la degradación en el rendimiento. La aplicación evaluada es la multiplicación de matrices, donde la escritura sobre la matriz de salida debe garantizar que únicamente un thread acceda a la vez, y así evitar sobreescrituras.

En la figura 5.8 tenemos la ejecución de la aplicación utilizando las diferente herramientas de acceso a memoria compartida. Se han evaluado entre ellas y respecto a la ejecución serie (línea continua). El primer aspecto que se debe tener en cuenta, es la ejecución serie. Se puede observar que en la experimentación con un único thread, el tiempo de ejecución entre las diferentes primitivas varía. Esto significa que existe un overhead significativo en la utilización de las primitivas. Únicamente la definición *reduction* consigue poca intrusión, teniendo un tiempo de ejecución similar a la ejecución serie, a diferencia de las primitivas *critical* y *reduction*.

En cuanto a la ejecución paralela, se observa que la primitiva *critical* no consigue escalar y genera una degradación sostenida en el rendimiento de la aplicación. A diferencia de ésta, las primitivas *atomic* y *serie* consiguen aumentar el rendimiento en su ejecución paralela. No obstante al comparar estos tiempos se observa que *atomic* consigue un peor tiempo respecto a la ejecución serie. La última primitiva evaluada, la definición *reduction*, consigue escalar al aumentar el número de cores, y además obtiene mejor rendimiento que la ejecución serie.

Para evaluar con más detalle el comportamiento de la primitiva *reduction*, en 5.9 se compara el tiempo de ejecución paralelo con diversas métricas. La linea de referencia a la ejecución serie nos indica que la ejecución paralela obtiene un aumento en el rendimiento. Respecto a esta ejecución se ha calculado la ejecución ideal, que debería decrementar linealmente, sin embargo, los experimentos han sido ejecutados en un entorno con 2 cores, por tanto la escalabilidad se satura al alcanzar el número de procesadores disponible. Como se ha visto anteriormente, la utilización de

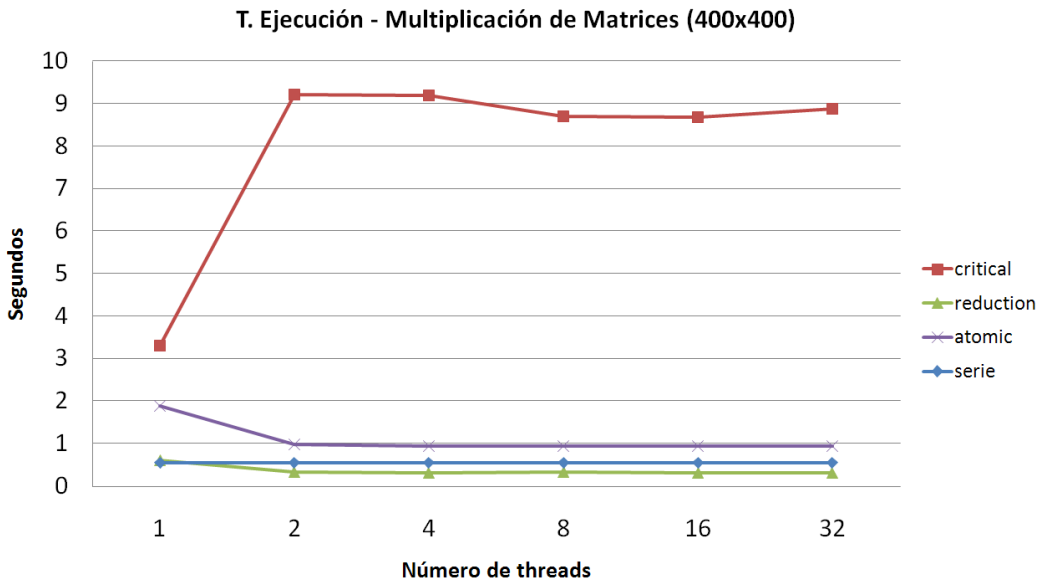


Figura 5.8: *Tiempo de ejecución para la multiplicación de matrices, en una arquitectura de dos cores*

OpenMP genera una intrusión en el código fuente original y por tanto un overhead. Se ha medido el overhead en la utilización de OpenMP mediante la linea de tendencia especulativa. Esta ha sido implementada eliminando la definición de la sección crítica dentro del código, y permitiendo race conditions. Aplicando la diferencia entre el tiempo paralelo de la primitiva reduction, y el tiempo paralelo de la ejecución especulativa obtenemos el overhead debido al acceso a memoria compartida, que en el caso de *reduction* es un overhead reducido, ya que esta operación minimiza el número de accesos a memoria creando copias locales a los threads y actualizando la variable original al final de la sección crítica.

### 5.3.3. Localidad de los datos

El experimento de localidad de los datos tiene como objetivo reflejar el impacto del acceso a los datos mediante la utilización de recursos compartido, las memorias cache multinivel. Se debe detectar el patrón de acceso a memoria de una aplicación para adaptarla a la arquitectura definida en el marco de trabajo. La aplicación analizada es la multiplicación de matrices por bloques. Para la consecución de este objetivo, se ha ejecutado una batería de experimentos para diferentes tamaños de bloque, de tamaños más grandes a tamaños más pequeños, buscando empíricamente el tamaño ideal que ajusta los datos a la estructura de niveles de caché. Para validar el experimento se ha utilizado la herramienta de profiling *ompp* integrada con PAPI, que permite la obtención

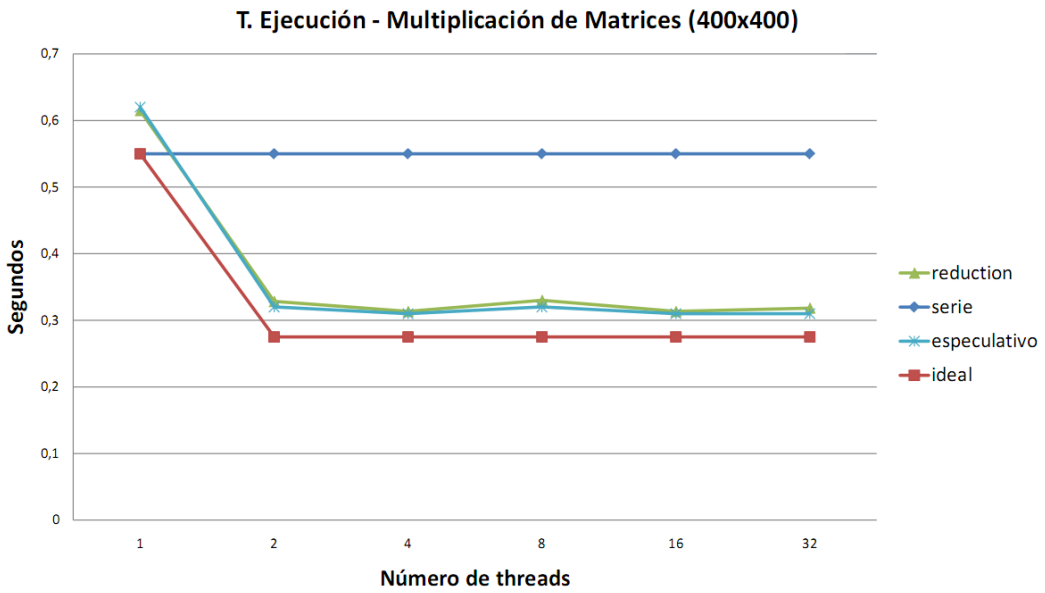


Figura 5.9: *Detalle de tiempo de ejecución para la multiplicación de matrices, de la directiva reduction.*

de contadores hardware y tiempos de ejecución por thread. Los valores proporcionados por PAPI consisten en contadores reales, integrados en el procesador, y que mediante PAPI es posible consultar para evaluar el comportamiento de la aplicación. Los contadores hardware utilizados son *L1_ICM* que proporciona el número de fallos de caché para la unidad de memoria caché L1 de instrucciones, *L1_DC* permite obtener los fallos de cache L1 y *L2_TCM* proporciona el número de fallos de caché de la memoria L2 unificada.

En la gráfica 5.10 se representan los tiempos de ejecución para este experimento. Se puede observar cómo se consigue paralelizar la aplicación para dos threads. El tiempo de ejecución se mantiene constante, hasta llegar a un tamaño de bloque de 8 x 8 elementos. En este punto, la localidad de los datos está afectando al rendimiento de la aplicación.

Para determinar cómo está afectando la localidad de los datos, a la memoria cache, se ha generado una traza que nos han permitido comparar el número de fallos de caché para los diferentes tamaños de bloque, en 5.11

Analizando los resultados, y conociendo que el entorno del marco de trabajo se dispone de una arquitectura de L1 de datos/instrucción de 256KB, una caché de L2 con 2MB. Ambas tiene un tamaño de línea de 64B (16 elementos float por línea), 8 vías con una política de asignación basada en asociatividad por conjuntos. En esta gráfica podemos observar que los fallos de cache aumentan exponencialmente a partir de un tamaño de bloque de 8KB. El motivo que se puede apreciar, es

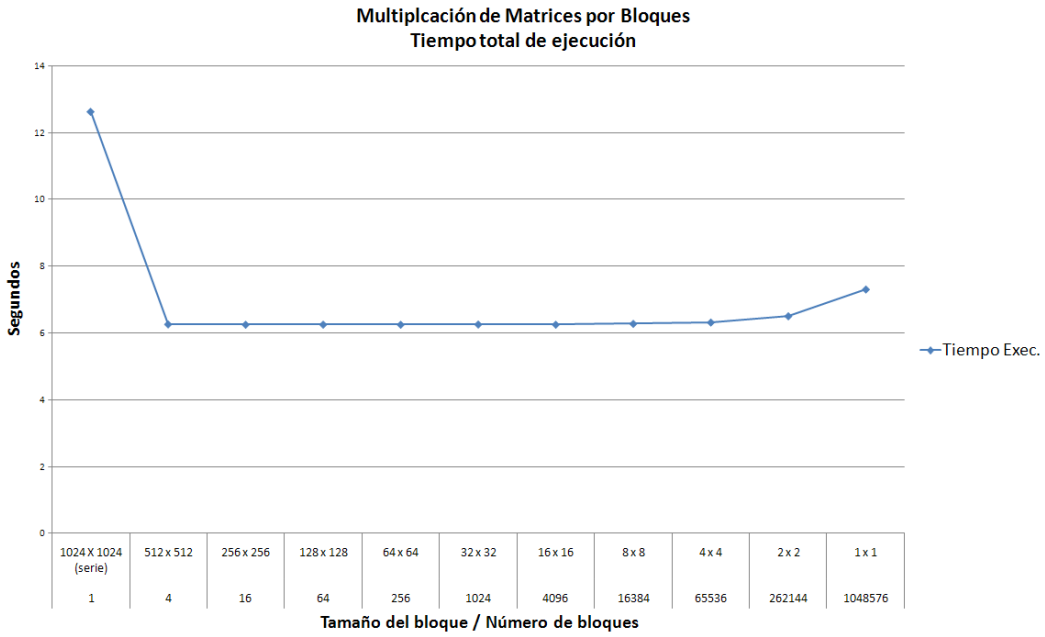


Figura 5.10: Tiempo de ejecución de la multiplicación de matrices por bloques, para una matriz de 1024 x 1024

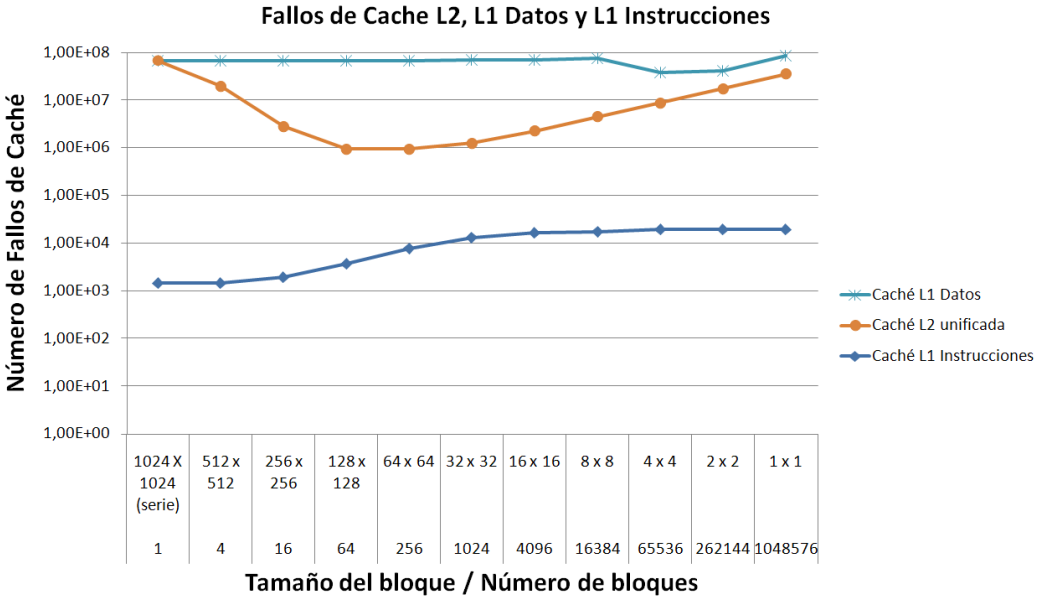


Figura 5.11: Número de fallos de caché para diferentes particiones de la matriz.

que no se está utilizando adecuadamente la cache en los diferentes niveles de memoria, que tienen la característica de tener menos latencia cuanto más cerca del procesador se encuentra. Por tanto aunque para bloques 4x4 los fallos en L1 disminuye, en L2 los fallos aumentan, y como se ha visto en 5.10, a partir de este punto la aplicación empieza a aumentar el tiempo de computo.

A partir de los resultados anteriores se rediseñó la aplicación para minimizar el número de fallos, teniendo en cuenta tanto la caché L1 como a la L2. Se ha dividido la matriz en 16x16 elementos, de manera que cada bloque tiene el mismo tamaño que las vias de la memoria caché L2, 256KB que contienen 64 elementos de la matriz. Para minimizar el acceso a la cache L1, se recorren los elementos de cada bloque, en sub-bloques, almacenando en la caché L1 lineas de 64 elementos. De esta forma se minimizan los fallos respecto a la ejecución anterior. En la tabla 5.1 se muestran los resultados obtenidos para esta ejecución sintonizada, comparada con con la ejecución correspondiente al mismo tamaño de bloque. Podemos observar como mejor el rendimiento, consiguiendo una ganancia de 6X, debido al efecto de minimizar los fallos de cache. Teniendo en cuenta que la ganancia respecto a la ejecución serie es de 12X.

Cuadro 5.1: Comparativa entre multiplicación por bloques y multiplicación por bloques optimizada a caché.

| **Tam. Bloque** | **Tiempo Ejec.** | **Ciclos** | **Cache L1 Instr.** | **Cache L1 Datos** | **Cache L2** |
|---|---|---|---|---|---|
| 64 x 64 | 6,249 seg | 1,62E+10 | 3,86E+03 | 3,40E+07 | 4,79E+05 |
| 64 x 64 ajustado. | 0,956 seg | 2,42E+09 | 4,75E+02 | 1,10E+06 | 7,41E+04 |

En la gráfica 5.12 se observa la ganacia para los diferentes parámetros hardware medidos entre la versión adaptada a los dos niveles de cache respecto a la versión con bloques de 64x64 elementos, que se ajusta únicamente al primer nivel de cache. Con la estrategia de adaptación a dos niveles se ha obtenido una mejora en el rendimiento notable. Sin embargo, el proceso de sintonización de las aplicaciones a este nivel es muy complicado debido a la dependencia de la arquitectura hardware y el patrón de comportamiento de la aplicación, y por tanto, al involucrar estos dos factores, únicamente podrá ser aplicable una sintonización si los niveles de memoria corresponden a la granularidad de partición que considera el algoritmo.

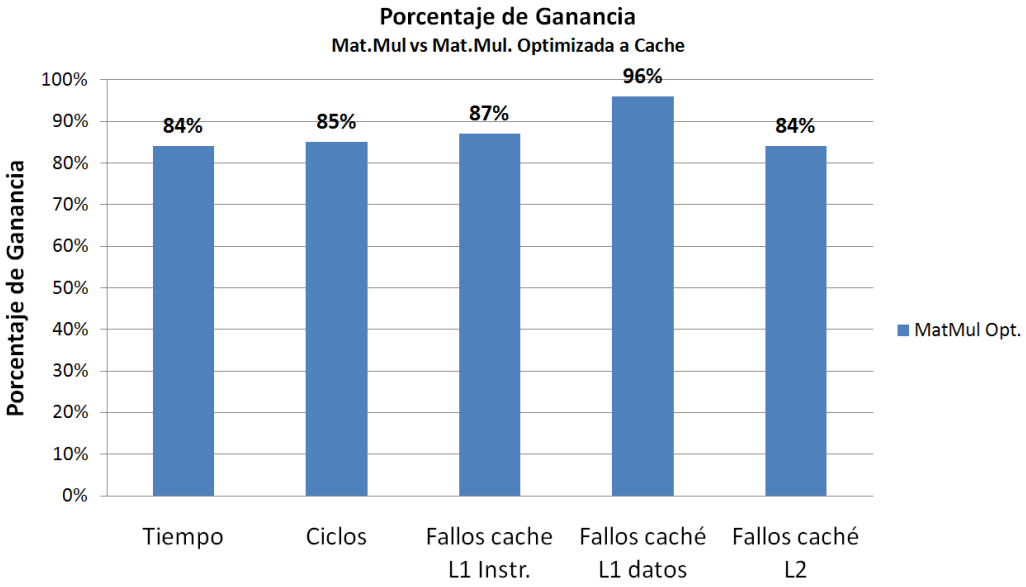


Figura 5.12: Ganancia con la versión adaptada a dos niveles de la jerarquía de caché, respecto a la versión adaptada al primer nivel de cache.

# Capítulo 6

# Conclusiones

El objetivo de esta tesina es sentar las bases de conocimiento que permitan definir un modelo de rendimiento para entornos multicore. Debido a la diversidad de sistemas multicore se ha definido un marco de trabajo acotado.

En este trabajo de investigación se han evaluado diversos factores de rendimiento para entornos multicore. El marco de trabajo acotado consiste en una arquitectura CMP (chip multiprocesor) de dos cores y un modelo de programación paralela que permite expresar el paralelismo de datos. El modelo de programación paralela analizado ha sido OpenMP que permite expresar el paralelismo de datos en entornos de memoria compartida. Se ha elegido OpenMP por ser un estándar de facto para estos entornos y porque es un modelo muy estudiado en el campo de HPC, gracias a la facilidad para expresar el paralelismo, además de ser un modelo en constante evolución. La versión más reciente del modelo OpenMP es capaz de implementar paralelismo funcional y paralelismo de datos. La versión utilizada en el marco de trabajo corresponde a la versión 2.5. De las numerosas implementaciones OpenMP se ha estudiado la desarrollada por GNU e integrada por el conjunto del compilador GCC y la librería de tiempo de ejecución libgomp.

El modelo OpenMP gestiona el paralelismo mediante la abstracción de la librería de threads, que necesita de una definición explícita del paralelismo. De esta manera, OpenMP permite al programador centrarse en expresar el paralelismo de la aplicación. Al estar íntimamente relacionado con la librería de threads, durante el análisis se ha estudiado la librería de threads, utilizada por libgomp para la creación y gestión de threads. La implementación NPTL (Native POSIX Threads Library), desarrollada por RedHat, integrada en los entornos Linux ha sido la implementación de threads analizada, que proporciona algunas extensiones respecto al estándar, como por ejemplo la definición de afinidad entre threads y cores. Además, esta implementación cumple con el estándar POSIX, que define una interfaz de programación de threads para mejorar la portabilidad

de aplicaciones entre entornos Unix-Like.

Definido el marco de trabajo se han analizado los factores de rendimiento del modelo OpenMP y de la librería de threads, algunos de los cuales no ha sido posible evaluar: por ejemplo el soporte en el kernel de Linux para modelos de thread N:M, que permiten definir los threads sobre dos unidades de planificación, a nivel del sistema operativo y a nivel de la capa de usuario. Alguna de las ventajas de ésta última es, la baja latencia en la creación y cambio de contexto entre threads.

Analizados los modelos de programación y arquitectura se han definido varios factores que afectan al rendimiento de aplicaciones multicore para el modelo OpenMP:

- *Políticas de planificación* que proporciona OpenMP para gestionar la repartición de carga de trabajo entre threads que intervienen en una región paralela. La utilización de diferentes políticas ha demostrado una variabilidad en el rendimiento, sin embargo, la planificación dinámica, que es la definida por defecto, siempre ha obtenido un mejor rendimiento para las cargas de trabajo aplicadas.

- *Herramientas de sincronización*, debido a que en entornos de memoria compartida el acceso a elementos compartidos se gestiona mediante ejecuciones concurrentes. El tiempo de concurrencia se debe minimizar para mejorar las prestaciones. De los mecanismo de sincronización evaluados, se ha visto que *reduction* ofrece el mejor rendimiento, con un overhead mínimo. El resto de mecanismos estudiados, no han ofrecido mejores resultados que la ejecución serie. Sin embargo, los elemento de sincronización permiten diferente flexibilidad en la definición de secciones críticas, y no siempre se va a poder utilizar el mecanismo *reduction*. Cabe destacar que una mejora de estos mecanismos proporcionaría una mejora en el rendimiento de las aplicaciones.

- *Localidad de los datos*, afecta al rendimiento debido a que las arquitecturas jerárquicas multinivel existenten en arquitecturas multicore se benefician de la localidad de los datos, tanto espacial como temporal, y que afectan al rendimiento de la aplicación en entornos multicore. Esta estrategia es la más difícil de sintonizar pero és la que obtiene una mejora en el rendimiento considerable.

## 6.1. Trabajo Futuro

Con la base de conocimiento adquirida en esta tesina, se plantean unas líneas abierta, de cara a la continuidad de la investigación.

- Se debe continuar estudiando el impacto de los factores que afectan al rendimiento
- Estudiar la posibilidad de alterar parámetros de la aplicación de forma dinámica para mejorar su rendimiento
- Estudiar los factores que será necesario monitorizar para detectar los problemas de rendimiento

# Bibliografía


[1] G. E. Moore, "Cramming more components onto integrated circuits," pp. 56–59, 2000.

[2] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, (New York, NY, USA), pp. 483–485, ACM, 1967.

[3] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, 1988.

[4] "Top 500 lista, june 2010." http://www.top500.org/lists/2010/06.

[5] "Jaguar at oak ridge national laboratory. opteron six core 2.6 ghz." http://www.nccs.gov/computing-resources/jaguar/.

[6] "Nebulae at site national supercomputing centre in shenzhen (nscs). intel em64t xeon x56xx (westmere-ep) 2660 mhz (10.64 gflops) 6 cores and nvidia tesla c2050 gpu." http://www.top500.org/system/10484.

[7] "Roadrunner at doe/nnsa/lanl. powerxcell 8i 3.2 ghz / opteron dc 1.8 ghz." http://www.lanl.gov/orgs/hpc/roadrunner/index.shtml.

[8] "Kraken at national institute for computational sciences/university of tennessee. opteron six core 2.6 ghz." http://www.nics.tennessee.edu/computing-resources/kraken.

[9] "Earth simulator at japan agency for marine -earth science and technology. nec 3200 mhz (102.4 gflops)." http://www.jamstec.go.jp/es/en/index.html.

[10] M. J. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, pp. 948 –960, sept. 1972.

[11] T. E. Anderson, B.Ñ. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: effective kernel support for the user-level management of parallelism," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 53–79, 1992.

[12] "The nanos group site, project nanos4." http://nanos.ac.upc.edu/content/nanos4.

[13] "Labri laboratoire bordelais de recherche en informatique, inria." http://runtime.bordeaux.inria.fr/marcel/.

[14] A. A. Nathan R. Fredrickson and Y. Qian., "Performance characteristics of openmp constructs, and application benchmarks on a large symmetric multiprocessor.," *In Proceedings of the 2003 International Conference on Supercomputing, pages 140-149, New York, June 23-26 2003.*

[15] "The epcc microbenchmarks page." http://www.epcc.ed.ac.uk/research/openmpbench/.

[16] "The nas benchmarks page." http://www.nas.nasa.gov/Software/NPB/.

[17] "The spec omp2001 benchmark suite." http://www.spec.org/hpg/omp2001/.

[18] G. Jost, H. Jin, D. A. Mey, and F. F. Hatay, "Comparing the openmp, mpi, and hybrid programming paradigms on an smp cluster," *Fifth European Workshop on OpenMP (EWOMP03) in Aachen, Germany, September 2003, http://www.nas.nasa.gov/News/Techreports/2003/PDF/nas-030019.pdf.*

[19] J. J. Costa, T. Cortes, X. Martorell, E. Ayguadé, and J. Labarta, "Running openmp applications efficiently on an everything-shared sdsm," pp. 35–42, 2004.

[20] J. P. Singh, W.-D. Weber, and A. Gupta, "Splash: Stanford parallel applications for shared-memory," *SIGARCH Comput. Archit. News*, vol. 20, no. 1, pp. 5–44, 1992.

[21] P. E. H. V. V. Dimakopoulos and G. C. Philos., "A microbenchmark study of openmp overheads under nested parallelism.," *Technical report, Department of Computer Science, University of Ioannina, Ioannia, Greece, 2008.*

[22] R.Ñanjegowda, O. Hernandez, B. Chapman, and H. H. Jin, "Scalability evaluation of barrier algorithms for openmp," (Berlin, Heidelberg), pp. 42–52, Springer-Verlag, 2009.

[23] "Computer science department, houston university." http://www2.cs.uh.edu/ openuh/.

[24] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[25] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," pp. 142 –151, sept. 2008.

[26] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "Cellss: a programming model for the cell be architecture," (New York, NY, USA), p. 86, ACM, 2006.

[27] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the starss programming model for platforms with multiple gpus," in *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, (Berlin, Heidelberg), pp. 851–862, Springer-Verlag, 2009.

[28] C. Bengtsson, M. Brorsson, H. Grahn, E. Hagersten, B. Jonsson, C. Kessler, B. Lisper, P. Stenström, and B. Svensson., "Multicore computing - the state of the art.," *SICS publications database (Sweden), available at http://eprints.sics.se/3546/01/SMIMulticoreReport-2008.pdf, accessed 3 October 2009, 2008.*

[29] L. V. Herlihy, M.P., "Distributed computing and the multicore revolution.," *ACM SIGACT News 39(1), 62-72 (2008)*.

[30] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Computing Surveys*, vol. 30, pp. 123–169, 1998.

[31] E. Cesar, A. Moreno, J. Sorribes, and E. Luque, "Modeling master/worker applications for automatic performance tuning," *Parallel Comput.*, vol. 32, no. 7, pp. 568–589, 2006.

[32] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, (New York, NY, USA), pp. 120–126, ACM, 1982.

[33] B. M. Luiz DeRose, "Profiling and tracing openmp applications with pomp based monitoring libraries," in *Euro-Par 2004 Parallel Processing*, pp. 39–46, 2004.

[34] M. G. Karl Fürlinger, "A profiling tool for openmp," in *OpenMP Shared Memory Parallel Programming*, pp. 15–23, 2008.

[35] J. Dongarra, J. London, S. Browne, S. Browne, J. Dongarra, N. Garner, N. Garner, K. London, P. Mucci, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications 2000 14: 189*, 2000.

[36] M. . P. B. . W. P. Geist, G.A. ; Heath, "A user's guide to picl a portable instrumented communication library," in *Oak Ridge National Lab., TN (USA)*, 1990.

[37] M. Heath and J. Etheridge, "Visualizing the performance of parallel programs," *Software, IEEE*, vol. 8, pp. 29 –39, sep 1991.

[38] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera, "Scalable performance analysis: The pablo performance analysis environment," in *In Proceedings of the Scalable parallel libraries conference*, pp. 104–113, IEEE Computer Society, 1993.

[39] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of mpi resources," *Supercomputer*, vol. 12, pp. 69–80, 1996.

[40] W. E. N. H.-C. H. Holger Brunst, Manuela Winkler, "Performance optimization for large scale computing: The scalable vampir approach," pp. 751–760, 2001.

[41] B. J. N. W.-E. A. D. B. B. M. Markus Geimer, Felix Wolf, "The scalasca performance toolset architecture," in *Concurrency and Computation: Practice and Experience*, pp. 702–719, 2010.

[42] E. K. Michael Gerndt, "Automatic memory access analysis with periscope," in *Computational Science – ICCS 2007*, pp. 847–854, 2007.

[43] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, pp. 287–331, 2006.

[44] I. Dorta, C. Leon, and C. Rodriguez, "Performance analysis of branch-and-bound skeletons," in *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*, p. 8 pp., 15-17 2006.

[45] F. Guirado, A. Ripoll, C. Roig, and E. Luque, "Performance prediction using an application-oriented mapping tool," in *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, pp. 184 – 191, 11-13 2004.

[46] E. L. Josep Jorba, Tomas Margalef, "Automatic performance analysis of message passing applications using the kappapi 2 tool," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 293–300, 2005.

[47] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T.Ñewhall, "The paradyn parallel performance measurement tool," *Computer*, vol. 28, pp. 37 –46, nov 1995.

[48] R. L. Ribler, H. Simitci, and D. A. Reed, "The autopilot performance-directed adaptive control system," *Future Generation Computer Systems*, vol. 18, no. 1, pp. 175 – 187, 2001.

[49] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: towards automated performance tuning," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, (Los Alamitos, CA, USA), pp. 1–11, IEEE Computer Society Press, 2002.

[50] M. Hussein, K. Mayes, M. Luján, and J. Gurd, "Adaptive performance control for distributed scientific coupled models," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, (New York, NY, USA), pp. 274–283, ACM, 2007.

[51] T. M. E. L. Anna Morajko, Oleg Morajko, "Mate: Dynamic performance tuning environment," *Euro-Par 2004 Parallel Processing*, pp. 98–107, 2004.

[52] A. Agarwal and M. Levy, "The kill rule for multicore," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pp. 750 –753, 4-8 2007.

[53] "Tilera." http://www.tilera.com.

[54] A. Leon, B. Langley, and J. L. Shin, "The ultrasparc t1 processor: Cmt reliability," in *Custom Integrated Circuits Conference, 2006. CICC '06. IEEE*, pp. 555 –562, 10-13 2006.

[55] "General-purpose computation on graphics hardware." http://gpgpu.org.

[56] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[57] J. A. Kahle, M.Ñ. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[58] "Intel core 2." http://www.intel.com/itcenter/products/core/core2/index.htm.

[59] "Amd phenom." http://www.amd.com/la/products/desktop/processors/phenom.

[60] "Intel core i7." http://www.intel.com/products/processor/corei7/.

[61] J. Merino, V. Puente, and J. Gregorio, "Esp-nuca: A low-cost adaptive non-uniform cache architecture," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1 –10, 9-14 2010.

[62] J. Psota and A. Agarwal, "rmpi: message passing on multicore processors with on-chip interconnect," in *HiPEAC'08: Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, (Berlin, Heidelberg), pp. 22–37, Springer-Verlag, 2008.

[63] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 453–464, IEEE Computer Society, 2008.

[64] "The posix specification (ieee std 1003.1,2004 edition)." http://www.unix.org/version3/ieee_std.html.

[65] F. Garcia and J. Fernandez, "Posix thread libraries," *Linux Journal, volume 2000*, p. 36.

[66] U. Drepper and I. Molnar, "The native posix thread library for linux.," *White Paper, Red Hat, 2003, http://people.redhat.com/drepper/nptl-design.pdf*.

[67] "The openmp specification, version 2.5.." http://www.openmp.org/mp-documents/spec25.pdf.

[68] "The openmp specification, version 3.0.." http://www.openmp.org/mp-documents/spec30.pdf.

[69] D.Ñovillo., "Openmp and automatic parallelization in gcc.," *In the Proceedings of the GCC Developers Summit, June 2006.*

[70] H. L. Chapman, B., "Enhacing openmp and its implementation for programming multicore systems.," *In Proceedings of the International Conference Parco 2007, pp. 3-18 (2007)*.

[71] G. M. Fuerlinger, K., "Analyzing overheads and scalability characteristics of openmp applications.," *Dayde, R. (ed.) VECPAR 2006, Rio de Janeiro, Brasil, vol. 4395, pp. 39-51. Springer, Heidelberg (2006)*.

[72] J. Ashburner, *Computational Neuroanatomy*. PhD thesis, University College London, 2000. http://www.fil.ion.ucl.ac.uk/spm/doc/theses/john/.

[73] J. Ashburner and K. Friston, "Image segmentation," in *Human Brain Function* (R. Frackowiak, K. Friston, C. Frith, R. Dolan, K. Friston, C. Price, S. Zeki, J. Ashburner, and W. Penny, eds.), Academic Press, 2nd ed., 2003. http://www.fil.ion.ucl.ac.uk/spm/doc/theses/john/chapter5.pdf.