UNIVERSIDAD DE MURCIA

DEPARTAMENTO DE INFORMÁTICA Y SISTEMAS

TESIS DOCTORAL

Técnicas de Modelado y Optimización del Tiempo de Ejecución de Rutinas Paralelas de Álgebra Lineal

Luis Pedro García González

2012

Directores:
Dr. Antonio Javier Cuenca Muñoz
Dr. Domingo Giménez Cánovas

A Mari Luz

Tanto si piensas que puedes,
como si piensas que no puedes,
estás en lo cierto

Henry Ford

Agradecimientos

A Domingo y a Javier por brindarme la oportunidad de trabajar con ellos, por la supervisión de esta tesis, por su paciencia, confianza y dedicación, y por los conocimientos compartidos sin los cuales habría sido imposible lograr la consecución de este trabajo. Ha sido un auténtico placer trabajar con vosotros y espero poder seguir haciéndolo en un futuro.

A Blas, Isidro y Lola por ser algo más que compañeros de trabajo, por su apoyo, por sus palabras de ánimos, y por esos momentos de tertulia disfrutados. A Paquita, mi buena amiga, le agradezco los buenos consejos que siempre me ha dado sin tener que pedírselos. A Alberto por preguntar de lo que iban mis trabajos de doctorado, aunque nada tuvieran que ver con los suyos. También aprovecho para agradecer a los compañeros con los que he tenido y tengo la suerte de trabajar en el SAIT, el interés mostrado por el transcurso de este trabajo, y por hacer que el día a día resulte más llevadero.

A mis padres, Luis y Pilar, por estar ahí cuando los he necesitado, por poner todos los medios a su alcance para que estudiara y por los valores que me han transmitido. No quiero dejar pasar la ocasión de agradecer a mi hermana Belén, a mis abuelos y a mis tíos el cariño y afecto demostrados.

A Mari Luz, por lo que ya hemos compartido, y por lo que nos queda por compartir. Esta tesis no habría sido posible sin ti, por lo que, en parte, también es tuya. Y a Pedro, mi hijo, por esa alegría que transmite y por esos momentos en los que, si se presentaba la ocasión, intentaba "ayudarme" con el ordenador.

Sinceramente, muchas gracias a todos.

Luis Pedro García González

Resumen

En este trabajo se describe una metodología de modelado del tiempo de ejecución de rutinas de álgebra lineal implementadas mediante los paradigmas de paso de mensajes y de memoria compartida, que refleja las características del software y del hardware de la plataforma en la que se ejecuta la rutina y que permite abordar su ajuste automático.

El modelo de tiempo de ejecución se construye a partir del estudio teórico de complejidad de la rutina, cuando se dispone de información sobre el algoritmo implementado, o a partir de funciones matemáticas en las que aparecen el tamaño del problema y los parámetros ajustables de las rutinas de álgebra lineal. En ambos casos el modelo es de aplicación en la selección de diferentes ajustes, tales como el número de procesadores a utilizar, el tamaño del bloque de cálculo, la selección de la mejor librería de entre las disponibles o de la selección del mejor algoritmo con el que resolver un problema.

En plataformas heterogéneas se propone una metodología que permita ejecutar el software desarrollado para plataformas homogéneas de forma óptima, introduciendo en el proceso de selección la asignación de procesos a procesadores. Se utilizan técnicas heurísticas, junto con el modelo que aproxima el tiempo de ejecución de la rutina, en la búsqueda de una solución a la mejor selección de procesos, asignación de procesos a procesadores, y selección de los parámetros ajustables de la rutina homogénea.

En rutinas de álgebra lineal que utilicen el paradigma de programación de memoria compartida, se considera también como parámetro ajustable la versión del ejecutable de la rutina generada por los compiladores disponibles en la plataforma multicore. El modelo de tiempo de ejecución proporciona información sobre la mejor selección de los parámetros ajustables de la rutina y sobre la versión compilada que se ejecutará en el menor tiempo.

Abstract

This work describes a methodology for modelling the execution time of linear algebra routines implemented with the programing paradigms of message-passing and shared memory. The methodology reflects the characteristics of the software and the hardware in the execution platform, and in this way the automatic tuning of the routine is achieved.

A theoretical model of the execution time of the routine is built when information about the algorithm is available. If this information is unavailable a model is obtained experimentally as a function of the problem size and the adjustable parameters in the routine. In both cases, the model is aplicable for selection of differents parameters, such as the number of processors to be used, the block size, the best basic library or the best available algorithm to solve a specific problem.

We propose a methodology to run efficiently on heterogeneous platforms the software designed for homogeneous platforms. The mapping of processes to processors is included in the selection process. To select the best number of processes, the processes to processors mapping and the selection of the adjustable parameters of the homogeneous routine, an heuristic approach is considered in combination with the theoretical model of the execution time.

An additional parameter is considered for shared-memory linear algebra routines: the executable version of the routine generated by the different available compilers on the multicore platform. The model provides information about the best selection of the adjustable parameters of the routine and about the compiled version with which the problem is solved in the shortest time.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

1.1 Introducción

La complejidad y el tamaño de los cálculos requeridos por las diferentes ramas de las ciencias e ingenierías crece al mismo ritmo con el que aparecen mejoras en la tecnología de los procesadores. La simulación numérica de problemas en ciencias e ingeniería demanda constantemente recursos computacionales, al precisar la resolución de problemas de mayor tamaño, con más exactitud y en menor tiempo. Por tanto, es preciso optimizar las aplicaciones que utilizan los científicos e ingenieros en la medida de lo posible, con el fin de obtener rendimientos cercanos al pico teórico ofrecido por el sistema en el que se ejecutan. Por otra parte, el trabajo necesario para optimizar un código real puede durar semanas e incluso meses y requiere de conocimientos en varias disciplinas, como son la arquitectura de computadores, las herramientas de programación y depuración, el análisis numérico, y el software matemático. Se trata de un procedimiento muy exigente que implica un elevado tiempo de desarrollo y que requiere amplios conocimientos del código a optimizar y de la plataforma en la que se ejecutará. Además, la labor de optimización realizada a medida para una plataforma concreta no tiene por qué ser válida, en principio, para otras plataformas de manera genérica. Este entorno de trabajo tan diverso ha obligado a cambiar la forma en la que tradicionalmente se viene optimizando el software para cálculo científico, con el fin de poder seguir el ritmo marcado por las necesidades de los usuarios y por la aparición de nuevo hardware. Dentro de este contexto surgen las técnicas de optimización automática como una herramienta valiosa que proporciona al software de computación científica cierta capacidad de adaptación al entorno de ejecución, pudiendo generar automáticamente un código optimizado para cada plataforma concreta por medio de la utilización de modelos y de la experimentación en la plataforma. Algunos ejemplos de proyectos en los que se han aplicado con éxito técnicas de optimización automática son: ATLAS [WPD01] para la generación de núcleos computacionales optimizados de álgebra lineal densa; SPARSITY [IYV04], donde se generan implementaciones optimizadas de núcleos para cálculos con matrices dispersas; FFTW

[FJ05], implementación adaptativa y optimizada de la transformada discreta de Fourier; y ABCLib_DRSSED [KKHY06], implementación de rutinas con optimización automática para la obtención de autovalores. La optimización automática del software es, por tanto, un campo de investigación activo y de interés científico, objeto de un *workshop* monográfico que se celebra anualmente: *International Workshop on Automatic Performance Tuning (iWAPT)* [iWA].

En esta tesis se aborda el problema del modelado del tiempo de ejecución de software de álgebra lineal densa en plataformas paralelas con el fin de disponer de este modelo como una herramienta que permita desarrollar técnicas de optimización automática del código que modela. Dado que el comportamiento de una aplicación está supeditado al sistema de cómputo en el que se ejecuta, el modelo de tiempo de ejecución deberá contener un modelo de la arquitectura de cómputo. Este modelo de la arquitectura estará conformado por un conjunto de parámetros que describen las características de la plataforma de cómputo y su influencia en el comportamiento del software de álgebra lineal, y que junto a los parámetros ajustables de la rutina (tamaño de bloque de cálculo, número de procesadores a utilizar y su topología lógica...) constituyen su modelo de tiempo de ejecución.

La metodología general propuesta para abordar la optimización automática de código paralelo de álgebra lineal, consiste en modelar el tiempo de ejecución de las rutinas y, a través de la experimentación en una plataforma paralela concreta, seleccionar automáticamente los parámetros ajustables de la rutina. También se aplica en la selección de un algoritmo entre varios candidatos, de manera que, para un tamaño de problema y plataforma, resuelve el problema en menor tiempo, así como en la selección de la mejor librería a utilizar que implemente las operaciones básicas utilizadas en la rutina entre las varias disponibles en la plataforma paralela. El modelo de tiempo de ejecución de la rutina se obtendrá de forma completamente analítica cuando se disponga de información sobre el algoritmo implementado, y en los casos en los que no se disponga de dicha información el modelo se obtendrá a partir de funciones polinómicas que aproximen el tiempo de ejecución. Adicionalmente, si con el modelo construido de forma analítica no se obtiene información de los parámetros ajustables de la rutina con la que el tiempo de ejecución es cercano al óptimo, se propone la utilización de funciones polinómicas para la generación de un nuevo modelo de tiempo de ejecución que permita la obtención de los parámetros ajustables.

1.2 Objetivos

El objetivo global de esta tesis se enmarca dentro del campo de investigación en modelado y optimización automática de software de álgebra lineal. Nuestro propósito es el de proporcionar una metodología que, por medio de modelos de tiempo de ejecución de las rutinas, permita aliviar el trabajo de optimización a los desarrolladores de este tipo de software, y que facilite su utilización al usuario, de tal forma que el tiempo de ejecución de este software no quede condicionado a los conocimientos que el usuario final tenga sobre la plataforma paralela, so-

bre el conjunto de librerías básicas de computación y comunicación disponibles, o sobre las características de generación de código paralelo de los compiladores disponibles.

1.2.1 Objetivos específicos

El objetivo global planteado se puede desglosar en los siguientes objetivos específicos:

- Desarrollar una metodología para la obtención de modelos de tiempo de ejecución de rutinas de álgebra lineal que sea de aplicación en diferentes plataformas paralelas homogéneas (arquitecturas de memoria compartida, arquitecturas de memoria distribuida y combinaciones de ambas), que permita reflejar los diferentes costes asociados a las operaciones básicas de álgebra lineal y primitivas de comunicación por paso de mensajes en función del algoritmo, del tamaño del problema en el que se utilizan o del mecanismo de comunicación empleado. Como se verá más adelante, es necesario tener en cuenta todos estos factores con el fin de que el modelo refleje las características del software y del hardware de la plataforma paralela, y por tanto sea de aplicación en la obtención de software con capacidad de optimización automática. Para la obtención de los modelos de tiempo de ejecución se han considerado dos métodos:
  - Realizar un estudio teórico de la complejidad de la rutina que nos permita obtener el modelo analítico de su tiempo de ejecución como una función del tamaño del problema a resolver, de los costes de las operaciones aritméticas del álgebra lineal, de los costes de las operaciones de comunicación y de los parámetros a seleccionar en tiempo de ejecución.
  - Emplear funciones polinómicas que aproximen el tiempo de ejecución de la rutina y que serán construidas a partir del modelo teórico o a partir de combinaciones del tamaño del problema y de los parámetros a seleccionar en tiempo de ejecución. En esta segunda aproximación se incluirán diferentes técnicas para la obtención de los coeficientes del polinomio con el fin de llegar a un compromiso entre la exactitud con la que el modelo aproxima el tiempo de ejecución y el tiempo de obtención del mismo.
- Ampliar el ámbito de aplicación de la metodología propuesta inicialmente para sistemas homogéneos a sistemas heterogéneos. El objetivo es obtener software eficiente de álgebra lineal para sistemas heterogéneos, aprovechando las versiones homogéneas de las rutinas básicas y evitando así la necesidad de volver a escribir el código de las mismas. Se plantea un esquema en el que se tomarán automáticamente un conjunto de decisiones (número de procesos a generar, la distribución heterogénea en la red de procesadores...) que permitirán que el software se ejecute en un tiempo cercano al óptimo. Con el fin de reducir el espacio de búsqueda de las decisiones a tomar de forma automática, se propone la utilización de diferentes técnicas heurísticas.

- Plantear una metodología que permita, mediante la aplicación de modelos de tiempo de ejecución, la selección del compilador que genera el código más eficiente para rutinas paralelas de álgebra lineal en sistemas con varios núcleos (*multicore*). Para alcanzar este objetivo se propone realizar una serie de experimentos que permitan cuantificar las capacidades del compilador para generar primitivas en una plataforma paralela. De esta manera se puede realizar una selección del código generado por el compilador que resulte más óptimo para un tamaño de problema y algoritmo en particular.

1.3 Metodología

En esta sección se describe la metodología utilizada a lo largo de este trabajo y que viene determinada por los objetivos que se han descrito en la sección anterior.

Se parte de un modelo general de arquitectura de cómputo para sistemas paralelos, con P procesadores conectados entre sí (veáse la sección 2.3). Tomando como referencia dicho modelo de sistema, se obtiene un modelo de tiempo de ejecución de la rutina de álgebra lineal. El modelo de tiempo de ejecución puede construirse a partir del estudio teórico del algoritmo implementado en la rutina o mediante funciones polinómicas. En ambos casos aparecerán un conjunto de parámetros algorítmicos, cuyos valores se seleccionarán de forma automática con el propósito de obtener un tiempo de ejecución de la rutina lo más óptimo posible. En el caso del modelo obtenido del estudio teórico de complejidad de la rutina de álgebra lineal, aparecerá un conjunto de parámetros del sistema cuyo valor se aproximará mediante experimentación, y en el caso del modelo obtenido mediante funciones polinomiales, aparecerán una serie de coeficientes que se aproximarán mediante experimentación directa sobre la plataforma paralela y aplicando técnicas de regresión lineal.

A continuación se presentan de forma más extensa las fases que conforman la metodología propuesta y que conducirán a la obtención de modelos de tiempo de ejecución con los que se pueden desarrollar técnicas de optimización automática de rutinas paralelas de álgebra lineal:

- **Fase de diseño**: se realiza la implementación de la rutina y se construye su modelo analítico del tiempo de ejecución, en el que quedarán reflejados los costes de las operaciones aritméticas, los costes de las primitivas del paradigma de programación paralela empleado (paso de mensajes o memoria compartida), y aquellos parámetros cuyos valores hay que seleccionar en el momento de ejecución de la rutina.

- **Fase de instalación**: se realiza la recogida de información acerca de cada uno de los parámetros del sistema. Si la rutina pertenece a los niveles inferiores de la jerarquía de librerías de álgebra lineal, la recogida de información se hará mediante experimentación directa. De lo contrario se solicitará esa información a las rutinas de niveles inferiores que la rutina a modelar invoca en su código. En esta fase, si no se dispone de información sobre

la rutina o la información disponible no conduce a tiempos de ejecución cercanos al óptimo, se construirá una familia de polinomios cuyos coeficientes se aproximarán mediante experimentación.

- **Fase de ejecución**: se utilizará el modelo de tiempo de ejecución para obtener los valores de los parámetros ajustables de la rutina con los que se consigue el menor tiempo teórico de ejecución. Una vez seleccionados los valores de estos parámetros se invocará a la rutina usando esos valores para los parámetros ajustables.

Es necesario realizar algunas consideraciones sobre las fases de diseño y de instalación, en las que se obtiene el modelo de tiempo de ejecución y se comprueba la bondad del modelo:

- En relación a la construcción del modelo: es conveniente analizar la posible dependencia de los costes de las operaciones aritméticas y de las operaciones de comunicación respecto al tamaño del problema al valor de los parámetros ajustables, y obtener un modelo que refleje esa dependencia.
- En relación a la recogida de información: en la fase de instalación se requiere obtener el valor de los costes de las operaciones utilizando experimentos que utilicen el mismo esquema de almacenamiento y patrón de acceso a los datos en memoria que utiliza la rutina a modelar. En relación al rango de valores para el tamaño de problema, en nuestra propuesta sólo es necesario contemplar aquellos tamaños considerados en el momento de la instalación. Si en la ejecución de la rutina el tamaño del problema no está contemplado en la instalación, se puede realizar una interpolación o utilizar en el modelo el valor más cercano.

1.4 Herramientas computacionales

En esta sección se describirá el entorno computacional que se ha utilizado para llevar a cabo la experimentación. En primer lugar se indicarán las características más relevantes del hardware que conforma las plataformas paralelas empleadas. A continuación se mostrarán las librerías y compiladores utilizados, que proporcionan las primitivas contempladas en los paradigmas de programación paralela. Finalmente se describirá el software de álgebra lineal que proporciona las operaciones básicas.

1.4.1 Hardware

Los sistemas computacionales utilizados en la fase experimental de esta tesis incluyen plataformas paralelas homogéneas y heterogéneas que presentan entre ellas diferencias significativas en sus características hardware, de tal forma que permiten reflejar las diferentes prestaciones

de cómputo tanto globales como locales a nivel de nodo, las diferentes redes de interconexión, el ratio entre la capacidad de cómputo y la capacidad de la red de interconexión o las características de las memorias de las plataformas computacionales paralelas disponibles. Por otra parte, y debido al tiempo transcurrido entre el inicio y la finalización de la tesis, alguna de las plataformas utilizadas ya no se pueden considerar sistemas actuales de cómputo paralelo. En cualquier caso, esta circunstancia no condiciona la utilidad del estudio realizado, sino que lo enriquece al permitir comprobar su utilidad en un amplio abanico de plataformas de cómputo a lo largo de un amplio período de tiempo.

A continuación se van a describir las características que las plataformas utilizadas presentaban en el momento de la realización de los estudios experimentales:

- Plataforma SUNEt, perteneciente al Grupo de Investigación de Computación Científica y Programación Paralela (PCG) de la Universidad de Murcia (UMU). Consistía en un cluster de estaciones de trabajo conectadas por una red Ethernet. Había cinco estaciones de trabajo SUN Ultra 1 y una estación de trabajo SUN Ultra 5. El sistema operativo era Solaris, el compilador era el de la plataforma y como librería de pasos de mensajes se empleó MPICH.

- Plataforma TORC, del Laboratorio ICL de la Universidad de Tennessee. Consistía de un cluster con 21 nodos de diferentes características. Había 10 nodos con 2 procesadores Intel Pentium III a 550 MHz cada uno, 3 nodos con procesadores Intel Pentium III a 600 MHz, 2 nodos con Intel Pentium II a 450 MHz, 3 nodos con procesadores AMD Athlon 1.2 GHz, 1 nodo Intel Pentium 4 a 1.7 GHz, y 1 nodo Intel Pentium 4 a 1.5 GHz. Disponía además de varias redes de interconexión: Fast-Ethernet, Myrinet y Giganet. La red de interconexión utilizada para nuestros experimentos fue la Fast-Ethernet. El sistema operativo era Linux, el compilador empleado fue el GNU gcc y la librería de paso de mensajes era MPICH.

- Plataforma HPC160, gestionada por el Servicio de Apoyo a la Investigación Tecnológica (SAIT) de la Universidad Politécnica de Cartagena (UPCT). Se trataba de un sistema paralelo de memoria distribuida que constaba de 4 nodos. Los nodos disponían de 4 procesadores Alpha EV68CB a 1 GHz con 8 Mbytes de caché de nivel 2. Los nodos estaban interconectados por una red Memory Channel. El sistema operativo era Tru64 UNIX. Se utilizaron el compilador y la librería MPI disponibles en el sistema y el compilador GNU gcc. Se realizaron experimentos tanto con un único nodo (HPC160smp), como con varios nodos (HPC160mc).

- Plataforma P4net, gestionada por el Servicio de Apoyo a la Investigación Tecnológica (SAIT) de la Universidad Politécnica de Cartagena (UPCT). Se trataba de un cluster con 4 nodos Intel Pentium 4 a 1.8 GHz. La red de interconexión era Fast-Ethernet. El sistema operativo era Linux y disponía del compilador GNU gcc y la librería de pasos de mensajes MPICH.

- Plataforma Rosebud, gestionada por el Grupo Interdisciplinar de Computación y Comunicaciones (INCO2-DSIC) del Departamento de Sistemas Informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia (UPV). Era una plataforma computacional que se había ido ampliando a lo largo del tiempo, por lo que se trataba de un sistema de naturaleza heterogénea. Rosebud constaba de 2 nodos con procesadores Intel Pentium 4 con diferentes velocidades de reloj, 2 nodos con dos procesadores Intel Xeon a 2.2 GHz cada uno, 2 nodos con procesadores Intel quad-core a 2.66 GHz y 2 nodos con cuatro procesadores Intel Itanium Montecito dual-core 2 a 1.4 Ghz en cada nodo, 8 cores por nodo, conectados con Gigabit-Ethernet. Los experimentos se realizaron utilizado los nodos con procesadores Intel Itanium Montecito. El sistema operativo era Linux, el compilador empleado era el GNU gcc y la librería de paso de mensajes LAM-MPI.

- Plataforma Sol, perteneciente al Grupo de Investigación de Computación Científica y Programación Paralela (PCG) de la Universidad de Murcia (UMU). El sistema disponía de 5 nodos de computación, 3 de ellos con dos procesadores Intel Xeon 5050 dual-core a 3.0 GHz por nodo (4 cores por nodo), y 2 con un procesador Intel Xeon 5050 dual-core a 3.0 GHz por nodo, lo que daba un total 16 cores. Los nodos estaban conectados por una red Gigabit-Ethernet. El sistema operativo era Linux, se utilizó el compilador GNU gcc y el de Intel icc. La librería de paso de mensajes era LAM-MPI.

- Plataforma Sagitario, perteneciente al Grupo de Investigación de Computación Científica y Programación Paralela (PCG) de la Universidad de Murcia (UMU). Se trata de un nodo Intel Pentium a 2.8 GHz con dos cores y 2 GBytes de RAM. El sistema operativo era Linux. Se realizaron experimentos con el compilador GNU gcc y con el compilador Intel icc.

- Plataforma Hipatia, gestionada por el Servicio de Apoyo a la Investigación Tecnológica (SAIT) de la Universidad Politécnica de Cartagena (UPCT). Se trataba de un cluster de 14 nodos con dos procesadores Intel Xeon Quad-Core a 2.80 GHz por nodo, 2 nodos con 4 Intel Xeon Quad-Core a 2.93 GHz y 2 nodos con 2 Intel Xeon 5160 Dual-Core a 3.00 GHz. La red de interconexión de los nodos era Infinibad 4X DDR2. El sistema operativo era Linux. Se realizaron experimentos con el compilador GNU gcc y con el compilador Intel icc. La librería de paso de mensajes fue una versión de MPICH proporcionada por el fabricante.

1.4.2 Software para cálculo intensivo

Para la obtención de los resultados experimentales y las rutinas de creación propia se ha empleado el lenguaje de programación ANSI C. Los ejecutables se han generado utilizando el compilador proporcionado por el fabricante, el de GNU, y ambos compiladores en aquellos experimentos en los que se requería. El desarrollo de rutinas que utilizan el paradigma de programación por paso de mensajes se ha realizado con MPI [SOHL$^+$98], y el de programación por

compartición de variables con OpenMP [DM98]. Las implementaciones utilizadas para MPI han sido varias: MPICH del Laboratorio Nacional de Argone y la Universidad Estatal de Mississippi [mpi], LAM-MPI de la Universidad de Ohio [lam, BDV94] además de las proporcionadas por el fabricante de cada plataforma paralela.

1.4.3 Librerías de álgebra lineal

Las operaciones de álgebra lineal constituyen el núcleo sobre el que se construye una gran parte del software que se utiliza en computación de alto rendimiento (*HPC*: High Performance Computing). Esto significa que el desarrollo de un conjunto de rutinas altamente eficientes y numéricamente estables que proporcionaran operaciones básicas para vectores y matrices, supondría una gran ayuda para los investigadores que utilizaran la computación científica como herramienta en sus trabajos. Con tal propósito surgió en 1979 la librería BLAS (*Basic Linear Algebra Subprograms*) [LHKK79], que ha llegado a convertirse en una interface de programación de aplicaciones estándar de facto y de la que se pueden encontrar implementaciones altamente eficientes para la mayoría de las arquitecturas de cómputo existentes. BLAS consta de tres niveles: nivel 1 de BLAS que proporciona operaciones vector-vector, nivel 2 de BLAS que proporciona operaciones matriz-vector y nivel 3 de BLAS que proporciona operaciones matriz-matriz.

La existencia de BLAS ha permitido la construcción de software como LAPACK [ABB+99], de gran utilidad en la resolución numérica de problemas en ciencias e ingenierías, al proporcionar rutinas para la resolución de sistemas de ecuaciones, de problemas de mínimos cuadrados, de autovalores y una serie de operaciones sobre matrices, como la factorización LU, QR, Cholesky... LAPACK se diseñó con el fin de aprovechar el buen rendimiento que las operaciones de nivel 3 de BLAS obtienen en las arquitecturas de computación de hoy en día.

Con la aparición de paradigmas de programación por paso de mensajes que explotaban las arquitecturas de computación paralela existentes, surgió la necesidad de diseñar una librería que proporcionara versiones paralelas de las rutinas de LAPACK. Con esa idea en mente se diseñó la librería ScaLAPACK [BCC+97]. Al igual que LAPACK, las rutinas de ScaLAPACK están basadas en algoritmos que operan por bloques con el fin de minimizar el movimiento de datos entre los diferentes niveles de la jerarquía de memoria, y también presenta una estructura jerárquica. ScaLAPACK está construida a partir de versiones de la rutinas de BLAS (PBLAS [CDW96]) que operan sobre matrices distribuidas atendiendo a un esquema cíclico por bloques [LJ93], y de una librería (BLACS [DW97]) que implementa tareas de comunicación que son realizadas con frecuencia en computaciones paralelas de álgebra lineal.

1.5 Estructura de la memoria

En el capítulo 2 se muestran los principales antecedentes históricos relacionados con el trabajo presentado en esta tesis. Se hará hincapié en aquellos que están relacionados directamente con el objetivo de nuestro trabajo, esto es, la obtención de modelos de tiempo de ejecución de rutinas de álgebra lineal densa. A continuación se describirá la propuesta de modelado que se ha usado como referencia para el desarrollo de este trabajo. Se comenzará mostrando el modelo de sistema para una plataforma genérica de computación paralela, y posteriormente se mostrará el modelo analítico de tiempo de ejecución de una rutina paralela de álgebra lineal, indicando el significado de cada uno de los componentes que conforman este modelo.

En el capítulo 3 se describe detalladamente nuestra propuesta de mejoras en el modelado de aplicaciones. Se comenzará haciendo una serie de consideraciones que es necesario tener en cuenta con el fin de obtener modelos que aproximen adecuadamente el tiempo de ejecución de una rutina paralela de álgebra lineal. Posteriormente, se describirán las distintas rutinas que se han utilizado en los experimentos y se mostrará la aplicación de las mejoras propuestas en el modelo analítico de su tiempo de ejecución de cara a una primera validación de dicho modelo. Para cada una de las rutinas de estudio, se mostrará para diversas plataformas hardware una comparación entre su tiempo de ejecución experimental y el tiempo teórico según el modelo propuesto. Simultáneamente a dicha comparación de tiempos (experimental vs. teórico), se constatará que mediante la utilización de la información proporcionada por el modelo propuesto sobre el comportamiento previsto de la rutina, es posible realizar de forma satisfactoria el ajuste automático de la misma y la obtención de tiempos de ejecución cercanos al óptimo.

En el capítulo 4 se aborda la obtención de modelos de tiempo de ejecución en aquellos casos en los que no se dispone de información sobre el algoritmo implementado en la rutina paralela de álgebra lineal, lo que complica la obtención de un modelo analítico de tiempo de ejecución. En primer lugar se mostrará el esquema seguido para la generación de polinomios que aproximen el tiempo de ejecución. A continuación se mostrará la metodología propuesta en la obtención de los coeficientes del polinomio. Finalmente se mostrarán algunos resultados experimentales obtenidos con diversas rutinas usando esta metodología de modelado como herramienta de optimización automática.

En el capítulo 5 ampliamos nuestra propuesta de modelado y optimización de rutinas de álgebra lineal a entornos de ejecución heterogéneos. Se combinará la metodología de modelado vista en los capítulos anteriores con la utilización de estrategias de asignación de procesos a procesadores, con el objetivo de que las rutinas diseñadas para ser ejecutadas en plataformas computacionales homogéneas, puedan ejecutarse de manera óptima en plataformas heterogéneas sin necesidad de modificar su código. Se comenzará el capítulo mostrando las modificaciones que ha sido necesario realizar a nuestra metodología de modelado con el fin de adaptarla al caso de plataformas heterogéneas. En la segunda parte del capítulo, plantearemos el problema de asignación de procesos a procesadores y, a continuación, mostraremos una propuesta que

permitirá abordar su solución. Finalmente, se mostrarán los resultados obtenidos en diversas plataformas heterogéneas.

En el capítulo 6 se estudia la aplicación de nuestra metodología de modelado a rutinas multithread y se incluye en el proceso de selección el código generado por los compiladores OpenMP disponibles en una plataforma multicore. En primer lugar se mostrará la metodología seguida para la obtención de los parámetros del sistema asociados con las capacidades de generación y gestión de threads básicos; a continuación se mostrará que con la ayuda de los modelos teóricos de tiempo de ejecución y el valor obtenido experimentalmente para los parámetros del sistema, se puede realizar una selección adecuada de la versión del ejecutable más idóneo y de otros parámetros del algoritmo, como el número de threads paralelos.

En el capítulo 7 se comienza destacando las principales conclusiones obtenidas con la realización de esta tesis. Se prosigue indicando los proyectos en cuyo marco se ha llevado a cabo este trabajo y las publicaciones realizadas durante su desarrollo. Finalmente, se concluye con la propuesta de algunas líneas futuras de trabajo.

Capítulo 2

Modelado y optimización de rutinas paralelas de álgebra lineal

2.1 Introducción

Desde la aparición de la computación paralela ha existido un gran interés en la realización de estudios teóricos sobre el comportamiento del software a ejecutar sobre este tipo de plataformas y en la construcción de modelos que permitieran analizar su comportamiento. Así, en los años 80 se realizó una importante labor investigadora de cara a la obtención de algoritmos paralelos eficientes, utilizando como principal herramienta el modelo *Parallel Random Access Machine* (PRAM) y sus variantes (véase la sección 2.2). Dicho modelo requiere que un algoritmo sea expresado completamente en términos de instrucciones que controlan el acceso individual a la memoria. Su utilización permitió a los investigadores comprender las características de los algoritmos paralelos bajo estudio. Se han propuesto diferentes extensiones al modelo PRAM, con el fin de reflejar la complejidad de la arquitectura hardware de un computador paralelo. Desafortunadamente las numerosas modificaciones al modelo PRAM original no permitieron abordar el problema de una forma unificada.

A finales de los años 80 y principios de los 90 se plantearon nuevas formas de acometer el problema de modelar el rendimiento de códigos paralelos. El enfoque adoptado consistía en evaluar directamente el rendimiento a partir de la estructura del código fuente [Val90, CKP+93], permitiendo modelar su rendimiento mediante una serie de parámetros y ecuaciones que los relacionaban. En general, estas metodologías permiten modelar el rendimiento de código paralelo del que se conoce su código fuente, o con una estructura regular en la que se van alternando bloques de computación con bloques de comunicación.

Posteriormente, con el fin de poder abordar el problema de modelar códigos paralelos más

complicados, se han venido utilizando técnicas estadísticas [Gau04], redes de Petri [WH97] y Redes Neuronales [IdSSM05]. Si bien estas técnicas son capaces de modelar el comportamiento de código paralelo complejo, presentan el problema del alto coste computacional para la resolución completa del modelo que plantean.

En este capítulo se realizará un repaso de algunos de los trabajos que se están desarrollando en la actualidad, y de los diferentes esquemas que se han venido utilizando en el problema de obtención de modelos, haciendo especial hincapié en aquellos trabajos que se aplican en la obtención de modelos de rutinas de álgebra lineal para plataformas paralelas y que han influido notablemente en el trabajo presentado en esta tesis. Se introducirá el modelo DLAM+ (véase la sección 2.3), el cual ha servido como punto de partida para desarrollar nuestra propuesta de mejoras en el modelado del tiempo de ejecución de rutinas paralelas de álgebra lineal. Finalmente se compararán otros trabajos actuales relacionados con nuestra propuesta y se expondrán las conclusiones a las que se ha llegado con este repaso del estado del arte en modelado y optimización de software científico.

2.2 Modelos generales

Modelo PRAM

En el año 1978 Fortune y Wylie [FW78] plantearon un modelo abstracto de computación paralela denominado PRAM (*Parallel Random Access Machine*). El modelo PRAM proporciona un modelo general de computación paralela en el que se hace una abstracción de los detalles de la arquitectura hardware y del estilo de programación, que se centra en el paralelismo disponible al problema que se intenta resolver computacionalmente. Su simplicidad y generalidad convirtieron al modelo PRAM en una herramienta de investigación aceptada ampliamente y en un referente para los trabajos que surgieron posteriormente en el modelado de plataformas paralelas.

El modelo PRAM se basa en una máquina de procesamiento paralelo idealizada, que consiste en P procesadores síncronos comunicándose a través de memoria compartida. Existe por tanto una unidad de control centralizada que marca la ejecución de las instrucciones. Cada procesador es capaz de ejecutar una instrucción o realizar una operación de comunicación por ciclo de reloj. Junto con la memoria compartida, cada uno de los procesadores cuenta con un conjunto de registros para almacenamiento local. En el modelo PRAM existen varias familias que han sido clasificadas según la semántica utilizada para acceder a la memoria compartida [KGGK94]: Lectura Exclusiva, Escritura Exclusiva (EREW PRAM); Lectura Concurrente, Escritura Exclusiva (CREW PRAM); Lectura Concurrente, Escritura Concurrente (CRCW PRAM); y Lectura Exclusiva, Escritura Concurrente (ERCW PRAM).

El modelo de coste en PRAM resultó ser de poca utilidad en la práctica, motivo por el cual aparecieron diferentes variantes en las que se proponían restricciones al modelo original,

de tal forma que se tuvieran en cuenta las características de la arquitectura hardware, como la jerarquía existentes en la memoria [ACFS92, HR92], la contención [KLAdH92], la latencia [PY88], el ancho de banda [ACS90] y el acceso asíncrono [Gib89]. En [Har94] se puede encontrar un estudio completo del modelo PRAM y de algunas de sus extensiones. Pese a sus limitaciones, siguen apareciendo nuevas versiones del modelo, y en propuestas más recientes [SZ09] el modelo original se extiende para poder ser aplicado a sistemas multicore.

Modelo BSP

En el año 1990 Valiant [Val90] describió una técnica para escribir programas paralelos eficientes y portátiles con un rendimiento predecible, denominada *Bulk Synchronous Parallelism* (BSP) y un modelo de computador paralelo llamado BSPC. El modelo BSPC describe un computador que consta de un número de parejas procesador-memoria, una red de interconexión con un ancho de banda limitado y un sistema de barreras de sincronización de coste fijo.

La computación en el modelo BSP se realiza a través de una serie de etapas (*supersteps*) paralelas, cada una de las cuales se divide en tres fases. En la primera fase, cada pareja procesador-memoria (P) realiza computaciones únicamente con datos locales. Esta actividad puede ser modelada utilizando el parámetro S de McColls [Mcc98] que representa el número de operaciones básicas (adición, multiplicación) que se pueden realizar por un procesador en un segundo. En la segunda fase (fase de comunicación), los procesadores comparten información y se puede enviar y recibir cualquier número de mensajes. El patrón de comunicación se define mediante una h-relación en la que cada proceso envía y recibe como máximo h mensajes. El parámetro h se suele utilizar también para incluir el tamaño total en bytes (m), esto es, h se suele utilizar como una abreviación para hm. El coste de las comunicaciones se modela con el parámetro g, que representa el tiempo requerido para que una h-relación se complete en un tráfico continuo de mensajes entre procesos aleatorios. El parámetro g se determina normalmente de forma empírica en la plataforma paralela bajo análisis. En la última fase, se realiza una barrera de sincronización. La duración de esta sincronización se modela con el parámetro l, el cuál también es determinado empíricamente.

El tiempo de ejecución de un *superstep* en el modelo BSP se calcula a partir del código fuente del programa y de los parámetros de la plataforma en la que se ejecutará. El modelo de coste sería:

$$T = \text{máx}(w_1, ..., w_i, ..., w_P) + \text{máx}(h_1 g, ..., h_i g, ..., h_P g) + l$$

donde w_i es el coste para una computación local en el proceso i.

En [SHM96] se puede encontrar una descripción más extensa del modelo BSP y una comparativa con otras técnicas. Con el propósito de obtener un modelo de coste más exacto, han

surgido diversas extensiones o variaciones a partir del modelo BSP original. En [Tis98, MT99] se propone incluir el tamaño de memoria como un cuarto parámetro. En [TK96] se plantea un modelo jerárquico conocido con el nombre de D-BSP. En [BPPS07] se analiza la utilización de un modelo de varios niveles con un número arbitrario de parámetros en cada nivel y en [Val08] se propone el modelo Multi-BSP que incluye explícitamente parámetros para el número de procesadores, tamaños de la memoria y de la caché, coste de las comunicaciones y coste en la sincronización, con el objetivo de adaptar el modelo BSP original a computaciones en sistemas multicore.

Modelo LogP

En el año 1993 Culler *et al.* [CKP+93] desarrollaron el modelo LogP a partir del BSP. A diferencia del modelo BSP, en el modelo LogP no se requiere una barrera global que separe la fase de computación de la de comunicación y, además, añade el concepto de red de interconexión con capacidad finita, en la que en un momento determinado existe una cantidad delimitada de mensajes en tránsito. En LogP se plantea un modelo de computador paralelo, formado por un conjunto de nodos (procesador-memoria) interconectados por una red de ancho de banda limitado y con una latencia significativa. Partiendo de dicho planteamiento, se propone un modelo de coste que caracteriza la red de interconexión, pero sin llegar a describir su estructura. Este modelo se compone de una serie de parámetros:

- *Capacidad de computación* proporcionada por el número de unidades de procesador-memoria, P.
- *Capacidad de comunicación* entre los procesadores $\frac{1}{g}$, siendo g (*gap*) el intervalo de tiempo entre la recepción/transmisión de dos mensajes consecutivos.
- *Latencia en la comunicación* entre procesadores, modelada por una constante L que representa una cota superior en el tiempo de comunicación de un mensaje pequeño medido en condiciones de descarga.
- *Eficiencia* entre comunicación y computación, que se modela con el parámetro o que representa la sobrecarga (*overhead*) que aparece durante la transmisión de un mensaje. El modelo supone que durante ese tiempo el procesador que inicia el envío o la recepción de un mensaje no puede realizar ninguna otra tarea.

El modelo básico asume que todos los mensajes son de una longitud fija y de poco tamaño. Como se ha comentado, el modelo asume que la red tiene una capacidad finita, y por tanto $\lceil L/g \rceil$ mensajes pueden estar en tránsito a lo sumo desde un procesador a cualquier otro en un momento dado. Si la red de interconexión está trabajando dentro de sus límite de capacidad, el tiempo para transmitir un mensaje pequeño será $2o + L$, con un *overhead* de o en el emisor y en el receptor, y con una latencia de L en la red. El ancho de banda disponible por procesador

es determinado por g y por la capacidad de la red, $\lceil L/g \rceil$. La red se trata como una *pipeline* de profundidad L, con un ratio de inicialización de g y una sobrecarga en los procesadores en cada extremo de o. Por tanto el coste para comunicar n mensajes cortos es:

$$T_{COM} = 2o + L + (n-1)\,\text{máx}(o, g)$$

Como ha ocurrido con otras propuestas de modelos genéricos, han aparecido extensiones al modelo original LogP con el fin de adaptar éste al nuevo hardware de computación y a las interfaces actuales de programación paralela. Algunas de estas extensiones se indican a continuación.

- Modelo LogGP [AISS95]: consiste en un modelo lineal que tiene en cuenta la posibilidad de envío de mensajes largos. Se introduce un parámetro nuevo, G, que representa el ancho de banda obtenido en el envío de mensajes largos.
- Modelo LoPC [FAV97]: utiliza directamente los parámetros del modelo LogP para predecir el coste de la contención en redes de estaciones de trabajo y sistemas multicomputador. Supone que la red de interconexión está libre de contención y que ésta tiene lugar en los recursos para procesamiento de mensajes en los nodos.
- Modelo P-LogP [KBG+01]: se trata de un modelo más detallado de aplicación en sistemas paralelos conectados por redes de área amplia y se diseña tomando como base el modelo LogP/LogGP. En el modelo P-LogP se tiene en cuenta la latencia, el ancho de banda, el número de sistemas paralelos, el número de procesadores en cada sistema paralelo y la longitud de los mensajes. En este modelo los parámetros se consideran una función de la longitud del mensaje.
- Modelo mLogP [CS03]: tiene en cuenta el impacto de la comunicación que ocurre en la memoria local de un nodo en sistemas distribuidos de memoria compartida. Se propone un modelo de comunicación punto-a-punto para predecir la latencia en operaciones de transferencia de datos en memoria.
- Modelo HLogGP [BP06]: es una extensión del modelo LogGP para poder ser aplicado en sistemas paralelos heterogéneos. Sustituye los parámetros escalares del modelo LogGP por parámetros vectoriales y matriciales, de tal forma que se tiene en cuenta que nodos distintos presentan valores distintos para estos parámetros.
- Modelo mPlogP [LZFD10]: se trata de una variación de mLogP para sistemas heterogéneos multicore. El modelo mPLogP utiliza parámetros vectoriales en lugar de escalares, con el fin de modelar la heterogeneidad de una arquitectura multicore.
- Modelo LoOgGP [Rod11]: es una generalización del modelo LogP en el que se considera que tanto el *gap* como el *overhead* presentan una dependencia lineal con el tamaño del

mensaje, mientras que la latencia y el número de procesadores son independientes de la cantidad de bytes de la comunicación.

Otros modelos generales

A continuación se reseñan otras propuestas que han sido de utilidad en el planteamiento de nuestra aproximación al problema de optimización de rutinas paralelas de álgebra lineal densa por medio del modelado de su tiempo de ejecución.

En 1994 Crovella y LeBlanc [CL94, Cro94] presentaron una propuesta para resolver el problema de estimación del rendimiento denominada *Lost Cycles Analysis* en la que se diferenciaba entre computación productiva y sobrecarga (*overhead*) paralela. El modelo pretendía ser completo de tal forma que capturara todas las posibles fuentes de sobrecarga y que a su vez las fuentes incluidas fueran mutuamente exclusivas. Usando este método, las fuentes de sobrecarga se dividen en varias categorías: desbalanceo de la carga de trabajo, paralelismo insuficiente, coste de sincronización, coste de comunicación y contención de recursos (tiempo de espera en el acceso a recursos hardware compartidos). Todas las categorías de sobrecarga se miden utilizando la métrica común *lost cycles*, y para cada una se obtiene un modelo analítico a partir de la realización de un pequeño número de experimentos.

Clement y Quinn [CQ95, CQ97] plantean la utilización de un modelo lineal función del tamaño del problema y del número de procesadores utilizados, en el que se tiene en cuenta el coste de la computación aritmética, el coste de las comunicaciones, el coste de los accesos a memoria y los efectos del compilador usado (número de instrucciones de bajo nivel generadas). La expresión analítica parametrizada resultante se introduce en un paquete de manipulación simbólica. Los autores muestran, utilizando técnicas de análisis estadístico multivariante, que lo parámetros pueden ser aproximados por distribuciones normales con una varianza constante en todos los casos.

Brewer [Bre95] desarrolla una herramienta para la construcción de librerías portátiles de alto nivel que contienen múltiples implementaciones de una única especificación. La selección de la implementación más rápida y de los parámetros optimizados para una carga de trabajo y plataforma dada se realiza de forma automática a partir de modelos estadísticos que utilizan técnicas de regresión lineal e información de ejecución de la librería. La herramienta proporciona para cada modelo una métrica que representa la exactitud del modelo.

Xu *et al.* [XZS96] proponen una metodología en la que se utiliza un modelo jerárquico basado en dos niveles. Un modelo en forma de grafo de tareas permite describir el comportamiento de los programas. Para caracterizar por completo la plataforma paralela se utilizan experimentos para la obtención de sus parámetros implícitos.

Rauber y Rünger [RR01] desarrollan un modelo computacional que describe la jerarquía de memorias en sistemas de memoria compartida distribuida. El modelo también es de aplicación

en sistemas de memoria compartida y de memoria distribuida, y permite modelar el tiempo de acceso a datos compartidos almacenados en diferentes niveles de la jerarquía de memorias y la transferencia de bloques de datos entre diferentes niveles de la memoria.

Engin Ipek *et al.* [IdSSM05] proponen la utilización de redes neuronales para construir un modelo que predice el rendimiento de aplicaciones a partir de un espacio de parámetros multidimensional definido por los argumentos pasados a la aplicación, haciendo hincapié en la eliminación de ruido en el conjunto de datos y en la elección de técnicas adecuadas de entrenamiento para la fase de aprendizaje de la red neuronal. Combinan un técnica de muestreo denominada *stratification* y un mecanismo de aprendizaje denominado *bagging* o *bootstrap aggregation*. Como resultado, la red neuronal "ve" puntos con valores pequeños muchas más veces que aquellos con valores absolutos mayores, de forma que se consigue minimizar el porcentaje de error en la predicción del rendimiento.

Seyed Masoud *et al.* [SSF+08] describen un modelo de tiempo de ejecución en el que se tiene en cuenta las propiedades de los recursos de la plataforma de ejecución. Dichos recursos pueden englobarse en tres categorías básicas: computación, comunicación y almacenamiento. Los recursos computacionales que afectan al tiempo de ejecución incluyen la velocidad de reloj de la CPU, el tamaño de la caché de nivel 2, y el ancho de banda del *front-side-bus* (FSB). Los parámetros de comunicación incluyen el ancho de banda máximo y la latencia de la red de interconexión. Por último, los parámetros de almacenamiento incluyen el tamaño de la memoria principal, la velocidad de acceso, etc. Si bien el número de parámetros puede parecer excesivo, en la práctica se ve reducido, y es posible obtener modelos aproximados que utilicen sólo la velocidad de reloj de la CPU y el número de nodos. Finalmente se representa el tiempo de ejecución mediante una combinación lineal de variables explicativas (términos básicos para un análisis de regresión) que son función de las propiedades de recursos estáticos, como la velocidad de reloj, multiplicados por parámetros que definen las características de la aplicación.

2.2.1 Modelos para las comunicaciones

Dado que en la ejecución de un programa paralelo utilizando el paradigma de programación por paso de mensajes se requiere el intercambio de información entre las unidades de procesamiento, es necesario analizar correctamente el coste de las comunicaciones a la hora de obtener su modelo de tiempo de ejecución. Por tal motivo, y dada la diversidad de factores que pueden influir en el coste de las comunicaciones (el modelo de programación, la topología de la red que interconecta el sistema de computación paralelo, los protocolos de software asociados...) se han llevado a cabo numerosos estudios sobre el modelado del coste de las comunicaciones de un programa paralelo.

Según el modelo propuesto por Hockney [Hoc82, Hoc94] las comunicaciones punto-a-punto en sistemas paralelos pueden describirse por medio de la siguiente expresión:

$$T_{COM}(n) = t_0 + \frac{n}{r_\infty}$$

donde t_0 es el tiempo de inicio de una comunicación (*startup time*), que se define como el tiempo necesario para enviar un mensaje de 0 bytes, n es la longitud del mensaje en bytes, y r_∞ es el ancho de banda asintótico de una comunicación, que se alcanza cuando el tamaño del mensaje tiende a infinito. Atendiendo al modelo de Hockney, $\frac{n}{r_\infty}$ representa el retraso en transmitir un mensaje de n bytes a través de una red con un ancho de banda asintótico de r_∞. Hockney introdujo dos parámetros adicionales, $n_{\frac{1}{2}}$, que es el tamaño de mensaje necesario para alcanzar la mitad del ancho de banda asintótico, y π_0 que indica el ancho de banda para mensajes cortos. Sólo dos de los cuatro parámetros, r_∞, t_0, $n_{\frac{1}{2}}$ y π_0 son independientes. Los otros dos pueden deducirse de estas relaciones: $t_0 = \frac{n_{\frac{1}{2}}}{r_\infty} = \frac{1}{\pi_0}$.

Tomando como referencia el modelo clásico de Hockney, diversos estudios han extendido su aplicación a sistemas de comunicaciones que han ido surgiendo, a comunicaciones colectivas, a sistemas heterogéneos, etc. A continuación se reseñarán algunas propuestas similares y contribuciones al modelo clásico de Hockney.

Kumar *et al.* [KGGK94] y Barnett *et al.* [BGP+94], partiendo del modelo clásico, realizan un estudio detallado del coste de las comunicaciones en diferentes configuraciones de redes de interconexión (anillo, malla de dos dimensiones e hipercubo) y esquemas de enrutamiento para el conjunto de comunicaciones colectivas de uso habitual.

Foster [Fos95] propone un modelo más detallado en el que se tiene en cuenta la competición por el ancho de banda entre varios procesos que envían datos simultáneamente. Se introduce un nuevo factor que denomina factor de escala (S). Este factor coincide con el número de procesos que comparten el ancho de banda, y refleja la idea de que el ancho de banda efectivo para cada proceso que comparte una misma red de comunicación es $\frac{1}{S}$ del ancho de banda total. El modelo para el coste de las comunicaciones queda definido por la siguiente ecuación:

$$T_{COM}(n) = t_0 + \frac{n}{r_\infty}S$$

Xu y Hwang [XH96] extienden el modelo de Hockney para que pueda ser de aplicación a operaciones colectivas. Según este modelo, el tiempo de una comunicación colectiva es:

$$T_{COM}(n,p) = t_0(p) + \frac{n}{r_\infty}(p)$$

donde p es el número de nodos que participan en la comunicación. Al igual que ocurría en el modelo de Hockney, T_{COM} es una función lineal del tamaño del mensaje, pero el tiempo de inicio de una comunicación y el ancho de banda asintótico son funciones, no necesariamente

lineales, del número de nodos.

Getov *et al.* [GHH97] muestran en su trabajo que, además de los parámetros hardware propuestos en el modelo clásico, existen una serie de parámetros que es necesario tener en cuenta con el fin de obtener una estimación más real del tiempo de las comunicaciones en un programa paralelo. Los factores a los que hacen referencia en su trabajo son el tipo de dato de los mensajes, el patrón de acceso a memoria, y la localización en la jerarquía de memorias de los datos a transmitir.

Tessera y Dubey [TD01] aplican una versión modificada del modelo de Hockney al estudio de las diferentes estrategias de comunicación disponibles en MPI (punto a punto vs. colectivas, protocolos bloqueantes vs. no bloqueantes...), y analizan la influencia de la estrategia seleccionada para la comunicación, en el tiempo total de ejecución de un programa paralelo.

Thakur y Gropp [TG03] realizan diversas modificaciones al modelo clásico y lo aplican en la determinación del mejor algoritmo en comunicaciones colectivas MPI en función del tamaño del mensaje a enviar.

Cham *et al.* [CHPVdG04] adaptan el modelo de Hockney de cara a evaluar el rendimiento de varios algoritmos para comunicaciones colectivas en una red de interconexión con topología de malla bidimensional.

2.3 Modelado del sistema paralelo de ejecución de rutinas de álgebra lineal

En 1996, Dackland y Kågström [DK96] plantean un modelo para sistemas paralelos de computación que ejecutan rutinas paralelas de álgebra lineal densa. El modelo propuesto se limita al conjunto de operaciones BLAS y operaciones de comunicación de tipo BLACS. Este modelo es posteriormente ampliado por Cuenca [Cue04] a cualquier librería de álgebra lineal y a cualquier librería estándar de paso de mensajes (modelo DLAM+).

Partiendo de las dos propuestas citadas, junto con las ideas extraídas de los modelos de arquitecturas paralelas mostrados en la sección anterior, planteamos una propuesta de mejoras en el modelado de rutinas paralelas de álgebra lineal con el fin de dotar a dicho software de la capacidad de optimización automática. Para alcanzar dicho objetivo se utilizará un modelo analítico del tiempo de ejecución que sea aplicable a sistemas de computación paralela. Con tal fin, se utilizarán una serie de parámetros que permitirán plasmar los efectos que, tanto las características de la plataforma hardware de cómputo paralelo, como las librerías básicas de cálculo numérico, y las librerías básicas de comunicaciones, puedan tener en el tiempo de ejecución del software.

Dada la diversidad de librerías numéricas de álgebra lineal, redes de interconexión y librerías

de comunicaciones que hoy en día pueden encontrarse en una plataforma paralela dada, y con el propósito de poder realizar una descripción detallada de nuestra propuesta de mejoras en el modelado, se hace conveniente realizar una división del modelo atendiendo a las capacidades de los elementos que pretende describir. De esta forma, tendremos un modelo para el cómputo, donde se recogerán las principales características de cómputo de la plataforma paralela y las librerías básicas de álgebra lineal; y un modelo para las comunicaciones, donde se intentarán reflejar las capacidades para las comunicaciones de la plataforma paralela y las librerías de comunicaciones.

Modelo para el cómputo

En una plataforma de cómputo, atendiendo a los niveles de las librerías BLAS, o de cualquier otra librería con una estructura similar, las operaciones de cálculo que se pueden realizar se clasificarían en tres niveles:

- Nivel 1: operaciones vector-vector, escalar-vector, con un coste $O(n)$.
- Nivel 2: operaciones matriz-vector, con un coste $O(n^2)$.
- Nivel 3: operaciones matriz-matriz, con un coste $O(n^3)$.

A las operaciones realizadas en cada nivel se les asocia un parámetro que representa el tiempo en llevar a cabo una operación en coma flotante, de tal forma que tendremos el parámetro k_1 para rutinas de nivel 1, el parámetro k_2 para rutinas de nivel 2 y el parámetro k_3 para rutinas de nivel 3. Con el fin de obtener un modelo que refleje adecuadamente el coste de la computación, es necesario realizar las siguientes consideraciones:

- El patrón de acceso a los datos de operaciones diferentes no tiene por qué ser el mismo aunque pertenezcan al mismo nivel. Esta variación en el patrón de acceso provocará variación en la localidad de los datos, pudiendo dar lugar a un coste distinto según la operación utilizada. Por tanto, es necesario para cada nivel i, con $i = 1, 2, 3$, distinguir entre valores de k_i. Tendremos por tanto un $k_{i,operacion}$.
- Con el fin de reflejar la existencia de diferentes versiones de BLAS para una plataforma dada, y la existencia de librerías con una estructura similar a BLAS, tal como ATLAS, es necesario considerar valores diferentes de $k_{i,operacion}$ en función de la librería básica de álgebra lineal que se utilice.
- Como consecuencia de la consideración anterior, el modelo también será de utilidad a la hora de elegir la librería más adecuada en aquellos casos en que se cuente con más de una implementación instalada de las operaciones de álgebra lineal básica.

- El parámetro $k_{i,operacion}$ no se considerará constante, sino que, dependiendo del tamaño del problema a resolver y de la forma en la que estén almacenados los datos, podrá tomar valores distintos, por lo que será necesario tener en cuenta esta variabilidad cuando se construya un modelo para rutinas de más alto nivel que utilicen la operación.

Modelo para las comunicaciones

En el modelo clásico de Hockney y en el modelo DLAM+, para modelar el coste de las comunicaciones se propone que el coste para transferir n palabras entre dos procesadores quede definido inicialmente por la siguiente ecuación:

$$T_{comm}(n, t_s, t_w) = t_s + nt_w \tag{2.1}$$

Siendo t_s el tiempo de inicio de una comunicación y t_w el tiempo para enviar un mensaje del tamaño de una palabra. De forma similar a lo realizado en el modelo de cómputo, es necesario plantear una serie de consideraciones con el fin de que el modelo de comunicaciones pueda reflejar adecuadamente el coste de la comunicación de una rutina cuando se ejecuta en una plataforma paralela:

- El modelo clásico de Hockney permite modelar adecuadamente una comunicación entre dos procesadores, y obtener el coste de una comunicación del tipo ping-pong. En el caso de comunicaciones colectivas, en el que intervienen más de dos procesadores, el tiempo de inicio de una comunicación y el tiempo para enviar un mensaje serán funciones del número de procesos, tal y como ya se planteaba en la propuesta de Xu y Hwang [XH96] y en DLAM+. La dependencia respecto al número de procesos vendrá fijada por la topología de la red de interconexión de la plataforma paralela y por el algoritmo implementado en la operación colectiva.
- Al igual que en los estudios realizados por otros autores (Getov *et al.* [GHH97], Kielmann *et al.* [KBG+01], Cameron y Sun [CS03]) y en DLAM+, los valores de t_s y t_w se considerarán variables en función del tamaño del mensaje a enviar y del patrón de acceso a los datos.
- Dado que las librerías de paso de mensajes utilizan en las operaciones punto a punto un protocolo de dos niveles [GLS99] en el que se selecciona el método de envío en función del tamaño del mensaje, y que en las comunicaciones colectivas se pueden implementar diferentes algoritmos de comunicación [KdSF+00], será necesario construir varios modelos para un mismo tipo de operación de comunicación y seleccionar uno u otro según el tamaño de mensaje enviado.
- Al tratarse de un modelo general para el coste de las comunicaciones, independiente de la librería de paso de mensajes y de la plataforma de cómputo paralelo, será de aplicación

a la hora de seleccionar la librería de paso de mensajes a utilizar en una plataforma que disponga de varias implementaciones.

El modelo de sistema descrito será el conjunto del modelo para la computación y del modelo para la comunicación anteriormente expuestos, y estará formado por los parámetros $k_{i,operacion}$ relacionados con las operaciones de computación, y por los parámetros t_s y t_w relacionados con cada una de las operaciones de comunicación.

2.4 Modelado de rutinas de álgebra lineal densa

Como se ha comentado en la sección anterior, el propósito de este trabajo es el de construir una metodología que, a través de la creación de modelos de tiempo de ejecución de las rutinas de álgebra lineal densa, permita minimizar los tiempos de ejecución de dichas rutinas y dotarlas de la capacidad de optimización automática. La propuesta de modelado debe por tanto reflejar las capacidades de cómputo y de comunicación de la plataforma, así como la utilización que el algoritmo implementado hace de esas capacidades. En esta sección se mostrará la formulación empleada en el modelo DLAM+ para el modelado de rutinas paralelas de álgebra lineal, y que ha servido de punto de partida para el desarrollo de una propuesta de mejora en el modelado de rutinas paralelas de álgebra lineal densa. El tiempo de ejecución de una rutina se puede modelar como:

$$T_{ejec} = f(n, SP, AP) \tag{2.2}$$

siendo en la anterior ecuación:

- n: tamaño del problema a resolver.
- SP: parámetros del sistema. Es el conjunto de parámetros descritos en la sección anterior y que reflejan las características del hardware de la plataforma paralela y del software básico al ejecutar una rutina paralela de álgebra lineal.
- AP: parámetros del algoritmo. Conjunto de parámetros cuyo valor se podrá escoger en el momento de ejecutar la rutina. En el caso de rutinas de álgebra lineal densa los parámetros típicos son el tamaño de bloque en la computación, el tamaño de bloque en la comunicación, el número de procesadores a utilizar y algunos parámetros que describan la topología lógica de los procesos.

Partiendo de las consideraciones realizadas en la sección anterior sobre los parámetros del sistema, habrá que tener en cuenta su dependencia del tamaño del problema y de los valores

considerados para los AP en el momento de ejecución de la rutina. Por tanto, no podrán considerarse valores constantes, sino que se relacionarán de la forma:

$$SP = g(n, AP) \tag{2.3}$$

La obtención de los valores de los parámetros del sistema y su dependencia respecto a los AP se realizará en tiempo de instalación, por medio de rutinas específicas que utilizarán las operaciones aritméticas y de comunicación básicas empleadas en el algoritmo y siguiendo el mismo esquema de acceso a los datos usado en la rutina a modelar. Por ejemplo, si la operación aritmética básica asociada a un parámetro del sistema actualiza bloques rectangulares contiguos de tamaño $\frac{m}{b_{compu}} \times n$ de una matriz de tamaño $m \times n$, en el procedimiento de obtención de valores experimentales la rutina básica se ejecutará sobre cada bloque contiguo con el fin de determinar las posibles variaciones en su valor respecto al tamaño del problema, posición en memoria y parámetros del algoritmo. Lo mismo ocurre con las comunicaciones: así, por ejemplo, si el algoritmo distribuye dos matrices utilizando una rutina de difusión y en otra parte utiliza envíos punto-a-punto, se obtendrá un par de valores (t_s, t_w) para la difusión y otro par de valores para la comunicación punto-a-punto, y además se deberá tener en cuenta que los valores de t_s y t_w para el envío de una matriz completa pueden ser distintos que para el de envío de bloques dentro de una matriz.

Al igual que las propuestas realizadas por Dackland y Kågström [DK96], Cuenca *et al.* [CGG04b] y por Yamamoto [Yam06], se aprovechará la estructura jerárquica de las librerías de álgebra lineal, de tal forma que el modelo de tiempo de ejecución de una rutina perteneciente a una librería de alto nivel se construirá partiendo de los modelos de las rutinas a las que llama en su código y que previamente se han modelado utilizando los parámetros básicos del sistema.

En los siguientes capítulos se mostrará la aplicación de esta formulación a diferentes rutinas de álgebra lineal, así como las mejoras en el modelado y las técnicas de remodelado que se proponen en este trabajo, y los resultados obtenidos aplicando el modelo para obtener rutinas con capacidad de optimización automática en diferentes plataformas paralelas y con diferentes librerías básicas de álgebra lineal y de comunicaciones por paso de mensajes.

2.5 Trabajos relacionados

La complejidad existente en la construcción de modelos que permitan optimizar el comportamiento del software cuando se ejecuta en plataformas paralelas ha motivado la aparición de un gran número de propuestas en este campo de investigación. En este capítulo se han mostrado alguna de ellas y, aunque la exposición no ha sido exhaustiva, sirve para dar una idea de la dificultad de encontrar una respuesta general a esta problemática. En este trabajo nos hemos centrado en la construcción de modelos de rutinas paralelas de álgebra lineal densa como

medio para la obtención de rutinas con capacidad de ajuste automático. Se mostrarán a continuación propuestas que están relacionadas, y posteriormente se realizarán algunos comentarios comparativos:

- **PHiPAC** [BACD97] utiliza un conjunto de algoritmos parametrizados de álgebra lineal para la multiplicación de matrices. Para cada algoritmo se realiza una búsqueda especificada previamente sobre el conjunto de valores que pueden adoptar los parámetros con el fin de encontrar la implementación óptima.

- En **ILIB** [KKK00] para cada rutina objeto de estudio (resolución de sistemas de ecuaciones, tridiagonalización de matrices en problemas de valores propios...), se extrae de su algoritmo una serie de puntos clave en los que se puede tomar una decisión, como, por ejemplo, grado de desenrrollado en un bucle o tipo de comunicación a utilizar. La búsqueda del valor óptimo de cada parámetro se realiza mediante ejecuciones de la rutina completa para los posibles valores de este parámetro, manteniendo el resto del conjunto de parámetros a un valor constante.

- En **FIBER** [KKHY03] se utilizan funciones polinomiales para aproximar el tiempo de ejecución de una rutina de álgebra lineal. La función polinomial se obtiene realizando ejecuciones de la rutina con uno de los parámetros fijo (tamaño del problema) y variando otro, por ejemplo desenrrollado de un bucle. A partir de los valores obtenidos por dicha función para distintos tamaños de problema, se obtiene otra función polinomial con la que se podrá realizar la selección óptima de los parámetros de la rutina.

- **OSKI** [VDY05] utiliza un generador de código parametrizado que encapsula posibles estrategias de ajuste (desenrrollado de bucles, variantes del algoritmo, tamaño de bloque...) para la multiplicación de matrices. En cada plataforma computacional de interés, la rutina se ajusta mediante búsqueda en un conjunto de valores predefinidos para los parámetros, se realizan ejecuciones de las rutinas y se selecciona la implementación más rápida.

En nuestra propuesta combinamos técnicas de modelado analítico, junto con el estudio experimental del comportamiento de la rutina en una plataforma de cómputo paralelo determinada. A diferencia de propuestas como PHiPAC o ILIB, en la que los ajustes se realizan para cada rutina, se utilizará una metodología que se aplica a todo el conjunto de rutinas paralelas de álgebra lineal, que, además, en ningún caso verán modificado su código original. Se ha combinado la utilización de modelos analíticos con la utilización de funciones polinomiales, de forma similar a lo realizado en FIBER, para aquellos casos en los que no se pueda obtener un modelo analítico de tiempo de ejecución de la rutina o cuando la información obtenida de los modelos de las rutinas de bajo nivel no proporcione resultados óptimos.

2.6 Resumen y conclusiones

En este capítulo se ha realizado una descripción del modelado de rutinas paralelas de álgebra lineal densa que emplearemos como punto de partida para nuestra propuesta de mejoras en la construcción de modelos con los que se pueda obtener el tiempo de ejecución teórico en una plataforma paralela, y que son de utilidad en la obtención automática de los valores de los parámetros ajustables de una rutina.

El capítulo comienza mostrando los antecedentes sobre el modelado en computación paralela y las diferentes aproximaciones existentes al problema de obtención de modelos de tiempo de ejecución de programas paralelos. Se ha continuado el capítulo describiendo nuestra propuesta en el campo del modelado de rutinas de álgebra lineal densa, con el propósito de proporcionar a este tipo de software de la capacidad de optimización automática, y se han mostrado características que diferencian nuestro trabajo de otras propuestas para la obtención de rutinas de álgebra lineal densa con capacidad de optimización automática. El modelo que se ha presentado consta de un conjunto de parámetros que permiten determinar la influencia de las características de la plataforma paralela, entendida ésta como la combinación del hardware y del software básico, en el tiempo de ejecución del software de álgebra lineal.

En los siguientes capítulos se describirá la metodología para la obtención del modelo analítico de tiempo de ejecución que será la herramienta principal para el ajuste de rutinas paralelas de álgebra lineal que se plantea en este trabajo. De igual forma, se detallarán las diferentes estrategias seguidas para la obtención de los parámetros del sistema y su dependencia respecto de la plataforma hardware, de las librerías básicas que se utilicen, del tamaño del problema y de las características propias del algoritmo utilizado en la rutina paralela de álgebra lineal.

Capítulo 3

Mejoras en el modelado de rutinas de álgebra lineal densa

3.1 Introducción

En este capítulo se muestra nuestra propuesta de mejoras en el modelado de rutinas de álgebra lineal que permite tener una estimación aproximada del tiempo de ejecución de estas rutinas. Como se ha visto en el capítulo anterior, partimos de una formulación del modelo en la que aparecen reflejados, el tamaño del problema a resolver, las características del sistema paralelo de cómputo a la hora de ejecutar rutinas paralelas de álgebra lineal a través de un conjunto de parámetros denominados parámetros del sistema (SP) y, por otro lado, una serie de parámetros ajustables denominados parámetros del algoritmo (AP). A partir de este planteamiento básico, se han realizado una serie de consideraciones que permiten introducir mejoras en el modelado del tiempo de ejecución de rutinas paralelas de álgebra lineal. Los modelos así obtenidos permitirán tomar una serie de decisiones, previas a la ejecución de las rutinas, encaminadas a la obtención de tiempos de ejecución óptimos en una plataforma dada.

En primer lugar, se detallarán las distintas rutinas que se han utilizado en los experimentos y se mostrará la aplicación de las mejoras propuestas en el modelo analítico de su tiempo de ejecución. Se efectuarán para cada una de las rutinas modeladas una serie de consideraciones que deben ser tenidas en cuenta cuando se plantea la ecuación que define el modelo de tiempo de ejecución y las decisiones que deberán ser tomadas para la obtención de los parámetros asociados al modelo, con el fin de obtener una mejor aproximación al tiempo de ejecución de una rutina paralela de álgebra lineal. Para cada una de las rutinas analizadas, sobre diversas plataformas hardware, se mostrará una comparación entre su tiempo de ejecución experimental y la aproximación obtenida utilizando el modelo propuesto. Conjuntamente a dicha comparación de tiempos, se mostrará que mediante la utilización del modelo propuesto es posible realizar de

forma satisfactoria el ajuste automático de la rutina y obtener tiempos de ejecución cercanos al óptimo.

3.2 Parámetros del modelo de tiempo de ejecución

El modelo básico de una rutina paralela de álgebra lineal se ha planteado tradicionalmente como un coste teórico para la computación junto a un coste teórico para la comunicación, con un único parámetro asociado con el coste de la computación, y un par de parámetros asociados con el coste de la comunicación: la latencia o tiempo de inicio de la comunicación, y el tiempo en enviar una palabra. Cuando el tamaño del problema es lo suficientemente grande y se ha realizado una estimación adecuada del valor del parámetro asociado al coste en la computación y al valor para cada uno de los dos parámetros asociados al coste en la comunicación, la aproximación al tiempo de ejecución de una rutina utilizando el modelo clásico suele dar resultados aceptables. Pero si se requiere de un modelado del tiempo de ejecución para los diferentes niveles de rutinas de álgebra lineal, y que sea de aplicación en diferentes plataformas computacionales y para diferentes tamaños de problema, es necesario plantear una formulación más detallada del modelo que sea capaz de reflejar las variaciones en los parámetros del sistema y su dependencia respecto a los parámetros del algoritmo.

En nuestro planteamiento se considera que el valor de los parámetros del sistema puede variar con el tamaño del problema a resolver, y por lo tanto puede ser necesario que el modelo se descomponga en varias partes con el propósito de tener en cuenta la variación del coste de las operaciones aritméticas con el tamaño y forma de la matriz en la se aplica la rutina. De esta forma se obtendrá un valor distinto del parámetro asociado con el coste aritmético en cada una de las partes en las que el modelo inicial haya sido descompuesto. Por ejemplo, si el coste aritmético de una rutina se modela inicialmente como $\frac{2}{3}kn^3$, puede ser conveniente descomponer el modelo en tres partes y expresar el coste de la rutina como $\frac{2}{9}k_{\frac{n}{6}}n^3 + \frac{2}{9}k_{\frac{3n}{6}}n^3 + \frac{2}{9}k_{\frac{5n}{6}}n^3$, donde $k_{\frac{in}{6}}$ sería el valor del parámetro del sistema asociado con una operación aritmética cuando opera con un tamaño de problema entre $\frac{(i-1)n}{6}$ y $\frac{(i+1)n}{6}$.

En las siguientes secciones se describirán las rutinas de álgebra lineal en las que se han aplicado las mejoras propuestas para la construcción de su modelo de tiempo de ejecución. Para cada rutina, en primer lugar, se expondrá su base teórica; posteriormente, se indicará el algoritmo que sigue y, finalmente, se mostrará la construcción de su modelo analítico de tiempo de ejecución. Las rutinas empleadas para mostrar nuestra propuesta pertenecen a los distintos niveles de la jerarquía clásica de librerías de álgebra lineal:

- Rutinas básicas secuenciales y paralelas:
  - Diferentes versiones de la multiplicación de matrices.

- Rutinas de mayor nivel secuenciales y paralelas:
  - Multiplicación de matrices con el algoritmo de Strassen.
  - Diferentes versiones de la factorización de Cholesky.

3.3 Multiplicación de matrices

La multiplicación de matrices es uno de los núcleos más importantes en aplicaciones científicas. Paquetes de software como LAPACK y ScaLAPACK están basados en el conjunto de rutinas de BLAS, del que forma parte fundamental la multiplicación de matrices (GEMM). La posibilidad de obtener con la rutina GEMM un rendimiento cercano al pico teórico de la computadora, aplicando técnicas como prefetching, loop unrolling o cache blocking [GH01], que aprovechan la jerarquía de memorias de las computadoras actuales, supuso por un lado la reescritura de las rutinas principales de LAPACK en términos de multiplicación entre bloques de matrices [AD89] y, por otro, que los principales vendedores de computadoras, cada vez que ofertaban una nueva plataforma de cálculo al mercado, tuvieran que proporcionar la correspondiente versión de BLAS con la rutina de GEMM específicamente optimizada.

La multiplicación paralela de matrices, al igual que la secuencial, sigue siendo objeto de estudio y en las últimas décadas se han propuesto diferentes enfoques con el objetivo de obtener implementaciones eficientes. Entre estas propuestas pueden citarse el algoritmo de Cannon [Can69], el algoritmo de Fox o de *broadcast-multiply-roll* [FOH87, FJL+88] y el algoritmo DNS [DNS81]. En PUMMA [CDW94] se implementa una variante por bloques y cíclica del algoritmo de Fox para mallas no cuadradas de procesos. SUMMA [dGW97] se utiliza en la rutina PDGEMM de PBLAS [CDW96] y continúa las ideas desarrolladas por Agarwal et al. [AGZ94] mostrando que una secuencia de actualizaciones de rango-k es altamente efectiva para la paralelización de la multiplicación y que el tiempo total de cálculo puede reducirse por medio del solapamiento de computaciones con comunicaciones. También en DIMMA [Cho97] se aplica la técnica de solapamiento, pero combinándola con la elección de un tamaño de bloque para la computación que permita obtener un rendimiento óptimo en las multiplicaciones locales realizadas con BLAS. Otros autores proponen reducir el número de comunicaciones utilizando distribuciones de procesadores en tres dimensiones [ABG+95] o reducir el coste de la computación utilizando la multiplicación de Strassen [Str69] localmente en cada nodo. En [GSdG95] se propone paralelizar el esquema de multiplicación recursivo de Strassen y en [NT04] seleccionar el mejor algoritmo de multiplicación entre varios en función del tamaño del problema o de los recursos disponibles.

Esta diversidad de algoritmos e implementaciones para la multiplicación de matrices, tanto en secuencial como en paralelo, y el que no haya un único algoritmo con el que siempre se obtenga el menor tiempo de ejecución para un sistema y problema en particular, justifica el abordar conjuntamente el problema de obtención de rutinas con optimización automática junto con el

de selección automática de algoritmos. En este apartado se discuten cinco algoritmos paralelos y uno secuencial de la multiplicación de matrices densas A y B para obtener el producto $C = AB$, y se aplica la técnica de optimización automática por medio de modelos teóricos de tiempo de ejecución parametrizados con determinación experimental de parámetros óptimos. El objetivo es obtener una multiplicación de matrices que sea adaptable en tiempo de instalación y de ejecución a través de los parámetros del algoritmo y con implementación múltiple, de tal forma que, en función de las características hardware de la plataforma y de las librerías básicas disponibles, se seleccione la que proporciona tiempos de ejecución óptimos. La efectividad del procedimiento se analiza a través de datos experimentales en diferentes plataformas. Adicionalmente, y con el fin de mostrar la influencia que el esquema de acceso a los datos y el tipo de rutina utilizada tiene sobre el valor de los SP de computación y de comunicación, se han incluido en el análisis algoritmos paralelos de la mayor sencillez posible para la multiplicación de matrices (algoritmo BSR y CBSR).

Los algoritmos que se han analizado son:

- **BSR**: Distribución Block-Striped de la matriz A y la matriz B Replicada en todos los procesos.
- **CBSR**: Distribución Checkerboard Block-Striped de la matriz A y la matriz B Replicada en todos los procesos.
- **Cannon**: Algoritmo de Cannon.
- **Strassen**: Algoritmo de Strassen secuencial.
- **StPBLAS**: Algoritmo de Strassen paralelo con rutinas PBLAS (**pdgemm** y **pdaxpy**)
- **StMS**: Algoritmo de Strassen paralelo con un esquema Maestro-Esclavo para las comunicaciones.

Los algoritmos paralelos analizados se comparan con la rutina **pdgemm** disponible en PBLAS y el secuencial con la rutina **dgemm** disponible en las librerías BLAS y ATLAS.

3.3.1 Obtención de valores experimentales

Los tiempos de ejecución que aparecen en este trabajo han sido obtenidos de la forma que se describe a continuación. Para matrices con tamaños igual o inferior a 1024 y, con el fin de minimizar los posibles errores al tomar la medida del tiempo, se calcula el promedio de 10 ejecuciones distintas de la rutina, descartando el valor mayor y el menor. Los tiempos de ejecución para matrices mayores son lo suficiente grandes, por lo que no ha sido necesaria la obtención de valores medios y, con el fin de reducir el error en la obtención de los valores experimentales, se han ejecutado los experimentos indicados tres veces para cada tamaño de matriz, y se ha utilizado el valor mínimo.

3.3.2 Descomposición en bloques de datos

En los algoritmos BSR, CBSR y Cannon se ha seguido el siguiente procedimiento, descrito en [Qui03], para asignar a procesos bloques contiguos de datos de las matrices. Si n es el número de elementos y p es el número de procesos, el primer elemento asignado al proceso i es $\lfloor \frac{in}{p} \rfloor$ y el último elemento asignado al proceso i es el elemento inmediatamente anterior al primer elemento asignado al proceso $i+1$, esto es, $\lfloor (i+1)\frac{n}{p} \rfloor - 1$. De esta forma, si tenemos 4 procesos (del 0 al 3) y 14 elementos, al proceso 0 le corresponderán 3 elementos, al proceso 1 le corresponderán 4 elementos, al proceso 2 le corresponderán 3 elementos y al proceso 3 le corresponderán 4 elementos. Para las rutinas **pdgemm** y StPBLAS se utiliza una distribución de las matrices cíclica por bloques [LJ93], al ser la utilizada en PBLAS.

3.3.3 Versión rowwise block-striped con replicación de la matriz B (BSR)

En esta sección se analiza un algoritmo para la multiplicación paralela de matrices con una descomposición de la matriz A en la que cada uno de los p procesos es responsable de un grupo contiguo de $\lfloor \frac{m}{p} \rfloor$ o $\lceil \frac{m}{p} \rceil$ filas de la matriz, y B es replicada en los p procesos (figura 3.1). Para la distribución de la matriz A se ha programado una rutina en la que el proceso 0 a partir de la matriz A forma bloques de tamaño $\frac{m}{p}k$ y envía estos bloques con **MPI_Send** al resto de los $p-1$ procesos. La matriz B es distribuida, también por el proceso 0, con un **MPI_Bcast** de tamaño kn. Una vez distribuidas A y B, cada proceso es responsable de realizar la multiplicación A_iB, donde A_i es de tamaño $\frac{m}{p}k$ y B es de tamaño kn, para lo cual se ha utilizado la rutina **dgemm** de BLAS nivel 3.

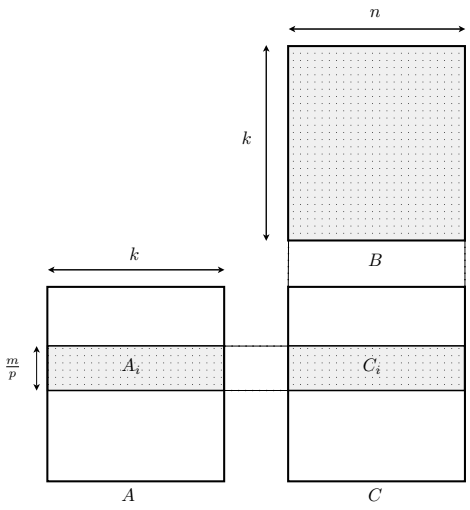


Figura 3.1: Descomposición de las matrices A, B y C.

En lo que se refiere a las necesidades de memoria, cada proceso requiere $\frac{m}{p}k$ elementos de memoria para los bloques de A y kn para la matriz B, además de $\frac{m}{p}k$ para el bloque resultante de C. Por lo tanto los p procesos necesitarán un total de $mk+knp+mn$ elementos de memoria.

A continuación se muestra el modelo de tiempo de ejecución de la rutina, en el que aparece reflejado el coste de las comunicaciones, que viene dado por el tiempo empleado en la distribución de las matrices A y B y por el coste computacional:

- **Distribución de A:**

$$t_s + \frac{m}{p}kt_w \tag{3.1}$$

- **Distribución de B:**

$$t_s + knt_w \tag{3.2}$$

- **Computación:**

$$\frac{2mkn}{p}k_{3,dgemm} \tag{3.3}$$

A continuación se mostrará, para diferentes plataformas computacionales, la obtención de los SP que aparecen en el modelo, y las consideraciones realizadas en la obtención de sus valores. También se comparará el tiempo teórico según el modelo propuesto y el tiempo de ejecución que realmente se obtiene. Adicionalmente, se comprobará que mediante la utilización del modelo se realiza una selección adecuada de los AP, lo que conduce a tiempos de ejecución de la rutina cercanos al óptimo.

Resultados experimentales

El único SP asociado con las operaciones de computación es el coste de la rutina **dgemm** de BLAS nivel 3, $k_{3,dgemm}$, utilizada para la multiplicación en cada proceso de un bloque de A de tamaño $\frac{m}{p}k$ por la matriz B de tamaño kn. Experimentalmente se ha comprobado que su valor no depende del tamaño del bloque de la matriz A y sí en cambio, del tamaño del problema. En la tabla 3.1 se muestran los valores del SP $k_{3,dgemm}$ para P4net y HPC160smp, obtenidos experimentalmente ejecutando la rutina **dgemm** con la versión de BLAS disponible en cada plataforma y empleando el mismo esquema de acceso a las matrices utilizado en la rutina paralela de multiplicación de matrices.

Respecto a los SP asociados con la comunicación (t_s, t_w), sólo se dispone del coste para la distribución de las matrices A y B, y el valor de t_s y el de t_w, será distinto según el tipo de rutina básica empleada en la comunicación de las matrices A y B, y para cada rutina, el valor de t_s y t_w podrá variar con el número de procesos y el tamaño del mensaje. En los experimentos realizados con la rutina de difusión (*broadcast*) de MPI los valores medidos han mostrado una dependencia respecto al número de procesos p involucrados en la comunicación de

la matriz B y se han mostrado constantes para los tamaños de mensaje que se manejan en esta rutina. Por otra parte, los valores de t_s y t_w asociados con las comunicaciones punto-a-punto se han obtenido mediante una rutina *ping-pong*, y los valores medidos dependen del tamaño del mensaje a enviar, que en este caso viene dado por el número de bloques p en el que se divide la matriz A.

En la tabla 3.2 se muestran los valores para los sistemas P4net y HPC160smp. Como se observa en la tabla, el comportamiento de las rutinas MPI es distinto en cada una de las plataformas. En P4net, cuando el número de procesos, p, es igual a 2, el coste de las comunicaciones es mayor usando la rutina de difusión que la rutina punto a punto, mientras que para $p = 4$ ocurre lo contrario. Por el contrario, en la plataforma HPC160smp siempre es menor el coste de las comunicaciones utilizando la rutina de difusión de MPI (con valores para t_s de 0 debido a las cifras decimales usadas). Estos resultados indican un mejor comportamiento del algoritmo de difusión de datos para la implementación de MPI en la plataforma HPC160smp que la implementación de MPI en P4net, para los tamaños de mensaje utilizados en las ejecuciones de la rutina.

	Tamaño del problema, n				
Sistema	512	1024	2048	4096	5120
P4net	0.00050	0.00043	0.00040	0.00039	0.00039
HPC160	0.00059	0.00057	0.00057	0.00061	0.00056

Tabla 3.1: Valores de $k_{3,dgemm}$ (en μsegundos) con BLAS optimizada, en P4net y HPC160smp, para varios tamaños de problema.

		Envío bloques		MPI_Bcast	
Sistema		t_s	t_w	t_s	t_w
P4net	$p = 2$	214	0.83	220	1.05
	$p = 4$	527	2.28	556	1.57
HPC160smp	$p = 2$	0.000	0.076	0.000	0.028
	$p = 4$	0.000	0.154	0.000	0.062

Tabla 3.2: Valores de t_s y t_w (en μsegundos), en P4net y HPC160smp, para la distribución de matrices en la rutina BSR.

Con el fin de comprobar la validez del modelo propuesto que aproxima el tiempo de ejecución de la rutina, se han realizado ejecuciones para diferentes tamaños de problema y número de procesos. En la tabla 3.3 se muestra una comparación para la plataforma P4net y en la tabla 3.4 para la plataforma HPC160smp entre los tiempos obtenidos ejecutando la rutina (tiempo experimental) y los tiempos proporcionados por el modelo (tiempo teórico) para las comunicaciones, la computación y el total, así como la desviación entre el valor experimental y el teórico calculada como $\frac{|t_{mod} - t_{exp}|}{t_{exp}}$.

	$p = 2$				$p = 4$			
n	distri A	distri B	computatión	total	distri A	distri B	computatión	total
	Tiempo experimental (segundos)							
512	0.1088	0.2735	0.0695	**0.4518**	0.1493	0.4115	0.0440	0.6048
1024	0.4356	1.0945	0.4526	**1.9827**	0.5974	1.6439	0.2243	2.4656
2048	1.7391	4.5528	3.5015	**9.7935**	2.3860	6.5824	1.6880	10.6563
4096	6.9244	17.5718	25.8593	50.3555	9.5627	26.3237	12.9001	**48.7865**
5120	10.7945	27.5764	51.7170	90.0880	14.9420	41.1789	24.8259	**80.9469**
	Tiempo teórico (segundos)							
512	0.1090	0.2755	0.0677	**0.4522**	0.1500	0.4121	0.0338	0.5959
1024	0.4354	1.1012	0.4637	**2.0003**	0.5982	1.6468	0.2318	2.4769
2048	1.7409	4.4042	3.4582	**9.6033**	2.3913	6.5856	1.7291	10.7060
4096	6.9628	17.6163	26.8829	51.4620	9.5636	26.3408	13.4414	**49.3458**
5120	10.8792	27.5253	51.8017	90.2063	14.9428	41.1572	25.9009	**82.0008**
	Desviación (%)							
512	0.16	0.73	2.57	0.09	0.44	0.16	23.11	1.47
1024	0.05	0.61	2.44	0.89	0.15	0.18	3.38	0.46
2048	0.10	3.26	1.24	1.94	0.22	0.05	2.44	0.47
4096	0.55	0.25	3.96	2.20	0.01	0.06	4.20	1.15
5120	0.78	0.19	0.16	0.13	0.00	0.05	4.33	1.30

Tabla 3.3: Comparación del tiempo de ejecución experimental y teórico (en segundos), en P4net, para la rutina rowwise block-striped (BSR).

	$p = 2$				$p = 4$			
n	distri A	distri B	computatión	total	distri A	distri B	computatión	total
	Tiempo experimental (segundos)							
512	0.0088	0.0068	0.0811	0.0967	0.0098	0.0156	0.0430	**0.0684**
1024	0.0400	0.0283	0.6250	0.6933	0.0371	0.0654	0.3701	**0.4726**
2048	0.1621	0.1162	4.8925	5.1708	0.1553	0.2578	2.5439	**2.9570**
4096	0.6387	0.4805	39.4365	40.5556	0.6377	1.0693	20.2622	**21.9692**
5120	0.9814	0.7383	75.3955	77.1152	1.0185	1.6630	37.9365	**40.6181**
	Tiempo teórico (segundos)							
512	0.0100	0.0073	0.0791	0.0964	0.0101	0.0164	0.0395	**0.0661**
1024	0.0400	0.0291	0.6162	0.6853	0.0405	0.0657	0.3081	**0.4142**
2048	0.1602	0.1162	4.9359	5.2123	0.1619	0.2626	2.4680	**2.8925**
4096	0.6406	0.4648	41.7006	42.8060	0.6475	1.0506	20.8503	**22.5484**
5120	1.0010	0.7263	75.4971	77.2243	1.0117	1.6416	37.7485	**40.4018**
	Desviación (%)							
512	13.89	6.26	2.41	0.32	3.60	5.07	7.96	3.33
1024	0.00	2.59	1.41	1.16	9.05	0.36	16.75	12.36
2048	1.20	0.00	0.89	0.80	4.25	1.88	2.98	2.18
4096	0.31	3.25	5.74	5.55	1.54	1.75	2.90	2.64
5120	1.99	1.62	0.13	0.14	0.67	1.29	0.50	0.53

Tabla 3.4: Comparación del tiempo de ejecución experimental y teórico (en segundos), en HPC160smp, para la rutina rowwise block-striped (BSR).

Como puede observase, con el modelo se obtiene una buena aproximación al tiempo de ejecución real de la rutina (salvo para tamaños de problema pequeños), con desviaciones para el tiempo total de ejecución entre un 0.09 % y un 2.20 % en P4net y entre un 0.14 % y un 12.36 % en HPC160smp. Adicionalmente, podemos también observar que con el modelo de tiempo de ejecución se pueden tomar decisiones acerca de los AP de esta rutina (en este caso el número de procesos p paralelos) con los que se obtienen tiempos de ejecución óptimos. En las tablas se han resaltado dichos tiempos óptimos y se puede comprobar que tanto los resultados experimentales como los teóricos muestran que en P4net para tamaños de problema pequeños ($n < 4096$) es preferible utilizar sólo 2 procesos y para tamaños de problema $n \geq 4096$ se consiguen mejores tiempos de ejecución utilizando 4 procesos. En HPC160smp siempre es preferible ejecutar la rutina utilizando 4 procesos, al disponer esta plataforma de una red de interconexión más rápida, y por tanto, el coste de las comunicaciones adquirir menos peso que el de la computación en todos los casos.

Podemos concluir que con nuestra propuesta de modelado se consigue una buena aproximación al tiempo de ejecución real de la rutina, y que por tanto se pueden tomar decisiones acertadas sobre cuales son los AP más aconsejables a la hora de ejecutar la rutina en diferentes plataformas y para distintos tamaños de problema. Por ejemplo, en P4net se observa que no siempre es aconsejable utilizar todos los procesadores disponibles, siendo la reducción del tiempo de ejecución promedio de un 6.4 %.

3.3.4 Versión checkerboard block con replicación de la matriz B (CBSR)

Esta versión de la multiplicación paralela de matrices utiliza una descomposición de la matriz A en una malla de $p = r \times c$ procesos en la que cada uno de los p procesos almacena localmente un bloque de la matriz A de tamaño $\lceil \frac{m}{r} \rceil \lceil \frac{k}{c} \rceil$, y la matriz B es replicada en los p procesos. El proceso $\{0,0\}$ distribuye, a los procesos en su misma columna, el panel correspondiente de tamaño $\lceil \frac{m}{r} \rceil k$, luego cada proceso distribuye por columnas los bloques de A de tamaño $\lceil \frac{m}{r} \rceil \lceil \frac{k}{c} \rceil$. En ambos casos se emplea la rutina **MPI_Send**. La matriz B es distribuida, también por el proceso $\{0,0\}$, con un **MPI_Bcast** de tamaño kn al resto de procesos.

El proceso $\{i,j\}$, responsable de calcular los elementos de $C_{i,j}$ del producto de matrices, requiere acceder a todos los bloques de la fila i de la matriz A, por lo que antes de realizar la multiplicación es necesario realizar una acumulación del bloque local de A entre todos los procesos en la misma fila, operación realizada con la rutina **MPI_Allgatherv** (figura 3.2). Luego cada proceso $\{i,j\}$ realiza la multiplicación $A_i B_j$ donde A_i es de tamaño $\frac{m}{r}k$ y B_j es de tamaño $k\frac{n}{c}$, para lo que se ha utilizado la rutina **dgemm** de BLAS nivel 3. Cada proceso necesita $\frac{m}{r}\frac{k}{c}$ elementos de memoria para los bloques de A y kn para la matriz B, además de $\frac{m}{r}\frac{k}{c}$ para el bloque resultante de C. De tal forma que los p procesos necesitan en total $mk+knp+mn$ elementos de memoria.

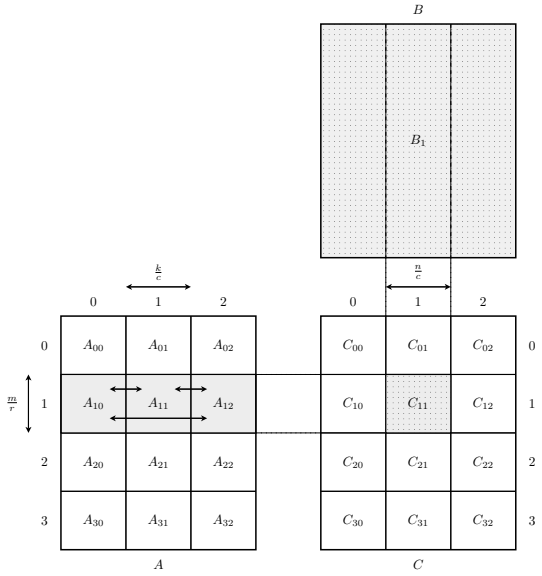


Figura 3.2: Descomposición de las matrices A, B y C en una malla de 4×3 procesos para el algoritmo checkerboard block (CBSR). Las flechas en las áreas sombreadas de la matriz A indican la acumulación de bloques de A previa a la multiplicación. Los números a la izquierda/derecha y arriba de las matrices A y C representan coordenadas de los procesos en la malla de 4×3.

Como ya se ha indicado, esta versión de la multiplicación paralela de matrices introduce un coste adicional en las comunicaciones, debido a que es necesario acumular los bloques de la matriz A en cada proceso. Por lo tanto se trata de una rutina con una mayor coste que la rutina BSR analizada con anterioridad. El motivo de incluirla en este trabajo es el de validar nuestra propuesta de modelado y verificar la necesidad de variar los valores de los SP en función de las rutinas básicas de comunicación empleadas, de tal forma que podamos obtener una buena aproximación al tiempo de ejecución de la rutina.

A continuación se muestra el modelo de tiempo de ejecución según nuestra propuesta de modelado de rutinas paralelas de álgebra lineal. En el modelo se refleja el coste de las comunicaciones, que viene dado por el tiempo empleado en la distribución de las matrices A y B y en la acumulación de los bloques de A, y el coste computacional de la multiplicación que se realiza en cada proceso:

- **Distribución de A**
  - Envío del proceso $\{0, 0\}$ de paneles de tamaño $\lceil \frac{m}{r} \rceil k$ a los $r-1$ procesos en la columna 0:

$$t_s + \frac{mk}{r} t_w \tag{3.4}$$

  - Envío de los procesos en la columna 0 de bloques de A de tamaño $\lceil \frac{m}{r} \rceil \lceil \frac{k}{c} \rceil$:

$$t_s + \frac{mk}{p} t_w \tag{3.5}$$

- **Distribución de B**

$$t_s + knt_w \tag{3.6}$$

- **Comunicación**
  - Acumulación, con la rutina **MPI_Allgatherv**, de los bloques de la matriz A en cada proceso:

$$t_s + \frac{mk}{p}t_w \tag{3.7}$$

- **Computación:**

$$\frac{2mkn}{p}k_{3,dgemm} \tag{3.8}$$

En la siguiente sección se muestra, para diferentes plataformas computacionales, la obtención de los SP que aparecen en el modelo y las consideraciones realizadas en la obtención de sus valores. Posteriormente, y con el fin de comprobar su validez, se comparará el tiempo teórico obtenido con el modelo antes de ejecutar la rutina y el tiempo experimental que resulta de la ejecución de la rutina en cada uno de las plataformas computacionales empleadas en nuestro estudio.

Resultados experimentales

El único SP asociado con las operaciones de computación es el coste de la rutina **dgemm** de BLAS, $k_{3,dgemm}$, para la multiplicación en cada proceso de la malla $r \times c$ de un bloque de A de tamaño $\frac{m}{r}k$ por un bloque de B de tamaño $k\frac{n}{c}$. Experimentalmente se comprobó que su valor no depende del tamaño de los bloques, y si del tamaño del problema, por lo que es posible utilizar en el modelo de tiempo de ejecución los mismos valores que ya se utilizaron en el algoritmo BSR: tabla 3.1 (P4net y HPC160smp).

En lo referente a los SP asociados con la comunicación (t_s, t_w), aparece un coste para la distribución de A, un coste para la replicación de B y un coste para la acumulación de los bloques de A. La distribución de A se realiza con comunicaciones punto-a-punto (rutina **MPI_Send** y **MPI_Recv**) en dos pasos. El primer paso consiste en el envío por columnas de procesos de paneles de A, y se trata del mismo procedimiento empleado en el algoritmo BSR, por lo que se utilizarán los valores de t_s y t_w ya obtenidos para la rutina BSR. El segundo paso consiste en la distribución por columnas de procesos de bloques de A de tamaño $\lceil\frac{m}{r}\rceil\lceil\frac{k}{c}\rceil$, y en los experimentos realizados se ha observado que el valor de t_s es el mismo que en el resto de comunicaciones punto-a-punto ya empleadas, pero sin embargo el valor de t_w medido es diferente, y toma los valores de $t_w = 0.73$ μsegundos en P4net y $t_w = 0.069$ μsegundos en HPC160smp. El procedimiento de replicación de B es el mismo que en el algoritmo BSR por lo que se utilizarán los valores para t_s y t_w ya obtenidos. Finalmente se tiene el coste asociado con la acumulación de bloques de A en cada proceso. La acumulación se realiza de forma simultánea en toda la malla lógica de procesos por medio de la rutina **MPI_Allgatherv**, y con el fin de

detectar posibles variaciones en los valores de los parámetros de comunicación en función de la topología lógica de la malla de $p = r \times c$ procesos, se han realizado mediciones del t_s y del t_w de la rutina **MPI_Allgatherv** para diferentes topologías de la malla de procesos: 1×2, 2×2, etc. El valor de t_s para la rutina **MPI_Allgatherv** ha mostrado un comportamiento constante y se ha comprobado que coincide con el t_s de la rutina **MPI_Bcast**. En cambio, el valor medido de t_w se ha comportado de forma distinta en cada una de las plataformas analizadas. En P4net se observa un comportamiento constante y, tanto para $p = 1 \times 2$ como para $p = 2 \times 2$, $t_w = 1.41$ μsegundos, mientras que en HPC160smp tenemos que $t_w = 0.11$ μsegundos con $p = 1 \times 2$ y $t_w = 0.13$ μsegundos con $p = 2 \times 2$.

	$p = 2 \times 2$			
n	distri A	distri B	comu+compu	total
	Tiempo experimental (segundos)			
512	0.147853	0.410528	0.128020	0.686401
1024	0.586179	1.642962	0.597694	2.826835
2048	2.348223	6.578731	3.156729	12.083683
4096	9.425755	26.388637	18.833914	54.648306
5120	14.742104	41.178493	35.274741	91.195338
	Tiempo teórico (segundos)			
512	0.157071	0.412122	0.126470	0.695663
1024	0.626964	1.646820	0.601686	2.875470
2048	2.506537	6.585613	3.207807	12.299957
4096	10.024827	26.340785	19.355634	55.721245
5120	15.663544	41.157164	35.141658	91.962366
	Desviación (%)			
512	5.87	0.39	1.21	1.35
1024	6.51	0.23	0.67	1.72
2048	6.32	0.10	1.62	1.79
4096	5.98	0.18	2.77	1.96
5120	5.88	0.05	0.38	0.84

Tabla 3.5: Comparación del tiempo (en segundos) de ejecución experimental y teórico, en P4net. Para la rutina checkerboard block (CBSR).

En la tabla 3.5 para P4net y en la 3.6 para HPC160smp se compara el tiempo obtenido experimentalmente con el tiempo proporcionado por el modelo para una malla de $p = 2 \times 2$ procesos. De igual forma que se ha hecho con la rutina BSR, se muestra la aproximación obtenida por el modelo en cada una de las fases consideradas de la rutina, de tal forma que aparece reflejada la capacidad del modelo para estimar correctamente tanto las comunicaciones como la computación. Como puede observarse en las tablas, el modelo proporciona una buena estimación del coste de las comunicaciones y de la computación y por tanto del tiempo de ejecución total de la rutina en ambas plataformas. En el peor de los casos la desviación respecto al tiempo real de ejecución total es tan solo de 1.96 % en P4net y de 7.51 % en HPC160smp.

$p = 2 \times 2$				
n	distri A	distri B	comu+compu	total
	Tiempo experimental (segundos)			
512	0.014648	0.014648	0.050780	0.080076
1024	0.062499	0.063475	0.378896	0.504870
2048	0.206049	0.260735	2.582941	3.049725
4096	0.853493	1.041965	20.334432	22.229890
5120	1.339809	1.630817	38.873013	41.843639
	Tiempo teórico (segundos)			
512	0.014526	0.016416	0.048539	0.080477
1024	0.058104	0.065662	0.343395	0.468157
2048	0.232415	0.262649	2.608499	3.104560
4096	0.929660	1.050592	21.411743	23.392997
5120	1.452594	1.641559	38.625681	41.720830
	Desviación (%)			
512	0.84	10.77	4.85	1.02
1024	7.56	3.33	9.43	7.51
2048	11.34	0.73	0.98	1.76
4096	8.19	0.82	5.30	5.23
5120	7.76	0.65	0.64	0.30

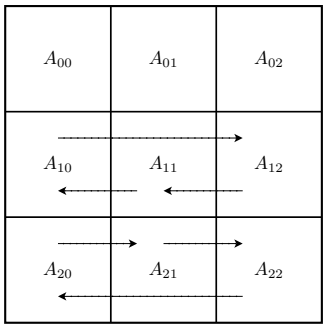
Tabla 3.6: Comparación del tiempo (en segundos) de ejecución experimental y teórico, en HPC160smp. Para la rutina checkerboard block (CBSR).

3.3.5 Algoritmo de Cannon

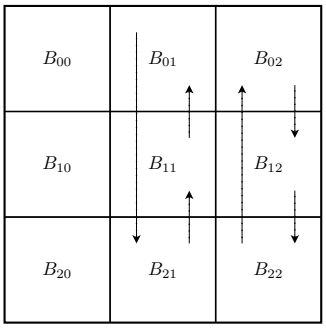
El algoritmo de Cannon para la multiplicación de matrices está basado en una descomposición checkerboard block de las matrices A y B en una malla cuadrada de $\sqrt{p} \times \sqrt{p}$ procesos. Inicialmente se asignan las submatrices $A_{i,j}$ y $B_{i,j}$ de dimensiones $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$ y $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ respectivamente, al proceso $p_{i,j}$. Si bien cualquier proceso en la fila i requiere todas las $\sqrt{p}$ matrices $A_{i,k}$ $(0 \leq k < \sqrt{p})$, es posible planificar las computaciones de los $\sqrt{p}$ procesos de la fila i de tal forma que, en un momento dado, cada proceso utilice un $A_{i,k}$ diferente. Estos bloques son rotados sistemáticamente entre los procesos después de cada computación, de tal forma que cada proceso obtenga el $A_{i,k}$ requerido. El mismo esquema, pero realizado sobre columnas, se aplica a los bloques de la matriz B.

Primero se alinean los bloques de A y B con el fin de que cada proceso pueda multiplicar sus matrices locales. Esta alineación inicial se obtiene desplazando todos los bloques $A_{i,j}$ a la izquierda i pasos y todos los bloques $B_{i,j}$ hacia arriba j pasos; en ambos casos cada fila y cada columna de procesos forma un anillo (figuras 3.3(a) y 3.3(b)). Después de ésta alineación, cada bloque de A se mueve un paso a la izquierda y cada bloque de B se mueve un paso hacia arriba (cíclicamente usando una topología lógica de toro), con lo que todos los procesos pueden volver a multiplicar sus bloques locales. El procedimiento finaliza después de $\sqrt{p} - 1$ desplazamientos y multiplicaciones, tal y como se muestra en las figuras 3.3(c), 3.3(d), 3.3(e) y 3.3(f).

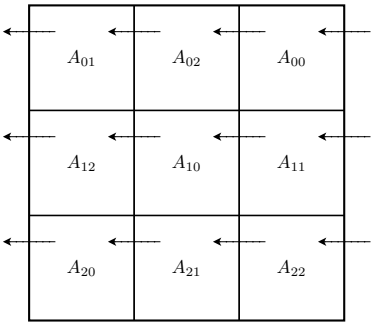
La distribución de las matrices A y B se realiza de igual forma que en la distribución



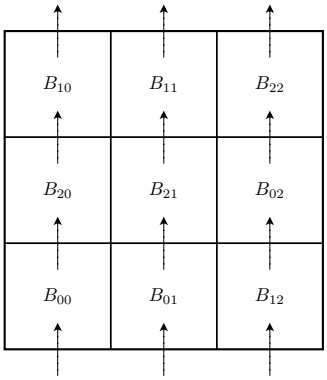
(a) Alineación inicial de la matriz A



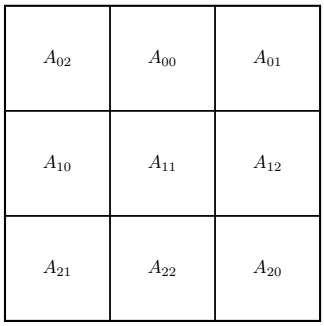
(b) Alineación inicial de la matriz B



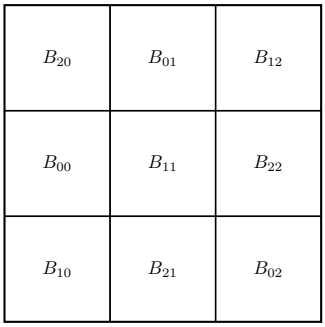
(c) Matriz A después del primer desplazamiento



(d) Matriz B después del primer desplazamiento



(e) Matriz A después del segundo desplazamiento



(f) Matriz B después del segundo desplazamiento

Figura 3.3: Esquema de comunicaciones en el algoritmo de Cannon. Las flechas indican movimiento de los bloques entre procesos.

checkerboard de A vista en el algoritmo anterior (CBSR). La rotación de los bloques de A y de B se realiza con la rutina **MPI_Sendrecv**. Cada proceso necesita $\frac{m}{\sqrt{p}}\frac{k}{\sqrt{p}}$ elementos de memoria para los bloques de A y $\frac{k}{\sqrt{p}}\frac{n}{\sqrt{p}}$ para los bloques de B, además de $\frac{m}{\sqrt{p}}\frac{n}{\sqrt{p}}$ para el bloque resultante de C. Los p procesos necesitan un total de $mk + kn + mn$ elementos de memoria. Por lo tanto los requerimientos de memoria de este algoritmo son independientes del número de procesos p, a diferencia de lo que ocurría con los algoritmos vistos previamente.

Una vez descrito el algoritmo de Cannon, se construye su modelo teórico de tiempo de ejecución en el que aparece reflejado el tamaño del problema, los parámetros seleccionables del algoritmo (AP), y los parámetros del sistema (SP) que definen el coste de las operaciones de comunicación y computación realizadas. El modelo parametrizado de coste de tiempo de ejecución de esta rutina queda definido por:

- **Distribución de A**
  - Envío del proceso $\{0,0\}$ de paneles de tamaño $\frac{m}{\sqrt{p}}k$ a los $\sqrt{p}-1$ procesos en la columna 0:
$$t_s + \frac{mk}{\sqrt{p}}t_w \tag{3.9}$$
  - Envío de los procesos en la columna 0 de bloques de A de tamaño $\frac{m}{\sqrt{p}}\frac{k}{\sqrt{p}}$:
$$t_s + \frac{mk}{p}t_w \tag{3.10}$$
- **Distribución de B**
  - Envío del proceso $\{0,0\}$ de paneles de tamaño $\frac{k}{\sqrt{p}}n$ a los $\sqrt{p}-1$ procesos en la columna 0:
$$t_s + \frac{kn}{\sqrt{p}}t_w \tag{3.11}$$
  - Envío de los procesos en la columna 0 de bloques de B de tamaño $\frac{k}{\sqrt{p}}\frac{n}{\sqrt{p}}$:
$$t_s + \frac{kn}{p}t_w \tag{3.12}$$
- **Comunicación:** Alineación inicial y rotación de los bloques de A y de B, con la rutina **MPI_Sendrecv**:
$$2\sqrt{p}t_s + \sqrt{p}\frac{mk+kn}{p}t_w \tag{3.13}$$
- **Computación:**
$$\frac{2mkn}{p}k_{3,dgemm} \tag{3.14}$$

donde aparecen un SP computacional ($k_{3,dgemm}$) que corresponde al coste computacional de la rutina secuencial de multiplicación de matrices y los SP de comunicaciones (t_s, t_w) de las

diferentes operaciones básicas de comunicación que es preciso realizar en cada uno de los pasos de la rutina (**MPI_Send** y **MPI_Sendrecv**). Al igual que se ha realizado con las rutinas anteriormente expuestas, se analizarán las posibles dependencias de los SP con el tamaño del problema n, y con los valores que se escojan de los AP que aparecen en esta rutina (número de procesadores p).

Resultados experimentales

En el algoritmo de Cannon cada proceso multiplica bloques locales de dimensión $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$ y $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$, en lugar de bloques rectangulares de dimensión $\frac{m}{p} \times k$ y $k \times n$ o $k \times \frac{n}{p}$ como ocurría en los anteriores algoritmos, por lo que el SP correspondiente al coste de la operación básica de multiplicación puede ser distinto. En la tabla 3.7 se han reflejado los valores obtenidos para diferentes tamaños de los bloques locales en cada una de las plataformas analizadas (P4net y HPC160smp). Estos valores se han obtenido con llamadas a la rutina **dgemm** que multiplica dos matrices para diferentes tamaños. Se puede observar en las tablas que los valores medidos varían con el tamaño del bloque local de una plataforma a otra y, además, que son distintos a los obtenidos para el coste de la rutina **dgemm** en los algoritmos vistos con anterioridad (tabla 3.1).

	Tamaño de bloque				
Sistema	128	256	512	1024	1280
P4net	0.000776	0.000566	0.000488	0.000423	0.000405
HPC160smp	0.000931	0.000640	0.000589	0.000575	0.000571

Tabla 3.7: Valores de $k_{3,dgemm}$ (en μsegundos) con BLAS optimizada, en P4net y HPC160smp, con varios tamaños de bloque para la rutina de Cannon.

En lo que se refiere a los SP de las comunicaciones, dado que la distribución de las matrices A y B se realiza como la de la matriz A en el algoritmo CBSR, podemos reutilizar los valores para t_s y t_w obtenidos anteriormente en la plataforma P4net para modelar el coste de la distribución de A y de B. En cambio para la plataforma HPC160smp es necesario calcular un coste para la distribución de ambas matrices A y B en lugar de considerarlo por separado, ya que el coste inicial de establecimiento de segmentos de memoria compartida y semáforos utilizados en la implementación de MPICH disponible en el HPC160smp provoca que el coste de enviar la primera matriz, matriz A, sea superior al de envío de la segunda matriz, matriz B. En HPC160smp para una malla de $p = 2 \times 2$ se obtiene un valor de $t_w = 0.0596$ μsegundos, mientras que el valor medido de t_s resulta despreciable frente al valor de t_w para los tamaños de mensaje que se manejan en esta rutina. Los parámetros t_s y t_w que aparecen en el coste de las rotaciones de los bloques de A y B son los asociados con el uso de la rutina **MPI_Sendrecv**. Se han realizado experimentos para una malla de $p = 2 \times 2$ procesadores en ambas plataformas y se ha obtenido un valor de $t_s = 220$ μsegundos y de $t_w = 1.39$ μsegundos en P4net. En la

plataforma HPC160smp el valor de $t_w = 0.062$ μsegundos, y el valor medido de t_s resulta tan pequeño que no se considerará contribución al coste de las comunicaciones. Al igual que ocurría con las versiones anteriores de la rutina, los valores de los SP de computación y comunicación varían de una plataforma a otra, y para una misma plataforma son diferentes en función de la rutina y de la forma en la que opere sobre los datos.

En la tabla 3.8 para la plataforma P4net y en la 3.9 para la plataforma HPC160smp se comparan el tiempo obtenido experimentalmente al ejecutar la rutina con el tiempo proporcionado por el modelo, para diferentes tamaños del problema en un malla de $p = 2 \times 2$ procesos. Para cada tamaño de problema se muestra la aproximación obtenida por el modelo para las comunicaciones y las computaciones. Como puede observarse, el modelo teórico propuesto, junto con las consideraciones realizadas para la obtención de los SP asociados con las comunicaciones y las computaciones, permite obtener una buena estimación del coste total de ejecución de la rutina en plataformas con diferentes características hardware.

	$p = 2 \times 2$				
n	distri A	distri B	comm	comu+compu	total
	Tiempo experimental (segundos)				
512	0.147589	0.147271	0.361001	0.408015	0.702875
1024	0.586129	0.585756	1.448780	1.690275	2.862160
2048	2.352746	2.358005	5.827288	7.508110	12.218861
4096	9.457222	9.441055	23.359507	36.230287	55.128564
5120	14.796357	14.907081	36.617797	63.915132	93.618570
	Tiempo teórico (segundos)				
512	0.157071	0.157071	0.365325	0.417373	0.731516
1024	0.626964	0.626964	1.458662	1.762470	3.016398
2048	2.506537	2.506537	5.832007	7.926775	12.939848
4096	10.024827	10.024827	23.325387	37.844411	57.894064
5120	15.663544	15.663544	36.445422	63.615726	94.942814
	Desviación (%)				
512	6.04	6.24	1.20	2.29	4.07
1024	6.51	6.57	0.68	4.27	5.39
2048	6.14	5.93	0.08	5.58	5.90
4096	5.66	5.82	0.15	4.46	5.02
5120	5.54	4.83	0.47	0.47	1.41

Tabla 3.8: Comparación del tiempo de ejecución experimental y teórico (en segundos), en P4net. Para la rutina de Cannon en una malla de $p = 2 \times 2$ procesos.

En lo referente al porcentaje de desviación entre el tiempo proporcionado por el modelo, y el que realmente resulta de ejecutar la rutina, puede observase que para la plataforma P4net oscila entre el 1.41 % y el 5.90 %. En la plataforma HPC160smp el valor de la desviación oscila entre el 1.14 % y el 47.22 %, aunque esta aproximación sólo ocurre para un tamaño de problema pequeño, siendo el resto de desviaciones no superiores al 4.72 %.

n	distri A y B	comm	comu+compu	total
	$p = 2 \times 2$			
	Tiempo experimental (segundos)			
512	0.019531	0.011718	0.053710	0.073241
1024	0.088865	0.052733	0.413075	0.501940
2048	0.374013	0.254877	2.735620	3.109633
4096	1.499961	1.041973	20.863715	22.363676
5120	2.296815	1.621063	40.063412	42.360227
	Tiempo teórico (segundos)			
512	0.023437	0.016281	0.078777	0.107829
1024	0.093748	0.065123	0.408851	0.525059
2048	0.374990	0.260493	2.791677	3.256507
4096	1.499961	1.041973	20.807077	22.666397
5120	2.343689	1.628083	39.939587	42.844774
	Desviación (%)			
512	20.00	38.94	46.67	47.22
1024	5.49	23.50	1.02	4.61
2048	0.26	2.20	2.05	4.72
4096	0.00	0.00	0.27	1.35
5120	2.04	0.43	0.31	1.14

Tabla 3.9: Comparación del tiempo de ejecución experimental y teórico (en segundos), en HPC160smp. Para la rutina de Cannon en una malla de $p = 2 \times 2$ procesos.

3.3.6 Algoritmo de Strassen

Strassen [Str69] introdujo un algoritmo para multiplicar matrices de dimensión $n \times n$ con un nivel de complejidad inferior al del algoritmo clásico ($O(n^3)$). Está basado en un esquema para el producto $C = AB$ de matrices particionadas en bloques. Sean A y B matrices de dimensión $n \times n$, con $n = 2^q$. Si se formula el producto de matrices en términos de operaciones entre submatrices, en lugar de operaciones entre filas y columnas, y particionamos las matrices A, B y el producto C en submatrices como sigue:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{3.15}$$

entonces el algoritmo convencional de coste $O(n^3)$ se puede expresar como:

$$\begin{array}{ll} \text{Llamadas recursivas} & \text{Post-adiciones} \\ P_1 = A_{11}B_{11} & C_{11} = P_1 + P_2 \\ P_2 = A_{12}B_{21} & C_{12} = P_3 + P_4 \\ P_3 = A_{11}B_{12} & C_{21} = P_5 + P_6 \\ P_4 = A_{12}B_{22} & C_{22} = P_7 + P_8 \\ P_5 = A_{21}B_{11} & \\ P_6 = A_{22}B_{21} & \\ P_7 = A_{21}B_{12} & \\ P_8 = A_{22}B_{22} & \end{array} \tag{3.16}$$

para lo que se requieren de 8 multiplicaciones recursivas y 4 adiciones. Strassen mostró cómo calcular C con sólo 7 multiplicaciones y 18 adiciones:

$$\begin{array}{llll} \text{Pre-adiciones} & & \text{Llamadas recursivas} & \text{Post-adiciones} \\ S_1 = A_{11} + A_{22} & T_1 = B_{11} + B_{22} & P_1 = S_1T_1 & C_{11} = P_1 + P_4 - P_5 + P_7 \\ S_2 = A_{21} + A_{22} & T_2 = B_{12} - B_{22} & P_2 = S_2B_{11} & C_{12} = P_3 + P_5 \\ S_3 = A_{11} + A_{12} & T_3 = B_{21} - B_{11} & P_3 = A_{11}T_2 & C_{21} = P_2 + P_4 \\ S_4 = A_{21} - A_{11} & T_4 = B_{11} + B_{12} & P_4 = A_{22}T_3 & C_{22} = P_1 + P_3 - P_2 + P_6 \\ S_5 = A_{12} - A_{22} & T_5 = B_{21} + B_{22} & P_5 = S_3B_{22} & \\ & & P_6 = S_4T_4 & \\ & & P_7 = S_5T_5 & \end{array} \tag{3.17}$$

con un coste de $O(n^{\log_2 7}) \simeq n^{2,807}$, inferior al algoritmo convencional, pero con mayores necesidades de memoria.

Se puede aplicar de nuevo el algoritmo de Strassen a cada una de las multiplicaciones de los bloques de dimensión $\frac{n}{2} \times \frac{n}{2}$ asociadas con los P_i. Si A y B son de dimensión $n \times n$, con $n = 2^q$, el algoritmo de Strassen se puede aplicar recursivamente, en un esquema de divide y vencerás [PSCL97]. En el nivel más bajo se llega a una multiplicación de elementos de la matriz, pero llegar hasta él no suele proporcionar buenos tiempos de ejecución, por lo que puede ser interesante obtener un tamaño de bloque, b, mínimo o nivel de recursión, l, en el que se deberá parar ésta y utilizar una multiplicación de matrices convencional.

Si se reorganizan las operaciones de suma y multiplicación en el algoritmo original de Strassen y se utiliza la matrix C para almacenar resultados intermedios, entonces el número de matrices necesarias para almacenar dichos resultados se reduce a 3:

Paso 1	**Paso 4**	**Paso 6**
$S_1 = A_{11} + A_{22}$	$T_1 = B_{21} - B_{11}$	$S_1 = A_{21} - A_{11}$
$T_1 = B_{11} + B_{22}$	$P_1 = A_{22}T_1$	$T_1 = B_{11} + B_{12}$
$P_1 = S_1T_1$	$C_{11} = C_{11} + P_1$	$P_1 = S_1T_1$
$C_{11} = P_1$	$C_{21} = C_{21} + P_1$	$C_{22} = C_{22} + P_1$
$C_{22} = P_1$		
Paso 2	**Paso 5**	**Paso 7**
$S_1 = A_{21} + A_{22}$	$S_1 = A_{11} + A_{12}$	$S_1 = A_{12} - A_{22}$
$P_1 = S_1B_{11}$	$P_1 = S_1B_{22}$	$T_1 = B_{21} + B_{22}$
$C_{21} = P_1$	$C_{11} = C_{11} - P_1$	$P_1 = S_1T_1$
$C_{22} = C_{22} - P_1$	$C_{12} = C_{12} + P_1$	$C_{11} = C_{11} + P_1$
Paso 3		
$T_1 = B_{12} - B_{22}$		
$P_1 = A_{11}T_1$		
$C_{12} = P_1$		
$C_{22} = C_{22} + P_1$		

(3.18)

El esquema de la ecuación 3.18 es el que se ha utilizado para la implementación secuencial y las paralelas realizadas en este trabajo. Los requerimientos de memoria del algoritmo de Strassen son superiores a los de una multiplicación convencional, debido a la necesidad de almacenar los resultados intermedios en cada nivel de recursión. Si para un nivel de recursión 1 se requiere de $\frac{17n^2}{4}$ elementos de memoria adicionales, para un nivel de recursión l las necesidades de memoria adicionales serán de $10\sum_{i=1}^{l} 7^{i-1}\frac{n^2}{4^i} + \sum_{i=1}^{l} 7^i\frac{n^2}{4^i}$.

Al igual que se ha realizado con las versiones de la multiplicación vistas con anterioridad, a partir de la descripción del algoritmo de Strassen se construye el modelo teórico que permitirá obtener una aproximación al tiempo de ejecución de la rutina que implementa dicho algoritmo. Dado que el coste para una multiplicación de matrices cuadradas de tamaño n^2 es $2n^3k_3$, y puesto que para un nivel de recursión l el tamaño de las matrices a multiplicar es $\frac{n}{2^l}$, cada multiplicación realizada tendrá un coste de $2\left(\frac{n}{2^l}\right)^3 k_3$ y dado que es preciso realizar 7^l multiplicaciones si el nivel de recursión es l, el coste total de las multiplicaciones será $7^l 2\left(\frac{n}{2^l}\right)^3 k_3$. Teniendo en cuenta también que el coste de una adición de matrices es n^2k_2 y que para un nivel de recursión l el tamaño de las matrices a sumar es $\frac{n}{2^l}$, cada adición de matrices tendrá un coste de $\left(\frac{n}{2^l}\right)^2 = \frac{n^2}{4^l}$, y puesto que para un nivel de recursión 1 el coste es $18\frac{n^2}{4}k_2$, para un nivel de recursión 2 es $\frac{18}{4}n^2\left(\frac{7}{4}+1\right)k_2$ y para un nivel de recursión 3 es $\frac{18}{4}n^2\left(\frac{7^2}{4^2}+\frac{7}{4}+1\right)k_2$, por tanto para un nivel de recursión l el coste será $\frac{18}{4}n^2\sum_{i=1}^{l}\left(\frac{7}{4}\right)^{i-1}k_2$, y el modelo de tiempo de ejecución para la rutina de Strassen queda como sigue:

$$T = 7^l 2 \left(\frac{n}{2^l}\right)^3 k_{3,dgemm} + \frac{18}{4} n^2 \sum_{i=1}^{l} \left(\frac{7}{4}\right)^{i-1} k_{2,add} \tag{3.19}$$

donde aparece un AP (nivel de recursión l) y dos SP computacionales correspondientes a rutinas de diferentes niveles de BLAS: $k_{3,dgemm}$, que representa el coste de una operación aritmética cuando se utiliza una multiplicación matricial, y $k_{2,add}$ que representa el coste de una operación aritmética cuando se utiliza la adición matricial. El valor de los SP, como en las rutinas vistas anteriormente, podrá depender del tamaño de problema, n, y del tamaño del bloque de cálculo resultante para el nivel de recursión l escogido.

Ajustes del algoritmo para dimensión impar

El algoritmo de Strassen se aplica de forma natural a matrices con tamaños que son potencia de 2, puesto que la matriz puede ser particionada en sus cuatro cuadrantes en cada nivel de recursión, mientras que en matrices de otras dimensiones es necesario introducir un trabajo extra. Existen diferentes estrategias para conseguir aplicar el algoritmo a estos casos. La más obvia consiste en llenar la matriz con ceros hasta que la dimensión sea una potencia de 2 (*static padding*); otras variantes consisten en rellenar con ceros sólo en aquel nivel de recursión que lo requiera (*dynamic padding*), o no considerar la fila/columna adicional y añadir su contribución en un paso final finalizada la multiplicación de Strassen (*dynamic peeling*), o simplemente utilizar menos pasos de recursión. En [DHSS94] se utiliza una versión de dynamic padding, y en [HLJT+96] una de dynamic peeling. En nuestra versión de Strassen, se implementan sumas generalizadas de matrices que permiten realizar su adición cuando sus dimensiones son distintas, añadiendo ceros si la matriz resultante es mayor, o acortando la dimensión de los sumandos cuando es menor. Para la multiplicación, se utiliza el valor menor del número de filas del primer operando y del número de columnas del segundo operando, de forma similar a como se hace en [DN05].

Resultados experimentales

Al tratarse de una rutina secuencial, los únicos parámetros del sistema que aparecen son los asociados con la computación para las multiplicaciones ($k_{3,dgemm}$) y las adiciones ($k_{2,add}$). La obtención de los valores de los SP se realiza por medio de experimentos que permiten medir el parámetro del sistema correspondiente y mostrar su dependencia respecto a los AP que aparezcan en el algoritmo. En el caso del parámetro $k_{3,dgemm}$, para un tamaño del problema n y un nivel de recursión l, se invocará a la rutina de BLAS para multiplicar matrices con dimensiones $b \times b$, siendo $b = \frac{n}{2^l}$. De forma similar se obtendrá el SP correspondiente a la operación de adición de matrices.

Como ya se ha indicado, la multiplicación de los bloques al finalizar la recursión se realiza con

la rutina **dgemm** de BLAS. El $k_{3,dgemm}$ asociado con esta operación sólo depende del tamaño de bloque b y, al tratarse del mismo tipo de multiplicación que en el algoritmo de Cannon, se pueden reutilizar los valores obtenidos previamente para las plataformas P4net y HPC160smp (tabla 3.7). En la tabla 3.10 se han reflejado los valores medidos para $k_{3,dgemm}$ en la plataforma Opteron, no mostrados anteriormente. En lo que se refiere al otro SP que aparece en nuestro modelo de tiempo de ejcución ($k_{2,add}$), en la tabla 3.11 se pueden ver los valores obtenidos en las plataformas P4net, HPC160smp y Opteron, mediante el procedimiento ya indicado, y su variación respecto al tamaño, b, de las matrices a sumar.

	Tamaño de bloque, b				
Sistema	128	256	512	1024	2048
Opteron	0.000428	0.000391	0.000351	0.000330	0.000319

Tabla 3.10: Valores de $k_{3,dgemm}$ (en μsegundos) con BLAS optimizada, en Opteron, con varios tamaños de bloque.

	Tamaño de bloque, b						
Sistema	64	128	256	512	1024	2048	2560
P4net	0.032959	0.025757	0.023911	0.022511	0.022645		
HPC160smp		0.029800	0.033527	0.028871		0.031199	0.029802
Opteron	0.012207	0.018250	0.022156	0.018341	0.017550	0.017515	

Tabla 3.11: Valores de $k_{2,add}$ (en μsegundos) en P4net, HPC160smp y Opteron, con varios tamaños de bloque.

Con el fin de comprobar la validez del modelo propuesto para la aproximación del tiempo de ejecución de la rutina secuencial de Strassen, se han realizado ejecuciones para diferentes tamaños de problema. Para cada tamaño de problema, se ha utilizando niveles de recursión entre 1 y 4. En cada caso se compara el tiempo de ejecución obtenido mediante la aproximación proporcionada por el modelo, con el tiempo que realmente tarda en ejecutarse la rutina en las plataformas P4net (tabla 3.12), Opteron (tabla 3.13) y HPC160smp (tabla 3.14). Para cada tamaño de problema se muestra también los tiempos obtenidos con la rutina **dgemm** de BLAS para la multiplicación de matrices y, de cara a mostrar la utilidad del modelo a la hora de poder seleccionar la rutina y el nivel de recursión con el que se obtiene el menor tiempo de ejecución, se han resaltado en negrita el menor tiempo de ejecución real y el menor tiempo de ejecución previsto por el modelo.

En la tabla 3.12 se puede observar que en el caso del sistema P4net, se obtienen mejores tiempos de ejecución con la rutina **dgemm** de BLAS optimizada que con Strassen, y el tiempo de ejecución se incrementa al aumentar el nivel de recursión l, hasta tamaños de problema $n = 2048$ en que es preferible utilizar Strassen con un sólo nivel de recursión. En Opteron

	P4net			
	Strassen			dgemm
n	$l=1$	$l=2$	$l=3$	
	Tiempo experimental (segundos)			
512	0.157456	0.200354	0.281269	**0.125375**
1024	0.975005	1.18352	1.480566	**0.890470**
2048	**6,548849**	7.234741	8.936342	6.743971
	Tiempo teórico (segundos)			
512	0.157734	0.210564	0.357702	**0.135378**
1024	0.981085	1.216962	1.595484	**0.927371**
2048	**6.539901**	7.292467	8.970030	6.916378
	Desviación (%)			
512	0.18	5.10	27.17	7.98
1024	0.62	2.83	7.76	4.14
2048	0.14	0.80	0.38	2.56

Tabla 3.12: Comparación de los tiempos de ejecución experimentales y teóricos (en segundos), en P4net. Para la rutina de Strassen y **dgemm** de BLAS optimizada.

	Opteron			
	Strassen			dgemm
n	$l=1$	$l=2$	$l=3$	
	Tiempo experimental (segundos)			
512	0.118755	0.135043	0.165111	**0.094953**
1024	0.745705	0.907532	1.023456	**0.711527**
2048	**5.29819**	5.52772	6.624437	5.504525
4096	41.146571	**39.401342**	41.160937	43.416609
	Tiempo teórico (segundos)			
512	0.117899	0.147255	0.168764	**0.094828**
1024	0.746077	0.929837	1.116897	**0.710990**
2048	**5.297985**	5.568715	6.927035	5.485237
4096	39.730501	**38.410911**	40.365709	43.296625
	Desviación (%)			
512	0.72	9.04	2.21	0.13
1024	0.05	2.46	9.13	0.08
2048	0.00	0.74	4.57	0.35
4096	3.44	2.51	1.93	0.28

Tabla 3.13: Comparación de los tiempos de ejecución experimentales y teóricos (en segundos), en Opteron. Para la rutina de Strassen y **dgemm** de BLAS optimizada.

	HPC160smp				
	Strassen				dgemm
n	$l=1$	$l=2$	$l=3$	$l=4$	
	Tiempo experimental (segundos)				
512	0.168942	0.179684	0.206051		**0.157224**
1024	**1.207977**	1.218726	1.297818		1.238258
2048	9.162911	8.678545	**8.481284**	8.74495	9.850399
4096	71.539648	64.567124	60.638488	**59.084807**	83.275756
5120	135.576501	123.145100	115.060295	**111.925609**	150.624064
	Tiempo teórico (segundos)				
512	0.185542	0.228267	0.319483		**0.158199**
1024	1.272441	1.361671	1.670457		**1.232390**
2048	9.185371	**9.107247**	9.884823	12.072498	9.871881
4096	71.444922	66.640698	**65.974536**	71.420833	83.401152
5120	135.795248	125.412521	**121.710839**	127.892474	150.994123
	Desviación (%)				
512	9.83	27.04	55.05		0.62
1024	3.66	11.73	32.60		0.47
2048	0.25	4.94	16.55	38.05	0.22
4096	0.13	3.21	8.80	20.88	0.15
5120	0.16	1.84	5.78	14.27	0.25

Tabla 3.14: Comparación de los tiempos de ejecución experimentales y teóricos (en segundos), en HPC160smp. Para la rutina de Strassen y **dgemm** de BLAS optimizada.

(tabla 3.13) se observa que es preferible utilizar **dgemm** de BLAS optimizada que Strassen, pero para tamaños de problema $n = 2048$ es preferible utilizar Strassen con un nivel de recursión, al igual que ocurría en el sistema P4net, y con $n = 4096$ el nivel de recursión óptimo cambia a 2. En HPC160smp (tabla 3.14) es preferible utilizar Strassen frente a **dgemm** optimizada para HPC160smp con un tamaño de problema $n \geq 1024$, pero a diferencia de P4net y Opteron, también varía el nivel de recursión óptimo, ahora $l = 3$, y para tamaños de problema $n \geq 4096$ el nivel de recursión óptimo es $l = 4$.

Como puede observarse, el nivel de recursión con el que se obtiene el menor tiempo de ejecución depende del tamaño del problema y del sistema en el que se ejecute la rutina secuencial de Strassen. En la mayor parte de los casos analizados, el modelo predice correctamente el tamaño del problema a partir del cual es aconsejable utilizar la rutina de Strassen frente a la rutina de multiplicación de matrices disponible en BLAS, así como el nivel de recursión con el que se obtiene el tiempo de ejecución óptimo. Por otra parte, se aprecia que la aproximación proporcionada por el modelo de tiempo de ejecución empeora al aumentar el nivel de recursión, siendo más evidente en la plataforma HPC160smp, donde se han empleado niveles de recursión más elevados que en el resto de plataformas. Esto se debe a que en el modelo se ha utilizado un único valor para el coste de cada una de las sumas que se realizan en la rutina de Strassen, y sin embargo para un nivel de recursión l el tamaño de las matrices a sumar varía entre $\frac{n}{2}$ y $\frac{n}{2^l}$ y, dada la dependencia del parámetro $k_{2,add}$ con el tamaño de las matrices a sumar, se podría haber utilizado un modelo más detallado en el que aparecieran diferentes valores del $k_{2,add}$ y mejorar así la aproximación obtenida por el modelo para esta plataforma, y realizando por

tanto una mejor selección. En cualquier caso, la diferencia entre el tiempo de ejecución óptimo y el tiempo de ejecución que se obtendría con la selección realizada a partir de la información proporcionada por el modelo es pequeña, y en el caso del sistema HPC160smp la media de las desviaciones entre ambos tiempos de ejecución es de 2.6 %.

3.3.7 Strassen paralelo. Versión PBLAS (StPBLAS)

Una forma de paralelizar el algoritmo de Strassen, consiste en seguir su esquema de 7 multiplicaciones y 18 adiciones y utilizar rutinas paralelas disponibles en paquetes estándar de software, como ScaLAPACK, para las multiplicaciones y adiciones teniendo en cuenta que es necesario mantener la distribución de datos impuesta por el paquete seleccionado, que en caso de ScaLAPACK es la cíclica por bloques. En la figura 3.4 se muestra una distribución de datos cíclica por bloques en una malla de 2×2 procesos. Los números a la izquierda y arriba de las matrices representan coordenadas en la malla. A la izquierda tenemos los 4 cuadrantes de la matriz A y a la derecha los cuatro cuadrantes de la adición, S_1, realizada en el primer paso de la multiplicación de Strassen (ecuación 3.18).

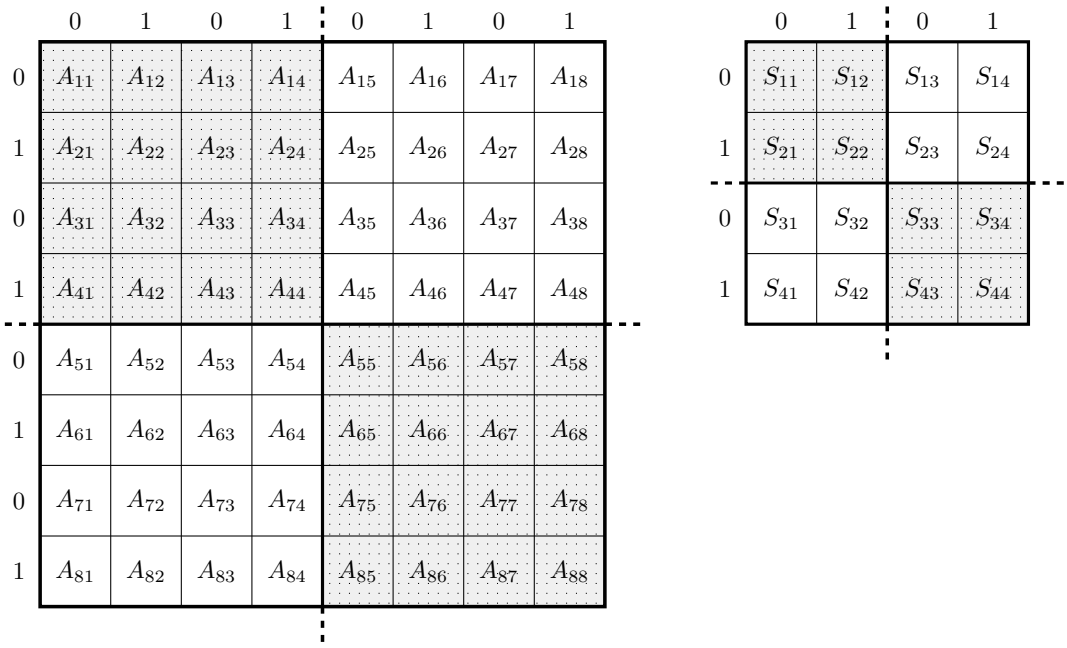


Figura 3.4: Distribución de datos cíclica por bloques. Una matriz con 8×8 bloques es distribuida en una malla de 2×2 procesos. Las áreas sombreadas son los dos bloques de la matriz A utilizados en la adición, S_1, del primer paso de la multiplicación de Strassen. A la derecha se muestra el resultado de la distribución de datos de la adición S_1 en la malla de 2×2 en el siguiente nivel de recursión.

Si imponemos a la hora de ejecutar la rutinas que el número de bloques por fila y columna sea un múltiplo del número de procesos por fila y columna, entonces no habrá comunicaciones

para las adiciones. El algoritmo de Strassen utilizando rutinas de ScaLAPACK se puede aplicar recursivamente, pero ahora es necesario restringir la condición anterior a que sea un múltiplo par del número de procesos por fila y columna, pues de lo contrario habría comunicaciones en las adiciones a partir del segundo nivel de recursión. Cada proceso de la malla realizará la adición de los bloques correspondientes a la distribución cíclica por bloques: $S_{11} = A_{11} + A_{55}$, $S_{12} = A_{12} + A_{56}$, etc. En el nivel de recursión dos, la matriz S_1 ya estará distribuida en bloques de igual forma que la matriz A del nivel de recursión 1, y la adición se volverá a realizar sin necesidad de comunicaciones en los bloques $S_{11} + S_{33}$, $S_{12} + S_{34}$, etc.

Los parámetros del algoritmo (AP) son el número de procesos $p = r \times c$, el tamaño de bloque para la distribución cíclica por bloques b, y el nivel de recursión l del algoritmo de Strassen. Los parámetros del sistema (SP) serán el coste de las multiplicaciones locales realizadas en la rutina **pdgemm**, $k_{3,dgemm}$, y el de las sumas en la rutina **pdgeadd**, $k_{2,add}$, de PBLAS. Los parámetros del sistema asociados con las comunicaciones son la latencia o tiempo de inicio de la comunicación, cuando r procesos se comunican simultáneamente con c procesos t_s^{r-c}, y cuando c procesos se comunican simultáneamente con r procesos, t_s^{c-r}, y el coste de enviar una palabra en la malla de procesos, t_w^{r-c} y t_w^{c-r}. En lo que sigue se supondrá, sin que esto suponga pérdida de generalidad, que las matrices y la malla de procesos son cuadradas, de dimensiones $n \times n$ y $\sqrt{p} \times \sqrt{p}$ respectivamente, que el tamaño de bloque es seleccionado de tal forma que no hay comunicaciones en las adiciones realizadas con la rutina **pdgeadd** y que sólo tendremos el coste de las comunicaciones en la rutina **pdgemm**.

Con las consideraciones realizadas el modelo del tiempo de ejecución quedaría:

- **Computación:**

$$7^l \frac{2}{p} \left(\frac{n}{2^l}\right)^3 k_{3,dgemm} + 18\frac{n^2}{p} \sum_{i=1}^{l} \frac{7^{i-1}}{4^i} k_{2,add} \tag{3.20}$$

- **Comunicación:**

$$7^l \left(2 \left\lceil \frac{n}{2^l b} \right\rceil t_s^{\sqrt{p}-\sqrt{p}} + \frac{2n^2}{4^l \sqrt{p}} t_w^{\sqrt{p}-\sqrt{p}}\right) \tag{3.21}$$

donde ahora aparecen tres AP (nivel de recursión l, número de procesadores p, y tamaño de bloque para la distribución de las matrices b), en lugar del único AP que se tenía en la versión secuencial de la rutina. En lo que se refiere a los SP, al igual que con la versión secuencial, se dispone de dos SP computacionales ($k_{3,dgemm}$ y $k_{2,add}$), y además al tratarse de una rutina paralela aparecen los SP de comunicaciones (t_s, t_w). El valor de los SP, como en las rutinas vistas anteriormente, podrá depender del tamaño de problema, n, y del valor seleccionado para los AP descritos.

Resultados experimentales

El modelo se ha validado experimentalmente en las plataformas P4net y HPC160smp, y se han obtenido en cada plataforma los valores de los parámetros del sistema y su dependencia respecto a los del algoritmo. En el caso del parámetro $k_{3,dgemm}$ asociado con la multiplicación realizada localmente en cada proceso de la malla, se ha implementado una versión "outer-product" por bloques de la multiplicación de matrices con la rutina **dgemm** de BLAS3 y utilizado un tamaño de bloque de 32 tal y como se realiza en la rutina **pdgemm** de la versión de PBLAS utilizada; en los experimentos realizados se ha comprobado que este parámetro sólo depende del tamaño de la matriz a multiplicar localmente. Los valores obtenidos experimentalmente para P4net y HPC160smp se muestran en las tablas 3.15 y 3.16, respectivamente.

$k_{3,dgemm}$ en PBLAS para P4net						
n	64	256	512	1024	1536	2560
	0.001953	0.000970	0.000996	0.000995	0.000988	0.000996

Tabla 3.15: Valores de $k_{3,dgemm}$ (en μsegundos) con BLAS optimizada en la rutina **pdgemm** de PBLAS para P4net, con varios tamaños de matriz.

$k_{3,dgemm}$ en PBLAS para HPC160smp						
n	256	512	1024	1536	2560	3072
	0.000714	0.000644	0.000640	0.000995	0.000988	0.000996

Tabla 3.16: Valores de $k_{3,dgemm}$ (en μsegundos) con BLAS optimizada en la rutina **pdgemm** de PBLAS para HPC160smp, con varios tamaños de matriz.

En las tablas 3.15 y 3.16 puede apreciarse la dependencia del valor del SP de computación $k_{3,dgemm}$ con el tamaño del problema y con la plataforma. Además, si comparamos estos valores del SP de computación $k_{3,dgemm}$ con los obtenidos para el mismo SP de computación en las rutinas de multiplicación de matrices vistas anteriormente, se puede comprobar que son distintos. Esta dependencia indica la necesidad de realizar un análisis para cada rutina de los SP que aparecen en su modelo, lo que refuerza nuestro planteamiento sobre mejoras en el modelado de rutinas mostrado en el capítulo anterior.

En lo referente al parámetro $k_{2,add}$ asociado con la rutina **pdgeadd** de PBLAS, se han obtenido sus valores mediante la ejecución de la rutina **pdgeadd** en las mismas condiciones en las que se va a validar experimentalmente el modelo, esto es, en una malla de $p = 2 \times 2$ y seleccionando el tamaño de bloque para la distribución de matrices de tal forma que no haya comunicaciones durante su adición. Los experimentos realizados muestran que el valor de este parámetro depende del tamaño de las matrices a sumar. Los valores obtenidos para distintos tamaños de matriz se muestran en la tabla 3.17 para P4net y en la tabla 3.18 para HPC160smp. Como puede apreciarse en las tablas, y de forma similar a lo comentado sobre el

parámetro $k_{3,dgemm}$, hay variación en el valor del parámetro $k_{2,add}$ con respecto a los tamaños de las matrices a operar, con respecto a la plataforma y con respecto a la rutina que utiliza la operación de adición.

$k_{2,add}$ en PBLAS para P4net							
n	128	256	512	768	1024	1536	2048
	0.022522	0.023270	0.023422	0.025704	0.022544	0.024798	0.031068

Tabla 3.17: Valores de $k_{2,add}$ (en μsegundos) con BLAS optimizada en la rutina **pdgeadd** de PBLAS para P4net, con varios tamaños de matriz.

$k_{2,add}$ en PBLAS para HPC160smp							
n	256	512	768	1024	1536	2048	2560
	0.029800	0.029800	0.034768	0.035390	0.035596	0.036786	0.036209

Tabla 3.18: Valores de $k_{2,add}$ (en μsegundos) con BLAS optimizada en la rutina **pdgeadd** de PBLAS para HPC160smp, con varios tamaños de matriz.

En cuanto a los parámetros del sistema asociados con las comunicaciones, t_s y t_w, sus valores se han obtenido mediante experimentos que simulan el envío de bloques de tamaño $n \times b$ realizados por una columna o fila de procesos al resto de procesos en su misma fila o columna y utilizando las rutinas de comunicación proporcionadas por la librería BLACS, tal y como se realiza en la rutina. Los valores obtenidos experimentalmente para una malla de $p = 2 \times 2$ procesos son de $t_s = 370.524$ μsegundos y $t_w = 0.709810$ μsegundos en P4net. En HPC160smp se ha obtenido que $t_s = 92.696$ μsegundos y que $t_w = 0.065002$ μsegundos. Como se puede observar, los valores de los SP de comunicación obtenidos para las rutinas de BLACS, son diferentes a los obtenidos para las rutinas básicas de comunicación de MPI empleadas en las multiplicaciones de matrices vistas con anterioridad. De ahí la importancia de realizar un estudio experimental de los valores de los SP de comunicación y comprobar su variación en función de la librería de comunicación empleada.

En la tabla 3.19 (P4net) y en la tabla 3.20 (HPC160smp) se compara el tiempo obtenido experimentalmente con el tiempo proporcionado por el modelo para una malla de $p = 2 \times 2$ procesos, para niveles de recursión l de 1, 2 y 3; y para diferentes tamaños de problema n. En negrita aparecen resaltados el menor tiempo de ejecución real y el menor tiempo de ejecución aproximado por el modelo. Se han realizado pruebas con diferentes tamaños de bloque para la distribución, pero la diferencia de tiempos de ejecución variando este parámetro es pequeña (un 1 %) y sólo se muestran los resultados para $b = 64$.

Los resultados reflejan que en P4net es siempre preferible utilizar un nivel de recursión $l = 1$. En HPC160smp para tamaños de problema pequeños con $n < 3072$ es mejor utilizar un sólo nivel de recursión, y para tamaños de problema con $n \geq 3072$ utilizar un nivel de recursión 2, salvo para $n = 5120$ en el que es preferible utilizar un nivel 3. En la plataforma HPC160smp el

	P4net Strassen PBLAS		
n	$l=1$	$l=2$	$l=3$
	Tiempo experimental (segundos)		
512	**0.525376**	0.700110	
1024	**1.815157**	3.296943	4.901434
2048	**9.222891**	11.931264	24.009070
3072	**24.715618**	33.355596	49.610680
4096	**51.600152**	65.679293	84.528084
	Tiempo teórico (segundos)		
512	**0.407083**	0.761308	
1024	**1.827532**	2.877208	4.984113
2048	**9.142147**	12.899089	20.246833
3072	**24.567550**	31.535856	47.827317
4096	**51.296548**	64.581415	90.880069
	Desviación (%)		
512	22.52	8.74	
1024	0.68	12.73	1.69
2048	0.88	8.11	15.67
3072	0.60	5.46	3.59
4096	0.59	1.67	7.51

Tabla 3.19: Comparación de los tiempos de ejecución experimentales y teóricos (en segundos), en P4net. Para Strassen con PBLAS utilizando $b = 64$ y $p = 2 \times 2$.

	HPC160smp Strassen PBLAS		
n	$l=1$	$l=2$	$l=3$
	Tiempo experimental (segundos)		
1024	**0.411122**	0.468738	
2048	**2.767520**	2.788013	3.119059
3072	9.121223	**8.815204**	9.307378
4096	21.333430	**20.499468**	20.651807
5120	41.086701	38.851529	**38.037099**
6144	70.749382	**66.608620**	66.857656
7168	109.889266	103.728778	**99.031633**
	Tiempo teórico (segundos)		
1024	**0.500108**	0.635130	
2048	**3.059522**	3.412719	4.568527
3072	**9.477941**	10.073681	12.325609
4096	**21.856261**	21.980797	24.988895
5120	41.504228	**40.949497**	43.996547
6144	70.003190	**67.857265**	70.174509
7168	111.829724	**106.244496**	107.967351
	Desviación (%)		
1024	21.64	35.50	
2048	10.55	22.41	46.47
3072	3.91	14.28	32.43
4096	2.45	7.23	21.00
5120	1.02	5.40	15.67
6144	1.05	1.87	4.96
7168	1.77	2.43	9.02

Tabla 3.20: Comparación de los tiempos de ejecución experimentales y teóricos (en segundos), en HPC160smp. Para Strassen con PBLAS utilizando $b = 64$ y $p = 2 \times 2$.

modelo no selecciona correctamente el nivel de recursión óptimo para los tamaños de problema 3072, 4096, 5120 y 7168, pero la diferencia entre el tiempo de ejecución óptimo y el tiempo de ejecución siguiendo la selección del nivel de recursión realizada por el modelo es pequeña (un 3.47 % de media).

Se ha mostrado que el modelo permite seleccionar el nivel de recursión con el que se obtienen tiempos de ejecución cercanos al óptimo, aunque dicho nivel varíe de una plataforma a otra y con el tamaño del problema. De forma similar a lo ocurrido con la multiplicación de Strassen secuencial, se observa que al aumentar el nivel de recursión empeora la aproximación obtenida con el modelo, debido a la simplificación realizada al modelar las adiciones que ocurren en la rutina. En cualquier caso, como ya hemos comentado, el modelo permite decidir el nivel de recursión óptimo en la mayoría de los casos analizados.

3.3.8 Strassen paralelo. Versión maestro-esclavo (StMS)

En esta sección se analiza una versión paralela diferente del algoritmo de Strassen. El esquema seguido se muestra en el algoritmo 3.1, y consiste en que un proceso maestro (proceso p_0) se encarga de realizar las pre-adiciones y enviar los datos a multiplicar al resto de procesos esclavos (p_1, p_2,...), que realizan las multiplicaciones convencionales y devuelven el resultado al proceso maestro. Éste recibe los datos y realiza las post-adiciones. Por último se indica a los procesos que no hay más trabajo que hacer.

Si consideramos el caso más sencillo, en el que se tiene un único nivel de recursión, el número de multiplicaciones a distribuir será 7 y habrá por tanto que decidir el número de procesos a lanzar, lo que introduce un nuevo parámetro del algoritmo respecto a la versión secuencial, el número de procesos p utilizados para la computación paralela. Se supondrá que habrá como máximo 8 procesos, ya que lanzar más procesos en un sistema homogéneo con una asignación de un proceso por procesador no supondría una reducción en el tiempo de ejecución de la rutina, debido a que los procesadores adicionales estarían ociosos.

En la tabla 3.21 se muestra la asignación de multiplicaciones a los procesos esclavos p_1, p_2, p_3, etc. En la tabla se resaltan el proceso esclavo que más trabajo tiene, o el último que recibe los datos a multiplicar y por tanto el que se tendrá en cuenta para definir la ecuación que modelará el tiempo de ejecución de la rutina.

Para $p = 2$, 3, 4 y 7 el proceso p_1 es el que más multiplicaciones tiene que realizar. Con $p = 5$ los procesos p_1, p_2 y p_3 tienen que realizar el mismo número de multiplicaciones, pero el p_3 es el último que recibe los datos y por tanto el último en acabar. Cuando $p = 6$ ocurre lo mismo que con $p = 5$, pero ahora es p_2 el que realiza mayor número de multiplicaciones, y hay sólo que esperar a que p_1 reciba los datos para su siguiente multiplicación. Finalmente, con $p = 8$ todos los procesos realizan una sola multiplicación pero p_7 es el último en recibir los datos a multiplicar, y por tanto el último en acabar. Por consiguiente tenemos 4 casos distintos

Algoritmo 3.1 Strassen en paralelo, versión maestro-esclavo (StMS).

```
if proc p_0 then
  for i = 1 to min(tasks, nprocs) do
    Compute S_0, T_0
    Send(S_0,i), Send(T_0,i)
  end for
  for j = 1 to j < tasks do
    Recv(P_0, Sender)
    if i ≤ tasks then
      Compute S_0, T_0
      Send(S_0,Sender), Send(T_0,Sender)
      Update(C)
    else
      Send(0,Sender)
    end if
  end for
else if proc ≠ p_0 then
  loop
    Recv(S_0,p_0)
    if tasks = 0 then
      break
    end if
    Recv(T_0,p_0)
    P_0 = S_0 × T_0
    Send(P_0,p_0)
  end loop
end if
```

	p_1	p_2	p_3	p_4	p_5	p_6	p_7
$p=2$	**7**						
$p=3$	**4**	3					
$p=4$	**3**	2	2				
$p=5$	2	2	**2**	1			
$p=6$	2	**2**	1	1	1		
$p=7$	**2**	1	1	1	1	1	
$p=8$	1	1	1	1	1	1	**1**

Tabla 3.21: Distribución de multiplicaciones a procesos para la rutina de Strassen paralela versión maestro-esclavo (StMS).

y 4 modelos para el tiempo de ejecución de la rutina:

- **Caso 1:** (Para $p = 2, 3, 4, 7$)

$$\left\lceil \frac{7}{p-1} \right\rceil \left(2\left(\frac{n}{2}\right)^3 k_{3,dgemm} + 2\frac{n^2}{4}k_{2,add} + 2t_s + 2t_w\frac{n^2}{4} + t_s + t_w\frac{n^2}{4} \right) + \frac{n^2}{4}k_{2,add} \quad (3.22)$$

- **Caso 2:** (Para $p = 5$)

$$2\left\lceil \frac{7}{p-1} \right\rceil \left(\frac{n}{2}\right)^3 k_{3,dgemm} + 15\frac{n^2}{4}k_{2,add} + 13t_s + 13t_w\frac{n^2}{4} \quad (3.23)$$

- **Caso 3:** (Para $p = 6$)

$$2\left\lceil \frac{7}{p-1} \right\rceil \left(\frac{n}{2}\right)^3 k_{3,dgemm} + 11\frac{n^2}{4}k_{2,add} + 10t_s + 10t_w\frac{n^2}{4} \quad (3.24)$$

- **Caso 4:** (Para $p = 8$)

$$2\left\lceil \frac{7}{p-1} \right\rceil \left(\frac{n}{2}\right)^3 k_{3,dgemm} + 15\frac{n^2}{4}k_{2,add} + 15t_s + 15t_w\frac{n^2}{4} \quad (3.25)$$

donde aparece un AP (número de procesadores p), dos SP computacionales ($k_{3,dgemm}$ y $k_{2,add}$), y los SP de comunicaciones (t_s, t_w). El valor de los SP, como en las rutinas vistas anteriormente, podrá depender del tamaño de problema, n, y del valor seleccionado para el AP descrito.

Resultados experimentales

Los parámetros del sistema asociados con las comunicaciones, t_s y t_w, que aparecen en el modelo, corresponden a la rutina **MPI_Send** utilizada en el algoritmo paralelo de Strassen para el envío de los bloques a multiplicar. El modelo ha sido comprobado en la plataforma HPC160smp con un valor de $t_w = 0.0279$ μsegundos. Este sistema conecta nodos de 4 procesadores con una Red MemoryChannel, por lo que es necesario considerar un valor de t_w distinto cuando se utilizan procesadores en varios nodos ($p = 5, 6, 7$ y 8). El valor obtenido es de $t_w = 0.0796$ μsegundos, y el valor de t_s es el mismo que ya se ha utilizado en los algoritmos anteriores.

En la tabla 3.22 se compara el tiempo obtenido experimentalmente con el tiempo proporcionado por los modelos de tiempo de ejecución, para diferentes tamaños de problema n y número de procesos p en la plataforma HPC160. La tabla muestra que los modelos teóricos propuestos para cada caso permiten obtener una buena estimación del coste total de ejecución de la rutina. El valor medio del porcentaje en la desviación entre el tiempo proporcionado por el modelo y el que realmente resulta de ejecutar la rutina, para las 35 combinaciones de tamaño de problema y número de procesos comparados, es de un 5.26 %.

n	$p=2$	$p=3$	$p=4$	$p=5$	$p=6$	$p=7$	$p=8$
	Tiempo experimental (segundos)						
512	0.166987	0.108396	0.076170	0.087889	0.090819	0.128904	0.119137
1024	1.219695	0.711900	0.546864	0.531230	0.526357	0.552720	0.510729
2048	9.198979	5.309461	4.019453	3.353452	3.340756	3.160075	2.468686
4096	71.201274	40.990362	30.879286	23.228067	23.230021	22.527735	14.861022
5120	137.541282	78.090190	58.701991	43.319418	43.232564	41.921990	26.585390
	Tiempo teórico (segundos)						
512	0.175168	0.107904	0.080928	0.095081	0.089348	0.076168	0.093496
1024	1.275844	0.740213	0.555160	0.627413	0.486308	0.454086	0.455054
2048	9.487064	5.472519	4.104389	3.328537	3.227463	3.083907	2.499458
4096	72.295981	41.526268	31.144701	23.175673	22.761135	22.166429	14.970820
5120	140.256621	80.425651	60.319238	43.734720	43.159052	42.373920	26.949044
	Desviación (%)						
512	4.90	0.45	6.25	8.18	1.62	40.91	21.52
1204	4.60	3.98	1.52	18.11	7.61	17.85	10.90
2048	3.13	3.07	2.11	0.74	3.39	2.41	1.25
4096	1.54	1.31	0.86	0.23	2.02	1.60	0.74
5120	1.97	2.99	2.68	0.96	0.17	1.08	1.37

Tabla 3.22: Comparación de los tiempos de ejecución experimentales y teóricos (en segundos), en HPC160. Para la rutina de Strassen paralela versión maestro-esclavo (StMS).

	$p=2$			$p=3$			$p=4$			$p=5$			$p=6$			$p=7$			$p=8$		
n	ex.	mo.	dev. (%)	ex.	mo.	dev. (%)	ex.	mo.	dev. (%)	ex.	mo.	dev. (%)	ex.	mo.	dev. (%)	ex.	mo.	dev. (%)	ex.	mo.	dev. (%)
512	$7.^o$	$7.^o$	0	$6.^o$	$6.^o$	0	$1.^o$	$2.^o$	13.3	$2.^o$	$5.^o$	26.2	$4.^o$	$4.^o$	0	$3.^o$	$3.^o$	0	$5.^o$	$1.^o$	36.1
1024	$7.^o$	$7.^o$	0	$6.^o$	$6.^o$	0	$4.^o$	$4.^o$	0	$3.^o$	$5.^o$	3.9	$2.^o$	$3.^o$	0.9	$5.^o$	$1.^o$	7.6	$1.^o$	$2.^o$	3.0
2048	$7.^o$	$7.^o$	0	$6.^o$	$6.^o$	0	$5.^o$	$5.^o$	0	$4.^o$	$4.^o$	0	$3.^o$	$3.^o$	0	$2.^o$	$2.^o$	0	$1.^o$	$1.^o$	0
4096	$7.^o$	$7.^o$	0	$6.^o$	$6.^o$	0	$5.^o$	$5.^o$	0	$3.^o$	$4.^o$	0.01	$4.^o$	$3.^o$	0.01	$2.^o$	$2.^o$	0	$1.^o$	$1.^o$	0
5120	$7.^o$	$7.^o$	0	$6.^o$	$6.^o$	0	$5.^o$	$5.^o$	0	$4.^o$	$4.^o$	0	$3.^o$	$3.^o$	0	$2.^o$	$2.^o$	0	$1.^o$	$1.^o$	0

Tabla 3.23: Orden de selección, de mejor a peor, en HPC160. Para la rutina de Strassen paralela versión maestro-esclavo (StMS).

En la tabla 3.23 se muestra para la plataforma HPC160 cual sería el orden de selección, de mejor a peor, que resultaría al comparar los tiempos de ejecución obtenidos variando el número de procesos paralelos p desde dos hasta ocho (se dispone de 8 procesadores). Para cada tamaño de problema, n, se indica el orden predicho por el modelo (mod.), el orden según los tiempos de ejecución experimentales (ex.) obtenidos a posteriori, y la desviación (dev.) en porcentaje entre los tiempos de ejecución experimentales para cada selección. Un valor de 0 en la desviación significa que el orden seleccionado por el modelo coincide con el obtenido experimentalmente. En la tabla se puede observar que la mejor selección es ejecutar la rutina con un número de procesos igual al de procesadores disponibles ($p = 8$), salvo cuando el tamaño del problema n es 512, en cuyo caso la mejor elección es ejecutar la rutina con 4 procesos. El orden de selección de p varía con el tamaño del problema y con el número de procesadores disponibles, pero con los modelos de tiempo de ejecución se consigue realizar un selección satisfactoria de p en casi todos los casos. En aquellos casos en los que la selección del mejor valor de p no coincide con el valor experimental, la desviación en el tiempo de ejecución es pequeña, siendo la media de las desviaciones de sólo un 10.12 %.

Como se observa en la tabla 3.22, la diferencia entre los tiempos de ejecución experimentales para $p = 5$, $p = 6$ y $p = 7$ es pequeña, con un 0.38 % de media entre $p = 5$ y $p = 6$, un 4 % de media entre $p = 6$ y $p = 7$ y un 3.23 % de media entre $p = 5$ y $p = 7$. Esto se debe a que el número de multiplicaciones a realizar en los procesos con más trabajo (tabla 3.21) es la misma en dichos casos y, dado que la plataforma HPC160 dispone de un sistema de comunicaciones rápido, el peso relativo del coste de las multiplicaciones es mayor que el de las comunicaciones. Por tanto se pueden aproximar los casos 2 y 3 al caso 1, y finalmente se podría modelar la rutina con sólo 2 modelos de tiempo de ejecución, que serían los correspondientes al caso 1 y al caso 4.

3.3.9 Comparativa entre las rutinas de multiplicación de matrices

En esta sección se realizará una comparativa entre las 6 rutinas de la multiplicación de matrices vistas anteriormente y con la multiplicación de matrices disponible en las librerías BLAS (**dgemm**) y PBLAS (**pdgemm**). Para la comparativa se van a utilizar versiones optimizadas de BLAS (BLASopt) para cada una de las plataformas computacionales empleadas previamente en la validación experimental de los modelos de tiempo de ejecución propuestos. Los resultados de los tiempos de ejecución que se van a comparar son los obtenidos con un número de procesos p igual a 4 (en el caso de rutinas paralelas). Para las rutinas que implementan el algoritmo de Strassen, en las que es posible utilizar diferentes niveles de recursión, se mostrarán los tiempos que resultarían de utilizar un l con el que el modelo nos indica que el tiempo de ejecución sería el más pequeño. Los tiempos que se muestran no incluyen el coste que supone la distribución de las matrices antes del inicio de la rutina de multiplicación de matrices (no se dispone de dicho coste para la rutina de PBLAS, StPBLAS y StMS), de tal forma que para la comparación únicamente se muestran los tiempos resultantes de la computación y de las comunicaciones

necesarias en las multiplicaciones.

En la figura 3.5 para las plataformas P4net y HPC160, se muestra a modo de resumen los tiempos obtenidos en la ejecución de las rutinas de multiplicación de matrices (Experimental), con los tiempos proporcionados por los modelos (Teórico) para diferentes tamaño del problema n.

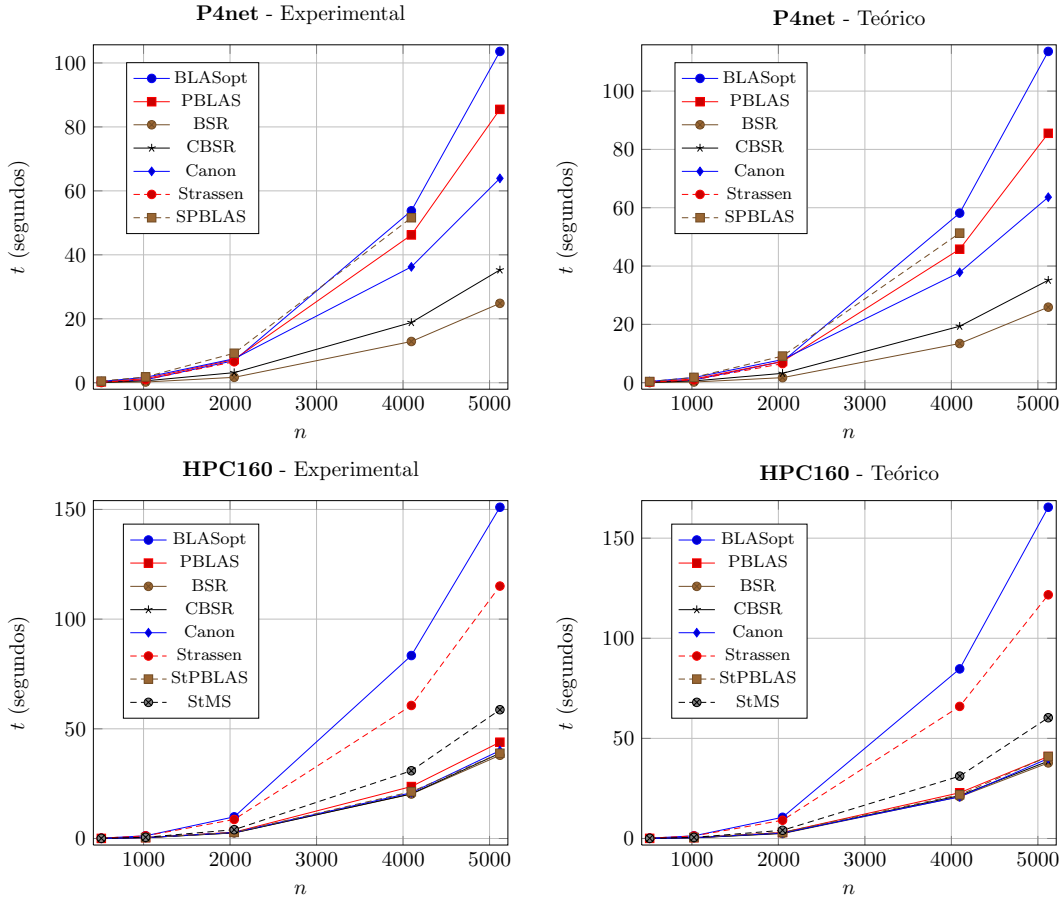


Figura 3.5: Comparación de los tiempos de ejecución experimentales y teóricos (en segundos) para **dgemm** de BLAS optimizada (BLASopt), **pdgemm** de PBLAS y las rutinas BSR, CBSR, Cannon, Strassen y StPBLAS, para distintos tamaños de problema, con $p = 4$. En P4net y HPC160.

Como se puede observar en la gráfica correspondiente a la plataforma P4net, debido al alto coste de las comunicaciones, los menores tiempos de ejecución se obtienen con la rutina que emplea una distribución block-striped de la matriz A y con la matriz B replicada en todos los procesos (rutina BSR) ya que el algoritmo implementado no requiere comunicación de datos entre procesos. La rutina con una distribución de A del tipo checkerboard block-striped (rutina CBSR) y la rutina que implementa el algoritmo de Cannon (rutina Cannon) proporcionan tiempos de ejecución peores que la rutina BSR, debido al coste adicional que supone el intercambio

de bloques necesarios para realizar la multiplicación, pero también inferiores a la multiplicación secuencial disponible en BLAS y la paralela disponible en PBLAS. La rutina secuencial de Strassen (Strassen), para tamaños de problema de 2048, proporciona menores tiempos de ejecución que la multiplicación disponible en BLAS y para tamaños de problema de 512, 1024 y 2048 mejores que la versión paralela de PBLAS. La implementación paralela de Strassen utilizando rutinas de PBLAS (rutina StPBLAS) es la que peores tiempos de ejecución proporciona en esta plataforma para el número de procesos y tamaños de problema analizados, de forma similar a lo que ocurría con la versión secuencial de Strassen, que tampoco mejoraba en todos los casos los tiempos de ejecución obtenidos con la multiplicación de matrices disponible en BLAS. Sólo para un tamaño de problema de 4096 el tiempo de ejecución de esta rutina es inferior al tiempo de ejecución dado por la rutina disponible en BLAS.

En la plataforma HPC160 el comportamiento es distinto al de la plataforma P4net, dado que la red que se utiliza para el intercambio de datos entre los procesos es más rápida. Como podemos observar en la figura 3.5, los tiempos de ejecución de la versiones paralelas son muy similares, además, y a diferencia de lo que ocurría en la plataforma P4net, la versión secuencial de la rutina de Strassen (rutina Strassen) mejora los tiempos de ejecución de la multiplicación disponible en la versión de BLAS (BLASopt) optimizada para la plataforma HPC160 si se utiliza el nivel de recursión adecuado. Lo mismo ocurre con la versión de Strassen paralela que utiliza la multiplicación y adición disponible en PBLAS (rutina StPBLAS), y podemos ver que sus tiempos de ejecución son inferiores a los de la multiplicación paralela disponible en PBLAS. Sólo para tamaños de problema de 1024 el tiempo de ejecución de la rutina StPBLAS es mayor que el de PBLAS, aunque la diferencia es tan sólo de un 5.1 %. La rutina que implementa una versión maestro-esclavo de la multiplicación de matrices de Strassen (rutina StMS) es la que requiere mayores tiempos de ejecución para un número de procesos $p = 4$, aunque se puede observar que sus tiempos de ejecución son inferiores a las versiones secuenciales analizadas.

n	BLASopt	PBLAS	BSR	CBSR	Cannon	Strassen	StPBLAS
512	16.09 %	0.30 %	23.11 %	1.21 %	2.29 %	0.18 %	22.52 %
1024	2.01 %	0.28 %	3.38 %	0.67 %	4.27 %	0.62 %	0.68 %
2048	5.12 %	0.14 %	2.44 %	1.62 %	5.58 %	0.14 %	0.88 %
4096	8.18 %	0.35 %	4.20 %	2.77 %	4.46 %		0.59 %
5120	9.65 %	0.03 %	4.33 %	0.38 %	0.47 %		

Tabla 3.24: Desviación entre los tiempos de ejecución experimentales y los proporcionados por los modelos en P4net. Para **dgemm** de BLAS optimizada (BLASopt), **pdgemm** de PBLAS y las rutinas BSR, CBSR, Cannon, Strassen y StPBLAS con $p = 4$.

En lo que se refiere a la estimación de los tiempos de ejecución proporcionada por los modelos, podemos ver en la tabla 3.24 para P4net que la desviación entre la aproximación proporcionada por el modelo y el tiempo medido una vez ejecutada la rutina, va desde un 0.03 % hasta un 23.11 % y que para HPC160 (tabla 3.25) varía entre un 0.27 % y un 48.31 %, si bien la desviaciones mayores ocurren en los casos en los que el tamaño del problema es pequeño.

n	BLASopt	PBLAS	BSR	CBSR	Cannon	Strassen	StPBLAS	StMS
512	4.58 %	22.95 %	7.96 %	4.41 %	48.31 %	9.83 %		6.25 %
1024	7.39 %	1.17 %	16.75 %	9.37 %	0.81 %	3.66 %	21.64 %	1.52 %
2048	7.26 %	4.17 %	2.98 %	0.99 %	2.08 %	4.94 %	10.55 %	2.11 %
4096	1.56 %	3.52 %	2.90 %	5.30 %	0.27 %	8.80 %	2.45 %	0.86 %
5120	9.57 %	6.72 %	0.50 %	0.64 %	0.31 %	5.78 %	3.40 %	2.68 %

Tabla 3.25: Desviación entre los tiempos de ejecución experimentales y teóricos en HPC160. Para **dgemm** de BLAS optimizada (BLASopt), **pdgemm** de PBLAS y las rutinas BSR, CBSR, Cannon, Strassen y StPBLAS con $p = 4$.

En resumen, observando los resultados obtenidos con las rutinas de multiplicación de matrices, se puede concluir que la metodología presentada para la obtención de los modelos de tiempo de ejecución permite decidir la rutina que proporciona tiempos de ejecución más pequeños y realizar una selección adecuada de los parámetros del algoritmo (AP) con los que se obtienen mejores tiempos de ejecución para distintos tamaños de problema a resolver y en plataformas con diferentes características.

3.4 Factorización de Cholesky

En esta sección aplicaremos la metodología de obtención de modelos de tiempo de ejecución a rutinas de álgebra lineal que pertenecen a los niveles superiores de la jerarquía clásica de librerías de álgebra lineal y que utilizan en su implementación rutinas de niveles inferiores. Se comenzará con dos versiones de la factorización de Cholesky que trabajan sin bloques. A continuación se analizará una versión por bloques y finalmente se analizará una versión paralela y por bloques. Aunque en las actuales plataformas de computación las rutinas de álgebra lineal que trabajan con bloques presentan en general un mejor comportamiento que las que no los utilizan, hemos incluido en este trabajo versiones sin bloques de alguna rutina con el fin de mostrar la aplicación de nuestra propuesta en la obtención de modelos de tiempo de ejecución detallados con los que se pueda aproximar adecuadamente el tiempo de ejecución en aquellos casos en los que se utilizan rutinas básicas de los niveles más bajos de la jerarquía de librerías de álgebra lineal y para las que suele ser más complicado obtener una buena aproximación a su tiempo de ejecución. Para cada una de las rutinas a las que se le aplicará nuestra metodología, en primer lugar veremos su base teórica y el algoritmo que sigue, posteriormente se describirán las librerías básicas de computación y comunicación que se han utilizado para su implementación, tras lo cual se pasará a construir su modelo teórico-experimental de tiempo de ejecución. Como en las rutinas vistas anteriormente, en el modelo aparecerán reflejados los parámetros del sistema (SP) y los del algoritmo (AP), y finalmente se validará experimentalmente el modelo construido en diferentes plataformas computacionales y con diferentes versiones de las librerías.

La factorización de Cholesky es un caso particular de la factorización de matrices que se apli-

ca cuando la matriz a factorizar es simétrica y definida positiva, y suele aparecer en problemas de mínimos cuadrados lineales, simulaciones de Montecarlo [Gen03] o en filtros de Kalman [Sor70]. Si la matriz $A \in R^{n \times n}$ es simétrica y definida positiva, entonces existe una única matriz triangular inferior $G \in R^{n \times n}$ con entradas diagonales positivas, tal que $A = GG^T$. Esta rutina se utiliza como paso previo en la resolución de un sistema de ecuaciones $Ax = b$, cuando A es simétrica y definida positiva, de tal forma que si calculamos la factorización de Cholesky y resolvemos los sistemas triangulares $Gy = b$ y $G^T x = y$, entonces $b = Gy = G(G^T x) = (GG^T)x = Ax$.

3.4.1 Factorizaciones de Cholesky sin bloques

El primer algoritmo implementado para la factorización de Cholesky es una versión producto matriz-vector en la que se sobrescribe A_{ij} con L_{ij} para todo $i \geq j$, calculando los nuevos valores de la columna $A_{j:n,j}$ de A para $j = 1, 2, 3, \ldots, n$ mediante las operaciones $A_{j:n,j} = A_{j:n,j} - A_{j:n,1:j-1} A_{j,1:j-1}^T$ y $A_{j:n,j} = A_{j:n,j} / \sqrt{A_{jj}}$. Esta última operación, que consiste en escalar las columnas $A_{j:n,j}$, tiene un coste lineal con n y el tiempo de ejecución es muy pequeño, por lo que no se considerará en el estudio. El coste del algoritmo es, por tanto, de $o(\frac{1}{3}n^3)$.

Algoritmo 3.2 Versión producto matriz vector de la factorización de Cholesky secuencial.

```
for j = 1 to n do
  if j > 1 then
    A(j:n, j) = A(j:n, j) − A(j:n, 1:j−1)A(j, 1:j−1)^T ⟵ dgemv
  end if
  A(j:n, j) = A(j:n, j)/√A(j, j)
end for
```

Para construir el modelo se identifican diferentes partes del algoritmo con diferentes rutinas básicas y sus correspondientes costes:

- **DGEMV** realiza la operación matriz-vector $A_{j:n,j} = A_{j:n,j} - A_{j:n,1:j-1} A_{j,1:j-1}^T$, utilizando la rutina **dgemv** de BLAS 2. El coste es de $o(2n^2)$.

El único SP que nos aparece es el coste de la operación de nivel 2 **dgemv** ($k_{2,dgemv}$), con lo que el tiempo de ejecución del algoritmo sería:

$$T = 2 \sum_{i=1}^{n-1} i(n-i) k_{2,dgemv} \tag{3.26}$$

y en una primera aproximación el tiempo teórico de ejecución del algoritmo podría calcularse como $\frac{1}{3}n^3 k_{2,dgemv}$, donde $k_{2,dgemv}$ se obtiene a partir del tiempo de ejecución de la rutina **dgemv** para una matriz de tamaño n, y del número de operaciones realizado por dicha rutina según la documentación de LAPACK. Con esta primera aproximación los resultados experimentales obtenidos no ofrecen una estimación del tiempo de ejecución satisfactoria, por lo que

resulta conveniente construir un modelo más detallado. Una mejor aproximación tiene en cuenta que el algoritmo utiliza submatrices rectangulares de de la matriz A, tamaño $(n-i) \times i$, para $i = 1, 2, \ldots, n-1$, y por tanto utiliza valores de $k_{2,dgemv}$ para distintos valores de i, es decir, para distintos tamaños de submatrices. Para el estudio se han tomado los valores $\frac{n}{8}$, $\frac{3n}{8}$, $\frac{5n}{8}$ y $\frac{7n}{8}$, con lo que tenemos los siguientes valores para $k_{2,dgemv}$: $k_{2,dgemv_{\frac{7n}{8}\times\frac{n}{8}}}$, $k_{2,dgemv_{\frac{5n}{8}\times\frac{3n}{8}}}$, $k_{2,dgemv_{\frac{3n}{8}\times\frac{5n}{8}}}$ y $k_{2,dgemv_{\frac{n}{8}\times\frac{7n}{8}}}$, siendo en este caso el modelo del tiempo de ejecución del algoritmo:

$$T = \frac{5}{96}n^3 k_{2,dgemv_{\frac{7n}{8}\times\frac{n}{8}}} + \frac{11}{96}n^3 k_{2,dgemv_{\frac{5n}{8}\times\frac{3n}{8}}} + \frac{11}{96}n^3 k_{2,dgemv_{\frac{3n}{8}\times\frac{5n}{8}}} + \frac{5}{96}n^3 k_{2,dgemv_{\frac{n}{8}\times\frac{7n}{8}}} \quad (3.27)$$

en el que sólo se han considerado los términos de orden superior y se han agrupado las operaciones en la anterior fórmula entre 1 y $\frac{n}{4}$, $\frac{n}{4}+1$ y $\frac{n}{2}$, $\frac{n}{2}+1$ y $\frac{3n}{4}$, $\frac{3n}{4}+1$ y $(n-1)$.

En el algoritmo 3.3 se puede ver un esquema de la versión producto externo en la que también se sobrescribe A_{ij} con L_{ij} para todo $i \geq j$, mediante las operaciones $A_{kk} = \sqrt{A_{kk}}$, $A_{k+1:n,k} = A_{k+1:n,k}/A_{kk}$, para $k = 1, 2, \ldots, n$, y $A_{j:n,j} = A_{j:n,j} - A_{j:n,k}A_{jk}$ para $j = k+1, k+2, \ldots, n$.

Algoritmo 3.3 Versión producto externo de la factorización de Cholesky secuencial.

for $k = 1$ to n **do**
  $A(k,k) = \sqrt{A(k,k)}$
  $A(k+1:n,k) = A(k+1:n,k)/A(k,k)$
  for $j = k+1$ to n **do**
    $A(j:n,j) = A(j:n,j) - A(j:n,k)A(j,k) \longleftarrow$ **daxpy**
  end for
end for

Las dos primeras operaciones no se considerarán en el modelo, dado que su tiempo de ejecución es pequeño respecto a la tercera operación, y sólo se identificará la última parte del algoritmo con las rutinas básicas utilizadas y sus costes:

- **DAXPY** realiza la operación producto de un vector por un escalar más otro vector (operación $A_{j:n,j} = A_{j:n,j} - A_{j:n,k}A_{jk}$ del algoritmo) utilizando la rutina **daxpy** de BLAS nivel 1. El coste es de $o(2n)$.

El único parámetro del sistema es el coste de la operación de nivel 1 **daxpy** (k_1), con lo que el tiempo de ejecución del algoritmo sería:

$$T = 2\sum_{i=1}^{n-1} i(n-i)k_{1,daxpy} \quad (3.28)$$

De forma similar a como se actuó en el algoritmo anterior, se aproxima la anterior fórmula teniendo en cuenta que el algoritmo accede a columnas de tamaño $(n-i)$ de la matriz A,

para $i = 1, 2, \ldots, n-1$, y utilizando únicamente los valores de $k_{1,daxpy}$: $k_{1,daxpy_{\frac{7n}{8}}}$, $k_{1,daxpy_{\frac{5n}{8}}}$, $k_{1,daxpy_{\frac{3n}{8}}}$ y $k_{1,daxpy_{\frac{n}{8}}}$.

El tiempo de ejecución del algoritmo se modela por tanto como:

$$T = \frac{5}{96}n^3 k_{1,daxpy_{\frac{7n}{8}}} + \frac{11}{96}n^3 k_{1,daxpy_{\frac{5n}{8}}} + \frac{11}{96}n^3 k_{1,daxpy_{\frac{3n}{8}}} + \frac{5}{96}n^3 k_{1,daxpy_{\frac{n}{8}}} \tag{3.29}$$

Una vez construidos los modelos de tiempo de ejecución para ambas rutinas de la factorización de Cholesky, en la siguiente sección se mostrará su validación experimental y se explicará el procedimiento para la obtención de los parámetros del sistema y la aplicación de los modelos en la selección de la librería con la que se obtienen mejores tiempos de ejecución.

Resultados experimentales

Para la validación experimental de los modelos que aproximan los tiempos de ejecución, se ha utilizado un sistema con procesadores Intel Pentium III y otro con procesadores Intel Pentium 4. Como librerías básicas de álgebra lineal se ha empleado una BLAS de referencia (BLASref) y una versión de BLAS optimizada para Pentium III y Pentium 4 (BLASopt).

En la tabla 3.26 (Pentium III y Pentium 4) se muestran los valores del *SP* $k_{2,dgemv}$ de la rutina **dgemv** de BLAS nivel 2 que aparece en el modelo de la versión producto matriz-vector. Puede observarse la dependencia de los valores del *SP* respecto al tamaño del problema y respecto a la forma de la submatriz. Por otra parte, se comprueba que los valores obtenidos para $k_{2,dgemv}$ con BLASref son peores que con BLASopt en Pentium III, mientras que en Pentium 4 los peores resultados se obtienen con BLASopt, aun tratándose de una versión específicamente optimizada para Pentium 4. Esto nos indica la importancia de un estudio experimental como el planteado de cara a una correcta selección de la librería básica.

En la tabla 3.27 (Pentium III y Pentium 4) se muestran los valores del *SP* $k_{1,daxpy}$ que aparece en el modelo de tiempo de ejecución de la factorización de Cholesky que implementa un algoritmo basado en productos externos. Los valores mostrados se han obtenido ejecutando la rutina **daxpy** en columnas de una matriz de tamaño n situadas en las posiciones $\frac{n}{8}$, $\frac{3n}{8}$, $\frac{5n}{8}$ y $\frac{7n}{8}$ de tamaños $\frac{7n}{8}$, $\frac{5n}{8}$, $\frac{3n}{8}$ y $\frac{n}{8}$, de forma similar a como se emplea en el algoritmo. Se han vuelto a utilizar las librerías BLAS de referencia (BLASref) y una versión optimizada para Pentium III y Pentium 4 (BLASopt). En Pentium III se obtienen mejores valores con BLASref, mientras que la versión optimizada de la librería (BLASopt) los obtiene para Pentium 4. Ahora ocurre lo contrario que con la rutina **dgemv**; en Pentium III es preferible utilizar BLASref y en Pentium 4 es preferible utilizar BLASopt; con lo que vemos que para rutinas diferentes puede ser preferible emplear librerías diferentes.

		Pentium III		Pentium 4	
n	Tamaño submatriz	BLASref	BLASopt	BLASref	BLASopt
256	224×32	0.0250	**0.0158**	**0.0049**	0.0060
	160×96	0.0225	**0.0139**	0.0039	**0.0037**
	96×160	0.0170	**0.0146**	0.0065	**0.0047**
	32×224	**0.0220**	0.0258	0.0083	**0.0050**
512	448×64	0.0274	**0.0086**	**0.0036**	0.0129
	320×192	0.0164	**0.0076**	**0.0038**	0.0100
	192×320	0.0166	**0.0076**	**0.0036**	0.0099
	64×448	0.0182	**0.0080**	**0.0059**	0.0113
1024	896×128	0.0199	**0.0079**	**0.0031**	0.0109
	640×384	0.0160	**0.0073**	**0.0032**	0.0124
	384×640	0.0160	**0.0074**	**0.0030**	0.0139
	128×896	0.0167	**0.0074**	**0.0074**	0.0151
2048	1792×256	0.0169	**0.0065**	**0.0028**	0.0093
	1280×768	0.0157	**0.0063**	**0.0033**	0.0111
	768×1280	0.0161	**0.0067**	**0.0033**	0.0099
	256×1792	0.0163	**0.0072**	**0.0041**	0.0106
4090	3584×512			**0.0027**	0.0073
	2560×1536			**0.0028**	0.0073
	1536×2560			**0.0029**	0.0077
	512×3584			**0.0031**	0.0091
8192	7168×1024			**0.0030**	0.0101
	5120×3072			**0.0029**	0.0103
	3072×5120			**0.0028**	0.0105
	1024×7168			**0.0029**	0.0123

Tabla 3.26: Valores de $k_{2,dgemv}$ (en μsegundos) con diferentes librerías, en Pentium III y Pentium 4, para distintos tamaños del problema y para diferentes tamaños de submatriz.

n	Tamaño columnas submatriz	Pentium III		Pentium 4	
		BLASref	BLASopt	BLASref	BLASopt
256	224	**0.0263**	0.0286	0.0089	**0.0085**
	160	**0.0250**	0.0256	**0.0094**	0.0100
	96	**0.0219**	0.0234	0.0677	**0.0656**
	32	**0.0312**	0.0438	0.0313	**0.0041**
512	448	0.0279	**0.0272**	0.0065	**0.0058**
	320	0.0219	**0.0213**	0.0066	**0.0066**
	192	0.0234	**0.0224**	0.0089	**0.0089**
	64	**0.0297**	0.0391	0.0023	**0.0023**
1024	896	**0.0278**	0.0285	0.0048	**0.0045**
	640	0.0208	**0.0205**	0.0050	**0.0045**
	384	**0.0201**	0.0206	0.0057	**0.0057**
	128	**0.0234**	0.0234	0.0012	**0.0012**
2048	1792	**0.0242**	0.0289	0.0040	**0.0037**
	1280	0.0217	**0.0211**	0.0038	**0.0036**
	768	**0.0203**	0.0207	0.0044	**0.0039**
	256	**0.0215**	0.0238	0.0078	**0.0078**
4096	3584			0.0034	**0.0030**
	2560			0.0034	**0.0030**
	1536			0.0040	**0.0033**
	512			0.0059	**0.0047**
8192	7168			0.0047	**0.0042**
	5120			0.0040	**0.0035**
	3072			0.0075	**0.0073**
	1024			0.0044	**0.0040**

Tabla 3.27: Valores de $k_{1,daxpy}$ (en μsegundos) con diferentes librerías, en Pentium III y Pentium 4, para los tamaños de las columnas de la matriz utilizados en el modelo y para cada tamaño del problema.

En la tabla 3.28 se muestran los tiempos de ejecución para las librerías utilizadas en Pentium III y Pentium 4, los calculados con el modelo y los obtenidos experimentalmente. Como se puede observar, en Pentium III y para el algoritmo producto matriz-vector los mejores tiempos de ejecución se obtienen con BLAS optimizada (BLASopt), mientras que en Pentium 4 es preferible utilizar BLAS de referencia (BLASref) en lugar de una versión optimizada para este procesador. En lo que se refiere al algoritmo producto externo, en la tabla 3.28 se puede observar que para los tamaños de problema analizados en Pentium III no conviene utilizar la versión optimizada de la librería (BLASopt) cuando el tamaño del problema se hace más grande, mientras que en Pentium 4 los mejores tiempos de ejecución se obtienen con la librería optimizada (BLASopt).

	producto matriz-vector				producto externo			
	BLASref		BLASopt		BLASref		BLASopt	
n	mod.	exp.	mod.	exp.	mod.	exp.	mod.	exp.
				Pentium III				
256	0.117	0.058	**0.081**	**0.041**	0.140	0.061	0.123	0.058
512	0.826	0.702	**0.350**	**0.335**	1.099	1.021	1.135	1.041
1024	5.983	5.733	**2.660**	**2.615**	7.889	8.749	7.952	8.872
2048	46.135	45.450	**18.931**	**19.199**	61.817	71.544	64.784	76.184
				Pentium 4				
256	0.032	0.020	**0.026**	**0.014**	0.192	0.024	0.188	0.021
512	**0.179**	**0.154**	0.476	0.416	0.446	0.218	0.442	0.209
1024	**1.350**	**1.119**	4.690	3.910	2.244	1.646	2.167	1.596
2048	**9.551**	**8.586**	29.667	28.793	13.316	13.048	12.550	12.687
4096	**65.604**	**69.214**	176.979	211.739	90.881	109.916	76.978	102.932
8192	**521.343**	**588.568**	1954.849	2245.958	983.491	805.458	917.109	775.526

Tabla 3.28: Comparación del tiempo (en segundos) de ejecución experimental con el tiempo de ejecución del modelo utilizando BLAS de referencia (BLASref) y BLAS optimizada (BLASopt), en Pentium III y Pentium 4. Para el algoritmo producto matriz-vector y producto externo.

La división del modelo en partes, con el fin de introducir el coste del parámetro en función del tamaño y forma de la submatriz en la que la rutina realiza las operaciones, junto con la obtención de los SP teniendo en cuenta el modo cómo las rutinas básicas actualizan los datos, predice bien los resultados experimentales y permite la selección automática de la librería que proporciona tiempos de ejecución óptimos, aunque dicha librería no sea la esperada. Como puede observarse, la selección de la librería básica depende de la rutina utilizada y del tamaño del problema, pero dicha selección es realizada adecuadamente utilizando la información proporcionada por nuestra propuesta de modelado.

Adicionalmente, los modelos obtenidos pueden utilizarse conjuntamente para predecir qué librería básica y qué algoritmo son preferibles utilizar. Ampliando la aproximación de otros trabajos [LSF97, NT04] donde se estudiaban polialgoritmos y se decidía el algoritmo a utilizar en la resolución de un problema, con nuestra propuesta se puede realizar una selección del algoritmo y de la librería básica de forma conjunta; aunque cada implementación en una librería podría

considerarse un algoritmo distinto, y viceversa, cada algoritmo podría estar implementado en una librería distinta.

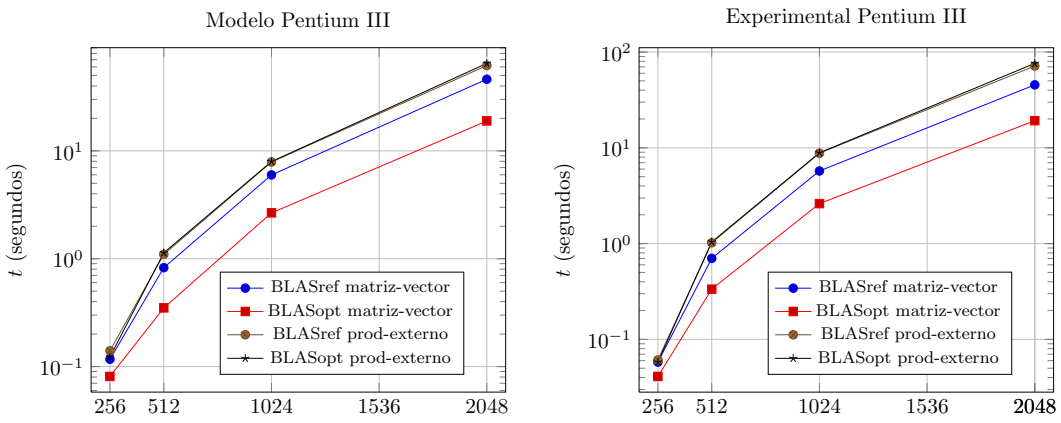


Figura 3.6: Selección del algoritmo y librería óptimos en Pentium III (tiempos en escala logarítmica).

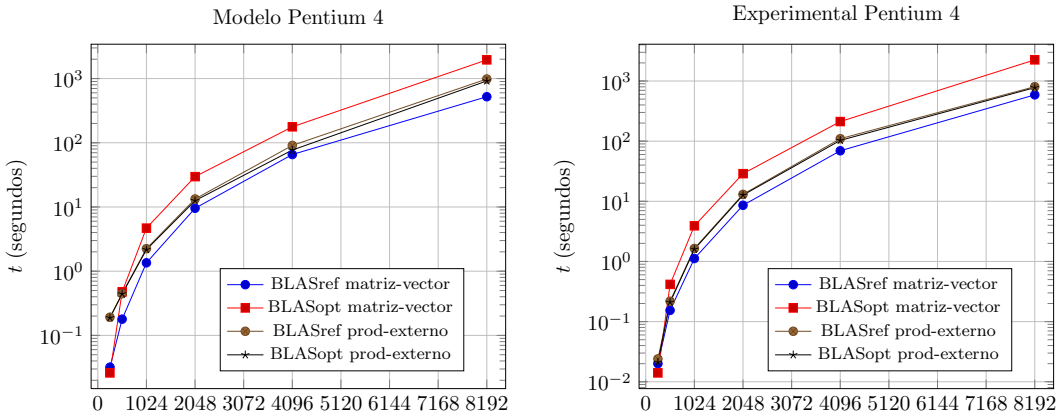


Figura 3.7: Selección del algoritmo y librería óptimos en Pentium 4 (tiempos en escala logarítmica).

En la plataforma con procesadores Pentium III observamos (tabla 3.28 y figura 3.6) que el orden de preferencia de los algoritmos y librerías es: matriz-vector con BLASopt, matriz-vector con BLASref, producto externo con BLASref y producto externo con BLASopt, salvo para tamaños de problema de 256, en que se intercambia el orden de los dos últimos. Como se puede comprobar, en todos los casos la predicción teórica (la que se realizaría utilizando la información proporcionada por los modelos de tiempo de ejecución) elige el algoritmo y librería en el orden apropiado. Lo mismo ocurre en la plataforma con procesadores Pentium 4 (tabla 3.28 y figura 3.7), aunque en este caso el orden de preferencia es: matriz-vector con BLASref, producto externo con BLASopt, producto externo con BLASref y matriz-vector con

BLASopt, salvo con tamaños de problema de 256, en el que la mejor selección es matriz-vector con BLASopt y manteniendo el orden de selección anterior en el resto de casos.

3.4.2 Factorización de Cholesky secuencial por bloques

En este apartado describiremos la versión por bloques de la rutina de factorización de Cholesky. El algoritmo implementado corresponde a la versión "dot product" de [GL96] en el que se sobrescribe la parte triangular inferior de A con la parte triangular inferior de G siendo su coste de $o(\frac{1}{3}n^3)$. Divididas A y G en los bloques A_{11}, A_{21}, A_{21}^T, A_{22}, G_{11}, G_{21} y G_{22} (figura 3.8); donde el bloque A_{11} es $b \times b$, A_{21} es $(n-b) \times b$, y A_{22} es $(n-b) \times (n-b)$ con G_{11} y G_{22} triangulares inferiores.

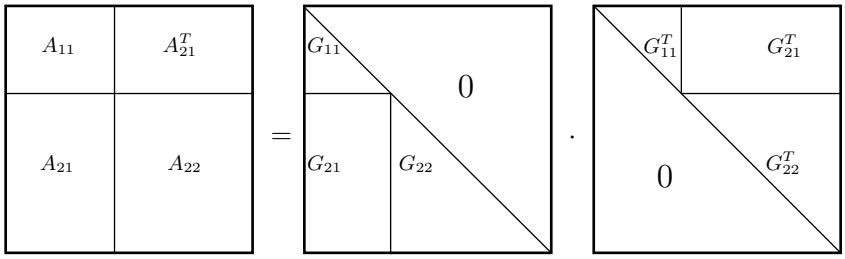


Figura 3.8: División por bloques de las matrices A y G en la factorización de Cholesky.

Supuesto G_{11}, el factor de Cholesky de A_{11}, conocido; tenemos las operaciones $G_{21} = A_{21}(G_{11}^T)^{-1}$ y, $\tilde{A}_{22} = A_{22} - G_{21}G_{21}^T = G_{22}G_{22}^T$. El algoritmo continuará aplicando estas mismas operaciones a la matriz $\tilde{A}_{22}$ (figura 3.9) de dimensión $(n-b) \times (n-b)$.

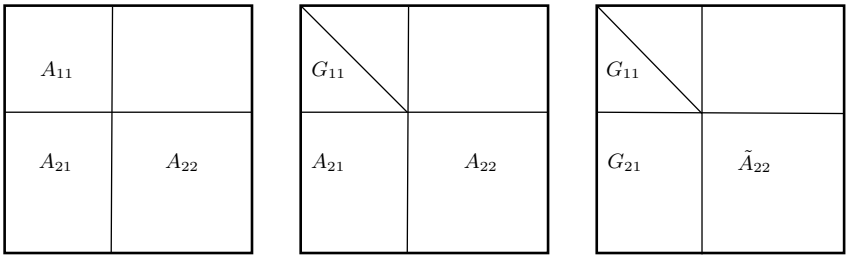


Figura 3.9: Proceso de la factorización de Cholesky por bloques en la matriz A.

Con el fin de seguir la estructura jerárquica clásica de las librerías de álgebra lineal, la rutina que hemos programado utiliza, de forma similar a su equivalente en la librería LAPACK, rutinas de la misma LAPACK y de BLAS. Para construir el modelo se identifican diferentes partes del algoritmo 3.4 con diferentes rutinas básicas y diferentes costes:

– **DPOTF2**: calcula la factorización de Cholesky en los bloques de la diagonal, utilizando la rutina de nivel 2 de LAPACK **dpotf2**. El coste es por tanto de $o(\frac{1}{3}b^3)$, y la operación

Algoritmo 3.4 Factorización de Cholesky secuencial por bloques.

for $j = 1$ to n/b **do**
  for $i = j$ to n/b **do**
    if $i = j$ **then**
      $A_{ij} = A_{ij} - \sum_{k=1}^{j-1} A_{ik}A_{jk}^T \longleftarrow$ **dsyrk**
      Calcular factorización de Cholesky para $A_{ij} \longleftarrow$ **dpotf2**
    else
      $A_{ij} = A_{ij} - \sum_{k=1}^{j-1} A_{ik}A_{jk}^T \longleftarrow$ **dgemm**
      Resolver $A_{ij}A_{jj}^T = A_{ij}$ para $A_{ij} \longleftarrow$ **dtrsm**
    end if
  end for
end for

que implementa es $A_{11} = G_{11}G_{11}^T$.

- **DTRSM**: calcula el factor para bloques por debajo del bloque de la diagonal, utilizando la rutina de nivel 3 de BLAS **dtrsm**. El coste es por tanto de $o(b^3)$, y realiza la operación $G_{21} = A_{21}(G_{11}^T)^{-1}$.

- **DSYRK**: actualiza bloques por debajo del bloque de la diagonal utilizando la rutina de nivel 3 de BLAS **dgemm**, y bloques de la diagonal utilizando la rutina BLAS de nivel 3 **dsyrk**. Los costes de estas rutinas serán por tanto de $o(2b^3)$ y $o(b^2(b+1))$ respectivamente. Las operaciones que se realizan con este algoritmo corresponden a $\tilde{A}_{22} = A_{22} - G_{21}G_{21}^T = G_{22}G_{22}^T$.

Para la construcción del modelo en este algoritmo, tenemos cuatro SP (coste de las rutinas de BLAS 3 y LAPACK utilizadas): $k_{3,trsm}$, $k_{3,gemm}$, $k_{3,syrk}$ y $k_{2,potf2}$; y un único parámetro del algoritmo (AP): b, tamaño del bloque utilizado. El tiempo de ejecución de este algoritmo puede ser modelado con:

$$\begin{aligned} T &= 2k_{3,gemm}\left(\frac{n^3}{6} - \frac{n^2 b}{2} + \frac{nb^2}{3}\right) + k_{3,trsm}\left(\frac{n^2 b}{2} - \frac{nb^2}{2}\right) \\ &\quad + k_{3,syrk}\left(\frac{n^2}{2} - \frac{nb}{2}\right)(b+1) + k_{2,potf2}\frac{nb^2}{3} \end{aligned} \tag{3.30}$$

En general los valores de k_2 y k_3 dependerán del tamaño del problema, n, y del tamaño del bloque de cálculo que se utilice, b. A continuación se obtendrán los valores de los SP mencionados y se determinará experimentalmente su dependencia en diferentes plataformas computacionales y con diferentes versiones de BLAS, y seguidamente se realizará la validación experimental de nuestra propuesta de modelo de tiempo de ejecución para esta versión de la factorización de Cholesky secuencial por bloques.

Resultados experimentales

Como ya se ha comentado anteriormente, en primer lugar es necesario obtener el valor de los SP que aparecen en el modelo de tiempo de ejecución y su dependencia respecto a n y b, para cada plataforma computacional y librería básica. Para la obtención de dichos valores se han diseñado una serie de experimentos en los que se realizan llamadas a cada una de las rutinas básicas empleadas y que proporcionan los valores de cada uno de los SP para diferentes tamaños de problema, tamaños de bloque de cálculo y librerías básicas disponibles. Hemos empleado como plataformas computacionales un sistema con procesador Pentium III y otro sistema con procesador Pentium 4, y como librerías básicas una BLAS de referencia (BLASref), una versión de BLAS optimizada para Pentium III y para Pentium 4 (BLASopt). Como valores representativos para el tamaño del problema, n, se han escogido los valores de 256, 512, 1024, 1280, 2048, 2560 y 4096. Para el tamaño de bloque, b, los valores escogidos han sido 16, 32, 64, 128 y 256. En las pruebas realizadas en ambos sistemas, se ha comprobado que el valor de los SP se muestra constante respecto al tamaño de la matriz y que su valor depende del tamaño del bloque de cálculo. En la tabla 3.29 ($k_{3,dtrsm}$), en la tabla 3.30 ($k_{3,dgemm}$), en la tabla 3.31 ($k_{3,dsyrk}$) y en la tabla 3.32 ($k_{2,dpotf2}$) se muestran los valores obtenidos en cada plataforma y con cada librería.

			Tamaño de bloque, b				
Sistema	Librería	n	16	32	64	128	256
PIII	BLASref	256,...,4096	0.0100	**0.0064**	0.0103	0.0199	0.0289
	BLASopt	256,...,4096	0.0083	0.0044	0.0030	0.0026	**0.0023**
P4	BLASref	256,...,4096	0.0049	0.0028	**0.0019**	0.0041	0.0048
	BLASopt	256,...,4096	0.0044	0.0023	0.0017	0.0014	**0.0012**

Tabla 3.29: Valores del SP $k_{3,dtrsm}$ (en μsegundos) con el tamaño de bloque óptimo, para la rutina Cholesky secuencial utilizando BLAS de referencia y BLAS optimizada. En Pentium III y Pentium 4.

			Tamaño de bloque, b				
Sistema	Librería	n	16	32	64	128	256
PIII	BLASref	256,...,4096	0.0098	**0.0060**	0.0121	0.0163	0.0160
	BLASopt	256,...,4096	0.0071	0.0029	0.0022	0.0019	**0.0018**
P4	BLAref	256,...,4096	0.0043	**0.0025**	0.0039	0.0042	0.0033
	BLASopt	256,...,4096	0.0029	0.0016	0.0009	0.0005	**0.0005**

Tabla 3.30: Valores del SP $k_{3,dgemm}$ (en μsegundos) con el tamaño de bloque óptimo, para la rutina Cholesky secuencial utilizando BLAS de referencia y BLAS optimizada. En Pentium III y Pentium 4.

Se puede observar que el valor de los SP asociados a las rutinas básicas varía de una rutina

			Tamaño de bloque, b				
Sistema	Librería	n	16	32	64	128	256
PIII	BLASref	256,...,4096	0.0126	**0.0080**	0.0113	0.0171	0.0166
	BLASopt	256,...,4096	0.0085	0.0057	0.0041	0.0030	**0.0024**
P4	BLASref	256,...,4096	0.0064	**0.0033**	0.0036	0.0052	0.0039
	BLASopt	256,...,4096	0.0055	0.0028	0.0014	0.0012	**0.0007**

Tabla 3.31: Valores del SP $k_{3,dsyrk}$ (en μsegundos) con el tamaño de bloque óptimo, para la rutina Cholesky secuencial utilizando BLAS de referencia y BLAS optimizada. En Pentium III y Pentium 4.

			Tamaño de bloque, b				
Sistema	Librería	n	16	32	64	128	256
PIII	BLASref	256,...,4096	0.0212	0.0100	**0.0086**	0.0121	0.0153
	BLASopt	256,...,4096	0.0212	**0.0096**	0.0131	0.0142	0.0118
P4	BLASref	256,...,4096	0.0469	0.0094	**0.0034**	0.0050	0.0045
	BLASopt	256,...,4096	0.0359	0.0077	**0.0026**	0.0046	0.0056

Tabla 3.32: Valores del SP $k_{2,dpotf2}$ (en μsegundos) con el tamaño de bloque óptimo, para la rutina Cholesky secuencial utilizando BLAS de referencia y BLAS optimizada. En Pentium III y Pentium 4.

a otra, incluso cuando éstas son del mismo nivel, que el valor de b con el que se obtiene el valor más pequeño para los SP es distinto en cada plataforma y que, además, también depende de la librería básica utilizada. De tal forma que se hace necesario construir un modelo de tiempo de ejecución detallado, en el que queden reflejadas las variaciones indicadas en los SP, tal y como aparece en nuestra propuesta de modelado.

Para la validación experimental del modelo de tiempo de ejecución propuesto, se han realizado ejecuciones de la rutina para diferentes tamaños del problema, utilizando en cada caso todos los tamaños de bloque seleccionados previamente durante la obtención de los SP, con el fin de comprobar si la selección del b óptimo (tamaño de bloque con el que se obtiene el menor tiempo de ejecución de la rutina) realizada con el modelo es correcta. Para cada caso se compara el tiempo de ejecución obtenido tras la ejecución de la rutina en cada plataforma computacional y librería básica, con la aproximación al tiempo de ejecución proporcionada por el modelo. En la tabla 3.33 (Pentium III) y en la 3.34 (Pentium 4) se muestra el menor tiempo de ejecución proporcionado por el modelo, el tamaño de bloque con el que se obtiene (blo.), el menor tiempo de ejecución obtenido experimentalmente (exp. inf), y la desviación (des.) entre ambos tiempos de ejecución. Si la selección del b óptimo no coincide, se muestra el tiempo experimental obtenido utilizando el tamaño de bloque óptimo predicho por el modelo (exp. mod). Como se puede observar, en el caso de utilizar BLAS de referencia el b óptimo es el mismo para todos los tamaños de matriz analizados y, en el caso de la versión optimizada para Pentium

III y Pentium 4, el b óptimo depende del tamaño de la matriz y no coincide con el valor del b óptimo observado con BLAS de referencia.

	BLASref					BLASopt				
	mod.		exp. (inf./mod.)		des.	mod.		exp. (inf./mod.)		des.
n	t	blo.	t	blo.	(%)	t	blo.	t	blo.	(%)
256	0.04	32	0.03/0.03	64/32	60/20	0.02	64	**0.01**	64	30
512	0.28	32	0.28	32	0.2	0.12	64	**0.11**/0.12	32/64	24/8
1024	2.19	32	2.20	32	0.1	0.85	128	**0.82**	128	3.5
2048	17.39	32	17.32	32	0.4	5.97	128	**5.81**	128	2.7

Tabla 3.33: Tiempo de ejecución (en segundos) con el tamaño de bloque óptimo, de la rutina Cholesky utilizando BLAS de referencia y BLAS optimizada. En Pentium III.

	BLASref					BLASopt				
	mod.		exp. (inf./mod.)		des.	mod.		exp. (inf./mod.)		des.
n	t	blo.	t	blo.	(%)	t	blo.	t	blo.	(%)
256	0.02	32	0.01/0.01	64/32	42/25	0.01	64	**0.01**	64	15
512	0.12	32	0.11	32	4	0.05	64	**0.05**	64	2
1024	0.91	32	0.92	32	2	0.31	128	**0.31**	128	0.4
2048	7.17	32	8.10	32	12	1.99	128	**2.01**	128	2
4096	56.98	32	70.91	32	20	13.29	256	**13.34**	256	0.4

Tabla 3.34: Tiempo de ejecución (en segundos) con el tamaño de bloque óptimo de la rutina Cholesky utilizando BLAS de referencia y BLAS optimizada. En Pentium 4.

Con la información mostrada en las tablas, se puede comprobar que en esta versión de la factorización de Cholesky donde se utilizan rutinas básicas de BLAS de nivel 2 y 3 y de LAPACK, los mejores tiempos de ejecución han sido obtenidos, para ambas plataformas computacionales y en todos los casos analizados, con las versiones optimizadas de las librerías.

El modelo de tiempo de ejecución propuesto permite predecir correctamente la librería y el tamaño de bloque óptimo, excepto en unos pocos casos en los que el incremento del tiempo de ejecución de la rutina, utilizando el tamaño de bloque con el que el tiempo de ejecución aproximado por el modelo es menor, ha sido en Pentium III del 14 % con BLASref y $n = 256$ y del 2 % con BLASopt y $n = 512$. En Pentium 4, este incremento ha sido del 11 % con BLASref para $n = 256$.

A continuación vamos a aplicar nuestra propuesta de modelado a una rutina paralela de la factorización de Cholesky, en la que aparecerán dos AP adicionales: el número de procesos paralelos y su configuración lógica en una malla de dos dimensiones.

3.4.3 Factorización de Cholesky paralela por bloques

El algoritmo es una variante de la versión secuencial por bloques ya utilizada anteriormente, en el que se realiza una distribución cíclica del trabajo en una malla de dos dimensiones de $p = r \times c$ procesos lógicos (figura 3.10), de la misma forma que se efectúa en la librería ScaLAPACK. En esta rutina se realizan llamadas a rutinas de BLAS y LAPACK para los cálculos aritméticos y de MPI para las comunicaciones.

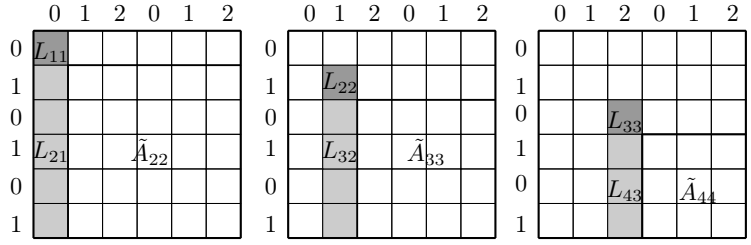
	0	1	2	0	1	2
0	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}
1	A_{21}	A_{22}	A_{23}	A_{24}	A_{25}	A_{26}
0	A_{31}	A_{32}	A_{33}	A_{34}	A_{35}	A_{36}
1	A_{41}	A_{42}	A_{43}	A_{44}	A_{45}	A_{46}
0	A_{51}	A_{52}	A_{53}	A_{54}	A_{55}	A_{56}
1	A_{61}	A_{62}	A_{63}	A_{64}	A_{65}	A_{66}

(a) distribución matriz 6×6 en 2×3 procesos

	0		1		2	
	A_{11}	A_{14}	A_{12}	A_{15}	A_{13}	A_{16}
0	A_{31}	A_{34}	A_{32}	A_{35}	A_{33}	A_{36}
	A_{51}	A_{54}	A_{52}	A_{55}	A_{53}	A_{56}
	A_{21}	A_{24}	A_{22}	A_{25}	A_{23}	A_{26}
1	A_{41}	A_{44}	A_{42}	A_{45}	A_{43}	A_{46}
	A_{61}	A_{64}	A_{62}	A_{65}	A_{63}	A_{66}

(b) bloques de la matriz en cada proceso

Figura 3.10: Distribución cíclica por bloques con $\frac{n}{b} = 6$ y $p = 2 \times 3$. Los números a la izquierda y arriba de la matriz representan coordenadas de los procesos en la malla 2×3.



(a) Primer paso (b) Segundo paso (c) Tercer paso

Figura 3.11: Distribución de los cálculos en los tres primeros pasos de la rutina de Cholesky paralela por bloques, con $\frac{n}{b} = 6$ y $p = 2 \times 3$. Los números a izquierda y arriba de la matriz representan coordenadas de los procesos en la malla de 2×3.

Calculado el factor L_{11} (factor de Cholesky de A_{11}) por el proceso {0,0} (figura 3.11 a), se tiene la operación $L_{21} = A_{21}(L_{11}^T)^{-1}$, realizada por los procesos de la columna 0 de la malla, y la operación $\tilde{A}_{22} = A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T$, realizada por todos los procesos en la malla. En los siguientes pasos se distribuirá el trabajo tal como se muestra en las figuras 3.11 b y 3.11 c.

Para el estudio de esta rutina, se ha realizado una implementación del algoritmo 3.5, donde *map_col* y *map_fil* son vectores con la asignación de columnas y filas de la matriz, respectivamente; y *coords* es un vector con las coordenadas del proceso dentro de la malla.

Algoritmo 3.5 Factorización de Cholesky paralela por bloques.

```
for j = 1 to j < n; j = j + b do
  if map_fil[j] == coords[0] and map_col[j] == coords[1] then
    Calcular la factorización de Cholesky para A_jj ⟵ dpotf2
    Send(A_jj,rowwise)
    for i = j + b to j < n; i = i + b do
      if map_fil[i] == coords[0] then
        Resolver A_ij A_jj^T = A_ij para A_ij ⟵ dtrsm
        Send(A_ij,rowwise)
      else
        Recv(A_ij)
      end if
    end for
    Send(A_{j+b:n,j},columwise)
  else if map_col[j] == coords[1] then
    Recv(A_jj)
    for i = j + b to j < n; i = i + b do
      if map_fil[i] == coords[0] then
        Resolver A_ij A_jj^T = A_ij para A_ij ⟵ dtrsm
        Send(A_ij,rowwise)
      else
        Recv(A_ij)
      end if
    end for
    Send(A_{j+b:n,j},columwise)
  end if
  Recv(A_{j+b:n,j})
  for h = j + b to n; h = h + b do
    if map_col[h] == coords[1] then
      for l = h to l < n; l = l + b do
        if map_fil[l] == coords[0] then
          if l == h then
            A_lh = A_lh − Σ_{k=1}^{h−1} A_lk A_hk^T ⟵ dsyrk
          else
            A_lh = A_lh − Σ_{k=1}^{h−1} A_lk A_hk^T ⟵ dgemm
          end if
        end if
      end for
    end if
  end for
end for
```

A continuación se identifican diferentes partes de la rutina con su equivalente en ScaLAPACK. Para cada parte se indican las rutinas básicas de BLAS y LAPACK utilizadas y sus costes, con el fin de obtener su modelo de tiempo de ejecución:

- **PDPOTF2**: el proceso $\{r_i, c_j\}$ con el bloque $b \times b$ diagonal A_{11}, realiza la factorización de Cholesky en A_{11} y obtiene el factor L_{11}, utilizando la rutina de nivel 2 de LAPACK **dpotf2**. El coste aritmético es de $o(\frac{1}{3}b^3)$, y sólo se intercambia información entre los procesos en el caso que A_{11} no sea definida positiva, por lo que no se considerará coste en la comunicación.
- **PDTRSM**: el proceso $\{r_i, c_j\}$ con el factor L_{11} lo difunde a todos los procesos en su misma columna, que obtienen por medio de la solución de un sistema de ecuaciones triangular, de tamaño $(n - ib) \times b$, la factorización de los bloques por debajo del bloque diagonal recientemente factorizado, factor L_{21}, utilizando la rutina de nivel 3 de BLAS **dtrsm**. El coste aritmético es de $o(b^3)$.
- **PDSYRK**: los bloques de L_{21} se difunden primero por columna de procesos, a continuación se difunden por filas de procesos, con lo que cada proceso puede ahora actualizar su parte de A_{22}; operación $\tilde{A_{22}} \leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T$. Esta operación se realiza usando las rutinas BLAS de nivel 3 **dsyrk** si el bloque es diagonal o utilizando **dgemm** si no lo es.

En esta versión paralela del algoritmo, además de los parámetros del sistema relacionados con el coste aritmético de las rutinas de LAPACK y BLAS3: $k_{2,potf2}$, $k_{3,trsm}$, $k_{3,gemm}$ y $k_{3,syrk}$; aparecen parámetros que identifican el coste de la comunicación entre los procesos (t_s, t_w). Los parámetros del algoritmo vienen identificados por el tamaño de bloque utilizado, b, y por el número de filas, r, y columnas, c, de la cuadrícula de procesos. En general, los valores de los parámetros del sistema dependerán de los del algoritmo y del tamaño n de la matriz, así como de la librería básica de álgebra lineal y de comunicaciones utilizadas.

La rutina utiliza para el envío de bloques de la matriz mensajes del tipo difusión. La comunicación de los bloques de la matriz entre procesos en una misma columna se realiza con tipos derivados [SOHL+98] y su coste es $t_s + b^2 t_{w_d}$ (siendo t_{w_d} el tiempo de envío para datos derivados). Para la comunicación del panel factorizado entre filas de procesos se utilizan tipos de datos simples con un coste de $b(t_s + bt_w)$.

El tiempo de ejecución de este algoritmo puede ser modelado con:

- **Computación:**

$$k_{2,dpotf2}\frac{nb^2}{3} + k_{3,dtrsm}\left[\frac{n}{r}(r-1) + \frac{n}{2}(\frac{n}{rb}-1)\right]b^2 + k_{3,dsyrk}\left\lceil\frac{1}{\sqrt{p}}\right\rceil(\frac{n^2-nb}{2})(b+1) + \frac{2}{p}k_{3,dgemm}\left(\frac{n^3}{6} - \frac{n^2b}{2} + \frac{nb^2}{3}\right) \quad (3.31)$$

- **Comunicación:**

$$\left(\frac{n}{b}-1\right)(t_s \ + \ b^2 t_{w_d}) \ + \ \frac{n}{2b}\left(\frac{n}{b}-1\right)(t_s + b^2 t_{w_d}) \ + \ \left(\frac{n}{b}-1\right)\left(bt_s + \frac{nb}{2}t_w\right) \quad (3.32)$$

Resultados experimentales

Para la validación experimental del modelo que aproxima el tiempo de ejecución de la rutina paralela por bloques de la factorización de Cholesky, se van a utilizar los sistemas P4net y HPC160. Con el sistema HPC160 se mostrarán los resultados obtenidos al utilizar memoria compartida para el paso de mensajes entre procesos en un mismo nodo del sistema, y los resultados cuando se utiliza la red MemoryChannel para el paso de mensajes entre procesos en distintos nodos. Se denominará sistema HPC160smp al primer caso y sistema HPC160mc al segundo. Se ha utilizado MPI como librería de paso de mensajes. En el caso de P4net, MPICH [mpi]; y una versión optimizada, tanto para MemoryChannel como para memoria compartida [cmp], de la misma librería, en el caso de HPC160smp y HPC160mc.

En la versión secuencial por bloques de la factorización de Cholesky ya se comprobó que los mejores tiempos de ejecución se obtenían con la versión optimizada de BLAS y LAPACK, por lo que sólo se obtendrán los valores experimentales de los parámetros del sistema, así como su dependencia de los del algoritmo, utilizando la versiones optimizadas de las librerías para Pentium 4 (BLASopt) y para el sistema HPC160 (CXML).

La estimación de cada uno de los SP aritméticos se ha realizado, como en los casos secuenciales, obteniendo valores medios con rutinas que utilizan patrones de acceso similares a los del algoritmo. Los parámetros del sistema que aparecen en el estudio teórico de la comunicación entre procesos se han estimado con rutinas que comunican los procesos en la malla 2D de igual forma que se realiza en el algoritmo.

	Tamaño bloque, b			
n	32	64	128	256
512	0.0045	0.0034	0.0063	0.0086
1024	0.0054	0.0046	0.0077	0.0103
2048	0.0067	0.0049	0.0076	0.0100

Tabla 3.35: Valores de $k_{2,dpotf2}$ (en μsegundos) para diferentes tamaños de matriz y tamaño de bloque, en Pentium 4 con la librería BLASopt.

En Pentium 4 con la librería BLASopt los valores de $k_{2,dpotf2}$ y $k_{3,dtrsm}$ dependen de los de n y de b (tabla 3.35 y tabla 3.36). El resto de parámetros aritméticos del sistema, $k_{3,dsyrk}$ y $k_{3,dgemm}$, pueden considerarse sólo función del tamaño de bloque b para tamaños del problema mayores de 1024. En la tabla 3.37 se muestran sus valores para los tamaños de bloque utilizados en la rutina.

	Tamaño de bloque, b			
n	32	64	128	256
512	0.0018	0.0016	0.0013	0.0012
1024	0.0020	0.0018	0.0014	0.0012
2048	0.0028	0.0021	0.0016	0.0012
4096	0.0037	0.0026	0.0019	0.0014
5120	0.0034	0.0024	0.0017	0.0014

Tabla 3.36: Valores de $k_{2,dtrsm}$ (en μsegundos) para diferentes tamaños de la matriz y tamaño de bloque, en Pentium 4 con la librería BLASopt.

Tamaño de bloque	32	64	128	256
$k_{3,dgemm}$	0.001862	0.000937	0.000572	0.000467
$k_{3,dsyrk}$	0.003492	0.001484	0.001228	0.000762

Tabla 3.37: Valores de $k_{3,dgemm}$ y $k_{3,dsyrk}$ (en μsegundos) para diferentes tamaños de bloque, en Pentium 4 con la librería BLASopt.

En el sistema HPC160, con la librería CXML, los valores de $k_{2,dpotf2}$ dependen de los de n y b (tabla 3.38) y, a diferencia de lo que ocurría en Pentium 4 con la librería BLASopt, en esta plataforma los valores de $k_{3,dtrsm}$ pueden considerarse sólo función de b. Para el resto de los parámetros del sistema, $k_{3,dsyrk}$ y $k_{3,dgemm}$, sus valores pueden considerarse únicamente función del tamaño de bloque b para tamaños del problema mayores de 1024. En la tabla 3.39 se muestran los valores obtenidos para los tamaños de bloque utilizados en la rutina.

	Tamaño de problema					
b	256	512	1024	2048	4096	7168
32	0.0000	0.0000	0.0028	0.0147	0.0101	0.0064
64	0.0042	0.0021	0.0024	0.0082	0.0034	0.0038
128	0.0017	0.0019	0.0033	0.0052	0.0025	0.0025
256	0.0016	0.0021	0.0027	0.0040	0.0023	0.0024

Tabla 3.38: Valores del SP $k_{2,dpotf2}$ (en μsegundos) para diferentes tamaños de la matriz y tamaño de bloque, utilizando la librería CXML en HPC160.

En lo que se refiere a los SP asociados con las comunicaciones para el envío de datos simples (t_s, t_w) y para el envío de datos derivados (t_{s_d}, t_{w_d}), en la plataforma P4net, debido al elevado coste de la comunicación respecto al de la computación, ha sido necesario tener en cuenta la dependencia de t_w y t_{w_d} respecto al tamaño del mensaje. En la tabla 3.40 se muestran los valores obtenidos experimentalmente para el envío entre procesos en una misma fila (t_w), y en la tabla 3.41 los valores correspondientes a envíos entre procesos en una misma columna (t_{w_d}). Los valores de t_s y t_{s_d} son aproximadamente iguales y se han considerado sólo dependientes del

Tamaño de bloque	32	64	128	256
$k_{3,dgemm}$	0.000824	0.000658	0.000610	0.000580
$k_{3,dsyrk}$	0.001628	0.001164	0.000807	0.000688
$k_{3,dtrsm}$	0.001617	0.001110	0.000841	0.000706

Tabla 3.39: Valores de los *SP* $k_{3,dgemm}$, $k_{3,dsyrk}$ y $k_{3,dtrsm}$ (en μsegundos) para diferentes tamaños de bloque en HPC160 con la librería CXML.

número de procesos utilizados en la comunicación, obteniéndose experimentalmente que $t_s = 55$ μsegundos para $p = 2$ y $t_s = 121$ μsegundos para $p = 4$.

	Número de procesos	
Tamaño de mensaje	2	4
1500	0.61	1.22
2048	0.77	1.45
$\geqslant 4000$	0.84	1.68

Tabla 3.40: Valores de t_w (en μsegundos) obtenidos experimentalmente en el envío a filas de procesos, para diferentes tamaños de mensaje y número de procesos. En P4net.

		Tamaño de bloque			
Sistema	Número de procesos	32	64	128	256
P4net	$p = 2$	0.97	0.84	1.00	1.10
	$p = 4$	1.60	1.90	1.60	1.64
HPC160smp	$p = 2$	0.019	0.024	0.020	0.019
	$p = 4$	0.047	0.048	0.045	0.041
HPC160mc	$p = 2$	0.095	0.091	0.089	0.90
	$p = 4$	0.190	0.176	0.179	0.183

Tabla 3.41: Valores de t_{w_d} (en μsegundos) obtenidos experimentalmente para el envío de bloques de tamaño b^2 a procesos en una misma columna, para diferentes tamaños de bloque y número de procesos. En P4net, HPC160smp y HPC160mc.

En HPC160smp y HPC160mc el coste de la comunicación respecto al de computación no es tan elevado, y para el envío entre procesos en una misma fila es suficiente considerar t_w función sólo del número de procesos. Experimentalmente se ha obtenido que $t_w = 0.011$ μsegundos para $p = 2$ y $t_w = 0.025$ μsegundos para $p = 4$ en HPC160smp, y $t_w = 0.072$ μsegundos para $p = 2$ y $t_w = 0.14$ μsegundos para $p = 4$ en HPC160mc. En el caso de envío de bloques de columnas ha sido necesario utilizar diferentes valores de t_{w_d} para distintos tamaños de bloque (tabla 3.41). La latencia se ha considerado, como en el caso de P4net, sólo función del número de procesos. Experimentalmente se ha obtenido que $t_s = 4.88$ μsegundos para $p = 2$ y $t_s = 9.77$ μsegundos para $p = 4$, tanto en HPC160smp como en HPC160mc. Como se puede comprobar, los valores

de los SP de comunicaciones dependen del tipo de rutina utilizada para la comunicación entre los procesos. Además su dependencia respecto a los AP varía de un sistema a otro y también con el tipo de mensaje. Como ya ocurría con los SP de computación, se vuelve a presentar con los SP de comunicación la necesidad de completar el modelo teórico con un estudio experimental como el que se ha planteado, con el fin de determinar adecuadamente en cada plataforma de cómputo sus valores y su dependencia respecto de los AP.

Utilizando los valores obtenidos para los parámetros del sistema, junto con los modelos de tiempo de ejecución planteados para la rutina de Cholesky paralela, en las figuras 3.12 (P4net) y 3.13 (HPC160smp) se compara el tiempo de ejecución obtenido experimentalmente (Experimental) con el obtenido usando el modelo (Teórico) para diferentes tamaños de problema, n, tamaño de bloque, b, y malla lógica 2D de procesos, $p = r \times c$. Como puede observarse en las gráficas correspondientes a los resultados experimentales, el tiempo de ejecución de la rutina varía con la malla 2D de procesos, y para una configuración dada de la malla 2D de procesos, con el tamaño de bloque para la distribución. De tal forma, que para un tamaño de problema n puede ser conveniente ejecutar la rutina con unos valores de $p = r \times c$ y b, mientras que para otro tamaño de problema los valores de $p = r \times c$ y b pueden ser distintos. Comparando las gráficas para cada plataforma, se puede observar que el número de procesos, su configuración lógica, y el tamaño de bloque con los cuales se obtienen los menores tiempos de ejecución, dependen del sistema en el que se ejecuta la rutina y del tamaño del problema. En este sentido, las gráficas obtenidas a partir del modelo teórico reflejan las variaciones indicadas, por lo que el modelo permitirá tomar las decisiones adecuadas sobre la mejor selección de los AP. En P4net, debido al alto coste de las comunicaciones, el tiempo de ejecución óptimo hasta tamaños de problema de 4096 se obtiene con un único proceso, mientras que para tamaños de problema mayores ($n = 5120$) se obtiene mejor tiempo de ejecución utilizando $p = 2 \times 1$ procesos, debido a que el tiempo de computación comienza a tener mayor peso que el tiempo de las comunicaciones. Sin embargo, en HPC160smp la mejor elección es siempre ejecutar el programa en paralelo con una distribución de los procesos $p = 4 \times 1$, con tiempos de ejecución 3.32 veces inferiores al de su ejecución secuencial. En HPC160mc se obtienen resultados similares a los de HPC160smp, pero con una distribución lógica de procesos óptima que depende del tamaño del problema.

En lo que se refiere a la desviación entre el tiempo obtenido en la ejecución de la rutina y el tiempo aproximado por el modelo ($\frac{|t_{mod} - t_{exp}|}{t_{exp}}$), en la tabla 3.42 para la plataforma P4net y en la tabla 3.43 para la plataforma HPC160smp, se muestra una comparación para cada tamaño de problema, tamaño de bloque y configuración de la malla lógica de procesos. Se omiten los valores para $n = 512$, $b = 256$ y configuraciones lógicas de procesos de $p = 1 \times 4$ y $p = 4 \times 1$ debido a que no son posibles.

Como se refleja en las tablas indicadas, el modelo propuesto para la computación y la comunicación permite obtener una buena aproximación al tiempo de ejecución real de la rutina. El porcentaje de desviación es en general pequeño y sólo en unos pocos casos y en la plataforma P4net, se puede observar desviaciones de hasta un 40 % cuando el tamaño del problema es

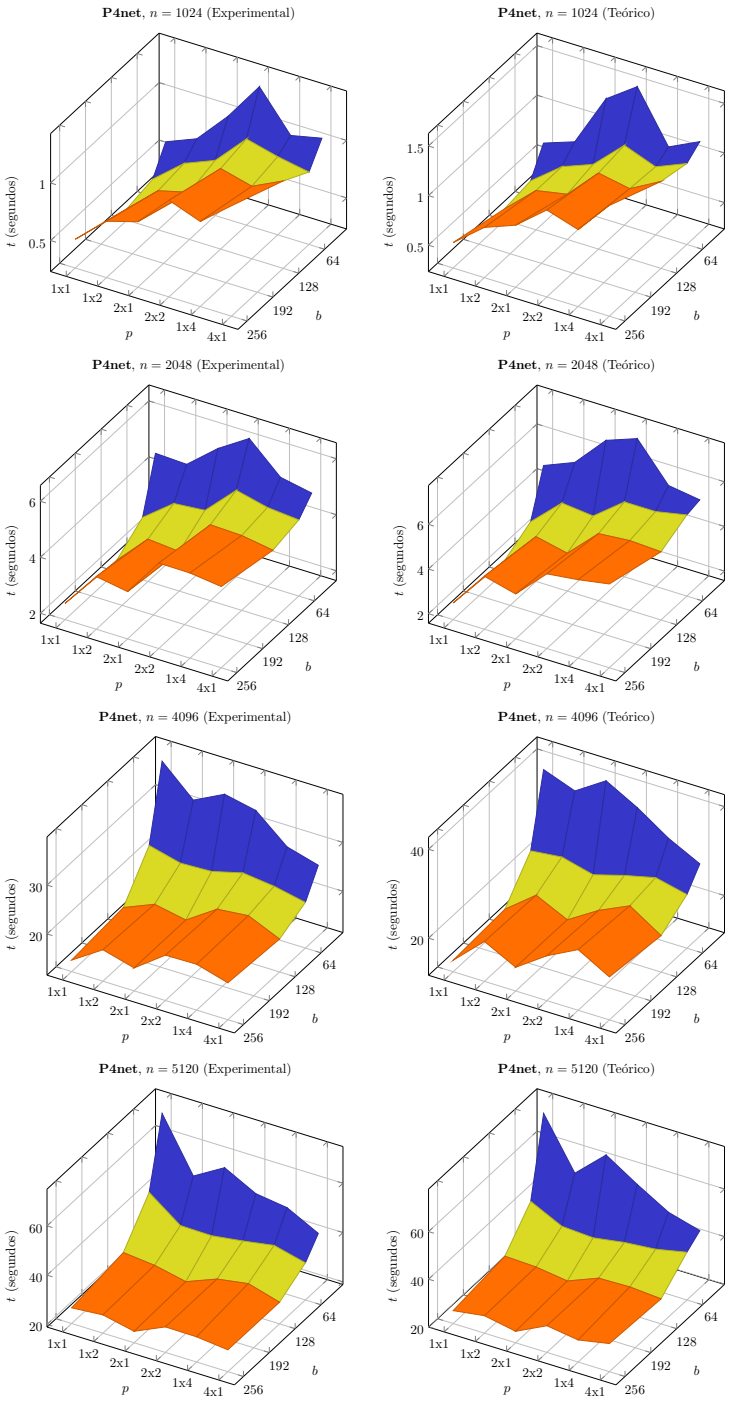


Figura 3.12: Comparación de los tiempos de ejecución experimental y los proporcionados por el modelo para distintos tamaños de problema, de bloque y de malla lógica de procesos, para la rutina de Cholesky en paralelo, en P4net.

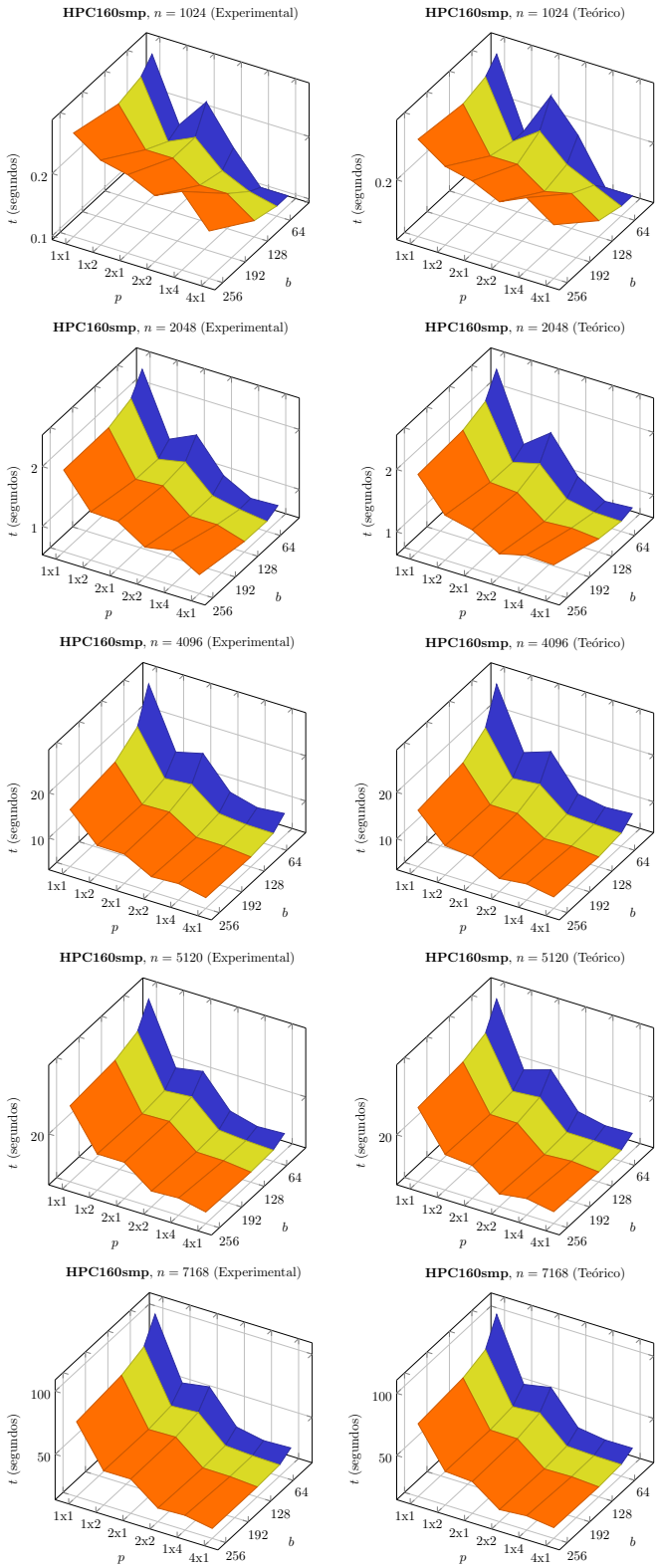


Figura 3.13: Comparación de los tiempos de ejecución experimental y los proporcionados por el modelo para distintos tamaños de problema, de bloque y de malla lógica de procesos, para la rutina de Cholesky en paralelo, en HPC160smp.

		Tamaño cuadrícula de procesos, $p = r \times c$					
n	b	1×1	1×2	2×1	2×2	1×4	4×1
512	32	0 %	25 %	29 %	21 %	7 %	6 %
	64	4 %	28 %	16 %	16 %	15 %	40 %
	128	0 %	19 %	11 %	12 %	12 %	13 %
	256	7 %	8 %	1 %	5 %		
1024	32	9 %	7 %	30 %	14 %	1 %	9 %
	64	4 %	6 %	6 %	6 %	2 %	21 %
	128	6 %	10 %	7 %	7 %	9 %	12 %
	256	2 %	2 %	8 %	4 %	4 %	13 %
2048	32	3 %	16 %	23 %	15 %	9 %	9 %
	64	0 %	12 %	5 %	5 %	10 %	19 %
	128	1 %	13 %	7 %	4 %	9 %	12 %
	256	4 %	12 %	8 %	5 %	9 %	17 %
4096	32	2 %	13 %	16 %	9 %	13 %	7 %
	64	0 %	11 %	2 %	3 %	14 %	13 %
	128	0 %	16 %	4 %	4 %	15 %	8 %
	256	2 %	15 %	4 %	4 %	21 %	11 %
5120	32	4 %	7 %	14 %	12 %	7 %	7 %
	64	5 %	3 %	2 %	3 %	4 %	9 %
	128	1 %	2 %	6 %	6 %	5 %	10 %
	256	0 %	3 %	5 %	6 %	15 %	14 %

Tabla 3.42: Desviación entre los tiempos de ejecución experimentales y los proporcionados por el modelo, para distintos tamaños de problema, tamaños de bloque y malla de procesos, para la rutina de Cholesky en paralelo, en P4net.

pequeño y la rutina utiliza un mayor número de procesos paralelos (caso con $n = 512$, $b = 64$ y $p = 4 \times 1$). De todas maneras, como veremos a continuación, el modelo sigue siendo útil a la hora de seleccionar los parámetros del algoritmo que proporcionan tiempos de ejecución óptimos.

Finalmente, el valor óptimo para los parámetros algorítmicos obtenidos experimentalmente (opt.) y los proporcionados por el modelo (mod.) para diferente número de procesadores, p, se muestran en la tabla 3.44 para P4net, y en la tabla 3.45 para HPC160smp y HPC160mc. También se muestra la desviación (dev.) entre el tiempo de ejecución experimental utilizando los parámetros proporcionados por el modelo y el tiempo de ejecución experimental inferior obtenido variando los valores de los parámetros del algoritmo. Un valor de 0 en la desviación indica que los valores de los parámetros seleccionados por el modelo coinciden con aquellos con los que se obtienen menores tiempos de ejecución. El tamaño de bloque seleccionado coincide con el óptimo en 27 de los 33 casos estudiados, y la topología lógica se selecciona correctamente en 22 de los 28 experimentos realizados. En los 10 casos en los que la selección de los parámetros no proporciona tiempos de ejecución óptimos, la desviación en los tiempos de ejecución experimentales es pequeña. La media de las desviaciones es de 5.1 % para los 10 casos en los que la selección realizada no es la óptima.

Como se ha mostrado en esta sección, el valor de los parámetros del algoritmo varía entre sistemas de diferentes características y con el tamaño del problema, pero con el modelo y con

		Tamaño cuadrícula de procesos, $p = r \times c$					
n	b	1×1	1×2	2×1	2×2	1×4	4×1
512	32	4 %	0 %	18 %	14 %	4 %	5 %
	64	9 %	14 %	13 %	20 %	15 %	13 %
	128	4 %	6 %	15 %	6 %	3 %	10 %
	256	1 %	2 %	2 %	4 %		
1024	32	1 %	2 %	10 %	14 %	8 %	7 %
	64	2 %	5 %	7 %	11 %	10 %	9 %
	128	1 %	3 %	0 %	3 %	4 %	6 %
	256	1 %	1 %	2 %	0 %	2 %	10 %
2048	32	1 %	1 %	7 %	6 %	6 %	9 %
	64	0 %	2 %	3 %	6 %	7 %	15 %
	128	0 %	2 %	2 %	1 %	5 %	19 %
	256	1 %	1 %	4 %	2 %	2 %	29 %
4096	32	2 %	0 %	3 %	2 %	5 %	2 %
	64	2 %	1 %	2 %	2 %	6 %	5 %
	128	1 %	1 %	1 %	1 %	5 %	6 %
	256	1 %	1 %	3 %	1 %	3 %	9 %
5120	32	1 %	2 %	3 %	1 %	5 %	4 %
	64	1 %	1 %	1 %	4 %	3 %	3 %
	128	1 %	1 %	2 %	4 %	2 %	4 %
	256	1 %	1 %	4 %	5 %	1 %	4 %
7168	32	1 %	0 %	1 %	0 %	4 %	1 %
	64	1 %	0 %	1 %	1 %	2 %	4 %
	128	1 %	1 %	1 %	1 %	2 %	2 %
	256	1 %	1 %	3 %	3 %	1 %	5 %

Tabla 3.43: Desviación entre los tiempos de ejecución experimentales y los proporcionados por el modelo, para distintos tamaños de problema, tamaños de bloque y malla de procesos, para la rutina de Cholesky en paralelo, en HPC160smp.

	$p = 1$			$p = 2$					$p = 4$				
	opt.	mod.	dev.	opt.		mod.		dev.	opt.		mod.		dev.
n	b	b	%	b	$r \times c$	b	$r \times c$	%	b	$r \times c$	b	$r \times c$	%
512	64	64	0	64	1×2	128	1×2	2.0	128	1×4	128	1×4	0
1024	128	128	0	128	1×2	64	1×2	3.4	64	4×1	64	1×4	1.2
2048	128	128	0	128	2×1	128	2×1	0	128	1×4	128	2×2	9.6
4096	256	256	0	256	2×1	128	2×1	0.1	128	4×1	128	4×1	0
5120	256	256	0	256	2×1	256	2×1	0	128	4×1	128	4×1	0
Media			0					1.1					2.2

Tabla 3.44: Selección de los parámetros del algoritmo para la rutina de Cholesky paralela por bloques en P4net.

	HPC160smp $p=4$					HPC160mc $p=4$					HPC160smp-mc $p=8$				
	opt.		mod.		dev.	opt.		mod.		dev.	opt.		mod.		dev.
n	b	$r \times c$	b	$r \times c$	%	b	$r \times c$	b	$r \times c$	%	b	$r \times c$	b	$r \times c$	%
512	32	4×1	32	4×1	0	32	4×1	32	2×2	81	32	2×4	32	2×4	0
1024	64	4×1	64	4×1	0	64	4×1	32	2×2	62	32	2×4	32	2×4	0
2048	64	4×1	64	4×1	0	64	4×1	32	2×2	1.3	64	2×4	32	2×4	17.2
4096	128	4×1	128	4×1	0	128	2×2	128	4×1	1.6	128	2×4	128	2×4	0
5120	128	4×1	128	4×1	0	128	2×2	128	2×2	0	64	2×4	64	2×4	0
7168	128	4×1	128	4×1	0	128	2×2	128	2×2	0	64	2×4	64	2×4	0
Media					0					24.3					2.9

Tabla 3.45: Selección de los parámetros del algoritmo para la rutina de Cholesky paralela por bloques en HPC160 utilizando memoria compartida (HPC160smp), MemoryChannel (HPC160mc) y ambas (HPC160smp-mc).

la utilización de diferentes costes para diferentes mecanismos de comunicación MPI se consigue una selección satisfactoria de los parámetros en todos los casos. Por lo tanto, la metodología propuesta pueda ser utilizada para obtener tiempos de ejecución cercanos al óptimo sin necesidad de la intervención del usuario de la rutina.

3.5 Resumen y conclusiones

En este capítulo se ha descrito nuestra propuesta de mejora de modelos de tiempo de ejecución de rutinas paralelas de álgebra lineal y se ha comprobado su efectividad en un conjunto significativo de rutinas. Para cada una de estas rutinas se ha realizado un estudio teórico de la complejidad del algoritmo que sigue y se ha obtenido un modelo del tiempo de ejecución. El modelo teórico planteado se ha completado con un estudio experimental en varias plataformas computacionales, y se ha comparado el tiempo proporcionado por el modelo con el tiempo de ejecución real, mostrando así el grado de aproximación obtenido. Adicionalmente se ha visto la utilidad del modelo para la selección de aquellos parámetros ajustables del algoritmo que determinan el tiempo de ejecución de la rutina que lo implementa, y se ha comprobado que con su ayuda se puede realizar una selección satisfactoria de estos parámetros ajustables. La selección del mejor valor de los parámetros no es una tarea sencilla, ya que su valor varía de una rutina a otra, y para una misma rutina depende de la plataforma en la que se ejecute, de la librería utilizada y del tamaño del problema a resolver.

A modo de conclusión se puede resumir que en nuestra propuesta de mejoras en el modelado del tiempo de ejecución de rutinas de álgebra lineal se han tenido en cuenta las siguientes consideraciones:

- Se ha diferenciado el coste asociado con rutinas básicas de diferentes niveles y entre rutinas de un mismo nivel que realizan operaciones distintas. Además se considera la influencia

que el algoritmo tiene sobre el coste de una rutina básica, de tal forma que consideramos posibles variaciones en el valor del coste para una misma rutina.

- Dada una plataforma paralela y diferentes librerías básicas, se ha tenido en cuenta la influencia de la librería en el tiempo de ejecución de la rutina de álgebra lineal y se han asociado costes distintos de la rutinas básicas en cada librería.
- El coste de las operaciones básicas puede variar con el tamaño del problema a resolver. En nuestro trabajo se propone descomponer el modelo de tiempo de ejecución con el fin de reflejar la variación en el coste de las operaciones básicas con el tamaño, forma y localidad de los datos en los que la rutina se aplica.
- De forma similar a lo realizado con el coste asociado a las rutinas básicas aritméticas, se consideran valores diferentes para los parámetros asociados con cada una de las rutinas básicas de comunicación.
- Con el fin de mejorar la aproximación al tiempo de ejecución, se tiene en cuenta la variación del coste de las comunicaciones con el tamaño del mensaje a enviar, y con los parámetros ajustables de la rutina: número de procesos, forma de la topología lógica de la malla de procesos, tamaño del bloque, etc.
- Se consideran las variaciones del coste de una misma rutina de comunicación en función de las diferentes implementaciones disponibles para una misma plataforma y también de las redes de interconexión empleadas.

En el siguiente capítulo se completará la propuesta realizada de mejoras en el modelado analítico de rutinas de álgebra lineal como base para la obtención de modelos de tiempo de ejecución, y se plantearán mecanismos que permitirán obtener aproximaciones al tiempo de ejecución en aquellos casos en los que el modelo analítico presentado en este capítulo no resulte de aplicación.

Capítulo 4

Utilización de técnicas de remodelado

4.1 Introducción

En este capítulo extendemos nuestra propuesta de mejoras en el modelado de rutinas de álgebra lineal con la aplicación de técnicas que permitirán obtener una estimación del tiempo de ejecución de una rutina de álgebra lineal en aquellos casos en los que el procedimiento general mostrado en el capítulo anterior no pueda ser aplicado, o bien cuando los resultados obtenidos no sean satisfactorios y, por tanto, éstos no sean de ayuda en la elección de los parámetros del algoritmo que conducen a la obtención de los mejores tiempos de ejecución.

En el capítulo anterior se partía de un estudio teórico del algoritmo que implementa la rutina, y a partir de este estudio se construía un modelo teórico-experimental (modelo analítico) con el que se aproximaba el tiempo de ejecución de la rutina que modela. El modelo teórico-experimental se expresaba como una función del tamaño del problema, n, de los parámetros del algoritmo, AP, y de los parámetros del sistema, SP. Los SP representaban el coste de realizar una operación con las rutinas de los niveles inferiores, y su valor era obtenido por medio de la experimentación. En este capítulo se propone que, si para una rutina o plataforma en particular, el modelo teórico junto con la obtención de los parámetros del sistema no permite una buena selección del valor de los parámetros del algoritmo, o bien desconocemos el modelo teórico de la rutina a modelar, se realice un remodelado, y que se apliquen modelos de tiempo de ejecución parametrizados basados en funciones polinomiales obteniendo el valor de sus coeficientes utilizando, por ejemplo, técnicas de regresión lineal [Bre95].

En primer lugar se describirá nuestra propuesta de remodelado y se realizará una breve introducción al concepto de regresión lineal, que nos servirá de base para mostrar a continuación

cuatro mecanismos para la obtención de los coeficientes del polinomio que modela la rutina. Posteriormente se realizará una validación experimental de la propuesta de remodelado con diversas rutinas de álgebra lineal sobre plataformas de diferentes características. En la validación experimental, se verán casos en los que se ha construido un nuevo modelo (remodelado) a partir de combinaciones del tamaño del problema, n, y de los AP de la rutina, cuando la información obtenida previamente por el modelo teórico no permita una correcta elección de los AP.

4.2 Propuesta de remodelado

Cuando la información proporcionada por el modelo analítico de tiempo de ejecución no permite realizar una correcta selección de los parámetros ajustables de la rutina paralela, proponemos la utilización de un nuevo modelo de tiempo de ejecución que puede ser construido bien a partir del modelo teórico, sustituyendo los SP que aparecen por polinomios que sean función de los parámetros del algoritmo, o bien a partir de combinaciones del tamaño del problema y parámetros del algoritmo. Por ejemplo, para una multiplicación de matrices de tamaño $n \times n$, con un coste de $O\left(n^3\right)$, es lógico suponer que un polinomio que aproxime su tiempo de ejecución tendrá la siguiente forma:

$$T(n) = \beta_3 n^3 + \beta_2 n^2 + \beta_1 n + \beta_0 \tag{4.1}$$

Si esta multiplicación se realizara por bloques de tamaño b, se formaría un polinomio en el que cada término sería una combinación de las potencias posibles del tamaño del problema $n : n^3, n^2, n$, y el tamaño de bloque $b : b^2, b, \frac{1}{b}$,... y el polinomio que aproxima el tiempo de ejecución de la rutina podría ser en este caso:

$$\begin{aligned} T(n) = \beta_{3,1} n^3 b + \beta_{3,0} n^3 + \beta_{3,-1} \frac{n^3}{b} + \beta_{2,1} n^2 b + \beta_{2,0} n^2 + \\ \beta_{2,-1} \frac{n^2}{b} + \beta_{1,1} n b + \beta_{1,0} n + \beta_{1-1} \frac{n}{b} + \beta_{0,0} \end{aligned} \tag{4.2}$$

A partir de esta formulación general para el modelo que aproxima el tiempo de ejecución, será necesario determinar, en cada plataforma computacional y para cada librería básica de álgebra lineal y de comunicaciones disponibles, los valores de los coeficientes que aparecen en el polinomio que aproxima el tiempo de ejecución de la rutina.

El procedimiento seguido para obtener el valor de los coeficientes resulta de gran importancia. Por una parte es aconsejable que el tiempo requerido en la obtención de los coeficientes, y por tanto del modelo, sea razonable, con el fin de no sobrecargar la plataforma de cómputo innecesariamente, y, por otra parte, dicho tiempo determinará la calidad de la aproximación al

tiempo de ejecución real de la rutina que proporcione el modelo obtenido. Para la determinación de estos coeficientes se propone la aplicación de cuatro métodos que hemos denominado: *FIxed Minimal Executions* (FI-ME), *VAriable Minimal Executions* (VA-ME), *FIxed Least Square* (FI-LS) y *VAriable Least Square* (VA-LS). La utilización de estos cuatro métodos, junto con un criterio de aplicación condicionado por el coste con el que se obtienen los coeficientes en cada uno, y guiado por la comparación con los resultados del tiempo de ejecución real de la rutina, configura nuestra propuesta para la obtención de técnicas de remodelado.

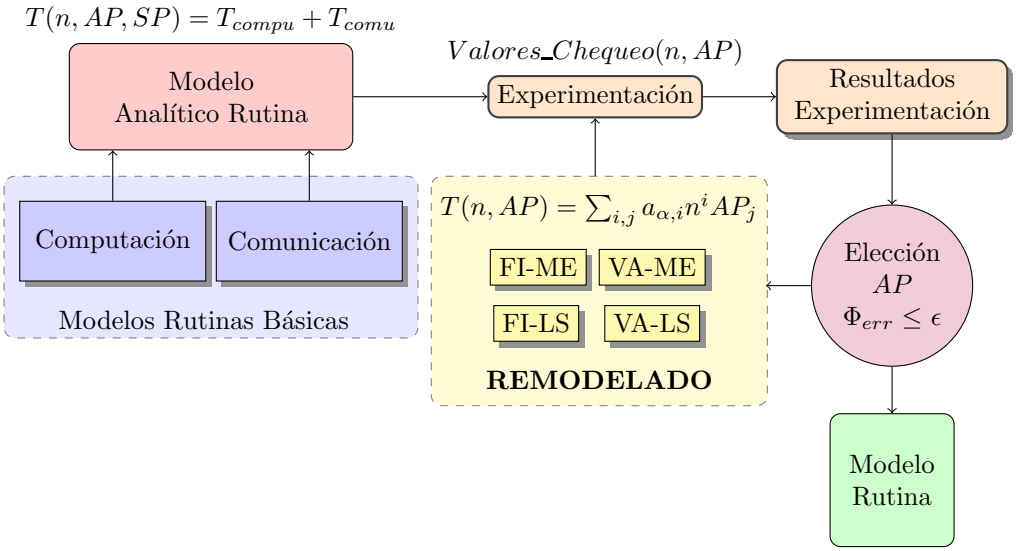


Figura 4.1: Esquema general para la obtención de un modelo de tiempo de ejecución de rutinas de álgebra lineal.

En la figura 4.1 se muestra el esquema general propuesto para la obtención de un modelo de tiempo de ejecución de rutinas de álgebra lineal. Se partirá del modelo analítico de la rutina, el cual obtendrá información de los modelos correspondientes a las operaciones básicas de computación y comunicación invocadas en su código. Los modelos de las rutinas básicas pueden ser modelos analíticos o modelos obtenidos por medio de alguno de los métodos indicados de obtención de los coeficientes. El modelo de tiempo de ejecución se comprueba a través de la experimentación con unos `Valores_Chequeo` de n y AP. Los resultados obtenidos durante la experimentación, se verifican con el fin de comprobar que la información que genera el modelo permite realizar una correcta selección de los AP de la rutina, y que la diferencia (Φ_{err}) entre los tiempos aproximados por el modelo y los tiempos reales de ejecución de la rutina para los `Valores_Chequeo` de n y AP, no supera un cierto valor de error (ϵ). En ese caso se da por válido el modelo; de lo contrario, se obtendrá un nuevo modelo de la rutina (remodelado) y se volverá a realizar su comprobación experimental.

4.2.1 Regresión lineal

Un modelo de regresión es una representación matemática de la relación entre la variable respuesta y unos valores de entrada o variables explicativas [Mon04]. El modelo más simple para una regresión lineal en la que sólo hay una variable explicativa es:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i \tag{4.3}$$

si suponemos que:

- x_i es el valor *i-ésimo* de la variable explicativa.
- $\bar{x}$ es la media de la variable explicativa.
- y_i es la respuesta medida correspondiente a x_i.
- $\bar{y}$ es la media de los valores medidos.
- β_0 y β_1 son los coeficientes, o parámetros, de la relación lineal siendo β_0 el valor de la ordenada en el origen y β_1 el de la pendiente.
- Las ϵ_i son variables aleatorias con distribución normal, media cero y varianza σ^2 o desviación estándar σ. Estas variables son independientes y representan errores en la estimación y en las medidas.

entonces, se puede considerar que y_i sigue una distribución normal con un valor esperado de $\beta_0 + \beta_1 x_i$ y con una varianza de σ^2.

Dado que en esta fase de remodelado de la rutina se propone construir un nuevo modelo utilizando la regresión lineal, se requiere de una técnica con la que se obtenga la mejor estimación para β_0 y β_1. El procedimiento tradicional consiste en utilizar la *estimación por mínimos cuadrados*, esto es, obtener el β_0 y el β_1 que minimicen la suma de los cuadrados de las desviaciones:

$$\sum_{i=1}^{n} \epsilon_i^2 \equiv \sum_{i=1}^{n} (y_i - \hat{y_i})^2 \equiv \sum_{i=1}^{n} [y_i - (\beta_0 + \beta_1 x_i)]^2 \tag{4.4}$$

donde $\hat{y_i}$ representa el valor predicho correspondiente a x_i. Para encontrar el β_0 y el β_1 que minimizan la ecuación 4.4, se obtienen las derivadas parciales y se igualan a cero. El conjunto de ecuaciones resultante se conoce como *ecuaciones normales*, y para el caso de una única variable la solución es:

$$\beta_1 = \frac{\sum_{i=1}^{n} (x_i - \bar{x}) y_i}{\sum_{i=1}^{n} (x_i - \bar{x})^2} \tag{4.5}$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x} \tag{4.6}$$

4.2.2 Métodos para la estimación de los coeficientes

A partir del método tradicional de obtención de los coeficientes de un modelo de regresión lineal visto anteriormente, en este trabajo se propone la utilización de cuatro mecanismos para la obtención de los coeficientes del polinomio que aproximará el tiempo de ejecución de una rutina de álgebra lineal y que serán de aplicación cuando el modelo general visto en el capítulo anterior no proporcione resultados satisfactorios. A continuación se describen los cuatro mecanismos propuestos:

- **FIxed Minimal Executions (FI-ME)**: se utiliza un único polinomio para aproximar el tiempo de ejecución de la rutina y se pretende que los coeficientes se puedan obtener con el mínimo número de experimentos con el fin de reducir el tiempo necesario en la obtención del modelo. Si hay d coeficientes a calcular se plantea un sistema de d ecuaciones con d incógnitas, donde las incógnitas son los coeficientes del polinomio, utilizando d combinaciones distintas del tamaño del problema y de los parámetros del algoritmo (n, AP). Para cada una de estas d combinaciones tenemos una ecuación en la que el término de la izquierda será el polinomio con el que se quiere modelar la rutina, en el que se sustituyen los valores de n y AP seleccionados, y el término independiente será el tiempo de ejecución experimental de la rutina para dichos valores. Finalmente se resuelve el sistema de ecuaciones y se obtienen los d coeficientes del polinomio.

- **VAriable Minimal Executions (VA-ME)**: se seleccionan i regiones de posibles combinaciones de (n, AP), y en cada región se utiliza un polinomio para aproximar el tiempo de ejecución de la rutina. Para cada una de estas regiones se aplica el método FI-ME.

- **FIxed Least Square (FI-LS)**: el tiempo de ejecución de la rutina es aproximado por un único polinomio. Los coeficientes de este polinomio se obtienen aplicando técnicas de regresión lineal (método de los mínimos cuadrados) con el fin de obtener un modelo que minimice la distancia entre el tiempo experimental y el teórico para un conjunto de valores de (n, AP) seleccionados.

- **VAriable Least Square (VA-LS)**: al igual que en el método VA-ME, se utilizan i polinomios para aproximar el tiempo de ejecución de la rutina, y los polinomios se obtienen usando el método FI-LS en cada una de las i regiones seleccionadas de posibles combinaciones de (n, AP).

4.2.3 Procedimiento para la aplicación de los métodos de obtención de los coeficientes

En este apartado se describe el procedimiento utilizado para la aplicación de los cuatro métodos propuestos para la obtención de los coeficientes del modelo de aproximación de tiempo de ejecución. Como se ha comentado al inicio del capítulo, el grado del polinomio que se utiliza para aproximar el tiempo de ejecución de la rutina puede tomarse inicialmente al mismo valor que el del coste de la rutina. Se trataría de una primera elección y cabe la posibilidad de elegir polinomios de grado mayor si el modelo no aproximara adecuadamente los tiempos de ejecución.

Una vez construido el polinomio con las potencias posibles del tamaño del problema, n, y los parámetros del algoritmo, AP, habrá que realizar la selección de un conjunto significativo de valores para n y para los diferentes AP, además de establecer el número i de intervalos de n, de AP, y el número de ejecuciones a realizar. El conjunto seleccionado de n, $[n_i, n_e]$, de AP, $[AP_i, AP_e]$, del número de intervalos de i, así como del número de ejecuciones a realizar, servirán como valores iniciales de experimentación. Esta selección puede ser propuesta en un principio por el diseñador de la rutina, y modificada posteriormente por el administrador del sistema en el momento de la instalación, con el fin de ajustar el modelo a las características de la plataforma de cómputo donde se ejecute la rutina y a las condiciones de uso. A continuación se describe el procedimiento seguido por cada uno de los métodos en la obtención de los coeficientes del polinomio:

- En el método *FI-ME*, el intervalo de experimentación correspondiente al tamaño del problema ($[n_i, n_e]$) se divide en tantas partes iguales como coeficientes tenga el polinomio. De esta forma, si hay d coeficientes el intervalo quedará dividido en:

$$\left\{ n_i + \frac{(n_e - n_i)}{(d+1)}, n_i + 2\frac{(n_e - n_i)}{(d+1)}, \ldots, n_i + d\frac{(n_e - n_i)}{(d+1)} \right\}$$

  y la rutina a modelar se ejecuta para cada uno de estos tamaños de problema y se obtiene su tiempo de ejecución experimental.

- Para el método *VA-ME*, una vez seleccionado un intervalo inicial de experimentación, ($[n_i, n_e]$), se divide en i subintervalos y cada uno de ellos a su vez se vuelve a dividir en d partes iguales, por lo que será necesario realizar d ejecuciones de la rutina en cada uno de los i subintervalos.

- Con el método *FI-LS*, partiendo de un tamaño del problema n_i se toman valores de tiempo de ejecución de la rutina a incrementos del tamaño del problema n_{inc} hasta llegar al n_e. El número de ejecuciones de la rutina necesarias para obtener los coeficientes será:

$$\frac{(n_e - n_i + n_{inc})}{n_{inc}}$$

- En el método *VA-LS*, al igual que ocurría con el método *VA-ME*, el intervalo inicial se divide en i subintervalos, y se define el número de tamaños de problema distintos, n_{dat}, con los que se calcularán los coeficientes de los i polinomios. A cada subintervalo le corresponderá un rango de tamaños de problema igual a $[n_i + jw, n_i + (j+1)w]$, donde $w = \frac{(n_e - n_i)}{i}$ y $j = 0, 1, 2, \ldots, i-1$. En cada subintervalo la rutina se ejecutará desde el tamaño inicial correspondiente al subintervalo hasta el tamaño final, en incrementos de tamaño del problema $\frac{w}{n_{dat}}$. En total será necesario realizar n_{dat} ejecuciones de la rutina en cada uno de los i subintervalos para la obtención de los i polinomios.

Para la generación del modelo, si bien se podría utilizar individualmente cualquiera de los métodos descritos, proponemos que se apliquen siguiendo un orden basado en el criterio de menor coste de obtención de los coeficientes; es decir, la secuencia de aplicación de los métodos comenzará con aquel que menos tiempo necesite para obtener los coeficientes, por lo tanto, primero se aplicará el método FI-ME, luego el VA-ME, después el FI-LS y por último el VA-LS. Siguiendo el orden indicado, la obtención del modelo teórico de tiempo de ejecución finalizará con el primero de éstos métodos que obtenga una cuota de error (Φ_{err}) inferior a un valor prefijado para un conjunto de valores de tamaños de problema y parámetros del algoritmo distintos a los utilizados para la obtención de los coeficientes (`Valores_Chequeo`). Dicha cuota de error se define como una diferencia en tanto por ciento entre la suma de los valores del tiempo de ejecución obtenidos usando el modelo (t_{teo}) y la suma de los tiempos de ejecución de la rutina (t_{exp}) para cada tamaño de problema y parámetros del algoritmo definidos en `Valores_Chequeo`:

$$\Phi_{err} = \left[1 - \left(\frac{\sum_{Valores\_Chequeo} t_{teo}}{\sum_{Valores\_Chequeo} t_{exp}}\right)\right] 100 \tag{4.7}$$

Con esta ecuación se obtiene una medida de la calidad del modelo construido con cada uno de los métodos empleados. Hay que tener en cuenta que no todas las diferencias entre los tiempos de ejecución experimentales y los proporcionados con el modelo tienen la misma importancia o peso. Así, una diferencia del 20 % para un tamaño de problema pequeño puede ser aceptable, mientras que para un tamaño de problema grande puede no serlo. Esta ecuación permite aplicar un mayor peso a las aproximaciones realizadas por el modelo para tamaños de problema grandes.

4.3 Aplicación del método

En este apartado, se mostrará, a modo de ejemplo, el esquema propuesto de obtención de modelos con la rutina secuencial de multiplicación de Strassen de matrices y con una versión de la multiplicación paralela de matrices.

Como ya se indicó en el capítulo 2, el software de álgebra lineal se organiza en forma de jerarquía de librerías, de tal forma que las rutinas cuentan con llamadas a rutinas pertenecientes a librerías de los niveles inferiores. Por ejemplo, en la rutina de multiplicación de Strassen aparece la multiplicación y la adición de matrices, que pertenecen a los niveles 3 y 2, respectivamente, de la jerarquía mencionada, de tal forma que para su modelo analítico se podrían aprovechar los modelos analíticos de tiempo de ejecución ya obtenidos previamente para la multiplicación y adición de matrices. Con el fin de mostrar nuestra propuesta de construcción de modelos a partir de funciones polinomiales y el procedimiento de obtención de los coeficientes, se construirán nuevos modelos para las rutinas de los niveles inferiores, es decir, se realizará un remodelado inicial de estas rutinas, en lugar de reutilizar la información que nos proporcionarían los modelos analíticos mostrados en el capítulo 3.

4.3.1 Multiplicación de Strassen de matrices

La descripción detallada de esta rutina y su modelo analítico de tiempo de ejecución como una función del tamaño del problema, n, y de los parámetros del algoritmo, AP, se mostraron en la sección 3.3.6. Por tanto, se utilizará el modelo ya construido en dicha sección:

$$T = 7^l 2 \left(\frac{n}{2^l}\right)^3 k_{3,dgemm} + \frac{18}{4} n^2 \sum_{i=1}^{l} \left(\frac{7}{4}\right)^{i-1} k_{2,add} \tag{4.8}$$

con la única diferencia de que ahora en lugar de los SP de las rutinas básicas, aparecen sus tiempos de ejecución:

$$T = 7^l t_{mult}\left(\frac{n}{2^l}\right) + 18 \sum_{i=1}^{l} 7^{i-1} t_{add}\left(\frac{n}{2^i}\right) \tag{4.9}$$

donde $t_{mult}(\frac{n}{2^l})$ es el tiempo de ejecución de la rutina básica **dgemm** para una multiplicación de matrices de tamaño $\frac{n}{2^l}$, y $t_{add}(\frac{n}{2^i})$ es el tiempo de ejecución de la rutina básica **daxpy** para una adición de matrices de tamaño $\frac{n}{2^i}$.

Resultados experimentales

Con el fin de evaluar la validez de nuestra propuesta de mejoras en el modelado de rutinas de álgebra lineal densa se han obtenido resultados experimentales en dos plataformas computacionales diferentes: Sol y HPC160. Para las rutinas básicas **dgemm** y **daxpy**, se han utilizado versiones optimizadas de BLAS específicas para cada plataforma.

En la construcción de los modelos de tiempo de ejecución para las rutinas **dgemm** y **daxpy**, se han realizado experimentos con polinomios de diferentes grados, utilizando como polinomio

de partida aquel que tenga un grado igual al coste de la rutina. En las pruebas experimentales realizadas en los sistemas Sol y HPC160, se ha observado que para la multiplicación de matrices se obtiene una buena aproximación al tiempo de ejecución con un polinomio de grado tres (mismo valor que el coste teórico de la rutina), mientras que para la adición de matrices se ha necesitado utilizar un polinomio de grado seis, (el coste teórico es de $O(n^2)$). Por lo tanto, se ha utilizado un polinomio de grado tres para $t_{mult}(\frac{n}{2^l})$ y un polinomio de grado seis para $t_{add}(\frac{n}{2^i})$. Los coeficientes de los polinomios pueden ser calculados con cualquiera de los cuatro métodos descritos en la sección 4.2.2, pero en ambos casos los mejores resultados fueron obtenidos con el método FIxed Least Square (FI-LS).

En el método FI-LS, para el cálculo de los coeficientes que aparecen en el modelo, hay que seleccionar unos valores iniciales de experimentación para n_i, para n_e y para n_{inc}. Se ha tomado para la multiplicación de matrices $n_i = 500$, $n_e = 10000$ y $n_{inc} = 500$. Por lo tanto se obtendrán tiempos de ejecución experimentales para los siguientes tamaños de problema: {500, 1000, 1500, 2000, ..., 10000}. Para la adición de matrices se ha tomado $n_i = 64$, $n_e = 2048$ y $n_{inc} = 64$, con lo que los tiempos de ejecución experimentales se obtendrán para tamaños de problema {64, 128, 192, 256, ..., 1984, 2048}.

Las funciones polinomiales obtenidas con el método FI-LS y que modelan las rutinas básicas se muestran en la tabla 4.1 para Sol y en la 4.2 para HPC160. En la plataforma HPC160, el coeficiente del término de mayor grado del modelo de tiempo de ejecución para la adición de matrices es negativo, pero los valores que proporciona el modelo son siempre positivos en el rango de tamaños de problema analizados, por lo que resulta de aplicación a la hora de obtener una aproximación al tiempo de ejecución de la rutina que modela.

$t_{mult}(\frac{n}{2^l}) =$	$1.338 \times 10^{-01} - 2.261 \times 10^{-04}n + 1.039 \times 10^{-07}n^2 + 3.963 \times 10^{-10}n^3$
$t_{add}(\frac{n}{2^i}) =$	$1.507 \times 10^{-03} - 2.952 \times 10^{-05}n + 1.521 \times 10^{-07}n^2 - 1.970 \times 10^{-10}n^3 +$ $1.614 \times 10^{-13}n^4 - 6.367 \times 10^{-17}n^5 + 9.687 \times 10^{-21}n^6$

Tabla 4.1: Modelo de tiempo de ejecución $t_{mult}(\frac{n}{2^l})$ y $t_{add}(\frac{n}{2^i})$ utilizando el método FI-LS, en Sol.

$t_{mult}(\frac{n}{2^l}) =$	$-9.517 \times 10^{-02} + 2.128 \times 10^{-04}n - 9.079 \times 10^{-08}n^2 + 1.136 \times 10^{-09}n^3$
$t_{add}(\frac{n}{2^i}) =$	$-9.983 \times 10^{-04} + 1.683 \times 10^{-05}n - 6.700 \times 10^{-08}n^2 + 1.624 \times 10^{-10}n^3 -$ $1.284 \times 10^{-13}n^4 + 4.698 \times 10^{-17}n^5 - 6.509 \times 10^{-21}n^6$

Tabla 4.2: Modelo de tiempo de ejecución $t_{mult}(\frac{n}{2^l})$ y $t_{add}(\frac{n}{2^i})$ utilizando el método FI-LS, en HPC160.

Utilizando el modelo teórico visto en la ecuación 4.9 conjuntamente con los modelos obtenidos para la multiplicación y adición de matrices mostrados en las tablas 4.1 y 4.2, se comprueba si se puede realizar una selección del parámetro del algoritmo (nivel de recursión, l) que proporcione tiempos de ejecución óptimos. Para la comprobación se utilizan tamaños de problema

distintos a los utilizados para la obtención de los modelos de tiempo de ejecución. En el caso del algoritmo de Strassen se han utilizado los siguientes valores para la comprobación del modelo (`Valores_Chequeo`):

- Parámetro del algoritmo: Nivel de recursión $l = \{1, 2, 3\}$
- Tamaño de matriz $n = \{3072, 4096, 5120, 6144\}$

n	l	Modelo	Experimental	Desviación (%)
3072	1	**11.75**	**12.86**	8.58
3072	2	13.90	13.63	1.99
3072	3	37.04	15.76	135.06
4096	1	**27.21**	**29.71**	8.41
4096	2	28.59	30.10	5.02
4096	3	48.76	33.34	46.26
5120	1	**53.14**	56.83	6.51
5120	2	53.53	**56.43**	5.13
5120	3	71.08	60.19	18.09
6144	1	96.48	96.32	0.17
6144	2	**95.39**	**93.69**	1.82
6144	3	110.40	98.39	12.21

Tabla 4.3: Comparación entre el tiempo de ejecución (en segundos) experimental con el tiempo de ejecución proporcionado por el modelo para el algoritmo de Strassen, en Sol.

En las tablas 4.3 y 4.4 se muestran el tiempo de ejecución proporcionado por el modelo (Modelo), el tiempo de ejecución experimental (Experimental) y la desviación entre ambos valores ($\frac{|t_{mod} - t_{exp}|}{t_{exp}}$) para los valores indicados de comprobación del modelo. Se han resaltado los tiempos experimentales y teóricos óptimos. Como puede observarse en las tablas, en los dos sistemas los modelos obtenidos realizan una predicción correcta del nivel de recursión l con el que se obtiene el tiempo de ejecución óptimo. Únicamente en el sistema Sol, y para matrices de tamaño 5120, el nivel de recursión óptimo y el proporcionado por el modelo son diferentes. Aunque en este caso el tiempo de ejecución de la rutina, al utilizar el valor de nivel de recursión proporcionado por el modelo, es sólo un 0.71 % superior al óptimo.

Se puede apreciar en las tablas que conforme se aumenta el nivel de recursión, l, la estimación proporcionada por el modelo empeora. En el sistema Sol esta situación se hace más palpable, y la desviación entre el tiempo proporcionado por el modelo y el tiempo de ejecución de la rutina varía desde 0.17 % hasta 135.06 %, con un $\Phi_{err} = 8.38$, mientras que en el sistema HPC160 la desviación adopta valores inferiores, variando entre 0.11 % y 36.46 %, con un $\Phi_{err} = 0.87$. El motivo de que esto ocurra es que conforme aumenta el nivel de recursión, aumenta el número de adiciones que es necesario realizar en el algoritmo de Strassen, y estas adiciones se realizan con tamaños de matriz cada vez más pequeños, por lo que en estos casos el modelo obtenido para la adición de matrices no aproxima bien el tiempo de ejecución. La estimación se puede

n	l	Modelo	Experimental	Desviación (%)
3072	1	29.96	29.70	0.89
3072	2	28.54	27.82	2.57
3072	3	**17.55**	**27.61**	36.46
4096	1	69.85	70.85	1.43
4096	2	66.04	64.55	2.30
4096	3	**57.82**	**62.56**	7.58
5120	1	135.03	134.67	0.26
5120	2	125.76	123.38	1.92
5120	3	**118.122**	**118.45**	0.28
6144	1	229.786	232.268	1.07
6144	2	211.104	210.876	0.11
6144	3	**201.150**	**199.326**	0.92

Tabla 4.4: Comparación entre el tiempo de ejecución (en segundos) experimental con el tiempo de ejecución proporcionado por el modelo para el algoritmo de Strassen, en HPC160.

mejorar con un modelo más detallado para la adición de matrices, en el que se contemplara un modelo diferente para distintos rangos de tamaño del problema, es decir, utilizar el método VA-LS en lugar del FI-LS. En el siguiente apartado, con el fin de mostrar nuestra propuesta de remodelado, en lugar de construir un nuevo modelo para la rutina básica de adición de matrices, se abordará la construcción de un nuevo modelo de la rutina de mayor nivel, es decir, se realizará un remodelado de la rutina de Strassen.

4.3.2 Remodelado de la rutina de Strassen

En este apartado, se mostrará, a modo de ejemplo, la aplicación de nuestra propuesta de remodelado a la rutina de multiplicación de matrices de Strassen. Como se ha visto en el apartado anterior, al utilizar la información proporcionada por los modelos de las rutinas de niveles inferiores en el modelo de la rutina de Strassen, la selección del nivel de recursión óptimo no era correcta en todos los casos analizados. El esquema que se propone consiste en definir, a partir del modelo teórico original, un conjunto de funciones polinomiales de grado tres que sirvan de base para el nuevo modelo teórico del algoritmo de Strassen. Este conjunto de funciones polinomiales es:

$$T(n,l) = 7^l 2\left(\frac{n}{2^l}\right)^3 M(l) + \frac{9}{2}n^2 A(l)\sum_{i=1}^{l}\left(\frac{7}{4}\right)^{i-1} \tag{4.10}$$

en el que aparecen dos coeficientes, $M(l)$ y $A(l)$, función del nivel de recursión y asociados con el coste de la multiplicación y adición de matrices respectivamente. Con el fin de obtener un valor para estos nuevos coeficientes se ha diseñado un conjunto de experimentos con la rutina

de Strassen en los que el parámetro l (nivel de recursión) se mantiene fijo y se varía el tamaño del problema n. Para cada l obtenemos una mejor aproximación de $M(l)$ y $A(l)$ utilizando el método de los mínimos cuadrados.

Para mostrar la funcionalidad de la propuesta se va a realizar el estudio experimental en el sistema Sol, al tratarse del sistema en el que la aproximación realizada por el modelo general no era la adecuada.

La tabla 4.5 muestra los valores obtenidos en el sistema Sol para $M(l)$ y $A(l)$, aplicando el proceso indicado. Se han escogido como valores iniciales de experimentación para l $\{1, 2, 3, 4\}$, y para n $\{512, 1024, 1536, 2048, 2560, 3072, 3584, 4096, 4608\}$.

l	$M(l)$	$A(l)$
1	2.222×10^{-10}	3.890×10^{-08}
2	2.244×10^{-10}	3.033×10^{-08}
3	1.986×10^{-10}	3.025×10^{-08}
4	3.477×10^{-10}	1.528×10^{-08}

Tabla 4.5: Valores de $M(l)$ y $A(l)$ para distintos niveles de recursión (l), en Sol.

Llegados a este punto, podríamos tener un modelo teórico del tiempo de ejecución para cada nivel de recursión, o bien intentar encontrar alguna dependencia en los valores obtenidos para $M(l)$ y $A(l)$ que permita tener una única fórmula para cualquier valor de l. Realizando un ajuste por mínimos cuadrados del conjunto de valores de $A(l)$, se obtiene una aproximación con un polinomio de primer grado $(a + bl)$, y aplicando el mismo ajuste a los valores de $M(l)$ el polinomio que ajusta sus valores es de segundo grado $(c + dl + el^2)$. Finalmente se obtiene que los coeficientes a, b, c, d y e que mejor aproximan $M(l)$ y $A(l)$ son:

- $M(l) = 1.907 \cdot 10^{-10} + 4.580 \cdot 10^{-11} \cdot l - 1.445 \cdot 10^{-11} \cdot l^2$.
- $A(l) = 4.378 \cdot 10^{-08} - 5.131 \cdot 10^{-09} \cdot l$.

Conseguimos, por tanto, disponer de un único modelo teórico para cualquier combinación de (n, AP) de la rutina de Strassen.

Con el fin de verificar la validez de este nuevo modelo, es necesario comprobar los resultados obtenidos experimentalmente con diferentes combinaciones de tamaño del problema[1] y parámetros del algoritmo. Se han seleccionados los siguientes valores representativos de ejecuciones del algoritmo para (`Valores_Chequeo`):

- Parámetros del Algoritmo: Nivel de Recursión $l = \{1, 2, 3\}$
- Tamaño de matriz $n = \{1664, 2176, 2688, 3200, 5120, 5632\}$

[1] El tamaño de problema utilizado para la comprobación del modelo es distinto del utilizado para la obtención de los coeficientes.

En la tabla 4.6 y en la figura 4.2 se muestran los tiempos de ejecución proporcionados por el modelo (Modelo) y los obtenidos experimentalmente (Experimental) para diferentes tamaños de problema y niveles de recursión, junto con la desviación entre ambos valores. Se han resaltado también los tiempos experimentales y teóricos óptimos. Como puede observarse, el modelo predice correctamente el nivel de recursión con el que se obtienen tiempos de ejecución óptimos, salvo para el tamaño de matriz 5120. En este caso el tiempo de ejecución obtenido con los valores proporcionados por el modelo es un 3.49 % superior al óptimo experimental. La desviación entre el tiempo de ejecución de la rutina y el tiempo proporcionado por el modelo adopta ahora valores que van del 0.17 % al 15.52 %, con un valor para la fluctuación ($\Phi_{err} = 0.37\,\%$) entre los valores experimentales y los obtenidos con el modelo, más pequeño que con el modelo visto con anterioridad.

n	l	Experimental	Modelo	Desviación (%)
1664	1	**1.99**	**2.27**	14.08
1664	2	2.37	2.73	15.52
1664	3	3.12	3.27	5.12
2176	1	**4.27**	**4.83**	13.02
2176	2	4.77	5.51	15.52
2176	3	6.08	6.25	2.74
2688	1	**7.87**	**8.80**	11.92
2688	2	8.40	9.67	15.23
2688	3	10.28	10.52	2.38
3200	1	**13.02**	**14.51**	11.92
3200	2	13.56	15.51	14.38
3200	3	16.00	16.30	1.87
5120	1	56.80	56.71	0.17
5120	2	**56.44**	57.01	1.00
5120	3	60.04	**55.09**	8.25
5632	1	75.78	74.92	1.12
5632	2	73.50	74.56	1.45
5632	3	**71.70**	**70.97**	1.03

Tabla 4.6: Comparación de los tiempos de ejecución experimentales (en segundos) con los tiempos de ejecución proporcionados por el nuevo modelo para el algoritmo de Strassen, en Sol.

En la figura 4.3 se comparan para la rutina de Strassen en la plataforma Sol, las desviaciones entre los tiempos de ejecución experimentales y los calculados con el modelo inicial y con el modelo obtenido aplicando la metodología de remodelado propuesta. Los tamaños de problema utilizados para la comparativa son los empleados en la verificación del modelo inicial. Como se puede observar en la figura, la desviación respecto al tiempo experimental es en general menor con el nuevo modelo de la rutina. Se aprecia sobre todo en el caso de nivel de recursión $l = 3$. Por tanto el uso del remodelado nos permite obtener un modelo de mayor calidad que mejora la estimación del modelo original.

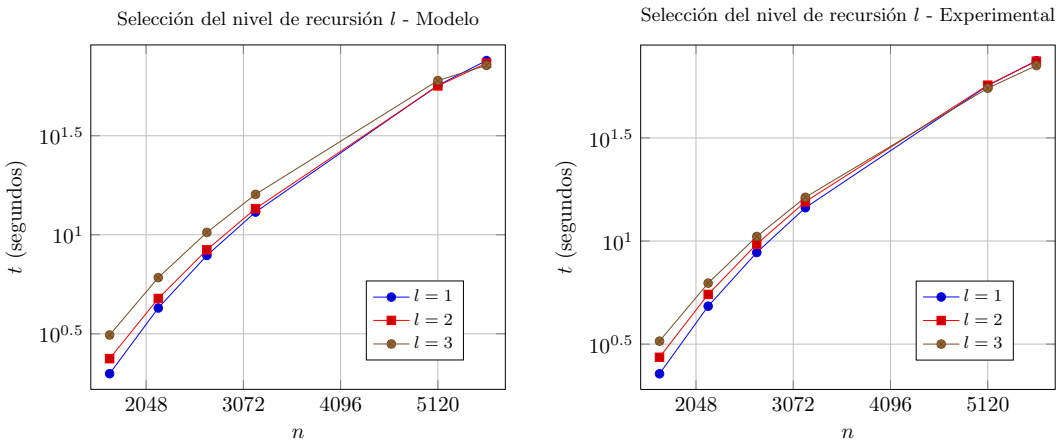


Figura 4.2: Selección del nivel de recursión l para el algoritmo de Strassen, en Sol (tiempos en escala logarítmica).

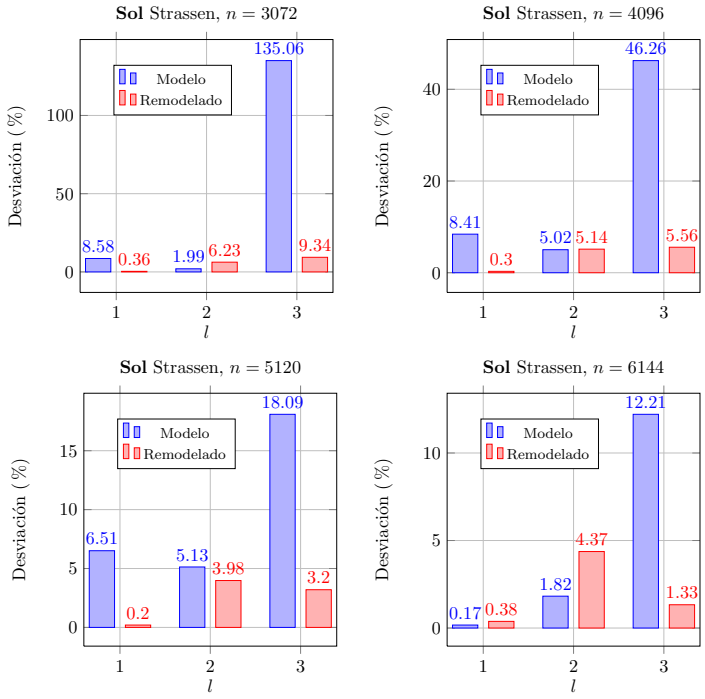


Figura 4.3: Comparación entre el modelo inicial (Modelo) y el nuevo modelo aplicando remodelado (Remodelado) de la rutina de Strassen para diferentes tamaños de problema, n, y niveles de recursión, l, en Sol.

Con los resultados obtenidos, se puede comprobar que los valores óptimos de los AP de una rutina varían de un sistema a otro y con el tamaño del problema. La posibilidad de construir nuevos modelos, permite mejorar la aproximación obtenida y realizar una selección satisfactoria de los AP.

4.3.3 Multiplicación paralela de matrices con replicación de la matriz B (BSR)

En este apartado se mostrará, a modo de ejemplo, la obtención de un nuevo modelo para la versión de la multiplicación paralela de matrices *rowwise block-striped* con replicación de la matriz B (BSR). La descripción de la rutina BSR y su modelo analítico se han mostrado en el capítulo anterior. Como en el caso de la multiplicación rápida de Strassen, para esta versión de la multiplicación paralela de matrices, partimos del modelo teórico (sección 3.3.3):

- **Distribución de A:**

$$t_s + \frac{n^2}{p} t_w \tag{4.11}$$

- **Distribución de B:**

$$t_s + n^2 t_w \tag{4.12}$$

- **Computación:**

$$\frac{2n^3}{p} k_{3,dgemm} \tag{4.13}$$

en el que ahora en lugar de utilizar los SP obtenidos para las rutinas básicas de computación y comunicación se utilizan los tiempos de ejecución proporcionados por sus respectivos modelos, de tal forma que el modelo de la rutina sería:

$$T(n,p) = t_{mult}\left(\frac{n^3}{p}\right) + (p-1)t_{Send}\left(\frac{n^2}{p}\right) + t_{Bcast}\left(n^2\right) \tag{4.14}$$

donde $t_{mult}(\frac{n^3}{p})$ es el tiempo de ejecución teórico de la rutina **dgemm** para una multiplicación secuencial de matrices de tamaño $\frac{n^3}{p}$, $t_{Send}(\frac{n^2}{p})$ es el tiempo de teórico para un mensaje punto a punto de tamaño $\frac{n^2}{p}$ y $t_{Bcast}(n^2)$ es el tiempo teórico para un mensaje de difusión de tamaño n^2.

Resultados experimentales

Con el fin de validar los modelos teóricos de esta rutina paralela se han obtenido resultados experimentales en dos plataformas distintas, Rosebud y Sol (descritas en el capítulo primero). En ambas plataformas se ha utilizado como librería básica un BLAS específico de la máquina

(Intel MKL). La librería utilizada para las comunicaciones entre procesos es MPI; en Rosebud se ha utilizado MPICH versión 2 y en Sol LAM-MPI.

Modelo para las computaciones

El modelo de las computaciones se reduce a obtener un modelo para la rutina de BLAS3 **dgemm**. Como ocurría con el algoritmo de Strassen los mejores resultados se obtienen con polinomios de tercer grado. Los coeficientes pueden calcularse usando alguno de los métodos ya descritos, pero la mejor aproximación se ha conseguido de nuevo con el método FIxed Least Square (FI-LS). Se ha comprobado si sería aprovechable el modelo de la rutina **dgemm** utilizado en el algoritmo de Strassen, pero debido a que en el algoritmo BSR se realizan multiplicaciones de matrices rectangulares, y no cuadradas, como ocurría en el algoritmo de Strassen, ha sido necesario obtener nuevos modelos, ya que de lo contrario la predicción era mucho peor. En Rosebud, los resultados obtenidos con el modelo de tiempo de ejecución de la rutina en la que los coeficientes se calculaban a partir de una ejecución de **dgemm**, proporcionaban tiempos teóricos muy inferiores a los experimentales. Con el fin de obtener un modelo más realista en Rosebud que reflejara adecuadamente la influencia que el esquema de acceso a memoria tiene sobre las rutinas básicas, se realizaron varias ejecuciones simultáneas de la rutina **dgemm** y se utilizó el valor medio de estas ejecuciones simultáneas para la obtención de los coeficientes del polinomio que aproxima su tiempo de ejecución.

Modelo para las comunicaciones

En lo que se refiere al modelo para las comunicaciones, aunque se puede utilizar un modelo lineal obtenido a partir de los datos de un simple mensaje de ping-pong entre dos procesos, la latencia, t_s, y el coste de envío de una palabra, t_w, tienen valores diferentes para una comunicación punto a punto que para un mensaje de difusión. Además, dado que t_s y t_w dependen del número de procesos utilizados en la comunicación, en general será necesario obtener funciones polinomiales para cada uno de los posibles tipos de envío que se realicen. En concreto, para este algoritmo de multiplicación de matrices y para la rutina que distribuye los bloques de la matriz A, se ha tenido en cuenta el esquema utilizado para acceder a los datos de la matriz antes de realizar su envío con la rutina **MPI_Send**, es decir, se ha construido un modelo para la rutina completa de envío de la matriz A y no sólo para la rutina **MPI_Send**. Adicionalmente, en Rosebud y para la rutina de difusión **MPI_Bcast**, cuando el número de procesos p era mayor que 12, fue necesario utilizar el método VAriable Least Square (VA-LS) para obtener un modelo de comunicaciones que proporcionara buenos resultados.

La tabla 4.7 compara los tiempos de ejecución proporcionados por el modelo (mod.) y los tiempos de ejecución experimentales (exp.) para la rutina **MPI_Bcast** cuando se utiliza el método FI-LS y cuando se utiliza el método VA-LS, junto con la desviación (dev.) para $p = 16$ procesos. Como se observa en la tabla, con el método FI-LS la desviación va desde el 2.10 %

hasta el 14.40 %, y aplicando el método VA-LS se consigue bajar la fluctuación en el error, adquiriendo ahora valores entre 0.04 % y 5.59 %.

n	512	1536	2048	3072	4096	5120	6144	6656
exp.	0.0528	0.4603	0.8157	2.0998	3.4852	5.4764	8.0366	9.0876
FI-LS	0.0495	0.4490	0.7986	1.7973	3.1956	4.9933	7.1905	8.4390
dev. (%)	6.13	2.46	2.10	14.40	8.31	8.82	10.53	7.14
VA-LS	0.0507	0.4605	0.8277	1.9823	3.5369	5.5356	7.9785	9.3666
dev. (%)	3.96	0.04	1.47	5.59	1.48	1.08	0.72	3.07

Tabla 4.7: Comparación de los tiempos (en segundos) experimentales y teóricos utilizando el método FI-LS y el método VA-LS para la rutina **MPI_Bcast** con $p = 16$ procesos, en Rosebud.

Discusión de los resultados experimentales

Como puede observarse en la figura 4.4, el modelo refleja el comportamiento de la rutina para diferentes tamaños de problema n y número de procesos p. Se puede ver tanto en la gráfica correspondiente al tiempo experimental (Experimental), como en la gráfica correspondiente al tiempo proporcionado por el modelo (Modelo), que el tiempo de ejecución de la rutina desciende hasta $p = 8$. A partir de dicho número de procesos las comunicaciones se realizan con dos tipos de red de interconexión: memoria compartida, cuando la comunicación es entre procesos en el mismo nodo y GigabitEthernet para la comunicación entre procesos en distintos nodos. Por ese motivo el tiempo de ejecución aumenta, volviendo a descender conforme se incrementa el número de procesos paralelos, pero sin llegar a obtener tiempos de ejecución inferiores a cuando se utilizan 8 procesos.

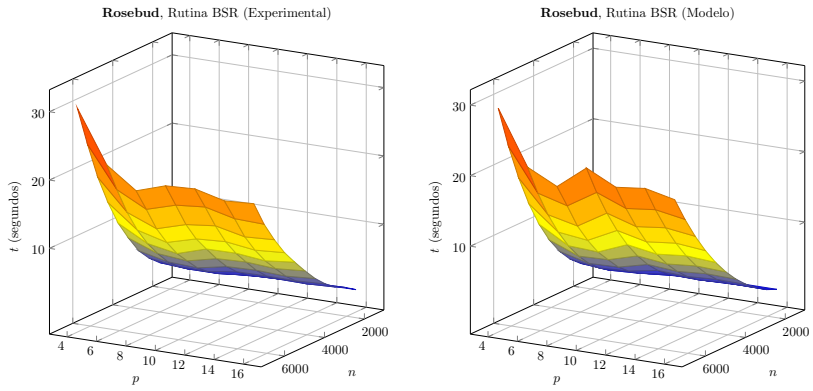


Figura 4.4: Comparación entre los tiempos de ejecución experimentales y los proporcionados por el modelo para la rutina BSR de multiplicación de matrices, con distintos tamaños de problema n y número de procesos p, en Rosebud.

La tabla 4.8 muestra el orden en que el modelo (mod.) selecciona el valor del parámetro

algorítmico p, y el orden de selección resultante a partir de los tiempos de ejecución experimentales (exp.) obtenidos en Rosebud. En la tabla se puede observar como el orden de selección de p varía con el tamaño del problema, pero con el modelo de tiempo de ejecución unido a la utilización del método VAriable Least Square (VA-LS) para modelar comunicaciones con diferentes tamaños de mensajes cuando se usa la rutina de difusión **MPI_Bcast** se consigue realizar una selección satisfactoria de p en casi todos los casos. Un valor de 0 en la desviación (dev.) significa que el valor de p seleccionado por el modelo coincide con el obtenido experimentalmente. En aquellos casos en los que la selección del valor de p no coincide con el valor experimental, la desviación en el tiempo de ejecución es muy pequeña, siendo la media de las desviaciones de sólo un 0.38 %.

	$p=4$			$p=8$			$p=12$			$p=16$		
n	exp.	mod.	dev. (%)	exp.	mod.	dev. (%)	exp.	mod.	dev. (%)	exp.	mod.	dev. (%)
1024	$2.^o$	$1.^o$	2.49	$1.^o$	$2.^o$	2.49	$3.^o$	$3.^o$	0	$4.^o$	$4.^o$	0
2048	$2.^o$	$2.^o$	0	$1.^o$	$1.^o$	0	$3.^o$	$3.^o$	0	$4.^o$	$4.^o$	0
2560	$2.^o$	$2.^o$	0	$1.^o$	$1.^o$	0	$3.^o$	$3.^o$	0	$4.^o$	$4.^o$	0
4096	$4.^o$	$4.^o$	0	$1.^o$	$1.^o$	0	$2.^o$	$3.^o$	0.37	$3.^o$	$2.^o$	0.37
5120	$4.^o$	$4.^o$	0	$1.^o$	$1.^o$	0	$3.^o$	$3.^o$	0	$2.^o$	$2.^o$	0
5632	$4.^o$	$4.^o$	0	$1.^o$	$1.^o$	0	$3.^o$	$3.^o$	0	$2.^o$	$2.^o$	0
6144	$4.^o$	$4.^o$	0	$1.^o$	$1.^o$	0	$3.^o$	$3.^o$	0	$2.^o$	$2.^o$	0
6656	$4.^o$	$4.^o$	0	$1.^o$	$1.^o$	0	$3.^o$	$3.^o$	0	$2.^o$	$2.^o$	0

Tabla 4.8: Orden de selección, de mejor a peor, de la rutina de multiplicación paralela de matrices (BSR), en Rosebud.

En la tabla 4.9, se muestra, para la plataforma Sol, el tiempo de ejecución experimental (exp.), el tiempo de ejecución proporcionado por el modelo (mod.), y la desviación (dev.) entre ambos para diferentes tamaños de problema y número de procesos. Los tiempos óptimos de ejecución obtenidos experimentalmente y los proporcionados por el modelo aparecen resaltados. Como se observa en la tabla, en general es preferible ejecutar en Sol la rutina con 8 procesos. Sólo para tamaños de problema menores de 1024 es recomendable ejecutar la rutina utilizando 4 procesos. Se puede comprobar con los resultados mostrados que el modelo predice correctamente el número de procesos con los que se obtienen tiempos de ejecución óptimos, y sólo cuando el tamaño de matriz es 1024 la predicción es incorrecta. No obstante, el tiempo de ejecución obtenido siguiendo la selección realizada por el modelo es únicamente un 5.41 % superior al tiempo óptimo experimental.

	n	1024	2048	2560	3072	3584	4096
$p = 2$							
exp.		0.413	2.695	4.913	8.254	12.694	18.826
mod.		0.444	2.760	5.081	8.424	12.972	18.911
dev. (%)		7.61	2.40	3.42	2.05	2.19	0.45
$p = 4$							
exp.		**0.351**	1.945	3.359	5.449	8.136	11.783
mod.		0.338	1.847	3.270	5.263	7.917	11.325
dev. (%)		3.58	5.04	2.63	3.41	2.68	3.89
$p = 6$							
exp.		0.367	1.870	2.997	5.037	6.822	9.488
mod.		0.362	1.791	3.055	4.775	7.000	9.796
dev. (%)		1.44	4.24	1.94	5.22	2.61	3.25
$p = 8$							
exp.		0.370	**1.740**	**2.750**	**4.436**	**6.166**	**9.033**
mod.		**0.312**	**1.507**	**2.549**	**3.949**	**5.753**	**8.008**
dev. (%)		15.68	13.38	7.32	10.97	6.70	11.36

Tabla 4.9: Comparación del tiempo (en segundos) experimental y teórico para diferentes números de procesos para la rutina de multiplicación paralela de matrices (BSR), en Sol.

4.3.4 Remodelado de la rutina de multiplicación paralela de matrices (BSR)

En esta sección se va a mostrar la aplicación de nuestra metodología a la obtención de un nuevo modelo para la rutina BSR. El método para construir este nuevo modelo consiste en definir una función polinomial para las diferentes combinaciones de n y AP. En este algoritmo el único AP que aparece es el número de procesos p, y al tratarse de una multiplicación de matrices con un coste de $O(n^3)$ se va a utilizar un polinomio de grado tres, de tal forma que la función polinomial sería de la siguiente forma:

$$\begin{aligned} T(n,p) = a_{3,1}n^3p + a_{3,0}n^3 + a_{3,-1}\frac{n^3}{p} + a_{2,1}n^2p + a_{2,0}n^2 + \\ a_{2,-1}\frac{n^2}{p} + a_{1,1}np + a_{1,0}n + a_{1,-1}\frac{n}{p} + a_{0,1}p + a_{0,0} + \frac{a_{0,-1}}{p} \end{aligned} \tag{4.15}$$

Se requiere, por tanto, el cálculo de doce coeficientes con alguno de los métodos indicados en la sección 4.2.2. En las pruebas realizadas los mejores resultados fueron obtenidos nuevamente con el método FIxed Least Square (FI-LS). La tabla 4.10 muestra los valores obtenidos para los doce coeficientes en Rosebud con $p = \{2, 4, 6\}$ y $n = \{1000, 1500, 2000, 2500, 3000, 3500, 4000\}$.

$a_{0,-1} = -8.921443e^{-01}$	$a_{0,0} = 6.421089e^{-01}$	$a_{0,1} = -8.618004e^{-02}$
$a_{1,-1} = 1.483827e^{-03}$	$a_{1,0} = -1.059608e^{-03}$	$a_{1,1} = 1.282712e^{-04}$
$a_{2,-1} = -7.001814e^{-07}$	$a_{2,0} = 5.088882e^{-07}$	$a_{2,1} = -3.868396e^{-08}$
$a_{3,-1} = 4.377307e^{-10}$	$a_{3,0} = -5.207178e^{-11}$	$a_{3,1} = 4.620444e^{-12}$

Tabla 4.10: Coeficientes para el nuevo modelo de tiempo de ejecución de la rutina BSR, en Rosebud.

Al igual que en los casos anteriores, con objeto de comprobar la validez de este nuevo modelo teórico de tiempo de ejecución, se realizan ejecuciones para diferentes combinaciones de tamaño de problema y parámetros del algoritmo. Se han seleccionado los siguientes valores representativos (`Valores_Chequeo`):

- Parámetros del algoritmo: número de procesos $p = \{2, 4, 6, 8\}$
- Tamaño de matriz $n = \{1024, 2048, 3584, 4608, 5632, 6656\}$

n / $p = 2$	1024	2048	3584	4608	5632	6656
exp.	0.238	1.794	9.012	18.824	34.014	55.781
mod.	0.235	1.752	8.955	18.695	33.710	55.133
dev. (%)	0.91	2.35	0.64	0.68	0.89	1.16
$p = 4$						
exp.	0.167	1.147	5.181	10.497	18.640	30.135
mod.	0.164	1.118	5.237	10.490	18.316	29.205
dev. (%)	1.98	2.55	1.09	0.08	1.74	3.09
$p = 6$						
exp.	**0.159**	1.109	4.227	8.202	14.145	22.453
mod.	**0.153**	0.978	4.117	7.935	13.497	21.117
dev. (%)	4.19	11.86	2.60	3.26	4.58	5.95
$p = 8$						
exp.	0.165	**1.066**	**4.029**	**7.559**	**12.517**	**19.499**
mod.	0.157	**0.962**	**3.647**	**6.793**	**11.322**	**17.490**
dev. (%)	5.00	9.71	9.48	10.14	9.54	10.30

Tabla 4.11: Comparación del tiempo (en segundos) de ejecución experimental y teórico, con el nuevo modelo, para diferentes número de procesos y tamaños del problema, en Rosebud.

La tabla 4.11 muestra el tiempo de ejecución proporcionado por el modelo (mod.) y el tiempo experimental (exp.) para diferentes tamaños de problema y número de procesos, junto con la desviación (dev.) entre ambos tiempos. Hay que indicar que se ha realizado la comprobación del modelo con un número de procesos mayor ($p = 8$) que el utilizado para obtener los coeficientes ($p = 6$). Por tanto el procedimiento es capaz de modelar el tiempo de ejecución con valores de experimentación fuera de su rango inicial. No se han obtenido tiempos de ejecución para

valores de número de procesos mayores que 8, dado que como se vio en la sección anterior los tiempos de ejecución de la rutina eran peores. Los tiempos óptimos, tanto experimentales como teóricos, aparecen resaltados. En la tabla se observa que el modelo predice satisfactoriamente, en todos los casos analizados, el número de procesos con los que se obtienen tiempos de ejecución óptimos, con una desviación entre el tiempo experimental obtenido a posteriori en la ejecución de la rutina y el tiempo proporcionado por el modelo que varía de 0.08 % a 11.86 %.

En la figura 4.5 se comparan para la rutina de multiplicación BSR, las desviaciones entre los tiempos de ejecución experimentales y los proporcionados por el modelo inicial y el modelo que se ha obtenido aplicando la metodología de remodelado propuesta, en la plataforma Rosebud. Los tamaños de problema utilizados para la comparativa son los empleados en la verificación del modelo inicial. Puede observarse que la desviación respecto al tiempo experimental es en general inferior con el nuevo modelo de la rutina. Sólo en el caso de $p = 8$, el nuevo modelo tiene una desviación mayor. Este comportamiento se debe, tal y como se ha comentado anteriormente, a que el remodelado se ha realizado con valores de p inferiores a 8.

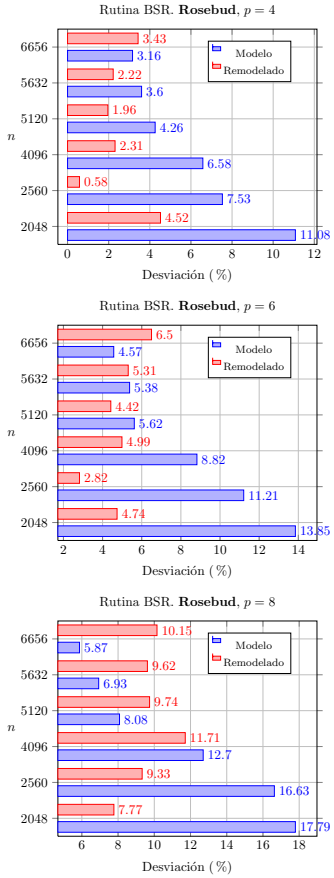


Figura 4.5: Comparación de las desviaciones respecto al tiempo experimental del modelo inicial (Modelo) y el nuevo modelo aplicando remodelado (Remodelado) de la rutina de multiplicación BSR, para diferentes tamaños de problema n y número de procesos p, en Rosebud.

4.4 Resumen y conclusiones

En este capítulo se ha mostrado un mecanismo para resolver el problema de obtención del tiempo de ejecución de una rutina de álgebra lineal en aquellos casos en los que se desconoce el modelo teórico de la rutina, o bien en los que el modelo analítico general no es lo suficientemente preciso. De esta manera, se ha mostrado cómo se ampliaría la metodología propuesta en el capítulo anterior añadiendo una metodología de remodelado.

En primer lugar se ha descrito el procedimiento a seguir para la obtención de modelos a partir de polinomios en los que cada uno de sus términos estaría formado por combinaciones de potencias del tamaño del problema y de los parámetros del algoritmo, y que constituye la base del mecanismo utilizado en el remodelado de la rutina. Para poder aplicar esta nueva metodología ha sido necesario introducir cuatro métodos para el cálculo de los coeficientes del polinomio que modela el tiempo de ejecución de la rutina y que permiten llegar a un compromiso entre el tiempo en el que se obtiene el modelo y la calidad de la aproximación que éste proporciona. Finalmente, se ha aplicado esta ampliación de la propuesta a la obtención de modelos de rutinas pertenecientes a los diferentes niveles de la jerarquía de librerías de álgebra lineal. Así mismo, se ha visto la forma de combinar el modelo teórico de una rutina de nivel superior con el procedimiento para la obtención de modelos polinomiales de las rutinas de los niveles inferiores que utiliza en su código. De igual manera, se ha mostrado la aplicación de la metodología de remodelado a rutinas de niveles superiores en aquellos casos en los que la utilización de la información obtenida de los modelos de las rutinas de niveles inferiores no proporcionaba resultados satisfactorios.

Capítulo 5

Modelado en sistemas heterogéneos de rutinas de álgebra lineal

5.1 Introducción

En los capítulos anteriores se ha mostrado como se puede realizar una selección de los parámetros ajustables de las rutinas de álgebra lineal con la metodología de modelado propuesta, de tal forma que se consiguen tiempos de ejecución cercanos al óptimo en plataformas homogéneas, en las que la rutina realiza una distribución lo más uniforme posible del trabajo entre sus procesadores. En este capítulo se amplía el campo de aplicación de nuestra propuesta de modelado del tiempo de ejecución de rutinas de álgebra lineal, al combinar el uso del modelo de tiempo de ejecución con la utilización de estrategias de asignación de procesos a procesadores. El propósito principal es que las rutinas de álgebra lineal diseñadas para sistemas homogéneos se ejecuten de manera eficiente en sistemas heterogéneos, sin necesidad de modificar el código de las mismas.

En primer lugar se describirán las modificaciones que es preciso realizar a la metodología presentada en los capítulos anteriores con el fin de adaptarla al caso de plataformas heterogéneas. Para ello será necesario introducir en el proceso de selección la asignación de procesos a procesadores, así como considerar la influencia que una determinada asignación de procesos a procesadores tiene sobre el valor de los parámetros del sistema. Posteriormente se planteará el problema de la asignación de procesos a procesadores, para a continuación describir una propuesta que permita abordar una solución a dicho problema de asignación.

5.2 Distribución del trabajo en sistemas heterogéneos

Las plataformas heterogéneas de computación están formadas por procesadores con distinta capacidad de cómputo, por lo que para conseguir rutinas de álgebra lineal que obtengan un buen rendimiento en este tipo de plataformas de computación es necesario que la distribución del trabajo a realizar por cada uno de su procesadores sea también heterogénea. Con el fin de resolver el problema de la distribución de trabajo entre procesadores, se han venido considerando tradicionalmente los siguientes enfoques [KL01]:

- La estrategia HoHe: en la que se mantiene la distribución homogénea de los procesos de un programa paralelo sobre los procesadores de la plataforma heterogénea, siendo la distribución de datos heterogénea sobre los procesos. Es decir, cada procesador se encarga de ejecutar un sólo proceso, pero la cantidad de datos asignada a cada proceso dependerá de la capacidad de cómputo del procesador que lo alberga.
- La estrategia HeHo: que emplea una distribución heterogénea de los procesos de un programa paralelo sobre los procesadores de la plataforma heterogénea y se mantiene la distribución de datos homogénea en los procesos. Es decir, todos los procesos realizan la misma cantidad de cómputo, mientras que a cada procesador se le asignan más o menos procesos en función de su capacidad de cómputo.

En lo que se refiere a la estrategia HoHe, existen numerosos trabajos, entre los que se encuentran el de Kalinov y Lastovetsky [KL01], que proponen modificaciones a la distribución cíclica por bloques tradicional para adaptarla a plataformas heterogéneas; el de Dovolnov *et al.* [DKK03], quienes proponen una serie de heurísticas para la distribución heterogénea de datos; el de Beaumont *et al.* [BBP+01], en el que se implementan algoritmos como el de la multiplicación de matrices para plataformas heterogéneas; el de Ohtaki *et al.* [OTBS04], donde se utiliza una descomposición de datos recursiva de tal forma que se obtiene un balanceo de la carga eficiente y un incremento efectivo del nivel de recursión del algoritmo de Strassen. Si bien en todos estos trabajos se consiguen rutinas altamente eficientes, todos ellos conllevan el doble esfuerzo de tener que realizar un nuevo diseño de la rutina y de escribir el código correspondiente para que cada proceso maneje diferentes cantidades de datos dependiendo de la capacidad del procesador donde se ejecute.

Por otra parte, con la estrategia HeHo se posibilita que las rutinas paralelas ya diseñadas para obtener un alto rendimiento en plataformas homogéneas con una distribución homogénea de procesos sobre los procesadores y distribución homogénea de datos sobre procesos (estrategia HoHo [KK05]) se ejecuten de forma eficiente en plataformas heterogéneas. Sin embargo, la estrategia necesita que se realice una correcta distribución del trabajo, lo que implica una selección de los procesadores a usar, del número de procesos paralelos, y de la asignación de procesos a procesadores. La búsqueda de la distribución de trabajo más adecuada se puede

realizar con la ayuda de modelos teóricos que aproximen, para cada posible selección, el tiempo de ejecución de la rutina, dado que el coste computacional que añade el cálculo de la aproximación que proporcionan suele ser bajo en comparación al tiempo de ejecución real de la rutina. Como ya se mostró en capítulos anteriores, en los modelos teóricos de tiempo de ejecución aparecen una serie de parámetros que representan las características de la plataforma de cómputo. En el caso de plataformas heterogéneas, distribuidas o de carga variable, los valores de estos parámetros variarán de un procesador a otro, estática o dinámicamente, y dependerán también de la asignación de procesos a procesadores. De tal forma que se plantea un problema, el de la asignación general, que es del tipo NP-completo [ZR87, LL94] y por tanto la selección óptima de procesadores, procesos y su asignación; se dificulta especialmente, siendo normal la búsqueda de soluciones aproximadas por medio de la aplicación de algún método heurístico [ZR87, FMT03, SKRS03, VTS+04].

En los últimos años la resolución de este tipo de problemas ha generado un gran interés entre la comunidad científica y se están dedicando importantes esfuerzos a su resolución. Así por ejemplo, puede citarse el trabajo de Kalinov y Klimov [KK05], que consideran el rendimiento de cada procesador como una función del número de procesos que ejecuta y del volumen de datos procesado por el procesador; el trabajo de Kishhimoto y Ichikawa [KI04], en el que se establece una fórmula aproximada del tiempo de ejecución total de una rutina, extrayendo un factor constante a partir de medidas de resultados experimentales con tamaños de problema pequeños y utilizando el método de los mínimos cuadrados; el trabajo de Muttoni *et al.* [MCGZ04], en el que se extiende, bajo ciertas condiciones, un modelo analítico de plataformas paralelas a los sistemas grid, con el propósito de realizar una óptima selección de procesadores y asignando un proceso a cada procesador seleccionado.

Como se ha visto en los capítulos anteriores, por medio de la metodología de modelado propuesta para rutinas de álgebra lineal ejecutándose en plataformas homogéneas, se podía realizar una selección automática de los parámetros del algoritmo, obteniendo unos tiempos de ejecución de las rutinas cercanos al óptimo, y sin necesidad de volver a escribir el código de las mismas. Recordemos que las decisiones que había que tomar en el caso homogéneo eran: el número de procesadores a utilizar (al tratarse de un sistema homogéneo, cada proceso se asigna a un procesador), la topología lógica de los procesos y el tamaño de bloque para la computación y para la comunicación. En un entorno de trabajo heterogéneo, como el que se aborda en este capítulo, se debe decidir además: el número de procesos a generar y el número de procesos que se asignan a cada procesador. Si consideramos una plataforma heterogénea y una rutina homogénea, se puede reutilizar el modelo ya existente para seleccionar los parámetros óptimos, y utilizar una estrategia de asignación con el fin de realizar la asignación de procesos a procesadores. La plataforma heterogénea se va a considerar estática, es decir, que la carga del sistema no varía con el tiempo. Con el propósito de adecuar la metodología propuesta a esta nueva situación, se puede considerar el valor de cada parámetro del sistema, SP, como el valor máximo para todos los procesadores (o entre cada pareja de procesadores cuando se trate de un SP asociado con el coste de las comunicaciones), es decir, cuando ejecutemos una

rutina homogénea en una plataforma heterogénea el coste de realizar una operación básica se considerará como el correspondiente al procesador más lento, pudiendo modificar a posteriori el valor del SP considerado en función de la carga de trabajo del sistema. Aunque el problema de asignación de procesos a procesadores se ha tratado como un problema de asignación de un grafo de tareas [SKK03], en nuestra propuesta se utiliza un enfoque diferente, y junto con las decisiones tomadas satisfactoriamente por el modelo de tiempo de ejecución (número de procesos, topología lógica...), se propone una asignación de procesos a procesadores heterogénea en función de las capacidades de computación y comunicación de los procesadores. Este planteamiento ya ha sido aplicado con éxito a la distribución del trabajo de esquemas de programación dinámica homogéneos en plataformas heterogéneas [CGMG05].

5.3 Modelo de tiempo de ejecución en plataformas heterogéneas

Como vimos en el capítulo 2, el modelo de tiempo de ejecución de una rutina de álgebra lineal ejecutándose en una plataforma homogénea, en la que todos sus procesadores son de iguales características y están conectados homogéneamente, se formula como:

$$t(n) = f(n, AP, SP) \tag{5.1}$$

donde aparece el tamaño del problema n a resolver, los parámetros algorítmicos AP y los parámetros del sistema SP. Los SP determinan la influencia del sistema de computación en la ejecución de la rutina y representan tanto el tiempo requerido por un procesador en realizar una operación utilizando una rutina de BLAS (parámetros $k_{nivel,operacion}$), como el coste de las comunicaciones entre procesadores (parámetros t_s y t_w). El valor de cada SP se obtiene de manera experimental en cada plataforma y es el mismo para toda la plataforma. Los AP que aparecen en plataformas homogéneas son el número de procesadores a utilizar entre los disponibles en el sistema, su distribución lógica, por ejemplo en malla 2D de r filas y c columnas de procesos, y el tamaño de bloque cuando se utiliza una distribución de datos cíclica por bloques o del tipo *"checkerboard"* [GGKK03]. Como se comprobó en los capítulos anteriores, el valor de los SP depende del tamaño del problema a resolver y del valor de los AP, de tal forma que la anterior formulación para el modelo de tiempo de ejecución, quedaría como:

$$t(n) = f(n, AP, g(n, AP)) \tag{5.2}$$

Como ya se ha comentado, en un sistema heterogéneo puede ser preferible utilizar un número distinto de procesos que de procesadores, y asignar un número distinto de procesos a cada procesador en función de su capacidad de cómputo y de comunicación. De esta forma, se requiere

considerar un nuevo AP que permita reflejar la heterogeneidad de la asignación de procesos a procesadores. Este nuevo parámetro es en realidad un vector de parámetros, $d = (d_1, \ldots, d_P)$, en el que cada d_i representa el número de procesos asignado a cada procesador, siendo P el número de procesadores disponibles en el sistema y $D = \sum_{i=1}^{P} d_i$ el número total de procesos generados para resolver la rutina paralela modelada.

La formulación que se ha venido utilizando hasta ahora para el modelo de tiempo de ejecución se vuelve más complicada en el caso heterogéneo. Por una parte el valor de los diferentes SP asociados con la computación variará de un procesador a otro, y además estos valores también variarán con el número de procesos asignados a cada procesador (conforme asignemos más procesos a un procesador dado, más tiempo necesitará éste para realizar cada cómputo), y por otra parte, el valor de los SP asociados con la comunicación será también diferente para cada una de las distintas parejas de procesadores (el coste de las comunicaciones entre dos procesadores se verá afectado por el número de procesos comunicados entre esos procesadores). En una primera aproximación se podría considerar un único valor para toda la plataforma heterogénea, dada la sincronización implícita existente en las rutinas paralelas de álgebra lineal, y así para cada SP de computación habría un sólo valor que sería el máximo de los valores de ese SP en cada uno de los procesadores, y para los SP de comunicación el máximo entre parejas de procesadores. De cara a plantear un modelo lo más simple posible, se considerará que en el caso de que a un procesador i se le asignen d_i procesos, el valor de sus SP se verá aumentado d_i veces. El valor estimado inicialmente para los SP, se podrá modificar a posteriori en tiempo de ejecución en función de la disponibilidad del procesador o de la red de interconexión, por medio de interpolación lineal. La rutina podrá ser ejecutada con el valor seleccionado para los AP con ayuda del modelo de tiempo de ejecución. Por consiguiente, no se considera en la ejecución de la rutina la heterogeneidad de la plataforma, y sólo se tiene en cuenta en la selección del número de procesos y de procesadores y en la asignación de procesos a procesadores.

5.3.1 Modelado de la rutina paralela de factorización LU

En este apartado se mostrará, mediante una serie de experimentos, la validez de los modelos que aproximan el tiempo de ejecución de una rutina de álgebra lineal homogénea con las consideraciones realizadas para los valores de los SP cuando la plataforma de ejecución es heterogénea. La rutina utilizada es una versión paralela de la factorización LU por bloques, y ha sido implementada siguiendo los algoritmos presentados en [GL96] y [CDO+96], de manera que sería equivalente a la rutina **pdgetrf** de la librería ScaLAPACK. Se emplearán los sistemas SUNEt y TORC (descritos en la sección de herramientas computacionales del capítulo 1) para la validación experimental. Primero se mostrará el modelo teórico de tiempo de ejecución de la rutina LU y posteriormente se comparará la aproximación obtenida con el modelo propuesto con los tiempos de ejecución obtenidos en cada sistema heterogéneo.

Modelo de tiempo de ejecución de la rutina LU

Antes de iniciar la factorización LU de la matriz A, ésta debe estar distribuida siguiendo una distribución cíclica por bloques entre los $r \times c$ procesos de la plataforma heterogénea, configurados lógicamente con una topología en malla 2D. En la rutina se realizan llamadas a una serie de rutinas de ScaLAPACK, PBLAS y LAPACK para cálculos aritméticos y de MPI para las comunicaciones. En el primer paso, para realizar la factorización LU en el bloque A_{11} se usará la rutina **pdgetf2** del nivel 2 de ScaLAPACK en el proceso $\{0, 0\}$. Para resolver los sistemas de ecuaciones: $L_{21}U_{11} = A_{21}$ y $L_{11}U_{12} = A_{12}$ se usará la rutina **pdtrsm** del nivel 3 de PBLAS en los procesos de la fila 0 y de la columna 0 de la malla, respectivamente. Por último para resolver la ecuación: $L_{21}U_{12} + L_{22}U_{22} = A_{22}$, reformulada como $\tilde{A}_{22} = A_{22} - L_{21}U_{12}$, se usará la rutina **pdgemm** del nivel 3 de PBLAS en todos los procesos de la malla. En los siguientes pasos se irá distribuyendo el trabajo entre los procesos de las consecutivas filas y columnas de la malla 2D.

El modelo de tiempo de ejecución para esta rutina será:

- **Computación:**

$$T_{COMPU} = \frac{2}{3}\frac{n^3}{p}k_{3,dgemm} + \frac{r+c}{p}n^2bk_{3,dtrsm} + \frac{1}{3}nb^2k_{2,dgetf2} \tag{5.3}$$

- **Comunicación:**

$$T_{COMU} = \frac{2ng}{b}t_s + \frac{2n^2g}{p}t_w \tag{5.4}$$

con $g = max(r, c)$. En el modelo aparecen tres SP computacionales ($k_{3,dgemm}$, $k_{3,dtrsm}$, $k_{2,dgetf2}$) que corresponden al coste computacional de una operación básica realizada por cada una de las rutinas de niveles inferiores que se utilizan, y los SP de comunicaciones (t_s, t_w) que corresponde al coste de la comunicación entre parejas de procesadores para las rutinas básicas de comunicación MPI. El valor de los SP mostrados dependerá del tamaño del problema, n, y de los valores que se escojan para los AP (bloque de cálculo, b, número de procesos, p, asignación de procesos a procesadores, d, y forma de la malla lógica 2D de r filas por c columnas de procesos).

5.3.2 Resultados experimentales

Como se ha comentado, los experimentos se van a llevar a cabo en las plataformas SUNEt y TORC. A continuación se indicará la nomenclatura utilizada para referirnos a los procesadores de cada una de estas plataformas:

- SUNEt: 5 SUN Ultra 1 (P_0 - P_4) y 1 SUN Ultra 5 (P_5). La SUN Ultra 5 es aproximadamente dos veces más rápida que una SUN Ultra 1 en cálculos con rutinas de álgebra

lineal. La red de interconexión es muy lenta (Ethernet), por lo que resulta complicado obtener ejecuciones paralelas que sean más rápidas que las secuenciales ejecutadas en la SUN Ultra 5.

- TORC: Se utilizará para los experimentos la red de interconexión de 100 Mbits (FastEthernet) y un total de 19 procesadores, de los cuales 8 son procesadores Intel duales (DPIII) (P_0 - P_{15}), 1 es un procesador Intel Pentium III a 600 MHz (SPIII) (P_{16}), otro es un procesador Intel AMD Athlon (ATH) (P_{17}) y el último es un procesador Intel Pentium 4 a 1.7 GHz (17P4) (P_{18}).

Los valores medidos para los SP computacionales en cada plataforma, con la librería ATLAS [ATL] como librería básica, se muestran en la tabla 5.1 ($k_{3,dtrsm}$ y $k_{3,dgemm}$) y en la tabla 5.2 ($k_{2,dgetf2}$). Como puede observarse en las tablas, los valores de k_3 ($k_{3,dgemm}$ y $k_{3,dtrsm}$) dependen en general del tamaño de bloque b, mientras que el valor de k_2 resulta independiente del valor de b.

		Tamaño de bloque, b			
Sistema	Procesador	16	32	64	128
SUNEt	SUN1	0.007	0.006	0.006	0.006
	SUN5	0.004	0.003	0.003	0.003
TORC	DPIII	0.004	0.003	0.003	0.003
	SPIII	0.004	0.002	0.003	0.002
	ATH	0.001	0.001	0.001	0.001
	17P4	0.001	0.001	0.001	0.001

Tabla 5.1: Valores (en μsegundos) de k_3 ($k_{3,dgemm}$ y $k_{3,dtrsm}$) con ATLAS, para la rutina LU con varios tamaños de bloque, en SUNEt y TORC.

		Tamaño de bloque, b			
Sistema	Procesador	16	32	64	128
SUNEt	SUN1			0.070	
	SUN5			0.050	
TORC	DPIII			0.015	
	SPIII			0.015	
	ATH			0.012	
	17P4			0.012	

Tabla 5.2: Valores de $k_{2,dgetf2}$ (en μsegundos), con ATLAS, para la rutina LU con varios tamaños de bloque, en SUNEt y TORC.

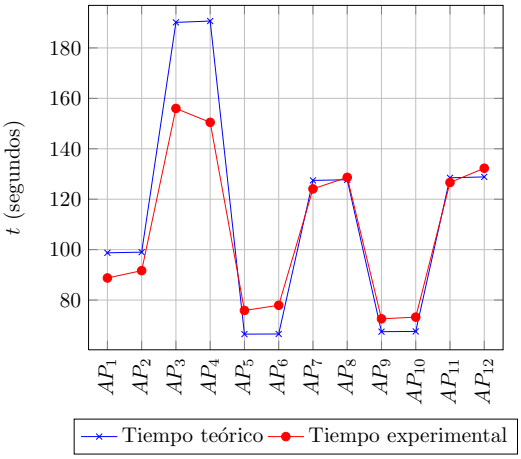
En lo que se refiere a los valores medidos para los SP de comunicaciones, han resultado constantes respecto al tamaño de problema. En SUNEt son aproximadamente $t_s = 170$ μsegundos y $t_w = 7.5$ μsegundos. En cuanto a la plataforma TORC, los valores medidos son $t_s = 10$

μsegundos y $t_w = 0.25$ μsegundos entre procesos en el mismo procesador, y $t_s = 100$ μsegundos y $t_w = 0.72$ μsegundos entre procesos en diferentes procesadores. En ambas plataformas los valores de los SP relacionados con la comunicación se han obtenido realizando experimentos con una rutina de *ping-pong*, empleando la librería de MPI instalada en cada plataforma.

Como se puede comprobar, el valor de los SP computacionales es distinto para cada tipo de procesador, no depende del tamaño del problema a resolver, y sí del valor de los AP. En cuanto a los SP relacionados con las comunicaciones, su valor varía con el tipo de red de interconexión utilizada y en menor medida, con el tipo de procesador. En TORC la velocidad de las comunicaciones es prácticamente igual entre los diferentes procesadores, ya que la red utilizada es la misma, excepto si hay comunicación entre procesadores duales, en cuyo caso la velocidad de comunicación será mayor al realizarse en memoria compartida. En SUNEt la velocidad de las comunicaciones es también prácticamente constante entre parejas de procesadores, su red de interconexión es más lenta que la de TORC, y al ser del tipo Ethernet aparecerán colisiones conforme se incremente el número de procesos, por lo que podría ser conveniente considerar la dependencia de t_s y t_w respecto a dicho número. Como se mostrará a continuación, estas diferencias entre los valores de los SP de una plataforma a otra darán lugar a una selección distinta para los parámetros algorítmicos con los que se obtienen mejores tiempos de ejecución de la rutina de álgebra lineal.

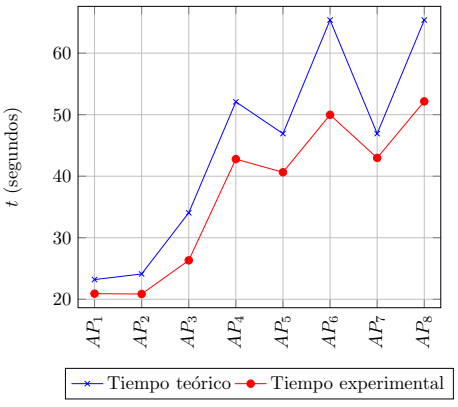
Utilizando el modelo de tiempo de ejecución mostrado para la rutina de factorización LU (ecuaciones 5.3 y 5.4), los valores de los SP obtenidos en cada una de las plataformas (tabla 5.1 y 5.2), y las consideraciones realizadas sobre sus valores cuando se asigna a un procesador más de un proceso, se puede realizar una estimación del tiempo de ejecución de la rutina para diferentes combinaciones de los AP (asignación de procesos a procesadores, topología lógica y tamaño de bloque). En la figura 5.1 para SUNEt y en la 5.2 para TORC, se muestran los tiempos de ejecución de la rutina LU proporcionados por el modelo de tiempo de ejecución, y el tiempo obtenido experimentalmente para diferentes combinaciones de valores de los AP. Como se observa en las figuras, en ambas plataformas los tiempos de ejecución teóricos están próximos a los experimentales y, para una asignación de procesos a procesadores dada, el modelo selecciona adecuadamente el resto de parámetros algorítmicos.

En SUNEt (figura 5.1) para un tamaño de problema $n = 2048$ y una asignación de 8 procesos a 6 procesadores, la combinación de los AP que proporciona el menor tiempo de ejecución para los casos considerados es la AP_9, que utiliza un tamaño de bloque de 32, una topología lógica de 2×4 y una asignación de procesos a procesadores de $(P_0, P_1, P_2, P_3, P_4, P_5) = (1, 1, 1, 1, 2, 2)$. Se observa que para una asignación dada de procesos a procesadores es preferible utilizar topologías de procesos menos rectangulares y tamaños de bloque más pequeños. La desviación entre el tiempo aproximado por el modelo y el tiempo experimental adopta valores entre el 1 % y el 21 %, siendo la media de estas desviaciones para las 12 combinaciones de los AP consideradas del 9.25 %.



	Asignación $p = 8, P = 6$ $(P_0, ..., P_5)$	$p = r \times c$	b
AP_1	(3,1,1,1,1,1)	2×4	32
AP_2	(3,1,1,1,1,1)	2×4	64
AP_3	(3,1,1,1,1,1)	1×8	32
AP_4	(3,1,1,1,1,1)	1×8	64
AP_5	(2,1,1,1,1,2)	2×4	32
AP_6	(2,1,1,1,1,2)	2×4	64
AP_7	(2,1,1,1,1,2)	1×8	32
AP_8	(2,1,1,1,1,2)	1×8	64
AP_9	(1,1,1,1,2,2)	2×4	32
AP_{10}	(1,1,1,1,2,2)	2×4	64
AP_{11}	(1,1,1,1,2,2)	1×8	32
AP_{12}	(1,1,1,1,2,2)	1×8	64

Figura 5.1: Comparación entre el tiempo experimental y el proporcionado por el modelo en SUNEt, con diferentes combinaciones de los AP, para una asignación de $p = 8$ procesos en $P = 6$ procesadores, con $n = 2048$.



	Asignación $p = 8, P = 19$ $(P_0, ..., P_{18})$	$p = r \times c$	b
AP_1	(1,0,1,0,1,0,1,0,1,0,1,0,1,0,0,0,0)	4×2	32
AP_2	(1,0,1,0,1,0,1,0,1,0,1,0,1,0,0,0,0)	8×1	32
AP_3	(1,0,1,0,1,0,1,0,1,0,0,0,0,0,0,2,0)	4×2	32
AP_4	(1,0,1,0,1,0,1,0,1,0,0,0,0,0,0,2,0)	8×1	32
AP_5	(1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,2,2,1)	4×2	32
AP_6	(1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,2,2,1)	8×1	32
AP_7	(1,0,1,0,1,0,1,0,1,0,0,0,0,0,0,2,0,0)	4×2	32
AP_8	(1,0,1,0,1,0,1,0,1,0,0,0,0,0,0,2,0,0)	8×1	32

Figura 5.2: Comparación entre el tiempo experimental y el proporcionado por el modelo en TORC, con diferentes combinaciones de los AP, para una asignación de $p = 8$ procesos en $P = 19$ procesadores, con $n = 4096$.

En la plataforma TORC (figura 5.2) para un tamaño de problema $n = 4096$ y una asignación de 8 procesos a 19 procesadores, la combinación de los AP que proporciona el menor tiempo de ejecución es la AP_1, que consiste en utilizar un valor para el tamaño de bloque de 32, de una topología lógica de 4×2 y de una asignación de procesos a procesadores de $(P_0, ..., P_{18}) = (1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0)$. De forma similar a lo que ocurría en SUNEt, se observa que para una asignación dada de procesos a procesadores es preferible utilizar topologías de procesos menos rectangulares. La desviación entre el tiempo aproximado por el modelo y el tiempo experimental toma valores entre el 9 % y el 31 %, siendo la media de estas desviaciones para las 8 combinaciones de los AP consideradas del 19.82 %.

Podemos concluir que en ambas plataformas heterogéneas, la curva del tiempo teórico de ejecución proporcionada por el modelo tiene una forma similar a la del tiempo de ejecución experimental para las diferentes combinaciones de los AP consideradas, por lo que la estimación que obtiene nos permitirá tomar decisiones, antes de ejecutar la rutina, sobre la mejor combinación de la asignación de procesos a procesadores, topología lógica de los procesos y tamaño de bloque. En la siguiente sección se mostrará nuestra propuesta para encontrar el valor de los AP (D, d, b y $r \times c$) que minimizan el tiempo de ejecución de una rutina de álgebra lineal homogénea cuando se ejecuta en una plataforma heterogénea.

5.4 Asignación de procesos a procesadores

En nuestra propuesta de modelado y optimización de rutinas de álgebra lineal para plataformas heterogéneas, se debe seleccionar el número de procesos paralelos (D) y su asignación a procesadores (d). La solución se representa por un vector a, con al menos un valor válido (que correspondería al caso secuencial en el que un proceso se asigna a un procesador) y con un número de componentes teóricamente ilimitado (dado que en principio es posible tener un número ilimitado de procesos). El valor a_i representa el procesador donde se asigna el proceso i. El espacio de posibles soluciones para la asignación se representa por un árbol en el que cada nivel corresponde a un proceso, y cada nodo de este nivel, a uno de los P procesadores en el que el proceso puede ser asignado. En cada nivel l se considerará la asignación de l procesos. El número de componentes de a se verá limitado por el máximo nivel que se puede alcanzar en el árbol. El tamaño del árbol se podría hacer más pequeño si cada nodo representara a todos los procesadores que tuvieran iguales características, y cuando se seleccione un nodo de un tipo de procesador, repartir los procesos equitativamente entre todos los procesadores de ese tipo. Sin embargo, en rutinas de álgebra lineal hay que encontrar la mejor topología lógica ($r \times c$) para los procesos, por lo que se hace necesario contemplar en el árbol a cada procesador individualmente. La solución puede constar de cualquier número de procesos, y cada procesador puede tener más de un proceso asignado; además, el orden en el que los procesos se asignan a los procesadores no es importante. Por tanto, el árbol de posibles soluciones puede ser inicialmente combinatorio, dado que de cara a la solución final no importa el orden en el que son elegidos

sus nodos al recorrerlo, y con repeticiones, ya que a un procesador puede asignársele más de un proceso. En la figura 5.3 se muestra, a modo de ejemplo, un árbol de soluciones de este tipo, para una asignación con P procesadores y hasta 3 procesos (se desciende en el árbol de asignaciones hasta el nivel 3).

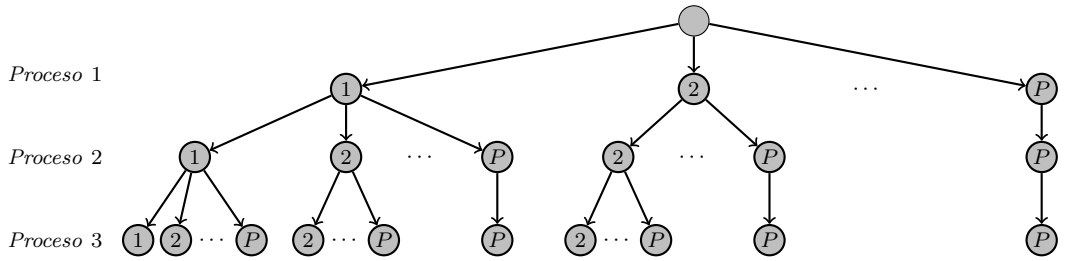


Figura 5.3: Árbol de asignación para P procesadores y hasta 3 procesos.

Por tanto, surge un problema de optimización, cuya solución puede obtenerse utilizando esquemas de búsqueda tales como los algoritmos de avance rápido (*greedy*), con retroceso (*backtracking*) o de ramificación y poda (*branch and bound*) en el árbol de asignaciones planteado [BB96, CLRS01], siendo el tiempo de ejecución de la rutina con la asignación correspondiente a cada nodo la función a optimizar. El tiempo de ejecución de la rutina se aproxima en cada nodo del árbol por su modelo analítico, donde los valores de los SP serían los correspondientes a la plataforma cuando se seleccionan los AP: D (número total de procesos) y d (número de procesos asignados a cada procesador), representados por el nodo. Adicionalmente para el nodo seleccionado (donde los valores de d y D están determinados), habrá que encontrar el valor del resto de los AP de la rutina que minimizan su tiempo de ejecución, que en el caso de rutinas de álgebra lineal serían: la mejor topología lógica 2D (r y c, con $D = r \times c$) y el mejor tamaño de bloque b para la distribución cíclica por bloques. En cada nodo del árbol se podría considerar otro problema de optimización, que consistiría en la búsqueda de la mejor asignación de procesadores en la malla lógica 2D de $D = r \times c$ procesos. El resultado de esta nueva búsqueda permitiría conseguir una reducción en el coste de las comunicaciones. En este trabajo, con el fin de no aumentar excesivamente el tamaño del árbol de asignaciones y por consiguiente el tiempo empleado en recorrerlo, nos limitaremos a la búsqueda de la mejor forma ($r \times c$) para la malla lógica 2D de procesos.

En lo referente a la utilización de un tipo u otro de árbol, cada nivel de un árbol de asignación corresponde a un proceso (de los D considerados) que se usará en la ejecución de la rutina paralela, y cada nodo de este nivel del árbol representa uno de los procesadores de la plataforma heterogénea al que el proceso puede ser asignado. En la figura 5.4 se representa un árbol combinatorio con repeticiones para tres procesadores y hasta tres procesos (tres niveles), y en la figura 5.5 se representa un árbol permutacional con repeticiones también para tres procesadores y hasta tres procesos (tres niveles). Como puede observarse, el número de nodos en el árbol permutacional es mayor que en el combinatorio, por lo que para un mismo número de procesos

y procesadores el tiempo en recorrerlo será mayor. Por otra parte el árbol permutacional nos permitirá considerar combinaciones para la asignación de procesos a procesadores que con el árbol combinatorio se descartarían.

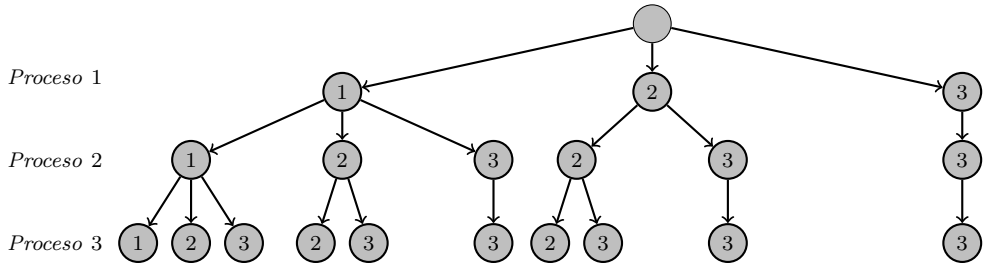


Figura 5.4: Árbol de asignación combinatorio con repeticiones para 3 procesadores y hasta 3 procesos.

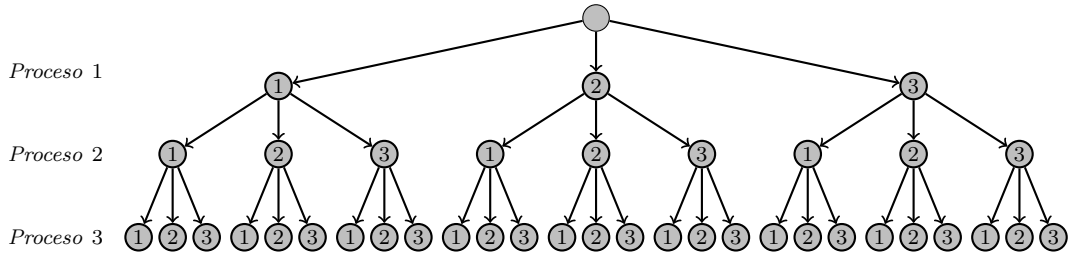


Figura 5.5: Árbol de asignación permutacional con repeticiones para 3 procesadores y hasta 3 procesos.

El problema de asignación que se plantea es reconocido como NP [LL94], por lo que encontrar una solución general al problema de optimización resultaría costoso. Sin embargo, la utilización de alguna estrategia con la que se eliminen nodos en el árbol de soluciones nos permite abordar su solución. Con dicho propósito, se asocian con cada nodo del árbol tres estimaciones del tiempo de ejecución: el Tiempo de Ejecución Estimado ($EET(nodo)$), el Menor Tiempo de Ejecución ($LET(nodo)$) y el Mayor Tiempo de Ejecución ($GET(nodo)$). El problema de optimización consiste en encontrar el nodo con el menor valor para $EET(nodo)$. Para cada nodo, $LET(nodo)$ y $GET(nodo)$ son una posible cota inferior y superior para la solución óptima de sus descendientes y se utilizarán para limitar el número de nodos evaluados y la altura del árbol. El valor de $LET(nodo)$ se podría obtener, por ejemplo, considerando que todos los procesadores no descartados participan en la computación. Una posible forma de obtener el valor de $GET(nodo)$ sería con una técnica de avance rápido, seleccionando en cada paso de la asignación a uno de los procesadores disponibles que proporcione el menor tiempo de ejecución teórico. En lo que sigue se verán otras posibles opciones para obtener los valores de $LET(nodo)$ y $GET(nodo)$.

En esquemas de *backtracking* o de *branch and bound* se define la cota Menor Tiempo de Ejecución Estimado ($MEET$) como $MEET = \text{mín}_{nodo \in \text{nodos evaluados}} GET(nodo)$. $MEET$ es la mejor cota superior para el tiempo de ejecución teórico que podemos estimar en un momento dado, y su valor se comparará en cada nodo con el de $LET(nodo)$, de tal forma que si $LET(nodo) \geqslant MEET$ no se continuará la búsqueda en las ramas de ese nodo.

Cuenca *et al.* proponen en [CGMG05] la utilización de cinco métodos para el recorrido del árbol de soluciones, y lo aplican a la distribución del trabajo de esquemas de programación dinámica homogéneos en plataformas heterogéneas. Sin embargo, los cinco métodos propuestos no reflejan por completo el conjunto de posibles decisiones a tomar que permitan obtener la mejor combinación de los AP que aparecen en rutinas de álgebra lineal. En este trabajo, se presenta un planteamiento modificado de su metodología, obteniendo algunas posibles aproximaciones:

- Método de Backtraking (**BTM**). Se trata de un método de búsqueda con retroceso (*backtracking*). En cada nodo los valores de GET y EET coinciden, y el valor de $LET = LET_{ari} + LET_{com}$. El valor de LET_{ari} se obtiene a partir del tiempo secuencial dividido por el máximo *speed-up* que se podría obtener al emplear todos los procesadores que todavía no se han descartado. En lo que se refiere al valor de LET_{com}, se podría considerar en cada nodo que su valor es el mismo que el valor utilizado para EET_{com}, al tratarse de una posible cota inferior del coste de las comunicaciones para los descendientes del nodo. Sin embargo, puede ocurrir que el valor de EET_{com} para un nodo de nivel l (se generan l procesos), podría ser mejorado por alguno de sus descendientes, debido a una mejor topología lógica resultante de la utilización de un número de procesos mayor que l. Por consiguiente, el valor de LET_{com} para un nodo se obtendrá como el mínimo valor entre el valor de su EET_{com} y un valor ficticio de EET_{com}. El valor ficticio de EET_{com} se calculará suponiendo la mejor forma para la malla lógica 2D de procesos que podría obtenerse desde este nodo.

- Método de Backtraking con Greedy (**BGRM**). Consiste en utilizar un esquema *greedy* para calcular el valor de GET en cada nodo. Se calcula el valor de EET para cada uno de los descendientes del nodo y aquel con el valor más bajo de EET se incluye en la solución. La búsqueda finaliza cuando el valor más bajo para el EET de los descendientes, es mayor o igual al del padre. El valor de LET se obtiene como en el método anterior.

- Método Greedy (**GRM**). Se utiliza un esquema *greedy* para el cálculo de LET. Con el fin de obtener una cota inferior, se incluirá en la solución aquel descendiente de un nodo que menos incremente el coste de las operaciones aritméticas, y se mantendrá constante el coste de las operaciones de comunicación. El proceso continúa hasta que el tiempo de ejecución aproximado en un nodo es mayor que el del padre. El valor de GET se obtiene al igual que en el método anterior. Ahora, la diferencia entre los valores de LET y GET es inferior a la de los métodos anteriores, de tal forma que se consigue reducir el número de nodos a examinar. Sin embargo, con este método es posible descartar una rama que proporcione la solución óptima.

- Método Greedy con árbol Combinatorio (**GCTM**). Consiste en aplicar un esquema *greedy* teniendo en cuenta que el árbol con el que se trabaja es combinatorio con repeticiones (figura 5.4). De forma similar a lo realizado en el método **BGRM**, se calcula el valor de EET para cada uno de los descendientes del nodo y aquel con el valor más bajo de EET se incluye en la solución. En este método, además, se propone un enfoque que permita relajar la condición establecida en el método **BGRM** de que cuando el valor más bajo para el EET de los descendientes es mayor o igual al del padre la búsqueda finaliza, de tal forma que no se descarten nodos con los que sería posible obtener mejores topologías lógicas de procesos. La búsqueda en profundidad continúa a través del mejor descendiente de cada nodo y no finaliza hasta alcanzar el nivel máximo que haya sido establecido de antemano para el árbol. Por otra parte, al considerar un árbol combinatorio, cuando se descarta un procesador ya no hay posibilidad de volver a asignarlo y, aunque este nuevo enfoque permite obtener una solución rápidamente, la solución proporcionada puede estar lejos de ser la óptima.
- Método Greedy con árbol Permutacional (**GPTM**). En este último método se aplica el esquema *greedy* en un árbol de permutaciones con repeticiones (figura 5.5) con el fin de evitar los inconvenientes del método anterior. De tal forma que, si un procesador se descarta en un paso, el procesador se vuelve a considerar en pasos sucesivos.

Como se ha comentado anteriormente, el valor de LET_{ari} se obtiene a partir del tiempo secuencial proporcionado por el modelo teórico de tiempo de ejecución de la rutina, dividido por el máximo *speed-up* que se podría obtener al emplear todos los procesadores que todavía no se han descartado. El cálculo del máximo *speed-up* se hará utilizando el siguiente procedimiento: suponiendo que tenemos una plataforma heterogénea con 10 procesadores y una selección de procesadores $u = (0, 0, 0, 0, 1, 1, 1)$ (tercer nivel del árbol de asignaciones, procesadores del 1 al 4 sin asignación, y con un proceso asignado a lo procesadores 5, 6 y 7, lo que indica que los procesadores del 1 al 4 han sido descartados, pero no los procesadores del 5 al 10), entonces el máximo *speed-up* total se podría obtener con una selección de procesadores $u = (0, 0, 0, 0, 1, 1, 1, 1, 1, 1)$ (se incluyen todos los procesadores no descartados previamente), y la fórmula para calcularlo sería:

$$S_{total} = \sum_{i=1}^{P} u_i s_i \tag{5.5}$$

donde s_i sería el *speed-up* relativo del procesador i. Para el cálculo del *speed-up* relativo de cada procesador se pueden utilizar diferentes métodos, por ejemplo, resolviendo con la rutina un problema de tamaño pequeño en cada uno de los diferentes tipos de procesadores que haya en la plataforma heterogénea, o utilizando el valor de los SP computacionales para las rutinas de los niveles inferiores.

En lo que se refiere a la determinación del valor ficticio de EET_{com} considerando la mejor

topología lógica de procesos que se podría obtener a partir de un nodo, hay que tener en cuenta que la mejor topología lógica de procesos viene determinada por el algoritmo implementado por la rutina y por la red de interconexión [CDD+95]. Por lo tanto, la elección se puede realizar a partir de su modelo teórico para las comunicaciones, junto con los resultados experimentales obtenidos de la ejecución de la rutina para un tamaño de problema pequeño, y utilizando diferentes combinaciones de r y c.

Con el propósito de mostrar los beneficios que se obtendrían al incluir una técnica que permita seleccionar, sin intervención del usuario, la mejor combinación de los AP de una rutina de álgebra lineal homogénea cuando se ejecuta en un sistema heterogéneo, el resultado de las decisiones obtenidas de forma automática utilizando cada uno de los cinco métodos propuestos anteriormente se va a comparar con los resultados que obtendrían tres tipos de usuario de una plataforma heterogénea. Las características que definirían el perfil de cada uno de los tres tipos de usuario serían:

- Un usuario voraz (GU), que para resolver su problema emplearía todos los procesadores disponibles en la plataforma heterogénea, con un proceso por procesador. Este tipo de usuario obtendría en la mayoría de los casos un tiempo de ejecución que no se acercaría al óptimo.
- Un usuario más conservador (CU), que decidiría usar un número de procesadores menor que el disponible (por ejemplo la mitad de los disponibles) con conocimientos sobre los beneficios que conlleva la utilización de códigos paralelos y que es consciente de que un elevado número de procesadores puede dar lugar a un incremento del coste de las comunicaciones y por tanto, de unos mayores tiempos de ejecución y pérdida de beneficio.
- Un usuario experto (EU), que podría decidir utilizar un número diferentes de procesos (1, la mitad o todos) y en los procesadores más adecuados, en función de la granularidad de la computación. Este usuario experto, tendría un perfil que incluiría amplios conocimientos en computación heterogénea, en software de álgebra lineal y en el sistema que se utiliza para resolver su problema.

En lo referente al resto de AP que es necesario considerar en la ejecución de una rutina de álgebra lineal (topología lógica de los procesos y tamaño del bloque), los tipos de usuarios descritos tomarán la mejor decisión proporcionada por nuestro método en cada caso con el número de procesos que hayan seleccionado.

5.5 Resultados experimentales

En este apartado se mostrará, mediante una serie de experimentos, la validez de combinar los modelos que aproximan el tiempo de ejecución de una rutina de álgebra lineal homogénea con

la metodología presentada para la distribución de trabajo en una plataforma heterogénea. Se comprobará que la metodología propuesta realiza una correcta selección de los parámetros del algoritmo y se comparará el resultado de las decisiones obtenidas con dicha metodología con el que obtendrían los tres tipos de usuarios modelados. En primer lugar se mostrarán los resultados obtenidos en las plataformas SUNEt y TORC, para posteriormente validar la metodología en una serie de plataformas heterogéneas que se han simulado a partir de las anteriores, y con configuraciones más complejas.

La rutina con la que se va a realizar la comprobación experimental es una versión paralela de la factorización LU por bloques (vista anteriormente en la sección de modelado). Para el cálculo del EET en un nodo del árbol de soluciones se usarán las ecuaciones 5.3 y 5.4, junto con los valores para los SP de computación y comunicación obtenidos experimentalmente en las plataformas SUNEt y TORC (tablas 5.1 y 5.2).

En lo que se refiere al cálculo del *speed-up* relativo para cada procesador, será el valor promedio de ejecución de la factorización LU respecto al procesador más lento, que se calculará a partir del modelo de tiempo de ejecución para la computación (ecuación 5.3) con $r \times c = 1$ (ejecución secuencial), junto con el valor medio de k_3 para los diferentes tamaños de bloque b considerados.

Para el cálculo del valor de EET_{com}, se usará el modelo con el que se aproxima el coste de las comunicaciones (ecuación 5.4). En lo referente al valor del EET_{com} ficticio para un nodo, se supondrá, como se ha comentado, la mejor topología lógica ($r \times c$) que se puede obtener a partir de ese nodo. Atendiendo al modelo de tiempo de ejecución de la rutina LU, la mejor topología lógica sería aquella en la que r sea lo más similar posible a c ($g = \text{máx}(r, c)$), de tal forma que, a partir de ese nodo, se irá descendiendo en el árbol de soluciones hasta que se llegue a una topología los más cuadrada posible, sin que esto suponga un incremento en el coste de las comunicaciones.

En la tabla 5.3 (plataforma SUNEt) y en la tabla 5.4 (plataforma TORC), se comparan los resultados obtenidos aplicando los cinco métodos propuestos, con los obtenidos por los tres tipos de usuario. Para cada método, se muestra la selección realizada de los AP (asignación de procesos a procesadores d, tamaño de bloque b y topología lógica $r \times c$), el tiempo de ejecución experimental de la rutina LU en segundos (tiempo solución), el tiempo en encontrar la solución en segundos (t.e.s) y la altura máxima del árbol de soluciones (nivel), que se corresponde con el número máximo de procesos paralelos establecidos. En la plataforma TORC, con mayor número de procesadores que la plataforma SUNEt, se ha utilizado para los métodos de búsqueda BTM y BGRM árboles de menor nivel con el fin de reducir el tiempo en encontrar la solución.

Se observa en las tablas que la selección de los AP realizada por los métodos propuestos proporciona, en general, mejores tiempos de ejecución para la rutina que la realizada por los usuarios modelados. Los métodos más voraces (GCTM y GPTM) alcanzan rápidamente la solución, pero a costa de obtener en algún caso peores resultados que los usuarios modelados.

Método	asignación de procesos a procesadores $d = (P_0, ..., P_5)$	b	topología lógica	tiempo solución	t.e.s	nivel
BTM	(1,1,1,1,1,1)	64	2×3	718.94	0.02	25
BGRM	(1,1,1,1,1,1)	64	2×3	718.94	0.04	25
GRM	(1,1,1,1,1,1)	64	2×3	718.94	0.02	25
GCTM	(1,1,0,0,0,1)	128	1×3	887.85	0.0001	25
GPTM	(1,1,0,0,0,1)	128	1×3	887.85	0.0005	25
CU	(1,1,0,0,0,1)	128	1×3	1047.13		
GU	(1,1,1,1,1,1)	64	2×3	887.85		
EU	(1,1,1,1,1,1)	64	2×3	887.85		

Tabla 5.3: Resultados obtenidos con los métodos de selección automática y los obtenidos por un usuario conservador, uno voraz y uno experto, para la rutina LU, con un tamaño de problema $n = 7680$, en SUNEt.

Cuando la plataforma tiene un número no muy elevado de procesadores y es poco heterogénea, los métodos más exhaustivos (BTM y BGRM) obtienen resultados aceptables sin requerir excesivo tiempo en tomar la decisión, debido a que pueden llegar a alcanzar suficiente profundidad en el árbol de asignación y obtener soluciones cercanas a la óptima. Como veremos a continuación, la situación cambia cuando estos métodos más exhaustivos se aplican en plataformas con mayor número de procesos y mayor grado de heterogeneidad.

Método	asignación de procesos a procesadores $d = (P_0, ..., P_{18})$	b	topología lógica	tiempo solución	t.e.s	nivel
BTM	(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0)	64	3×5	17.91	3.08	15
BGRM	(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0)	64	3×5	17.91	3.08	15
GRM	(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0)	64	4×4	15.27	0.06	25
GCTM	(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0)	64	1×1	43.16	0.0012	30
GPTM	(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0)	64	4×4	15.27	0.01	30
CU	(1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1)	64	3×3	23.77		
GU	(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)	32	1×19	33.57		
EU	(1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1)	64	3×3	23.77		

Tabla 5.4: Resultados obtenidos con los métodos de selección automática y los obtenidos por un usuario conservador, uno voraz y uno experto, para la rutina LU, con un tamaño de problema $n = 2048$, en TORC.

Con el fin de comprobar los resultados que se obtendrían con la metodología propuesta y su validez en plataformas con mayor número de procesadores y más heterogéneas, se han realizado experimentos simulados en una serie de plataformas ficticias. Estas plataformas ficticias están diseñadas a partir de las plataformas heterogéneas SUNEt y TORC, con incrementos y variaciones de procesadores. El nombre y la configuración de cada sistema simulado es:

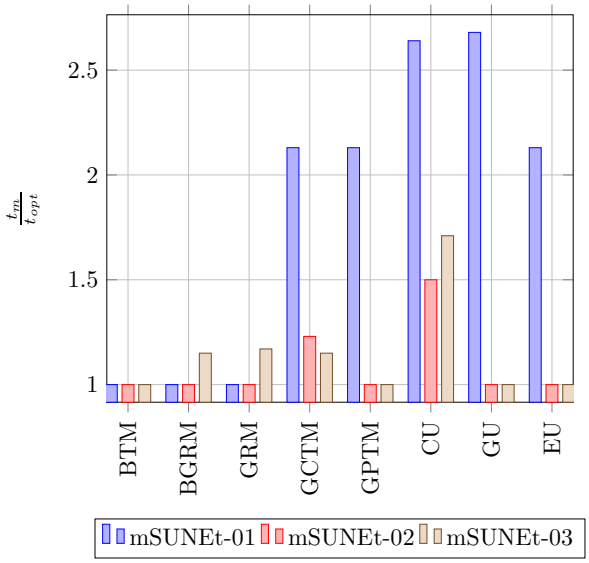
- **mSUNEt-01**. Partiendo de la plataforma SUNEt, se incrementa el número de SUN Ultra

5 a 5. El número total de procesadores sería 10 y habría 2 tipos distintos de procesador.

- **mSUNEt-02**. A partir de SUNEt, el número de SUN Ultra 1 y SUN Ultra 5 se incrementa en 10 cada uno. El número total de procesadores sería 20 y habría 2 tipos distintos de procesador.
- **mSUNEt-03**. A partir de SUNEt, el número de SUN Ultra 5 se incrementa a 10 y se añaden 5 procesadores nuevos dos veces más rápidos que una SUN Ultra 5. El número total de procesadores sería 20 y habría 3 tipos distintos de procesador.
- **mTORC-01**. Partiendo de la plataforma TORC, se incrementa el número de 17P4 a 11. El número total de procesadores sería 29 y habría 4 tipos de procesadores distintos.
- **mTORC-02**. Partiendo de la plataforma TORC, se incrementa el número de DPIII, SPIII y ATH a 10 y el de 17P4 a 20. El número total de procesadores sería 50 y habría 4 tipos de procesadores distintos.
- **mTORC-03**. A partir de TORC, se incrementa el número de DPIII a 10, el de SPIII a 15, el de ATH a 5 y el de 17P4 a 10, además se incluyen 6 procesadores, 10 de cada uno, con características distintas a los anteriores. El número total de procesadores sería 100 y habría 10 tipos de procesadores distintos.

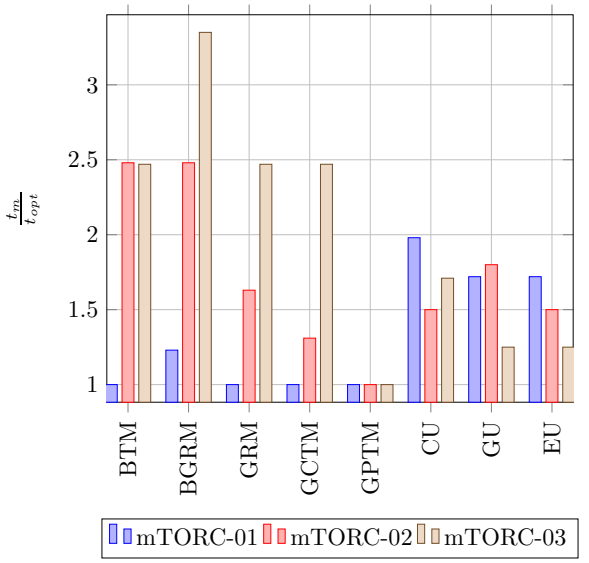
Los resultados de aplicar la metodología propuesta en las plataformas simuladas se pueden ver en la figura 5.6 y en la 5.7. En ellas se muestra, para cada método o usuario modelado, el cociente entre el tiempo de ejecución obtenido con la selección de los AP realizada con la metodología propuesta y el mejor de todos los tiempos de ejecución obtenidos (un valor de 1 indica que ambos tiempos coinciden), el tiempo en obtener la solución (t.e.s) en segundos, y la altura máxima del árbol de soluciones (nivel), que se corresponde con el número máximo de procesos paralelos establecidos. Se observa en las figuras que, cuando se incrementa el número de procesadores en SUNEt (mSUNEt-01 y mSUNEt-02) o en TORC (mTORC-01 y mTORC-02) y además la heterogeneidad (mSUNEt-03 y mTORC-03), los métodos exhaustivos necesitan recorrer más nodos en el árbol de asignaciones, lo que conlleva que el tiempo requerido en proporcionar una solución sea muy elevado en comparación con el utilizado al aplicar estos métodos en las plataformas SUNEt y TORC, con menos procesadores y menor heterogeneidad. Además, si a estos métodos se les restringe el recorrido en profundidad hasta un cierto nivel con el fin de reducir el tiempo en encontrar una solución, las soluciones obtenidas suelen ser peores que las de los usuarios modelados en plataformas con mayor número de procesadores (mTORC-02) y más heterogéneas (mTORC-03), al quedar zonas del árbol de soluciones sin explorar.

En lo referente a los métodos basados en esquemas voraces (GRM, y principalmente GCTM y GPTM), se observa que en general obtienen una solución aceptable en poco tiempo. En plataformas con un elevado número de procesadores o con un alto grado de heterogeneidad, como mTORC-02 y mTORC-03 (figura 5.7), los métodos GCTM y GPTM son capaces de profundizar



Sistema	Método	t.e.s	nivel
mSUNEt-01	BTM	3.7	20
	BGRM	0.02	20
	GRM	0.03	25
	GCTM	0.0005	25
	GPTM	0.0025	25
mSUNEt-02	BTM	46.7	20
	BGRM	88.4	20
	GRM	0.5	20
	GCTM	0.0004	25
	GPTM	0.004	25
mSUNEt-03	BTM	110.9	20
	BGRM	251.1	20
	GRM	0.04	25
	GCTM	0.0005	25
	GPTM	0.006	25

Figura 5.6: Comparación del cociente entre el mejor tiempo de ejecución y el obtenido con la aplicación de los métodos propuestos y la decisión realizada por un usuario conservador, voraz y experto en los sistemas simulados, para la rutina de factorización LU. Tiempo en encontrar la solución (t.e.s) en segundos y nivel de profundidad alcanzado. En las plataformas simuladas a partir de SUNEt, con tamaño de problema $n = 20000$.



Sistema	Método	t.e.s	nivel
mTORC-01	BTM	20.4	15
	BGRM	59.5	15
	GRM	0.7	20
	GCTM	0.0007	25
	GPTM	0.012	25
mTORC-02	BTM	259.4	15
	BGRM	792.3	15
	GRM	7.5	25
	GCTM	0.01	30
	GPTM	0.07	30
mTORC-03	BTM	109.2	10
	GRM	169.7	10
	GRBM	1274.3	5
	GCTM	0.08	25
	GPTM	2.3	40

Figura 5.7: Comparación del cociente entre el mejor tiempo de ejecución y el obtenido con la aplicación de los métodos propuestos y la decisión realizada por un usuario conservador, voraz y experto en los sistemas simulados, para la rutina de factorización LU. Tiempo en encontrar la solución (t.e.s) en segundos y nivel de profundidad alcanzado. En las plataformas simuladas a partir de TORC, con tamaño de problema $n = 20000$.

lo suficiente en el árbol de asignaciones y por lo tanto la solución que obtienen proporciona mejores resultados que la realizada por los usuarios modelados, de tal forma que para este tipo de casos resultan ser más adecuados que los métodos más exhaustivos. La aplicación de un esquema *greedy* en un árbol de permutaciones con repeticiones (método GPTM) ha proporcionado los mejores resultados globales. Sólo en el caso de la plataforma simulada mSUNEt-01 la solución obtenida ha sido peor que la del resto de los métodos, aunque igual a la proporcionada por un usuario experto. En lo que se refiere a los tiempos requeridos en encontrar la solución, el método GPTM ha requerido de media 0.4 segundos, siendo inferior al de los métodos más exhaustivos, aunque unas 40 veces más lento de media que el método GCTM, que considera árboles combinatorios.

En resumen, con los resultados obtenidos para la rutina paralela de factorización LU, podemos afirmar que la metodología propuesta permite, en general, realizar una selección automática de los AP mejor que la realizada por los usuarios modelados, y por tanto se consigue que la rutina se ejecute en un tiempo menor, en diversas plataformas heterogéneas con diferente número de procesadores y distinto grado de heterogeneidad.

5.6 Resumen y conclusiones

En este capítulo se ha descrito nuestra propuesta para resolver el problema de ajuste de rutinas homogéneas de álgebra lineal a las condiciones de una plataforma heterogénea de la forma más óptima posible y sin intervención del usuario.

Se ha comenzado introduciendo el problema de distribución del trabajo en plataformas heterogéneas y asignación de procesos a procesadores, y se ha ampliado nuestra propuesta de utilización de modelos de tiempo de ejecución con el fin de adaptarla a plataformas heterogéneas. Dado que en nuestra propuesta se pretende reutilizar el código ya existente de álgebra lineal desarrollado para plataformas homogéneas, la metodología de modelado planteada en los capítulos previos sigue siendo de aplicación, pero considerando la asignación de procesos a procesadores como un nuevo AP y su influencia en los valores de los SP, de tal forma que el modelo permita aproximar adecuadamente el tiempo de ejecución. Se ha continuado en el capítulo con una metodología que permita abordar el problema de la asignación y se ha combinado con los modelos de tiempo de ejecución de tal forma que la selección del mejor valor para los AP se pueda realizar sin intervención del usuario. La propuesta se ha aplicado a la rutina paralela de factorización LU, en plataformas con diferente grado de heterogeneidad. Su aplicación a otras rutinas paralelas de álgebra lineal se haría de forma similar, utilizando la metodología de modelado vista en los capítulos anteriores.

Capítulo 6

Modelado en sistemas multicore de rutinas de álgebra lineal

6.1 Introducción

En los capítulos anteriores se proponía una metodología de modelado para rutinas de álgebra lineal escritas bajo un interface de paso de mensajes como la librería MPI, comprobándose que con la utilización de nuestra metodología de modelado se podía realizar una selección de las parámetros ajustables de la rutina al ejecutarse, lo que permitía obtener tiempos de ejecución cercanos a los óptimos en plataformas de memoria distribuida homogéneas y heterogéneas. En este capítulo la aplicación de nuestra metodología de modelado y optimización se amplia a rutinas de álgebra lineal paralelas multithread para plataformas multicore. Se incluirá en el proceso de selección el código generado por los compiladores OpenMP disponibles en la plataforma multicore, de tal forma que cuando haya varios compiladores OpenMP disponibles se seleccione el código o partes del código multithread que conduzca a mejores tiempos de ejecución de la rutina. Por consiguiente, se considerará como nuevo parámetro del algoritmo (AP) la elección de la versión generada del código por cada compilador.

6.2 Generación de código multithread en sistemas multicore

La industria de la computación ha cambiado notablemente en los últimos años. De una tendencia en el diseño de procesadores basada en la ampliación exponencial de la frecuencia de reloj, se ha pasado a otra basada en los chips multiprocesador (CMP), también conocidos

como procesadores multicore [Gee05], con el propósito de buscar un compromiso entre rendimiento, eficiencia energética y fiabilidad [ABC+06]. La aparición de estos sistemas multicore ha permitido que las plataformas de computación paralelas sean más accesibles a la comunidad científica, lo que ha supuesto un auge en la popularidad de la programación paralela. El usuario de estos sistemas multicore desea obtener el máximo rendimiento, y a la vez reducir en lo posible el trabajo requerido para la recodificación del código secuencial ya existente, que además está validado en la resolución de sus problemas. En este tipo de sistemas es posible desarrollar un código paralelo a partir del secuencial utilizando el paradigma de memoria compartida, lo que en general resulta ser más sencillo que transformarlo en un código de paso de mensajes. Con el paradigma de memoria compartida se puede expresar un paralelismo en los datos con técnicas explícitas de creación de threads como Pthreads [But97], pero se trata de un proceso invasivo ya que la computación debe ser separada en funciones que se mapean a threads y el trabajo debe ser explícitamente dividido entre dichos threads. Una alternativa consiste en describir el paralelismo al compilador utilizando OpenMP [DM98], por medio de una sintaxis basada en directivas. En este caso, si el compilador no entiende las directivas OpenMP, éstas se ignoran y el código se compila sin errores. De tal forma que con OpenMP el trabajo de convertir un código secuencial en uno paralelo se puede abordar de forma incremental y es relativamente poco invasivo, lo que supone una ventaja sobre otros métodos de programación paralela para sistemas multicore.

En el mercado están disponibles diferentes compiladores OpenMP, ya sean comerciales o gratuitos, por lo que suele ser habitual disponer de más de un compilador OpenMP y diferentes versiones del mismo compilador, en un sistema multicore. Dada esta diversidad de compiladores, puede ocurrir que un compilador sea capaz de generar un ejecutable eficiente a partir del código secuencial, mientras que otro optimizará mejor el uso de múltiples threads. Además, se puede dar el caso de que dado un compilador (o una de sus versiones) y una plataforma multicore, el ejecutable generado se comporte de forma eficiente cuando el número de threads sea inferior al de cores disponibles pero, para esa misma plataforma y compilador, el rendimiento del ejecutable generado decrezca de forma considerable al incrementar el número de threads. Esta situación puede darse cuando un algoritmo requiere de un determinado número de threads, debido a una mejor distribución del trabajo, y se ejecuta en una plataforma con un número de cores inferior al de threads solicitados, o cuando una plataforma multicore debe ejecutar simultáneamente rutinas diferentes, y la cantidad de threads en ejecución supera a la de cores disponibles. Por consiguiente, no se puede afirmar de forma inequívoca qué compilador generará el ejecutable óptimo para una plataforma o código en particular y, como se comprobará experimentalmente, la mejor versión para cada código puede ser generada por distintos compiladores.

En el entorno de trabajo descrito, en el que se dispone de la posibilidad de generar diferentes versiones ejecutables de un mismo código (una por cada compilador disponible), puede ser conveniente disponer de un mecanismo que permita seleccionar el ejecutable más eficiente en función de las características del problema a resolver y de la plataforma de ejecución, aplicando un enfoque similar al que ya se viene utilizando con los polialgoritmos [JKP95, SB95, NT04] y

las polilibrerías [AAV+04] para obtener la solución más eficiente de problemas con un alto coste computacional. Como se mostró en los capítulos anteriores, con la metodología de modelado de rutinas de álgebra lineal se podía realizar una correcta selección de los parámetros ajustables de la rutina (AP), y ahora se propone considerar como un nuevo AP la versión compilada de dicha rutina, junto con otros AP como el número de threads, tamaños de bloque, etc. y, cuando haya que resolver un problema, seleccionar el ejecutable más efectivo, así como el resto de AP.

En la siguiente sección se describirá la metodología propuesta para la selección del ejecutable más efectivo, y se mostrará un mecanismo que permitirá comparar la efectividad de los compiladores instalados en una plataforma a la hora de generar un código multithread con OpenMP.

6.3 Propuesta de selección y modelado

La metodología propuesta consiste, en primer lugar, en realizar una comparación de las capacidades de los compiladores instalados en una plataforma. Para este primer objetivo, se propone utilizar un conjunto de rutinas de *benchmarking* (BR), que consisten básicamente en llamadas a directivas OpenMP. Posteriormente, para cada rutina **XR** multithread a ejecutar, se usarán un conjunto de datos apropiados (la información sobre los SP obtenida a partir de las BR, el modelo de tiempo de ejecución para la rutina **XR** y el tamaño del problema a resolver) con el fin de realizar una selección del valor de los AP (la versión compilada a utilizar, el número de threads a generar en cada situación...) con los que se obtienen tiempos de ejecución cercanos al óptimo.

En la figura 6.1 se muestra un esquema de nuestra propuesta para una rutina **XR** en una plataforma con varios compiladores ($Compilador_1$, $Compilador_2$,..., $Compilador_N$). Para cada compilador disponible en la plataforma, se obtendrá una versión de las BR (BR_1, BR_2,..., BR_N), y con su ejecución en la plataforma multicore se caracterizarán las capacidades de gestión y generación de threads de cada compilador, por medio de los correspondientes SP (SP_1, SP_2,..., SP_N), junto con otros SP que aparecen en rutinas de álgebra lineal, como el coste de ejecución de las rutinas básicas. El modelo de tiempo de ejecución (T_{XR}) de la rutina XR multithread como una función de los SP y AP, para un tamaño de problema N_R, permitirá realizar la selección de la mejor versión de XR (XR_1, XR_2,..., XR_N), el número de threads a generar, y otros AP de la rutina. A continuación se describirán las características de las rutinas de BR propuestas que permitirán obtener los SP relacionados con la generación y gestión de threads de cada compilador, y de los modelos de tiempo de ejecución de cada una de ellas.

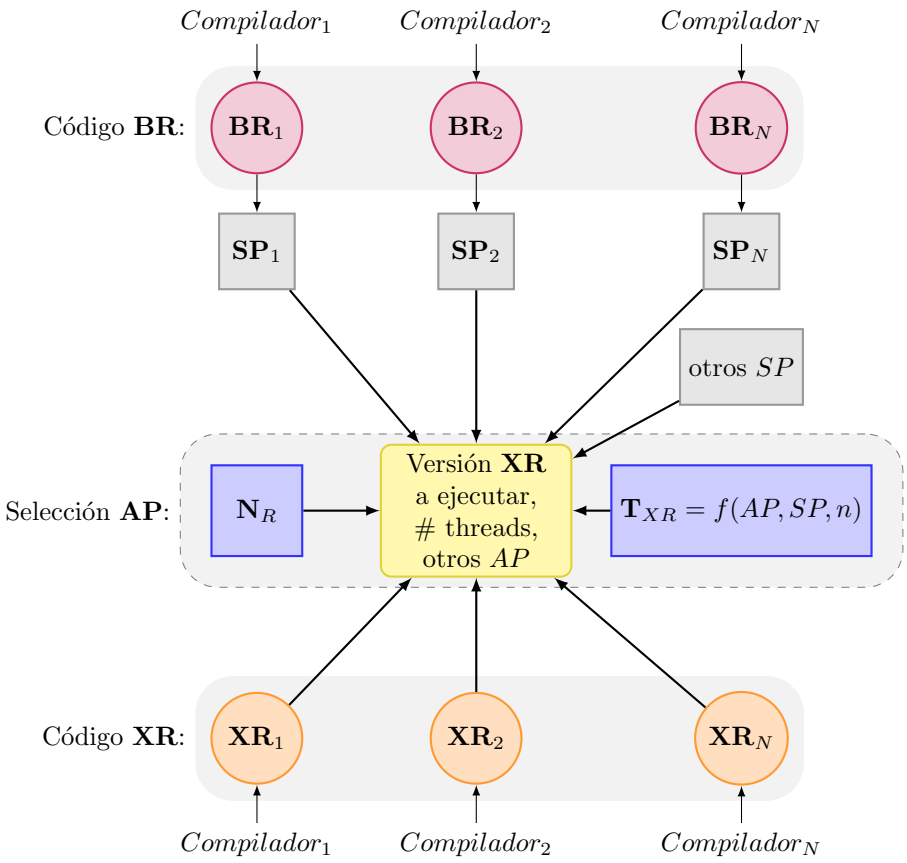


Figura 6.1: Esquema de la metodología propuesta para una rutina **XR** en una plataforma con varios compiladores.

6.3.1 Análisis empírico y parámetros OpenMP del sistema

En esta sección se describirán las rutinas básicas (BR) con las que se propone comparar las capacidades de los compiladores OpenMP en la creación y gestión de threads. Para cada rutina, se analizarán los resultados experimentales obtenidos con diferentes compiladores en varias plataformas multicore, y se propondrá un modelo de tiempo de ejecución con el que se pueda realizar una comparación del comportamiento de diferentes compiladores en la creación de threads, reparto homogéneo de trabajo entre threads, y barreras de sincronización de threads. Se han implementado un conjunto de rutinas propias con el fin de poder gestionar los algoritmos y por tanto el modelo de tiempo de ejecución correspondiente, en lugar de utilizar otras ya existentes, tales como el EPCC OpenMP constructs benchmark [BO01] o el SPEC OMP [SGJ+02].

Para el análisis empírico, se han seleccionado varios escenarios que consisten en nodos individuales de las plataformas descritas en la sección referente a las herramientas computacionales. Para denominar a cada nodo multicore se utilizará la siguiente nomenclatura: P2c de la plataforma Sagitario, A4c de la plataforma HPC160, X4c de la plataforma Sol y X8c de la plataforma Hipatia. En la tabla 6.1 se indican las características de cada nodo y los compiladores disponibles. Dado que en alguno de estos escenarios no había posibilidad de habilitar el soporte para hyperthreadring, todo el estudio se ha realizado sin tener en cuenta esta característica. En lo que ser refiere al nivel de optimización indicado al compilador, se ha utilizado el nivel `-O3`. Las rutinas que se van a analizar son:

- *R-generate*. Esta rutina consiste en crear una serie de threads con una cantidad fija de trabajo por thread. El propósito es comparar el tiempo de creación y gestión de threads.
- *R-pfor*. La rutina consiste en paralelizar un bucle `for` sencillo. En cada iteración se realiza una cantidad de trabajo significativo. El propósito es el de comparar el tiempo empleado en distribuir dinámicamente un conjunto de tareas homogéneas entre threads.
- *R-barriers*. Esta rutina consiste en imponer una primitiva de barrera, a continuación de una área de trabajo paralelo. Su objetivo es el de comparar el tiempo en realizar una sincronización global de todos los threads.

En la figura 6.1 puede observarse que las BR se utilizarán para obtener en cada plataforma un conjunto de SP por compilador disponible: SP_1 para el $Compilador_1$, SP_2 para el $Compilador_2$, etc. Este conjunto de SP aparecerá posteriormente en la formulación del modelo de tiempo de ejecución de una rutina de álgebra lineal (T_{XR}). La aproximación al tiempo de ejecución de la rutina de álgebra lineal obtenida con los SP correspondientes a cada compilador disponible permitirá seleccionar la versión del ejecutable más eficaz (XR_1, XR_2...) y el resto de AP que aparecen en una rutina de álgebra lineal.

Como ya se ha comentado, el propósito del modelo de tiempo de ejecución es el de utilizarlo para comparar el comportamiento de los ejecutables generados con cada compilador, y poder

Nodo	Características	Compiladores	
P2c	Intel Pentium 2.8 GHz, **2 cores** 1 MB L2, 2 GB RAM, Linux 2.6	icc 10.1	gcc 4.1.1
A4c	Alpha EV68CB, 1.0 GHz, **4 cores** 1.75 MB L2, 16 GB RAM, Tru64 Unix	cc 6.3	gcc 4.3
X4c	Intel Xeon 3 GHz, **4 cores** 2 MB L2, 4 GB RAM, Linux 2.6	icc 10.1	gcc 4.2.3
X8c	Intel Xeon 2 GHz, **8 cores** 4 MB L2, 32 GB RAM, Linux 2.6	icc 10.1	gcc 3.4.6

Tabla 6.1: Nodos multicore en los que se han realizado los experimentos.

predecir su tiempo de ejecución en diferentes situaciones. Por tal motivo su formulación será sencilla sin pretender describir en profundidad la arquitectura hardware de la plataforma o la implementación multithread de cada compilador.

A continuación, para cada una de las BR descritas, se mostrará el modelo de tiempo de ejecución y el conjunto de SP que aparecen en su formulación y que caracterizarán las capacidades de cómputo de la plataforma en la ejecución de rutinas OpenMP con cada compilador disponible, y que influyen en el tiempo de ejecución de las BR. Para cada BR se analizarán los resultados experimentales obtenidos en cada plataforma y con cada compilador. Este análisis servirá para justificar el modelo de tiempo de ejecución de cada rutina básica y los SP que aparecen en su formulación.

Rutina *R-generate*

En lo referente a la rutina *R-generate*, en la tabla 6.2 se puede apreciar, por un lado, que para la plataforma A4c, tanto con el ejecutable generado con el compilador `cc`, como con el generado con el compilador `gcc`, se obtienen en general tiempos de ejecución similares cuando el número de threads es inferior al de cores, pero cuando se incrementa el número de threads, el tiempo de ejecución de la versión generada con `cc` crece rápidamente, mientras que el de la versión obtenida con `gcc` mantiene su rendimiento (el tiempo de ejecución crece menos aunque haya más threads que gestionar). Este comportamiento se puede deber a que el coste para la gestión de threads se controla de forma más efectiva con `gcc`, lo que se hace más evidente cuando el tiempo disponible para utilizar cada core tiene que compartirse entre varios threads. Por otro lado, vemos que para las plataformas P2c, X4c y X8c el ejecutable obtenido con `gcc` resulta ser más rápido que el obtenido con `icc` para cualquier número de threads.

Como puede observarse en los experimentos realizados con la rutina *R-generate*, para conseguir el mejor rendimiento es necesario tener en cuenta el número de threads generados, con el fin de seleccionar la mejor versión del ejecutable en cada posible escenario.

Rutina			Número de threads					
			1	2	4	8	12	16
R-generate ($N = 100$)	P2c	icc	0.000207	0.002708	0.001692	0.124567	0.105436	0.145509
		gcc	0.000079	0.000197	0.000226	0.000362	0.000431	0.000566
	X4c	icc	0.000213	0.000591	0.109017	0.222876	0.240063	0.276859
		gcc	0.000091	0.001450	0.003949	0.007761	0.011122	0.014349
	A4c	cc	0.000977	0.001953	0.002930	0.035156	0.102537	0.101560
		gcc	0.000977	0.000977	0.001953	0.002930	0.002929	0.003906
	X8c	icc	0.000131	0.001547	0.003852	0.201918	0.203106	0.203174
		gcc	0.000004	0.000042	0.000073	0.000122	0.000174	0.000228

Tabla 6.2: Comparación del tiempo de ejecución (en segundos) para la rutina *R-generate*, en diferentes plataformas y compiladores.

Cuando el número de threads generado es menor o igual que el número de cores disponibles, el tiempo de ejecución puede ser modelado por:

$$T_{R-generate} = PT_{gen} + NT_{work} \tag{6.1}$$

donde P es el número de threads a generar, T_{gen} es el tiempo necesario para generar un thread, N es el tamaño del problema y T_{work} es el tiempo unitario de trabajo, esto es, el tiempo necesario para realizar una operación en una CPU (incluyendo el tiempo en cargar los operandos y almacenar el resultado). De tal forma que el primer término se corresponde con el tiempo en crear los P threads, y el segundo término representaría el tiempo que cada thread necesita para realizar su trabajo, teniendo en cuenta que todos los threads están trabajando en paralelo y realizando cada uno N operaciones.

El modelo para el tiempo de ejecución cuando el número de threads supera al de cores disponibles se obtiene añadiendo un nuevo término que se corresponde con el tiempo necesario para realizar un cambio de contexto entre threads que comparten el mismo core:

$$T_{R-generate} = PT_{gen} + NT_{work}\frac{P}{C} + \frac{P}{C}\frac{NT_{work}}{T_{cpu}}T_{sw} = PT_{gen} + NT_{work}\frac{P}{C}\left(1 + \frac{T_{sw}}{T_{cpu}}\right) \tag{6.2}$$

donde, C es el número de cores disponibles, $\frac{P}{C}$ se corresponde con el número de threads asignado por core, y T_{cpu} es el tiempo de CPU asignado a cada thread entre dos cambios de contexto consecutivos, de tal forma que $\frac{NT_{work}}{T_{cpu}}$ es el número total de cambios de contexto de cada thread durante su ejecución. Finalmente, T_{sw} es el tiempo necesario para realizar un cambio de contexto del thread en ejecución en un core. De tal forma que el tiempo total dedicado a un cambio de

contexto en cada core (en paralelo para todos los cores) sería el número de threads por core multiplicado por el número de cambios de contexto por thread y por el tiempo necesario en realizar cada cambio de contexto.

Rutina *R-pfor*

En la tabla 6.3 se muestran los resultados obtenidos cuando se realizan 100 ejecuciones (×100) de la rutina *R-pfor* en los diferentes escenarios indicados. En general, el comportamiento observado es similar al de la rutina *R-generate*: cuando el número de threads es inferior o igual a la cantidad de cores disponibles, los tiempos obtenidos con el ejecutable generado con el compilador `icc` (o `cc` en A4c) son mejores que los obtenidos con el ejecutable generado con `gcc`, o al menos presentan un comportamiento similar. Sin embargo, cuando el número de threads es superior al de cores disponibles, los tiempos de ejecución de la versión de la rutina obtenida con `gcc` son entre 3 y 50 veces mejores que los obtenidos con la versión generada con el otro compilador. Por consiguiente, en cada plataforma, cuando hay que realizar una cantidad de trabajo homogénea (el conjunto de iteraciones del bucle `for`), el número de threads de ejecución determina la elección de la versión compilada con mejores prestaciones.

Rutina			Número de threads					
			1	2	4	8	12	16
R-pfor (×100)	P2c	icc	0.298028	0.152193	0.349722	0.350801	0.352281	0.352905
		gcc	0.297786	0.155407	0.163423	0.191674	0.201023	0.179113
	X4c	icc	0.002067	0.001532	0.001394	0.254343	0.386841	0.336576
		gcc	0.006952	0.005278	0.004675	0.009392	0.012408	0.016042
	A4c	cc	0.002930	0.001953	0.002930	3.324133	9.936266	13.441057
		gcc	0.032225	0.048827	0.062499	0.099607	0.142574	0.181636
	X8c	icc	0.000078	0.001904	0.003562	0.336080	0.337658	0.581781
		gcc	0.006568	0.005858	0.005978	0.008167	0.010539	0.012126

Tabla 6.3: Comparación del tiempo de ejecución (en segundos) para la rutina *R-pfor*, en diferentes plataformas y compiladores.

Cuando el número de threads generado es menor o igual que el número de cores disponibles, el tiempo de ejecución puede ser modelado por:

$$T_{R-pfor} = PT_{gen} + \frac{N_t}{P}T_{work} \tag{6.3}$$

donde P, al igual que en el modelo para la rutina anterior, es el número de threads generado, T_{gen} es el tiempo en generar un thread, N_t es el tamaño total del problema y T_{work} es el tiempo de CPU en computar una operación elemental. Por consiguiente, el primer término se corresponde con el tiempo en generar los P threads, y el segundo término se corresponde con

el tiempo necesario para realizar el trabajo en cada thread paralelo.

Cuando el número de threads es superior al de cores, el modelo de tiempo de ejecución sería:

$$T_{R-pfor} = PT_{gen} + \frac{N_t}{P} T_{work} \left(1 + \frac{PT_{sw}}{CT_{cpu}}\right) \tag{6.4}$$

donde los parámetros C, T_{cpu} y T_{sw} tienen el mismo significado que en el modelo para la rutina *R-generate* (ecuación 6.2).

Rutina *R-barriers*

Como se ha comentado, la rutina *R-barriers* consiste en imponer una primitiva de barrera a continuación de una área de trabajo paralelo. En la tabla 6.4 se recogen los resultados obtenidos para 100 ejecuciones de esta rutina (×100). Puede observarse que en la plataforma P2c no hay gran diferencia entre ambas versiones de la rutina básica. Sin embargo, en la plataforma X4c los tiempos de ejecución obtenidos con la versión generada por el compilador `icc` son unas 6 veces mejores para un elevado número de threads, mientras que para las otras dos plataformas el comportamiento es diferente. En A4c, cuando el número de threads no es mayor que el número de cores, la versión obtenida con el compilador `cc` es aproximadamente unas 2 veces mejor, mientras que cuando el número de threads es mayor, el rendimiento con la versión `cc` cae bruscamente comparado con el de la versión `gcc`. Finalmente, en la plataforma X8c, la versión `gcc` es mejor que la `icc` en todos los casos, y la diferencia entre ambas versiones aumenta con el número de threads a sincronizar.

Rutina			Número de threads					
			1	2	4	8	12	16
R-barriers (×100)	P2c	icc	0.00833	0.06254	0.33302	0.44293	0.57320	0.70641
		gcc	0.01923	0.03831	0.14083	0.27282	0.38638	0.52142
	X4c	icc	0.00723	0.07524	0.2257	0.81294	0.96502	1.33739
		gcc	0.02449	0.73481	2.2563	4.48060	6.02206	8.36155
	A4c	cc	0.01855	0.04883	0.1201	3.85537	12.08465	14.85411
		gcc	0.03027	0.09110	0.1807	0.38280	0.58201	0.78807
	X8c	icc	0.08434	19.64639	31.9543	69.49994	103.66320	133.70356
		gcc	0.02367	1.14024	2.0208	4.02098	6.106732	8.31250

Tabla 6.4: Comparación del tiempo de ejecución (en segundos) para la rutina *R-barriers*, en diferentes plataformas y compiladores.

El modelo de tiempo de ejecución para la rutina *R-barriers*, cuando el número de threads es menor o igual que el número de cores disponible, sería:

$$T_{R-barriers} = PT_{gen} + NT_{work} + PT_{syn} \tag{6.5}$$

donde T_{syn} es el tiempo de sincronización por thread.

Cuando el número de threads es superior al de cores, el modelo de tiempo de ejecución sería:

$$T_{R-barriers} = PT_{gen} + NT_{work}\left(1 + \frac{PT_{sw}}{CT_{cpu}}\right) + PT_{syn} \tag{6.6}$$

Como puede observarse en la ecuación anterior, el modelo para esta rutina es el mismo que el de la rutina *R-generate* (ecuación 6.2), al que se le ha añadido el último término, que corresponde con el tiempo de sincronización.

Para finalizar esta sección podemos concluir, a partir de los resultados experimentales y de los modelos de ejecución propuestos, que el comportamiento de las primitivas básicas OpenMP analizadas para la creación y gestión de los threads es bastante diferente en función de la plataforma y el compilador. Por esta razón, un ajuste manual por parte del usuario se convierte en una labor compleja, mientras que una metodología que permita tomar las decisiones apropiadas (qué versión del código compilada ejecutar y el número de threads a generar) en cada situación, facilitaría la obtención de tiempos de ejecución cercanos al óptimo.

En el siguiente apartado se mostrarán algunos resultados experimentales sobre la gestión de este nuevo parámetro algorítmico (versión compilada de la rutina), con diferentes rutinas de álgebra lineal.

6.4 Proceso de ajuste de rutinas de álgebra lineal

En esta sección se mostrará la aplicación de nuestra propuesta de ajuste de rutinas de álgebra lineal en plataformas multicore. A partir de las BR descritas en la sección anterior, se realizará una estimación de los SP (T_{gen}, T_{work} y $\frac{T_{sw}}{T_{cpu}}$), y, por medio del modelo de tiempo de ejecución de la rutina de álgebra lineal (T_{XR}), se obtendrá una imagen aproximada de su comportamiento para diferentes tamaños de problema y número de threads (figura 6.1). De tal forma que, para un tamaño de problema específico, se podrá decidir el valor más apropiado para los AP (versión compilada y número de threads) con el fin de reducir su tiempo de ejecución.

Las rutinas de álgebra lineal a las que se les va a aplicar la metodología de ajuste que se propone son: una rutina que implementa el método de relajación de Jacobi [GL96, AGMV08] y una rutina de Strassen para la multiplicación de matrices. Las plataformas y compiladores para comprobar la validez de la propuesta son los reflejados en la tabla 6.1.

6.4.1 Rutina R-jacobi

La rutina R-jacobi implementa el método de relajación de Jacobi para una malla 2D de $n \times n$ puntos. Con el propósito de obtener tiempos que puedan ser comparables, consiste únicamente de dos iteraciones consecutivas. En la figura 6.2 se muestran los tiempos de ejecución, variando el número de threads, obtenidos para un tamaño de problema de $n = 1000$. Como se puede apreciar, cuando el número de threads es inferior o igual al número de cores disponibles, el ejecutable obtenido con ambos compiladores tiene un comportamiento similar en las plataformas P2c, X4c y X8c. En estas plataformas, cuando el número de threads aumenta, el rendimiento con la versión `gcc` del ejecutable se mantiene, mientras que el rendimiento con la versión del ejecutable obtenida con el otro compilador decrece. Sin embargo, en la plataforma A4c el rendimiento con la versión `gcc` es inferior al de la otra versión hasta los 4 threads (número de cores disponibles), siguiendo un comportamiento similar al de las otras plataformas cuando el número de threads de ejecución es mayor que el de cores disponibles.

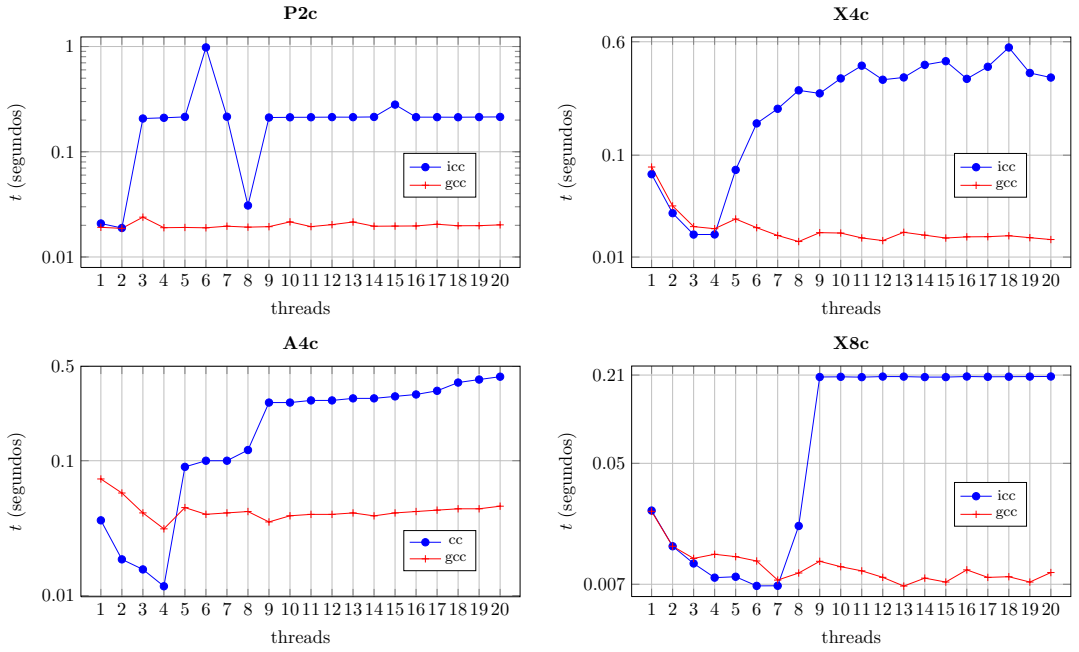


Figura 6.2: Tiempo experimental (en escala logarítmica) para la ejecución de la rutina R-jacobi, en diferentes plataformas y compiladores, para tamaño de problema $n = 1000$.

Como ha podido observarse con estos experimentos, la selección del mejor ejecutable dependerá del número de cores disponibles, de su arquitectura y del tamaño del problema.

El modelo de tiempo de ejecución para esta rutina, cuando el número de threads es menor o igual que el de cores disponibles, sería:

$$T_{R-jacobi} = PT_{gen} + \frac{11n^2}{P}T_{work} \tag{6.7}$$

y cuando el número de threads es superior al de cores, el modelo de tiempo de ejecución sería:

$$T_{R-jacobi} = PT_{gen} + \frac{11n^2}{P}T_{work}\left(1 + \frac{PT_{sw}}{CT_{cpu}}\right) \tag{6.8}$$

donde aparecen tres SP (T_{gen}, T_{work}, $\frac{T_{sw}}{T_{cpu}}$), correspondientes al coste de creación y generación de threads en OpenMP, y un AP (P, el número de threads paralelos).

A continuación se mostrará, para la rutina R-jacobi vista, el proceso completo de selección de los AP.

Medición de los SP

En la tabla 6.5, se muestra el valor obtenido para los SP en cada plataforma y con cada compilador. El valor de T_{gen} puede calcularse a partir del tiempo de ejecución de las rutinas *R-generate* y *R-pfor*, para un tamaño de problema de $N = 1$, de tal forma que el segundo término de las ecuaciones 6.1 y 6.3 es despreciable. Los valores de T_{work} se pueden obtener ejecutando un rutina secuencial sencilla que contenga operaciones básicas de CPU. Los valores mostrados en la tabla corresponden a una operación en doble precisión de números en coma flotante: suma o multiplicación. Finalmente, el cociente $\frac{T_{sw}}{T_{cpu}}$, que puede considerarse constante en una primera aproximación, puede calcularse a partir de los tiempos de ejecución de las rutinas *R-generate* y *R-pfor* para el caso de que el número de threads sea mayor que el de cores, utilizando las ecuaciones 6.2 y 6.4.

	P2c		X4c		A4c		X8c	
	icc	gcc	icc	gcc	cc	gcc	icc	gcc
T_{gen} (μs)	75	25	75	25	75	25	75	25
T_{work} (ns)	2	2	4	7	3	10	1.5	1.5
$\frac{T_{sw}}{T_{cpu}}$	2	1.5	15	0.8	15	1.8	1	0.4

Tabla 6.5: Parámetros del sistema (SP) para diferentes plataformas y compiladores.

Modelado del tiempo de ejecución

En la figura 6.3 se muestra el tiempo de ejecución estimado por los modelos para la rutina *R-jacobi* (ecuaciones 6.7 y 6.8) con los SP estimados con el procedimiento indicado y reflejados en la tabla 6.5, para un problema de tamaño $n = 1000$.

Comparando la forma de las gráficas para el tiempo de ejecución obtenido experimentalmente (figura 6.2) con las obtenidas a partir de los modelos de tiempo de ejecución (figura 6.3), se puede apreciar que tienen una forma similar, por lo que la metodología propuesta obtiene una imagen aproximada del comportamiento de la rutina. Como veremos a continuación, esta imagen permitirá realizar una selección de los AP, antes de ejecutar la rutina, obteniendo así tiempos de ejecución cercanos al óptimo.

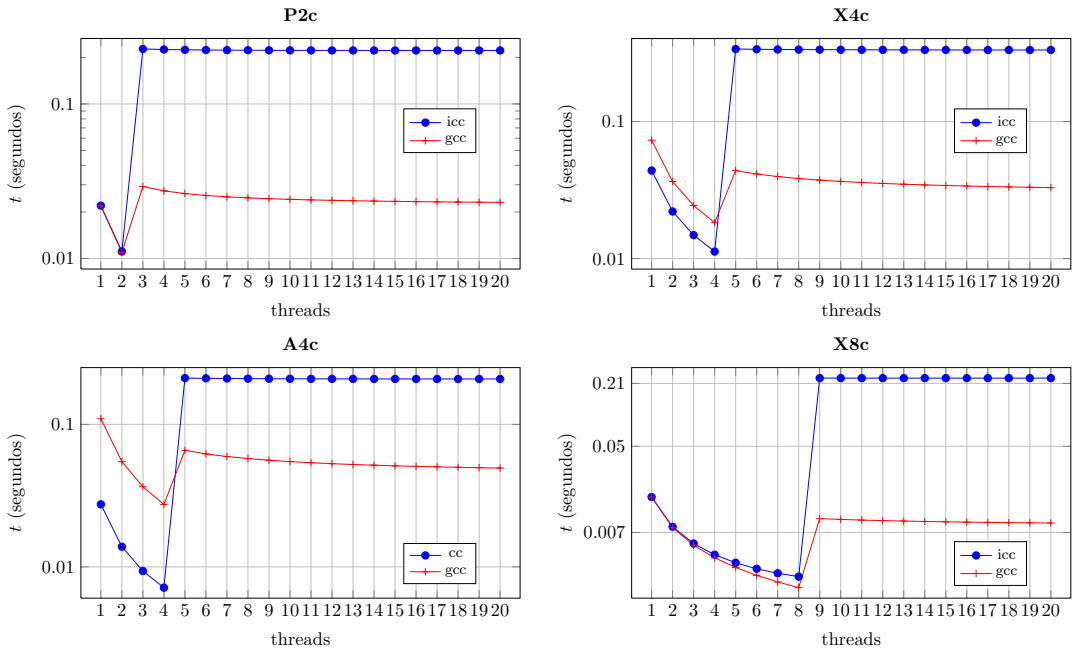


Figura 6.3: Tiempo teórico (en escala logarítmica) proporcionado por el modelo de tiempo de ejecución de la rutina R-jacobi, en diferentes plataformas y compiladores, para tamaño de problema $n = 1000$.

Selección de los valores de los AP

Con la utilización de los modelos teóricos de tiempo de ejecución propuestos se pueden tomar decisiones previas a la ejecución de la rutina. Por ejemplo, si se fija de antemano el AP número de threads paralelos se puede seleccionar qué versión compilada de entre las disponibles debe ejecutarse. En el caso más general se decidirá el valor de ambos AP: la versión compilada y el número de threads paralelos. En la tabla 6.6 se muestran los valores de los AP con los que el tiempo de ejecución aproximado por el modelo es menor. Como se puede apreciar, la versión seleccionada de la rutina y el número de threads depende del sistema en el que dicha rutina se va a ejecutar.

	P2c	X4c	A4c	X8c
versión compilada	gcc	icc	cc	gcc
número de threads	2	4	4	8

Tabla 6.6: Valores de los parámetros ajustables (AP) en diferentes plataformas y con diferentes compiladores, para la rutina de R-jacobi ($n = 1000$).

Discusión de los resultados

En este apartado se efectuará una comparación entre los resultados conocidos a posteriori, al ejecutar la rutina de R-jacobi con los posibles valores para los AP (versión compilada y número de threads), con los resultados del estudio teórico-experimental realizado. En la tabla 6.7 se compara el tiempo de ejecución óptimo, conocido a posteriori (Óptimo), el tiempo de ejecución que se obtendría con la selección de los AP realizada por el procedimiento indicado (Modelo) y el tiempo de ejecución resultante de una selección de los AP realizada por un usuario experto (Experto) con información sobre la plataforma, y que seleccionará el compilador de la plataforma y un número de threads igual al de cores disponibles.

	P2c	X4c	A4c	X8c
Óptimo	0.0187	0.0256	0.0117	0.0068
Modelo	0.0187	0.0286	0.0117	0.0084
Experto	0.0189	0.0286	0.0117	0.0181

Tabla 6.7: Comparación para la rutina R-jacobi de los tiempos de ejecución óptimo experimental, el obtenido con los AP seleccionados con el modelo, y los obtenidos con la selección de los AP realizada por un usuario experto, con un tamaño de problema de $n = 1000$. Tiempo en segundos.

Como puede observarse a partir de los resultados obtenidos, la metodología propuesta permite realizar una selección adecuada de los AP. En el caso de los sistemas X4c y X8c, el procedimiento no predice el óptimo. En el sistema X4c el óptimo experimental se obtiene para valores de los AP que corresponden con la utilización del compilador `gcc`, y número de threads paralelos de 8, mientras que en el sistema X8c el óptimo experimental se obtiene con la versión compilada `gcc`, y número de threads paralelos de 13. La desviación entre el óptimo experimental y el óptimo del modelo es de un 12 % en X4c y de un 24 % en X8c. La selección realizada por el usuario experto sería en X4c de un 12 % (coincide con la selección realizada por el modelo) y mucho mayor (165 %) en X8c. Estos resultados confirman la importancia de disponer de información experimental sobre el comportamiento de los compiladores en diferentes situaciones, junto con la información teórica proporcionada por los modelos de tiempo de ejecución. En cualquier caso, la utilidad de la metodología que proponemos se pondrá de manifiesto en rutinas en las que haya que decidir un mayor número de AP. Esta situación se mostrará a continuación con el algoritmo de Strassen para la multiplicación de matrices.

6.4.2 Rutina de multiplicación de matrices de Strassen

La descripción detalla del algoritmo de Strassen para la multiplicación de matrices se mostró en la sección 3.3.6. Como ya se comentó en dicha sección, se había implementado una versión secuencial utilizando rutinas básicas de BLAS (**dgemm** y **daxpy**). Partiendo de la versión secuencial, se ha escrito una versión paralela con dos niveles de recursión, utilizando primitivas OpenMP. El modelo de tiempo de ejecución para esta nueva rutina OpenMP de Strassen, cuando el número de threads es menor o igual al número de cores disponibles y con niveles de recursión del algoritmo de Strassen de uno y dos, sería:

$$T_{R-strassen} = PT_{gen} + 7\frac{2(\frac{n}{2})^3}{P}T_{mul} + \frac{9n^2}{4}T_{add} \tag{6.9}$$

$$T_{R-strassen} = P_1P_2T_{gen} + 49\frac{2(\frac{n}{2^2})^3}{P_2P_1}T_{mul} + 7\frac{\frac{18n^2}{4^2}}{P_1}T_{add} + \frac{9n^2}{2}T_{add} \tag{6.10}$$

y cuando el número de threads es superior al número de cores disponibles, el modelo de tiempo de ejecución sería:

$$T_{R-strassen} = PT_{gen} + 7\frac{2(\frac{n}{2})^3}{C}T_{mult}\left(1 + \frac{T_{sw}}{T_{cpu}}\right) + \frac{9n^2}{4}T_{add} \tag{6.11}$$

$$T_{R-strassen} = P_1P_2T_{gen} + 49\frac{2(\frac{n}{2^2})^3}{C}T_{mul}\left(1 + \frac{T_{sw}}{T_{cpu}}\right) + 7\frac{\frac{18n^2}{4^2}}{\text{mín}(P_1, C)}T_{add}\left(1 + \frac{T_{sw}}{T_{cpu}}\right) + \frac{9n^2}{2}T_{add} \tag{6.12}$$

donde aparecen cuatro SP (T_{gen}, $\frac{T_{sw}}{T_{cpu}}$, T_{mul}, T_{add}). T_{gen} y $\frac{T_{sw}}{T_{cpu}}$ corresponden con el coste de la creación y gestión de threads, T_{mul} y T_{add} se corresponden con el tiempo unitario (coste de las operaciones básicas de BLAS utilizadas) de trabajo para una multiplicación de matrices y para una adición de matrices respectivamente. C es el número de cores disponibles en la plataforma. En lo que se refiere a los AP, aparecen P, P_1 y P_2. P es el número de threads cuando el nivel de recursión del algoritmo de Strassen es uno, y P_1 y P_2 son el número de threads para el primer y segundo nivel de recursión del algoritmo de Strassen, cuando la rutina utilice dos niveles de recursión y paralelismo anidado.

En la figura 6.4 (izquierda) se muestra el tiempo de ejecución obtenido para la rutina R-strassen en los nodos multicore de la tabla 6.1, para un tamaño de problema de $n = 1000$. La ejecución de la rutina se ha realizado con un número de threads diferente para el primer y el segundo nivel de recursión del algoritmo, siendo P_1 el número de threads correspondiente al primer nivel y P_2 el número de threads correspondiente al segundo. Las versiones del ejecutable generadas con cada compilador tienen un comportamiento similar cuando el número de threads

es inferior o igual al número de cores disponibles en cada nodo. Se observan diferencias cuando el número de threads crece por encima del número de cores disponibles; en esta situación el tiempo de ejecución de la versión `gcc` tiende a ser mejor que el de la otra versión. Es importante resaltar que el número óptimo de threads en cada nivel de recursión cambia de una plataforma a otra, por lo que un ajuste manual de los valores de P_1 y P_2 se convierte en una labor compleja sin la ayuda de una metodología como la propuesta.

A continuación se mostrará, a modo de ejemplo, el proceso completo de selección de los valores de los AP para la rutina R-strassen.

Medición de los SP

En la tabla 6.8 se muestran los valores medios obtenidos para los SP. De igual forma que se hizo para la rutina R-jacobi, el valor de T_{gen} en cada plataforma y para cada compilador se puede obtener con la rutinas *R-generate* y *R-pfor*, para un tamaño de problema de $N = 1$. El tiempo requerido en generar los threads es similar en las cuatro plataformas, y depende del compilador utilizado. Por consiguiente, los threads generados a partir del código OpenMP tienen un comportamiento que varía en función del compilador.

Plataforma		T_{gen} (μs)	$\frac{T_{sw}}{T_{cpu}}$	T_{add} (ns)	T_{mul} (ps)
P2c	icc	75	$2 + 0.01P$	$20 + 0.05P$	$400 + 100P$
	gcc	25	$7 - 0.01P$	20	$400 + 0.1P$
X4c	icc	75	$0.9 + 0.3P$	$23 + 0.3P$	$140 + 10P$
	gcc	25	$0.9 + 0.01P$	$30 - 0.3P$	$140 - P$
A4c	cc	75	$0.8 + 0.2P$	$40 + P$	60
	gcc	25	$0.8 + 0.02P$	$40 - 0.1P$	$60 - 0.5P$
X8c	icc	75	$6 + 0.05P$	10	100
	gcc	25	$0.5 + 0.01P$	10	100

Tabla 6.8: Valores de los parámetros del sistema (SP) en diferentes plataformas y con diferentes compiladores.

En la tercera columna de la tabla 6.8 se muestran los valores para el cociente $\frac{T_{sw}}{T_{cpu}}$, que corresponde con el porcentaje de incremento del tiempo de ejecución debido al coste que introduce el cambio de contexto de los threads en ejecución en cada core. En una primera aproximación se había considerado su valor constante para cada compilador y plataforma y su valor se obtenía a partir de las BR vistas con anterioridad. Sin embargo, se ha comprobado experimentalmente que su valor depende del número de threads, por lo que resulta conveniente aproximarlo por una función lineal en P con coeficientes que variarán con cada plataforma y compilador. Por tanto, así se dispone de una nueva BR que permite calcular dichos coeficientes. El procedimiento seguido para calcular el valor de estos coeficientes, consiste en ejecutar una rutina básica

de álgebra lineal (una adición de vectores) con una cantidad de trabajo fijo por thread. La rutina se ejecuta variando el número de threads desde 2 hasta 64, y se obtiene el porcentaje de incremento producido entre el tiempo de ejecución experimental y el tiempo de ejecución teórico de la rutina. Con el compilador `gcc` los valores de $\frac{T_{sw}}{T_{cpu}}$ en todas las plataformas crecen lentamente e incluso disminuyen un poco. Por el contrario, con el otro compilador se observa un crecimiento evidente, lo que nos indica que la gestión de los threads por core resulta ser más complicada.

Los valores del parámetro T_{add} se muestran en la cuarta columna de la tabla 6.8. Su valor se corresponde con el coste unitario para una operación aritmética realizada en la adición de dos matrices de números reales en precisión doble. Para calcular su valor se ha utilizado una adición de matrices y se ha ido incrementando el número de threads para un tamaño de problema fijo.

Por último, los valores de T_{mul} se muestran en la quinta columna de la tabla 6.8. Este parámetro indica el coste unitario para una operación aritmética realizada en la multiplicación de dos matrices de números reales en precisión doble. De forma similar a lo realizado para el parámetro anterior, su valor se ha calculado utilizado una multiplicación de matrices y se ha ido incrementando el número de threads para un tamaño de problema fijo.

En lo referente a estos dos parámetros aritméticos (T_{add} y T_{mul}), se puede resaltar que el tiempo en realizar una operación básica dentro de una multiplicación de matrices es inferior al que se necesita dentro de una adición de matrices. Esto es debido a que el valor para cada uno de estos parámetros incluye, además del tiempo de CPU necesario para realizar la operación aritmética, el necesario para manejar los operandos y los resultados, lo que conlleva su movimiento entre los diferentes niveles de memoria. La multiplicación de matrices de BLAS está implementada por bloques y por consiguiente reutiliza más datos de los niveles superiores de la jerarquía de memorias (al realizar $O(n^3)$ operaciones en $O(n^2)$ datos) que la adición de matrices, que realiza $O(n^2)$ operaciones en $O(n^2)$ datos.

Cabe destacar que se han obtenido valores variables para el parámetro T_{add} con el número de threads para la versión `gcc`, de tal forma que para un tamaño de problema fijo, si se incrementa el número de threads, el valor de este parámetro disminuye, o no se incrementa como era de esperar, atendiendo a posibles colisiones producidas entre threads ejecutándose en un mismo core. Este comportamiento indica un uso más eficiente de los recursos CPU (las diferentes unidades funcionales) por parte de los threads ejecutando operaciones vectoriales como la adición de matrices. Por otra parte, el valor de T_{mul} se ha mostrado más constante, aunque se observa, al igual que con T_{add}, cierta tendencia a disminuir en la versión `gcc`.

Modelado del tiempo de ejecución

El siguiente paso consiste en sustituir el valor obtenido para los SP en el modelo teórico de tiempo de ejecución de la rutina, de tal forma que se obtiene una aproximación a su tiempo

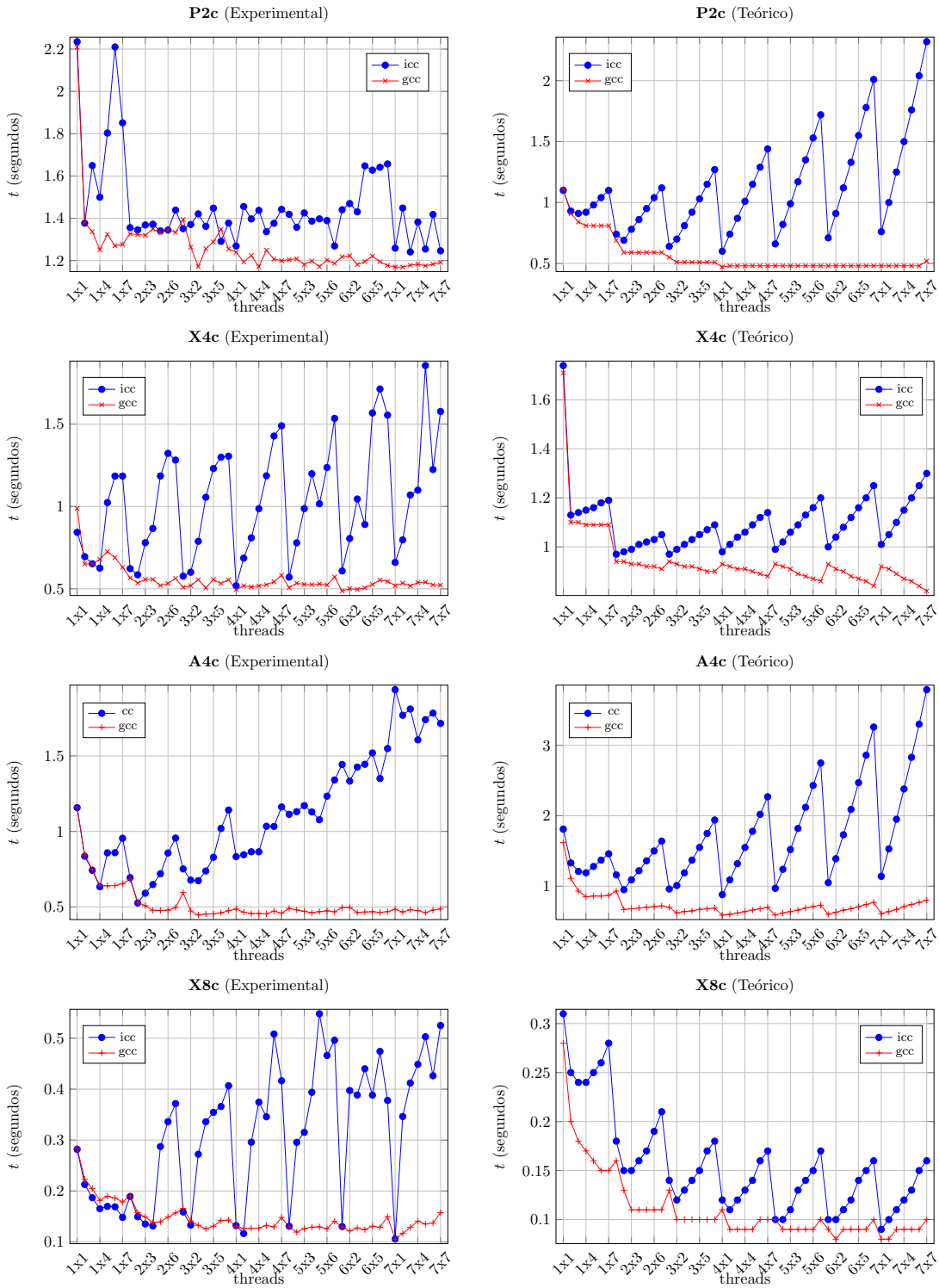


Figura 6.4: Tiempo experimental (izquierda) y teórico (derecha) de la rutina R-strassen, en diferentes plataformas y compiladores, para tamaño de problema $n = 1000$, con $P_1 \times P_2$ threads.

de ejecución para el tamaño del problema a resolver, antes de que ésta se ejecute. En la figura 6.4 (derecha) se muestran los tiempos de ejecución aproximados por el modelo de la rutina R-strassen para un tamaño de problema $n = 1000$, tomando los valores de los SP calculados en el paso anterior (tabla 6.8).

Si se comparan las gráficas mostradas en la figura 6.4 para el tiempo experimental (izquierda) y el teórico (derecha), se aprecia que el modelo de tiempo de ejecución propuesto no predice exactamente el comportamiento de la rutina. En cualquier caso, el propósito de la metodología de modelado propuesta es el de conocer a priori la evolución que seguiría el tiempo de ejecución de la rutina cuando el valor de los AP cambia, de tal forma que sea posible situar, dentro del espacio de posibles valores para los parámetros configurables, dónde se encontraría el menor tiempo de ejecución de la rutina.

Como puede observarse, tanto en el tiempo de ejecución experimental como en el aproximado por el modelo, las versiones de la rutina obtenidas con cada compilador muestran un comportamiento similar cuando el número de threads es inferior al de cores. Sin embargo, conforme el número de threads aumenta el comportamiento difiere. El tiempo de ejecución de la versión `gcc` de la rutina, muestra una tendencia a disminuir, debido a que los valores para los SP aritméticos T_{add} y T_{mul} también lo hacen; este comportamiento continúa hasta que se alcanza un valor mínimo, a partir del cual la tendencia cambia y el tiempo de ejecución se mantiene prácticamente constante o crece ligeramente. En lo que se refiere al tiempo de ejecución de la versión obtenida con los otros compiladores, se observa que su valor se incrementa con el número de threads generado, debido al comportamiento que presentan los SP aritméticos para estos compiladores (tabla 6.8).

Selección de los valores de los AP

Atendiendo a la información proporcionada por el modelo que aproxima el tiempo de ejecución de la rutina, se puede conocer a priori cuales serían los valores de los AP que proporcionarían tiempos de ejecución cercanos al óptimo. En la tabla 6.9 se muestran los valores de los AP para los que el modelo indica el menor tiempo de ejecución para la rutina R-strassen con un tamaño de problema de $n = 1000$.

	P2c	X4c	A4c	X8c
versión compilada	gcc	gcc	gcc	gcc
número de threads en el 1er nivel (P_1)	7	4	4	7
número de threads en el 2do nivel (P_2)	7	1	1	2

Tabla 6.9: Valores de los parámetros ajustables (AP) en diferentes plataformas y con diferentes compiladores, para la rutina de R-strassen ($n = 1000$).

La versión seleccionada de la rutina sería la `gcc` en las cuatro plataformas analizadas. Sin

embargo, el número de threads por nivel de recursión (P_1 y P_2) depende del sistema en el que se vaya a ejecutar la rutina, de tal forma que de nuevo nos encontramos ante un ejemplo de la dificultad de fijar un valor de los AP para cualquier plataforma, lo que refuerza la necesidad de utilizar una metodología como la que proponemos.

Discusión de los resultados

En este apartado se efectúa una análisis comparativo entre los resultados obtenidos en el tiempo de ejecución real de la rutina R-strassen siguiendo la metodología propuesta, y tres posibles casos:

- **OP**: que correspondería con el tiempo de ejecución óptimo, obtenido a posteriori, tras ejecutar la rutina para todos los posibles valores de los AP y tomando el menor.
- **HW**: que correspondería con el tiempo obtenido al utilizar los AP seleccionados por un posible usuario con conocimientos sobre las características de la plataforma. Este usuario seleccionaría el compilador específico de la plataforma, y un número de threads siempre igual al de cores disponibles.
- **SW**: que correspondería con el tiempo obtenido al utilizar los AP seleccionados por un posible usuario experto en las características del algoritmo. Este usuario seleccionaría siempre siete threads para cada nivel de recursión, con el fin de distribuir homogéneamente el trabajo entre todos los threads.

En lo que se refiere al caso **HW**, se mostrará el tiempo medio de las ejecuciones con las posibles combinaciones del número de threads (en una plataforma con 4 cores el número de threads considerado sería igual a 1×4, 2×2 y 4×1). Respecto al caso **SW**, dado que podría seleccionar cualquiera de los dos compiladores disponibles para la generación de su código, se mostrará el valor medio entre ambos.

Plataforma	OP	MOD	HW	SW	dev MOD (%)	dev HW (%)	dev SW (%)
P2c	1.17	1.19	1.37	1.22	2	17	4
X4c	0.49	0.50	0.55	1.31	2	12	167
A4c	0.45	0.49	0.65	1.20	9	44	167
X8c	0.11	0.16	0.12	0.32	45	9	191

Tabla 6.10: Tiempos de ejecución (en segudos) de la rutina R-strassen para un tamaño de problema de $n = 1000$, con diferentes combinaciones de los AP.

En la tabla 6.10 se comparan el tiempo real de ejecución de la rutina obtenido siguiendo la metodología propuesta (**MOD**), con el obtenido en los tres posibles casos considerados. Para

cada plataforma aparece la desviación entre el tiempo óptimo y el que se obtiene en cada caso. Como puede observarse, el conocimiento de la plataforma resulta ser más efectivo que el conocimiento sobre el algoritmo tal y como se ha definido. Por todo ello, la propuesta que se plantea en el presente trabajo, permite gestionar conjuntamente información tanto del algoritmo (modelo de tiempo de ejecución) como de las características de la plataforma (valores de los SP), y realiza una selección de los AP con la cual los tiempos de ejecución son cercanos al óptimo. Las desviaciones entre **MOD** y **OP** son mínimas, con un valor de la media de las desviaciones para las cuatro plataformas analizadas de un 14 %.

6.5 Resumen y conclusiones

En este capítulo se ha analizado una propuesta de modelado y optimización de rutinas de álgebra lineal paralelas multithread para plataformas multicore. Se ha comprobado que la selección de la versión generada por los diferentes compiladores disponibles en un sistema multicore contribuye a la aceleración de la solución de este tipo de rutinas y que su selección depende de factores como la rutina, el número de threads paralelos y el tamaño del problema a resolver. De esta manera, la selección de la versión compilada de una rutina se trata como otro nuevo parámetro algorítmico.

Para implementar el modelo de tiempo de ejecución de una rutina multithread ha sido necesario introducir información sobre las capacidades del compilador en la gestión y generación de threads, por lo que se han propuesto una serie de parámetros del sistema. Estos parámetros se obtienen a partir de un conjunto de rutinas básicas (BR), que consisten en primitivas y directivas OpenMP. De tal forma que, con la ejecución de las BR se obtiene experimentalmente el valor de los parámetros del sistema asociados a la creación y gestión de threads simples y coste de operaciones aritméticas para cada pareja plataforma-compilador. Finalmente, el modelo teórico que aproxima el tiempo de ejecución de la rutina permite comparar el comportamiento de la rutina para cada pareja plataforma-compilador y decidir en cada situación la mejor combinación para los parámetros algorítmicos (versión compilada de la rutina y número de threads).

La propuesta se ha aplicado a dos rutinas de álgebra lineal ejecutándose sobre plataformas multicore de diferente naturaleza y se han llevado a cabo comparaciones entre el tiempo aproximado por el modelo y el tiempo real de ejecución de cada rutina con diferentes compiladores y valores de los parámetros algorítmicos. Los tiempos de ejecución reales obtenidos atendiendo a la información proporcionada por nuestra metodología han sido muy cercanos a los óptimos, confirmándose así la importancia de disponer de información experimental sobre el comportamiento de los compiladores en distintas situaciones, junto con la proporcionada por los modelos de tiempo de ejecución.

Capítulo 7

Conclusiones y trabajo futuro

7.1 Conclusiones

La adaptación del software a las características de las diferentes plataformas de computación paralela disponibles, con el fin de aprovechar los recursos computacionales que ofrecen estas plataformas, supone un reto para los usuarios y para los desarrolladores. En esta tesis se propone la utilización de modelos del tiempo de ejecución como herramienta principal para el desarrollo de técnicas de optimización automática y ajuste del código modelado, de tal forma que se consiga minimizar su tiempo de ejecución.

En primer lugar, se ha mostrado la construcción de modelos parametrizados que aproximan el tiempo de ejecución basados en un estudio teórico de la rutina de álgebra lineal a modelar y en el coste asociado con el software que proporciona las operaciones básicas de computación y comunicación que utilizará la rutina. Estos costes de las operaciones básicas aparecen en el modelo en forma de parámetros del sistema, y su determinación se realiza experimentalmente en cada plataforma de cómputo y para cada software básico disponible. Con el propósito de que el modelo de la rutina refleje adecuadamente su comportamiento, se propone una formulación detallada para el modelo, que permita tener en cuenta las variaciones que aparecen en el valor de los parámetros del sistema en función de las rutinas básicas de computación y comunicación empleadas, así como de otros factores como el esquema de acceso a los datos, o la implementación de la operación básica en una librería concreta. Los resultados muestran que con esta metodología se puede realizar una selección adecuada de los parámetros ajustables que aparecen en rutinas de álgebra lineal. De igual manera, se muestra la aplicación de esta técnica a la selección del mejor algoritmo entre varios que se encuentren disponibles, y a la selección de la mejor librería básica, de entre las instaladas en la plataforma computacional.

A continuación, se ha mostrado una propuesta que complementa a la anterior, basada en una formulación para el modelo de tiempo de ejecución que utiliza combinaciones de potencias

del tamaño del problema y de los parámetros del algoritmo. El objetivo es que si con el modelo analítico de una rutina no se consigue realizar una selección adecuada de los parámetros del algoritmo (consideramos parámetros algorítmicos apropiados a aquellos que permitan obtener tiempos de ejecución cercanos a los óptimos), se construya un nuevo modelo para la rutina, es decir, se realice un remodelado. El remodelado se puede aplicar de forma progresiva, aprovechando la estructura jerárquica de las librerías de software de álgebra lineal de forma que, si la información que proporciona el modelo analítico de las rutinas de los niveles inferiores no se puede reutilizar en el modelo teórico de las rutinas de los niveles superiores, se realiza un remodelado de las rutinas de los niveles inferiores, y en el caso de que dicho remodelado siga sin proporcionar información válida se procederá a la construcción de un nuevo modelo para la rutina del nivel superior. Con esta nueva propuesta de formulación para el modelo, se requiere obtener el valor de los coeficientes que ajusten el modelo a las características de la plataforma computacional y software básico empleado en la rutina. El procedimiento seguido para obtener dichos coeficientes se basa en la aplicación de técnicas de regresión segmentada sobre valores generados experimentalmente para el tiempo de ejecución de la rutina. Se ha mostrado la aplicación de la metodología propuesta en el modelado de rutinas básicas de comunicación y computación en diferentes plataformas computacionales y con rutinas de los diferentes niveles de la jerarquía clásica de librerías de software de álgebra lineal.

Por otro lado, se ha mostrado que una metodología de modelado del tiempo de ejecución en combinación con estrategias de asignación de procesos a procesadores permite encontrar una solución al problema de ejecutar una rutina homogénea de la manera más óptima posible en un sistema heterogéneo, sin necesidad de modificar el código de la misma. La búsqueda de una solución en estas condiciones se complica notablemente, dado que ahora es necesario encontrar la mejor distribución del trabajo. Se han utilizado árboles que representan el espacio de posibles soluciones a la mejor selección del número de procesos y asignación de procesos a procesadores. En estos árboles, cada nivel representa un proceso, y cada nodo uno de los procesadores en los que el proceso puede ser asignado. Se plantea un problema de optimización en el que la función a optimizar es el tiempo de ejecución de la rutina (aproximada por su modelo analítico de tiempo de ejecución) con cada posible asignación de procesos a procesadores representada por cada nodo del árbol. Se han propuesto algunas estrategias de búsqueda exhaustiva y de avance rápido con las que recorrer el árbol de posibles soluciones, y se ha analizado su aplicación en plataformas con diferente grado de heterogeneidad. La aplicación de técnicas de búsqueda exhaustiva en plataformas de tamaño reducido y poco heterogéneas han proporcionado resultados satisfactorios con soluciones cercanas a la óptima. Sin embargo, en plataformas de mayor tamaño y más heterogéneas, su aplicación conduce a tiempos de búsqueda excesivos, resultando ser más adecuadas las técnicas propuestas de avance rápido al proporcionar en general mejores resultados que los métodos más exhaustivos, y en un tiempo razonable. Se ha comprobado que las decisiones realizadas con la metodología propuesta, sobre la mejor selección de los parámetros ajustables de la rutina homogénea cuando se ejecuta en una plataforma heterogénea, son mejores, en general, que las decisiones que realizaría el usuario, por lo que se puede concluir

que se ha alcanzado el objetivo propuesto.

Tras ello, se ha realizado un estudio sobre la aplicación de nuestra metodología de modelado del tiempo de ejecución a la elección de la mejor versión disponible para el ejecutable de una rutina de álgebra lineal multithread con código OpenMP, y se ha visto que dicha selección supone una mejora en las prestaciones del código de álgebra lineal multithread para plataformas multicore. La selección de la versión del ejecutable se realiza siguiendo la metodología de modelado propuesta a lo largo de este trabajo, de tal forma que la elección de un código generado por un compilador concreto se convierte en otro parámetro algorítmico más. Se ha comprobado que esta selección de la versión más óptima no siempre es la que a priori podría considerarse la mejor, en base a los conocimientos que se pudieran tener de la plataforma o el algoritmo, y puede ser totalmente diferente dependiendo de la rutina, del número de threads paralelos y del tamaño del problema a resolver, lo que justifica la utilización de la técnica propuesta.

Por último, cabría destacar que en nuestro estudio sobre técnicas de modelado del tiempo de ejecución, con el propósito de realizar el ajuste del software científico sin intervención del usuario, en plataformas computacionales homogéneas y heterogéneas, se han obtenido resultados significativos y diversas contribuciones con las que se pueden desarrollar técnicas de optimización automática, obteniéndose las mejores prestaciones posibles.

7.2 Resultados de la tesis

El desarrollo de los objetivos fundamentales de esta tesis se llevó a cabo dentro del marco de los siguientes proyectos regionales y nacionales:

- Proyecto CICYT, “Desarrollo y optimización de código paralelo para sistemas de audio 3D (TIC2003-08238-C02-02)”, coordinado con un grupo de investigación de la Universidad Politécnica de Valencia, desarrollado desde el 1 de diciembre de 2003 hasta el 30 de noviembre de 2006.
- Proyecto de la Fundación Séneca, Consejería de Cultura y Educación de la Región de Murcia, “Técnicas de optimización de rutinas paralelas y aplicaciones (02973/PI/05)”, desarrollado desde el 1 de enero de 2006 hasta el 31 de diciembre de 2008.
- Proyecto CICYT, “Construcción y optimización automáticas de bibliotecas de computación científica (TIN2008-06570-C04-04)”, coordinado con grupos de investigación de la Universidad Politécnica de Valencia, Jaume I de Castellón, Alicante y La Laguna, desarrollado desde el 1 de enero de 2009 hasta el 31 de diciembre de 2011. Figuran como EPO las empresas Hewlett-Packard Española y Aurorasat.
- Proyecto de la Fundación Séneca, Consejería de Cultura y Educación de la Región de Murcia, “Adaptación y optimización de código científico en sistemas computacionales jerárquicos (08763/PI/08)”, coordinado con un grupo de investigación de la Universidad

Politécnica de Cartagena, desarrollado desde el 1 de enero de 2009 hasta el 31 de diciembre de 2011.

Adicionalmente cabe destacar la participación en la siguiente Red de Investigación:

- Dirección General de Investigación, Ministerio de Educación y Ciencia, "Red de computación de altas prestaciones sobre arquitecturas paralelas heterogéneas (CAPAP-H, CAPAP-H2, CAPAP-H3) (TIN2007-29664-E, TIN2009-08058-E, TIN2010-12011-E)", desarrollado desde el 2007 al 2012.

A continuación se enumeran las publicaciones y comunicaciones que muestran los resultados relacionados con los objetivos marcados en esta tesis, con una breve descripción de su contenido:

- *Designing polylibraries to speed up linear algebra computations.* Pedro Alberti, Pedro Alonso, Antonio Vidal, Javier Cuenca, Luis-Pedro García, Domingo Giménez. Technical Report, UM-LSI 1-2003. [AAV+03]

  En este informe técnico se analiza el uso de polilibrerías de álgebra lineal densa y se propone una arquitectura para este tipo de librerías, en las que un programa llama a rutinas de diferentes librerías en función del tamaño del problema a resolver y la plataforma computacional utilizada. El objetivo de este trabajo es el desarrollo de una metodología de aplicación en el diseño de librerías de álgebra lineal densa. Se muestran resultados experimentales en diferentes sistemas y con distintas rutinas de álgebra lineal: multiplicación de matrices, factorización de matrices (QR, LU, Cholesky), etc. Los resultados obtenidos confirman que el diseño de polilibrerías permite acelerar los resultados de las computaciones. El contenido de este trabajo se incluye en el capítulo 3.

- *Empirical modelling of parallel linear algebra routines.* Javier Cuenca, Luis-Pedro García, Domingo Giménez, José González, Antonio Vidal. PPAM 2003. Czestochowa, Polonia, September 7-10, 2003. [CGG+04a]

  En este artículo se propone la combinación de estudios empíricos y modelos teóricos de rutinas paralelas de álgebra lineal con el fin de mejorar la estimación obtenida con el modelo de tiempo de ejecución. El modelo así obtenido puede ser utilizado en la toma de algunas decisiones que facilitan la reducción del tiempo de ejecución. Se muestran resultados con la factorización QR y la de Cholesky. El contenido de este artículo se corresponde con lo tratado en el capítulo 3.

- *Auto-optimization of linear algebra parallel routines: the Cholesky factorization.* PARCO 2005. Luis-Pedro García, Javier Cuenca, Domingo Giménez. [GCG05]

  En este artículo se realiza un estudio de las mejoras obtenidas en el modelo de tiempo de ejecución de rutinas paralelas de álgebra lineal cuando se tiene en cuenta la existencia de

diferentes costes en las operaciones de comunicación disponibles en MPI, y de la dependencia del coste de las comunicaciones con el volumen de información transmitido. Las mejoras introducidas en el modelado son analizadas con la factorización de Cholesky en plataformas en las que es posible utilizar simultáneamente más de una red de interconexión. Su contenido se corresponde con parte del capítulo 3.

- *Processes Distribution of Homogeneous Parallel Linear Algebra Routines on Heterogeneous Clusters.* Javier Cuenca, Luis-Pedro García, Domingo Giménez, Jack Dongarra. CLUSTER 2005. [CGGD05]

  Se aborda la optimización automática por medio de modelos de tiempo de ejecución de rutinas paralelas de álgebra lineal diseñadas para sistemas homogéneos en sistemas heterogéneos sin la necesidad de reescribir el código de la rutina. Se consigue con la metodología propuesta automatizar la toma de una serie de decisiones que conducen a la obtención de tiempos de ejecución cercanos al óptimo. El contenido de este trabajo se incluye en el capítulo 5 de esta tesis.

- *Parametrización de esquemas algorítmicos paralelos para autooptimización.* Francisco Almeida, Juan Manuel Beltrán, Murilo Boratto, Domingo Giménez, Javier Cuenca, Luis-Pedro García, Juan-Pedro Martínez-Gallar, Antonio M. Vidal. Proceedings XVII Jornadas de Paralelismo. 2006. [ABB+06]

  Se muestra la aplicación de modelos de tiempo de ejecución parametrizados en la obtención de rutinas con capacidad de optimización automática. Se analizan los resultados obtenidos en la aplicación a esquemas de programación dinámica, divide y vencerás (como el algoritmo de Strassen visto en esta tesis), *backtracking* y factorizaciones matriciales.

- *Including improvement of the execution time in a software architecture of libraries with self-optimisation.* Luis-Pedro García, Javier Cuenca, Domingo Giménez. ICSOFT 2007. [GCG07a]

  Se presenta la aplicación de la fase de remodelado dentro del contexto de las técnicas de optimización automática de rutinas de álgebra lineal con estructura jerárquica, tal como la librería ScaLAPACK. Dentro de este contexto, las rutinas en un nivel de la jerarquía usan información generada por rutinas de los niveles inferiores. Se puede dar el caso que la información generada en un nivel no sea lo suficientemente precisa para ser usada satisfactoriamente en los niveles superiores, por lo que se propone entonces la generación de un nuevo modelo. El contenido de este artículo se corresponde con lo tratado en el capítulo 4.

- *Using experimental data to improve the performance modelling of parallel linear algebra routines.* Luis-Pedro García, Javier Cuenca, Domingo Giménez. PPAM 2007. [GCG07b]

  En este artículo se estudia la aplicación apropiada de técnicas de regresión lineal para la obtención de un modelo que permita aproximar el tiempo de ejecución de una rutina paralela de álgebra lineal. Se hace un análisis de diferentes esquemas para la obtención

de los coeficientes del polinomio que aproxima el tiempo de ejecución de la rutina, con el propósito de conseguir llegar a un compromiso entre el tiempo de generación del modelo y la exactitud del mismo. Se muestran resultados de la generación del modelo con la multiplicación de matrices. El contenido de este trabajo se incluye dentro del capítulo 4 de esta tesis.

- *Comparing the behaviour of basic linear algebra routines on multicore platforms.* Javier Cuenca, Luis-Pedro García, Domingo Giménez, Manuel Quesada. 9th International Conference on Computational and Mathematical Methods in Science and Engineering, Minisymposium on High Performance Computing applied to Computational Problems in Science and Engineering. 2009. [CGGQ09]

  Se realiza un análisis de la influencia del compilador OpenMP en el comportamiento de diferentes rutinas paralelas de álgebra lineal densa en plataformas multicore. Se comprueba con este trabajo que la selección del compilador con el que se obtiene mejores tiempos de ejecución depende de factores como la rutina, el número de hilos de ejecución a generar, y el tamaño del problema a resolver. Se propone la utilización de un motor de policompilación que, basándose en el uso del modelo teórico del tiempo de ejecución de la rutina y de un conjunto de pruebas en las que se miden las capacidades del compilador y de su implementación de las primitivas OpenMP, seleccione en tiempo de ejecución la versión óptima del código compilado. En el capítulo 6 de esta tesis se incluyen los resultados mostrados en este trabajo.

- *Analysis of the influence of the compiler on multicore performance.* Javier Cuenca, Luis-Pedro García, Domingo Giménez, Manuel Quesada. PDP 2010. [CGGQ10]

  En este artículo se realiza un estudio teórico y experimental de la influencia del compilador en el rendimiento de rutinas paralelas de álgebra lineal en plataformas multicore. Se propone la utilización de una metodología que combinando el uso de modelos teóricos de tiempo de ejecución junto con la experimentación en una plataforma multicore en particular, permita realizar la selección del compilador OpenMP que genere el código con el que se obtiene mejores tiempos de ejecución. El contenido de este artículo se recoge en el capítulo 6.

- *A proposal for autotuning linear algebra routines on multicore platforms.* Javier Cuenca, Luis-Pedro García, Domingo Giménez. ICSS 2010. [CGG10]

  Se propone la aplicación de modelos de tiempo de ejecución de rutinas paralelas de álgebra lineal en la selección automática del compilador OpenMP capaz de generar el código o partes del código que mejor se adapta a una plataforma multicore determinada. Adicionalmente se estudia la aplicación del modelo para la selección de los parámetros del algoritmo, en este caso el número de hilos de ejecución a generar. El artículo se corresponde con los contenidos del capítulo 6.

7.3 Trabajo futuro

A continuación se presentan algunas de las líneas relacionadas con el trabajo realizado en esta tesis y que se consideran de interés a corto y medio plazo. En alguna de estas líneas ya se ha comenzado a trabajar en la actualidad:

- En el capítulo 4 se ha mostrado como combinar el modelo teórico que aproxima el tiempo de ejecución de una rutina de álgebra lineal con el uso de modelos obtenidos utilizando regresión segmentada. Se podría estudiar la posibilidad de utilizar funciones spline discretas en el remodelado, desarrollar una metodología para mejorar incrementalmente la estimación obtenida y emplear una metodología de selección de modelos como el Criterio de Información de Akaike (AIC) [Aka74].
- Modelado de rutinas que combinan diferentes tipos de paralelismo:
  - Ampliación de la metodología presentada a paralelismo con OpenMP y BLAS en sistemas NUMA de grandes dimensiones donde los cores comparten un sistema de memoria jerárquico. Experimentalmente se ha comprobado que se produce una degradación en las prestaciones de la rutina **dgemm** multithread de BLAS al ir incrementándose el tamaño de este tipo de sistemas. El objetivo es el modelado del tiempo de ejecución de rutinas de álgebra lineal con paralelismo anidado OpenMP y BLAS multithread, que permita desarrollar una metodología de optimización automática con la que mejorar las prestaciones de las versiones multithread disponibles en las librerías de software de álgebra lineal. En relación a esta propuesta, se dispone ya de los resultados contenidos en el trabajo: *Optimisation of dense linear algebra computation in large NUMA systems through auto-tuned nested parallelism.* Javier Cuenca, Luis-Pedro García, Domingo Giménez. PDP 2012. [CGG12].
  - Ampliación de la metodología presentada a paralelismo CPU+GPU. Se pretende extender la metodología presentada en la utilización de modelos de tiempo de ejecución parametrizados para conseguir modelar el tiempo de ejecución de una rutina cuando se ejecuta en un sistema híbrido formado por varios cores y una o más de una GPU. El objetivo es el de acelerar la resolución de problemas en ciencias e ingeniería combinando el uso de OpenMP con CUDA u OpenCL. De igual forma, se podría extender la metodología a combinaciones en las que se use MPI.
- Optimización automática de rutinas de álgebra lineal en plataformas heterogéneas:
  - Estudio de la aplicación de la metodología propuesta para el modelado del tiempo de ejecución en la que se hace uso de la estructura jerárquica de las librerías de software de álgebra lineal diseñadas para plataformas homogéneas (LAPACK, ScaLAPACK...), a librerías con estructura similar y diseñadas para plataformas heterogéneas como HeteroScaLAPACK [RL06].

- Ampliación de la propuesta para la búsqueda de la mejor combinación de los parámetros ajustables de rutinas de álgebra lineal homogéneas cuando se ejecutan en plataformas heterogéneas, mediante la utilización sistemática de heurísticas a través de métodos metaheurísticos como búsqueda dispersa, algoritmos genéticos, GRASP (*Greedy Randomized Adaptative Search Procedure*). Un siguiente paso, sería analizar la posibilidad de aplicar de forma combinada las técnicas metaehurísticas mencionadas.

Bibliografía

[AAV+03] P. Alberti, P. Alonso, A. Vidal, J. Cuenca, L. P. García, and D. Giménez. Designing polylibraries to speed up linear algebra computations. Technical Report UM-LSI 1-2003, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Murcia, march 2003.

[AAV+04] P. Alberti, P. Alonso, A. M. Vidal, J. Cuenca, and D. Giménez. Designing polylibraries to speed up linear algebra computations. *International Journal of High Performance Computing and Networking*, 1:75–84, August 2004.

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, third edition, 1999.

[ABB+06] F. Almeida, J. M. Beltran, M. Boratto, D. Giménez, J. Cuenca, L. P. García, J. P. Martínez-Gallar, and A. M. Vidal. Parametrizaci\'on de esquemas algorítmicos paralelos para autooptimización. In *XVII Jornadas de Paralelismo*, pages 443–448, 2006.

[ABC+06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, december 2006.

[ABG+95] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[ACFS92] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:12–20, 1992.

[ACS90] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theor. Comput. Sci.*, 71(1):3–28, 1990.

[AD89] E. Anderson and J. Dongarra. Evaluating block algorithm variants in LAPACK. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 3–8, 1989.

[AGMV08] F. Almeida, D. Giménez, J. M. Mantas, and A. M. Vidal. *Introducción a la programación paralela*. Cengage Learning Paraninfo, 2008.

[AGZ94] R. C. Agarwal, F. Gustavson, and M. Zubair. A high performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication. *IBM J. of Research and Development*, 38(6):673–681, 1994.

[AISS95] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. In *Proceedings of the 7th annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105. ACM, 1995.

[Aka74] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, December 1974.

[ATL] ATLAS. Automatically tuned linear algebra software. `http://math-atlas.sourceforge.net`.

[BACD97] J. Bilmes, K. Asanović, C. W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, pages 340–347, July 1997.

[BB96] G. Brassard and P. Bratley. *Fundamentals of algorithmics*. Prentice Hall, 1996.

[BBP+01] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster scalapack (dense linear solvers). *IEEE Transactions on Parallel and Distributed Systems*, 50(10):1052–1070, 2001.

[BCC+97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1997.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[BGP+94] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. Van de Geijn, and J. Watts. Building a high-performance collective communication library. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, Supercomputing '94, pages 107–116. ACM, 1994.

[BO01] J. M. Bull and D. O'Neill. A microbenchmark suite for OpenMP 2.0. In *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, pages 41–48, 2001.

[BP06] J. L. Bosque and L. Pastor. A parallel computational model for heterogeneous clusters. *IEEE Trans. Parallel Distrib. Syst.*, 17:1390–1400, December 2006.

[BPPS07] G. Bilardi, A. Pietracaprina, G. Pucci, and F. Silvestri. Network-oblivious algorithms. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[Bre95] E. A. Brewer. High-level optimization via automated statistical modeling. *SIGPLAN Notices*, 30:80–91, August 1995.

[But97] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[Can69] L. E. Cannon. *A cellular computer to implement the Kalman Filter Algorithm*. PhD thesis, Montan State University, 1969.

[CDD+95] J. Choi, J. Demmel, I. S. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. In *Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, PARA'95*, Lecture Notes in Computer Science, pages 95–106. Springer, 1995.

[CDO+96] J. Choi, J. Dongarra, L. Ostrouchov, A. Petitet, D. Walker, and R. Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Sci. Program.*, 5(3):173–184, 1996.

[CDW94] J. Choi, J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[CDW96] J. Choi, J. Dongarra, and D. W. Walker. PB-BLAS: a set of parallel block basic linear algebra subprograms. *Concurrency: practice and experience*, 8(7):517–535, September 1996.

[CGG+04a] J. Cuenca, L. P. García, D. Giménez, J. González, and A. Vidal. Empirical modelling of parallel linear algebra routines. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 3019 of *Lecture Notes in Computer Science*, pages 169–174. Springer, 2004.

[CGG04b] J. Cuenca, D. Giménez, and J. González. Architecture of an automatically tuned linear algebra library. *Parallel Computing*, 30:187–210, February 2004.

[CGG10] J. Cuenca, L. P. García, and D. Giménez. A proposal for autotuning linear algebra routines on multicore platforms. *ICCS*, 1(1):515–523, 2010.

[CGG12] J. Cuenca, L. P. García, and D. Giménez. Optimisation of dense linear algebra computation in large NUMA systems through auto-tuned nested parallelism. In *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2012)*, February 2012.

[CGGD05] J. Cuenca, L. P. García, D. Giménez, and J. Dongarra. Processes distribution of homogeneous parallel linear algebra routines on heterogeneous clusters. In *CLUSTER*, pages 1–10. IEEE, 2005.

[CGGQ09] J. Cuenca, L. P. García, D. Giménez, and M. Quesada. Comparing the behaviour of basic linear algebra routines on multicore platforms. In *9th International Conference on Computational and Mathematical Methods in Science and Engineering*, pages 341–349, 2009.

[CGGQ10] J. Cuenca, L. P. García, D. Giménez, and M. Quesada. Analysis of the influence of the compiler on multicore performance. In *PDP*, pages 170–174, 2010.

[CGMG05] J. Cuenca, D. Giménez, and J. P. Martínez-Gallar. Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems. *Parallel Computing*, 31(7):711–735, 2005.

[Cho97] J. Choi. A Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers. *IPPS*, 00:310–314, 1997.

[CHPVdG04] E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. Van de Geijn. On optimizing collective communication. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 145–155. IEEE Computer Society, 2004.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. LogP: towards a realistic model of parallel computation. *SIGPLAN Notices*, 28(7):1–12, 1993.

[CL94] M. E. Crovella and T. J. Leblanc. Parallel performance prediction using lost cycles analysis. In *Proceedings of Supercomputing*, pages 600–609, 1994.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd revised edition, 2001.

[cmp] Compaq MPI: Description. `http://www.hp.com/techservers/software/cmpidesc.html`.

[CQ95] M. J. Clement and M. J. Quinn. Multivariate statistical techniques for parallel performance prediction. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 446–455. IEEE Computer Society, 1995.

[CQ97] M. J. Clement and M. J. Quinn. Automated performance prediction for scalable parallel computing. *Parallel Computing*, 23:1405–1420, 1997.

[Cro94] M. E. Crovella. *Performance Prediction and Tuning of Parallel Programs*. PhD thesis, University of Rochester Department of Computer Science, August 1994. Available as TR 573 from URCSD.

[CS03] K. W. Cameron and X. Sun. Quantifying locality effect in data access delay: Memory logP. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 48.2–. IEEE Computer Society, 2003.

[Cue04] J. Cuenca. *Optimización Automática de Software Paralelo de Álgebra Lineal*. PhD thesis, Universidad de Murcia, 2004.

[dGW97] R. A. Van de Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[DHSS94] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith. GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix–matrix multiply algorithm. *J. Comp. Phys.*, 110:1–10, 1994.

[DK96] K. Dackland and B. Kågström. A hierarchical approach for performance analysis of ScaLAPACK-based routines using the distributed linear algebra machine. In *Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization*, PARA '96, pages 186–195. Springer-Verlag, 1996.

[DKK03] E. Dovolnov, A. Kalinov, and S. Klimov. Natural block data decomposition for heterogeneous clusters. In *17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 102–111. IEEE Computer Society, 2003.

[DM98] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science and Engineering*, 5:46–55, 1998.

[DN05] P. D'Alberto and A. Nicolau. Adaptive Strassen and ATLAS's DGEMM: A fast square-matrix multiply for modern high-performance systems. In *Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, volume 0 of *HPCASIA'05*, pages 45–52. IEEE Computer Society, 2005.

[DNS81] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM J. Computing*, 10:657–673, 1981.

[DW97] J. Dongarra and R. Whaley. A user's guide to the BLACS v1.1. Technical Report 94, LAPACK Working Note, May 1997. originally released March 1995.

[FAV97] M. I. Frank, A. Agarwal, and M. K. Vernon. LoPC: Modeling contention in parallel algorithms. *SIGPLAN Notices*, 32:276–287, June 1997.

[FJ05] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[FJL+88] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving problems on concurrent processors*. Prentice-Hall, Inc., 1988.

[FMT03] S. Fujita, M. Masukawa, and S. Tagashira. A fast branch-and-bound scheme for the multiprocessor scheduling problem with communication time. In *32nd International Conference on Parallel Processing Workshops (ICPP 2003 Workshops)*, pages 104–111. IEEE Computer Society, 2003.

[FOH87] G. C. Fox, S. W. Otto, and A. J. G. Hey. Matrix algorithms on a hypercube i: Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.

[Fos95] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.

[Gau04] H. Gautama. *A Probabilistic Approach to Symbolic Performance Modeling of Parallel Systems*. PhD thesis, Technische Universiteit Delft, 2004.

[GCG05] L. P. García, J. Cuenca, and D. Giménez. Auto-optimization of linear algebra parallel routines: The cholesky factorization. In *PARCO*, pages 229–236, 2005.

[GCG07a] L. P. García, J. Cuenca, and D. Giménez. Including improvement of the execution time in a software architecture of libraries with self-optimisation. In *ICSOFT (SE)*, pages 156–161, 2007.

[GCG07b] L. P. García, J. Cuenca, and D. Giménez. Using experimental data to improve the performance modelling of parallel linear algebra routines. In *PPAM*, pages 1150–1159, 2007.

[Gee05] D. Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38:11–13, May 2005.

[Gen03] J.E. Gentle. *Random number generation and Monte Carlo methods*. Statistics and computing. Springer, 2003.

[GGKK03] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to parallel computing*. Addison-Wesley, 2nd edition, 2003.

[GH01] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. Society for Industrial and Applied Mathematics, first edition, 2001.

[GHH97] V. Getov, E. Hernández, and T. Hey. Message-passing performance of parallel computers. In *EURO– PAR '97*, pages 1009–1016. Springer. LNCS, 1997.

[Gib89] P. B. Gibbons. A more practical PRAM model. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168. ACM, 1989.

[GL96] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Press, 3rd edition, 1996.

[GLS99] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface, second edition*. The MIT Press, 1999.

[GSdG95] B. Grayson, A. P. Shah, and R. A. Van de Geijn. A High Performance Parallel Strassen Implementation. Technical Report CS-TR-95-24, The University of Texas at Austion, january 1995.

[Har94] T. J. Harris. A survey of PRAM simulation techniques. *ACM Comput. Surv.*, 26(2):187–206, 1994.

[HLJT+96] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson. Implementation of Strassen's algorithm for matrix multiplication. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 32. IEEE Computer Society, 1996.

[Hoc82] R. W. Hockney. Characterization of parallel computers and algorithms. *Computer Physics Communications*, 26(3-4):285 – 291, 1982.

[Hoc94] R. W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20:389–398, March 1994.

[HR92] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation i. the model. *Journal of Parallel and Distributed Computing*, 16(3):212–232, November 1992.

[IdSSM05] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 196–205. Springer, 2005.

[iWA] The International Workshop on Automatic Performance Tuning WWW home page. `http://www.iwapt.org`.

[IYV04] E. J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing and Applications*, 18(1):135–158, 2004.

[JKP95] L. H. Jamieson, A. A. Khokhar, and J. N. Patel. Algorithm scalability: A polyalgorithmic approach. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages 90–95. CRC Press, 1995.

[KBG+01] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. H. Hofman. Network performance-aware collective communication for clustered wide-area systems. *Parallel Computing*, 27(11):1431 – 1456, 2001.

[KdSF+00] N.T. Karonis, B.R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, pages 377–384. IEEE Computer Society, 2000.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., 1994.

[KI04] Y. Kishimoto and S. Ichikawa. An execution-time estimation model for heterogeneous clusters. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, volume 2, page 105b. IEEE Computer Society, 2004.

[KK05] A. Kalinov and S. Klimov. Optimal mapping of a parallel application processes onto heterogeneous platform. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, volume 2, pages 123.2–. IEEE Computer Society, 2005.

[KKHY03] T. Katagiri, K. Kise, H. Honda, and T. Yuba. FIBER: A generalized framework for auto-tuning software. In *ISHPC*, pages 146–159, 2003.

[KKHY06] T. Katagiri, K. Kise, H. Honda, and T. Yuba. ABCLib_DRSSED: a parallel eigensolver with an auto-tuning facility. *Parallel Computing*, 32(3):231–250, 2006.

[KKK00] T. Katagiri, H. Kuroda, and Y. Kanada. A methodology for automatically tuned parallel tridiagonalization on distributed memory vector-parallel machines. In *VECPAR 2000: Proceedings of Vector and Parallel proccessing*, pages 265–277, 2000.

[KL01] A. Kalinov and A. L. Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. *Journal of Parallel and Distributed Computing*, 61(4):520–535, 2001.

[KLAdH92] R. M. Karp, M. Luby, and F. M. Auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 318–326. ACM, 1992.

[lam] LAM-MPI parallel computing. `http://www.lam-mpi.org`.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[LJ93] W. Lichtenstein and S. L. Johnsson. Block-cyclic dense linear algebra. *SIAM Journal on Scientific Computing*, 14(6):1259–1288, 1993.

[LL94] H. Lennerstad and L. Lundberg. Optimal scheduling results for parallel computing. *SIAM News*, 27:16–18, 1994.

[LSF97] J. Li, A. Skjellum, and R. D. Falgout. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency: Practice and Experience*, 9(5):345–389, 1997.

[LZFD10] L. Li, X. Zhang, J. Feng, and X. Dong. mPlogP: A parallel computation model for heterogeneous multi-core computer. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 679–684. IEEE Computer Society, 2010.

[Mcc98] W. F. Mccoll. Foundations of time-critical scalable computing. In *Proceedings of the 15th IFIP World Computer Congress. Osterreichische Computer Gesellschaft*, pages 93–107, 1998.

[MCGZ04] L. Muttoni, G. Casale, F. Granata, and S. Zanero. Optimal number of nodes for computation in grid environments. In *12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (EUROMICRO-PDP 2004)*, pages 282–289. IEEE Computer Society, 2004.

[Mon04] D. C. Montgomery. *Design and Analysis of Experiments*. Wiley, 6th edition, December 2004.

[mpi] MPICH-A Portable MPI Implementation. `http://www-unix.mcs.anl.gov/mpi/mpich`.

[MT99] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24(3-4):287–297, 1999.

[NT04] W. Nasri and D. Trystram. A poly-algorithmic approach applied for fast matrix multiplication on clusters. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, pages 234–241. IEEE Computer Society, 2004.

[OTBS04] Y. Ohtaki, D. Takahashi, T. Boku, and M. Sato. Parallel implementation of strassen's matrix multiplication algorithm for heterogeneous clusters. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*. IEEE Computer Society, 2004.

[PSCL97] P. Pauca, X. Sun, S. Chatterjee, and A. Lebeck. Architecture-efficient Strassen's matrix multiplication: A case study of divide-and-conquer algorithms. In *In International Linear Algebra Society (ILAS) Symposium on Algorithms for Control, Signals and Image Processing*, 1997.

[PY88] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 510–513. ACM, 1988.

[Qui03] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Science, first edition, 2003.

[RL06] R. Reddy and A. Lastovetsky. HeteroMPI+ScaLAPACK: towards a ScaLAPACK (dense linear solvers) on heterogeneous networks of computers. In *Proceedings of the 13th international conference on High Performance Computing*, HiPC'06, pages 242–253. Springer-Verlag, 2006.

[Rod11] D. Rodríguez. *Modelado Analítico del Rendimiento de Aplicaciones en Sistemas Paralelos*. PhD thesis, Universidad de Santiago de Compostela, 2011.

[RR01] T. Rauber and G. Rünger. A hierarchical computation model for distributed shared-memory machines. In *Proceedings of Parallel and Distributed Processing*, volume 0, pages 57–64. IEEE Computer Society, 2001.

[SB95] A. Skjellum and P. Bangalore. Driving issues in scalable libraries: Poly-algorithms, data distribution independence, redistribution, local storage schemes. In *Proceedings of the 7th (SIAM) Conference on Parallel Processing for Scientific Computing*, pages 734–737, 1995.

[SGJ+02] H. Saito, G. Gaertner, W. B. Jones, R. Eigenmann, H. Iwashita, R. Lieberman, G. M. van Waveren, and B. Whitney. Large system performance of SPEC OMP2001 benchmarks. In H. P. Zima, K. Joe, M. Sato, Y. Seo, and M. Shimasaki, editors, *Proceedings of the 4th International Symposium of High Performance Computing (ISHPC 2002)*, volume 2327 of *Lecture Notes in Computer Science*, pages 370–379. Springer, 2002.

[SHM96] D. B. Skillicorn, J. M. D. Hill, and W. F. Mccoll. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1996.

[SKK03] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high-performance scientific simulations. In J. Dongarra, I. Foster, G. Fox, W. Gropp,

K. Kennedy, L. Torczon, and A. White, editors, *Sourcebook of Parallel Computing*, pages 491–541. Morgan Kaufmann Publishers Inc., 2003.

[SKRS03] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan. Scheduling of parallel jobs in a heterogeneous multi-site environement. In *Job Scheduling Strategies for Parallel Processing, 9th International Workshop (JSSPP 2003)*, volume 2862 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2003.

[SOHL+98] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–The Complete Reference: Volume 1, The MPI Core, second edition.* The MIT Press, 1998.

[Sor70] H. W. Sorenson. Least-squares estimation: from Gauss to Kalman. *IEEE Spectrum*, 7:63–68, July 1970.

[SSF+08] S. M. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. J. Collazo-Mojica. A modeling approach for estimating execution time of long-running scientific applications. In *IPDPS*, pages 1–8. IEEE, 2008.

[Str69] V. Strassen. Gaussian elimination is not optimal. *NumerMath*, 13:354–356, 1969.

[SZ09] J. E. Savage and M. Zubair. Evaluating multicore algorithms on the unified memory model. *Sci. Program.*, 17(4):295–308, 2009.

[TD01] D. Tessera and A. Dubey. Communication policies performance: A case study. In *Ninth Euromicro Workshop on Parallel and Distributed Processing (PDP'01)*, pages 491–. IEEE Computer Society, 2001.

[TG03] R. Thakur and W. Gropp. Improving the performance of collective operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2840 in LNCS, Springer Verlag (2003) 10th European PVM/MPI User's Group Meeting*, pages 257–267. Springer Verlag, 2003.

[Tis98] A. Tiskin. The bulk-synchronous parallel random access machine. *Theor. Comput. Sci.*, 196(1-2):109–130, 1998.

[TK96] P. Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting (extended abstract). In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, pages 352–358. Springer-Verlag, 1996.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[Val08] L. G. Valiant. A bridging model for multi-core computing. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*, pages 13–28. Springer-Verlag, 2008.

[VDY05] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series. Institute of Physics Publishing, June 2005.

[VTS+04] D. P. Vidyarthi, A. K. Tripathi, B. K. Sarker, A. Dhawan, and L. T. Yang. Cluster-based multiple task allocation in distributed computing system. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, volume 14, page 239. IEEE Computer Society, 2004.

[WH97] H. Wabnig and G. Haring. PAPS – A tested for performance prediction of parallel applications. *Parallel Computing*, 22(13):1837–1851, 1997.

[WPD01] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[XH96] Z. Xu and K. Hwang. Modeling communication overhead: MPI and MPL performance on the IBM SP2. *Parallel Distributed Technology: Systems Applications, IEEE*, 4(1):9 –24, spring 1996.

[XZS96] Z. Xu, X. Zhang, and L. Sun. Semi-empirical multiprocessor performance predictions. *Journal of Parallel and Distributed Computing*, 39:14–28, 1996.

[Yam06] Y. Yamamoto. Performance modeling and optimal block size selection for the small-bulge multishift qr algorithm. In M. Guo, L. Yang, B. Di Martino, H. Zima, J. Dongarra, and F. Tang, editors, *Parallel and Distributed Processing and Applications*, volume 4330 of *Lecture Notes in Computer Science*, pages 451–463. Springer, 2006.

[ZR87] W. Zhao and K. Ramamritham. Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints. *The Journal of Systems and Software*, 7:195–205, September 1987.