Universidad de Murcia
Facultad de Informática
Máster en Nuevas
Tecnologías en Informática

Optimización de Rutinas Multinivel de Álgebra Lineal en Sistemas Multicore

Tesis de Máster

Itinerario de Arquitecturas de Altas Prestaciones y Supercomputación

Autor:
Jesús Cámara Moreno
jcm23547@um.es

Tutores:
A. Javier Cuenca Muñoz
Domingo Giménez Cánovas

Murcia, Septiembre de 2011

Índice General

Resumen

El propósito de esta tesis de máster es analizar el comportamiento de diferentes rutinas matriciales en sistemas multicore de memoria compartida usando varios niveles de paralelismo. Para lograr una implementación lo más eficiente posible, se realiza un proceso de auto-optimización capaz de seleccionar el número de threads a usar en cada nivel de paralelismo. Este proceso, mediante la aplicación de determinadas técnicas, va a permitir ajustar los parámetros algorítmicos de la rutina a las características físicas del sistema para que el usuario final pueda integrar estas rutinas en el código sin preocuparse del sistema en el que la aplicación se vaya a ejecutar, asegurando una ejecución paralela multinivel óptima.

En este trabajo se va a diseñar un prototipo de librería. Este prototipo incluye una rutina de multiplicación de matrices capaz de ejecutarse de forma óptima en el sistema usando dos niveles de paralelismo, invocando, a su vez, a rutinas programadas en niveles inferiores de diferentes librerías de álgebra lineal, como BLAS.

El primer nivel de paralelismo se establecerá mediante OpenMP, pues los sistemas sobre los que se realizarán las pruebas presentan una arquitectura basada en memoria compartida. En cambio, el segundo nivel de paralelismo se establecerá mediante llamadas a rutinas incluidas en librerías de álgebra lineal como MKL o ATLAS, ya que proporcionan una implementación multithreading de BLAS.

La implementación de este prototipo de librería se llevará a cabo evaluando el uso de múltiples niveles de paralelismo al ejecutar rutinas matriciales sobre sistemas multicore. Se pretende que este prototipo sea el punto de partida para su extensión con niveles adicionales de paralelismo y capacidad de auto-optimización, lo que requiere realizar un diseño modular de dicho prototipo. Puesto que el código desarrollado será de libre distribución, se han de proporcionar interfaces genéricas bien definidas, del estilo a las ofrecidas por librerías como BLAS, LAPACK, etc., que permitan su posible ampliación y adaptación con nuevas versiones desarrolladas en el futuro.

Capítulo 1. Introducción

1.1 Introducción

La reciente propagación de los sistemas multicore ha despertado un gran interés en la comunidad científica por el uso de la computación paralela en la resolución eficiente de problemas científicos y de ingeniería sin que ello requiera tener que reprogramar códigos secuenciales existentes. En este contexto, existen librerías como ATLAS [1] o MKL [2] que ofrecen implementaciones multithreading eficientes de rutinas matriciales básicas del tipo de las ofrecidas por BLAS [3], que van a permitir obtener códigos paralelos eficientes basados en el uso de dichas rutinas. Sin embargo, tener acceso a estas librerías no significa que sean utilizadas eficientemente por científicos no expertos en paralelismo. De ahí que surja la necesidad de estudiar y evaluar su comportamiento en los sistemas donde van a ser ejecutadas.

El propósito de esta tesis de máster es analizar el comportamiento de la rutina de multiplicación de matrices sobre distintos sistemas multicore de memoria compartida mediante el empleo de dos niveles de paralelismo, con el objetivo de obtener una ejecución lo más eficiente posible. Este incremento de eficiencia se conseguirá llevando a cabo un proceso de auto-optimización, que permitirá determinar el número de threads a emplear en cada nivel y ajustar los parámetros de la rutina a las características físicas del sistema de forma que, por un lado, el usuario final pueda integrarlas en su código sin preocuparse de las características del sistema donde se va a ejecutar y garantizando, por otro lado, una ejecución paralela multinivel lo más óptima posible.

El resultado final de este trabajo será la obtención de un prototipo de librería que integre la rutina de multiplicación de matrices y sea capaz de ejecutarse de forma óptima en el sistema empleando dos niveles de paralelismo. El primer nivel de paralelismo se establecerá mediante el API para la programación multiproceso de memoria compartida OpenMP [4], pues los sistemas sobre los que se realizarán las pruebas son sistemas con memoria compartida; el segundo nivel, en cambio, se especificará mediante llamadas a rutinas matriciales de librerías como MKL o ATLAS que, como ya se ha comentado, proporcionan una implementación multithreading de BLAS. La implementación de este prototipo se llevará a cabo evaluando la eficiencia de cada una de estas rutinas sobre los distintos sistemas objeto de estudio, evaluando, a su vez, el uso de varios niveles de paralelismo.

Se pretende que el prototipo diseñado de librería sea el punto de partida para su extensión con niveles adicionales de paralelismo y capacidad de auto-optimización. Todo esto obliga a realizar un diseño lo más modular posible. A su vez, la librería ha de ser capaz de invocar a rutinas de niveles inferiores implementadas para diferentes plataformas y sistemas, como las contenidas en ATLAS o MKL. Puesto que la librería a desarrollar será de libre distribución y código abierto, se ha de dotar con interfaces bien definidas, genéricas y de fácil utilización, similares a las de otras librerías como BLAS, LAPACK… que faciliten el acoplamiento con nuevas versiones desarrolladas una vez finalizada la primera versión.

1.2 Estado del arte y motivación

Esta tesis de máster se va a desarrollar en el marco del Grupo de Computación Científica y Programación Paralela de la Universidad de Murcia [5], que cuenta con una amplia experiencia en el desarrollo, optimización y auto-optimización de código paralelo, así como en la aplicación de la computación paralela en diversos ámbitos científicos.

En 1974 se inicia el desarrollo de librerías básicas de álgebra lineal como BLAS-1, que tras sucesivas aportaciones e implementaciones da lugar al desarrollo de BLAS-2, culminando en la versión de nivel 3 de BLAS en 1986.

Durante la década de los 90 tiene lugar la aparición de otras librerías basadas en BLAS, como es el caso de LAPACK [11], ScaLAPACK [12] y PLAPACK [13], así como librerías multi-threading también basadas en BLAS desarrolladas por fabricantes de computadores, como es el caso de la librería MKL de Intel.

Con el tiempo, el desarrollo e implantación de librerías paralelas de álgebra lineal ha permitido el desarrollo de software paralelo y ha contribuido notablemente a la extensión de la computación paralela de altas prestaciones en la práctica de muchos ámbitos científicos.

Tras un primer recorrido por diversas fuentes de información, nos encontramos ante la utilización, cada vez más patente, de técnicas de programación paralela para la resolución de problemas científicos y de ingeniería en una amplia variedad de campos de conocimiento experimental, como la climatología, dinámica de fluidos, genética o química, entre otros [6, 7, 8, 9, 10].

La relevancia del empleo de librerías matriciales paralelas para el desarrollo de aplicaciones radica en las posibilidades que ofrece respecto a la resolución de problemas de alto coste computacional en el menor tiempo posible, utilizando para ello grandes sistemas de computación, como clusters o multiprocesadores.

Un aspecto que ha permitido la extensión de la programación paralela en diversos ámbitos ha sido, sin lugar a dudas, el avance computacional producido desde los años 80, tanto en lo referente a avances en el diseño de multiprocesadores y computadores paralelos como al desarrollo de librerías y prototipos de software paralelo eficiente con capacidad de auto-optimización, como ATLAS, FFT [14] y SOLAR [15].

Estos avances han permitido a los investigadores proponer y comprobar nuevas técnicas y algoritmos paralelos para solucionar problemas de gran dimensión que requieren de altas capacidades computacionales de cálculo y procesamiento.

El vacío detectado en el desarrollo de librerías paralelas multinivel con capacidad de auto-optimización ha sido la principal motivación para acometer el desarrollo de este trabajo de tesis de máster, en el que se obtendrá un prototipo de librería que va a permitir resolver problemas de álgebra lineal de forma paralela utilizando varios niveles de paralelismo y con capacidad de auto-adaptación al sistema físico donde se utilice.

Aprovechando las iniciales de BLAS y teniendo en cuenta los dos niveles de paralelismo a emplear en el desarrollo de la librería, nombraremos al prototipo a desarrollar a partir de este momento como 2L-BLAS.

En siguientes subapartados se analizarán técnicas generales de optimización así como las posibilidades que existen para sistemas multicore, indicando las librerías que se utilizan para ello. De igual manera, se discutirá la metodología empleada en la aplicación de cada una de estas técnicas.

1.2.1 Técnicas generales de optimización

Los avances en el campo de la computación científica han llevado a la resolución de problemas de gran dimensión con una mayor precisión en los resultados. En este sentido, cabe destacar el desarrollo de librerías como ATLAS, que han permitido dotar al software de una cierta capacidad de auto-modificación de cara a adaptarse lo mejor posible a las condiciones del entorno donde se esté utilizando. Todo ello ha propiciado la aparición de proyectos de desarrollo de software paralelo de álgebra lineal con capacidad de auto-optimización. Entre ellos se puede citar el planteado y desarrollado en la tesis doctoral de J. Cuenca [15], en el que se aplican distintos tipos de ajustes automáticos (número de procesadores, topología lógica, tamaño de bloque, distribución del trabajo, polilibrerías y polialgoritmos) para plataformas paralelas genéricas con el objetivo de optimizar las prestaciones; o el proyecto OSKI (*Optimized Sparse Kernel Interface*) desarrollado en la Universidad de Berkeley en el grupo de investigación BeBOP [16] del profesor James Demmel, que consta de una colección de rutinas de bajo nivel que proporcionan kernels computacionales auto-optimizados para matrices dispersas sobre máquinas monoprocesador superescalares basadas en caché. Asimismo, recientes artículos de investigación [17,18] relacionados con esta temática han sido publicados por J. Cuenca y D. Giménez, miembros del grupo de investigación PCGUM de la Universidad de Murcia, donde presentan, por un lado, una propuesta para auto-optimizar rutinas de álgebra lineal sobre sistemas multicore y, por otro, un estudio de la optimización de la computación densa en álgebra lineal sobre sistemas NUMA mediante auto-optimización del paralelismo anidado.

Actualmente, debido a los continuos avances producidos en el desarrollo de la arquitectura de las unidades de procesamiento gráfico (GPU) por fabricantes como Intel, AMD y NVIDIA; y dado su elevado rendimiento en la realización de operaciones de punto flotante respecto a los procesadores multicore, se está produciendo su integración en sistemas multicore con el objetivo de incrementar las prestaciones en la resolución de problemas de alto coste computacional. Como consecuencia, está teniendo lugar la aparición y desarrollo de nuevas librerías dedicadas a la optimización en este tipo de sistemas, como es el caso de MAGMA [19], que tiene por objetivo desarrollar una biblioteca de álgebra lineal densa similar a LAPACK pero enfocada a arquitecturas heterogéneas/híbridas ("GPU+multicore"), diseñando algoritmos de álgebra lineal y marcos para sistemas manycore híbridos y GPUs que permitan a las aplicaciones explotar completamente las posibilidades que cada uno de los componentes híbridos ofrece.

1.2.2 Optimización en sistemas multicore

Las arquitecturas multicore han permitido a los desarrolladores optimizar aplicaciones mediante el particionamiento inteligente de la carga de trabajo entre los núcleos del procesador. De esta forma, el código de la aplicación puede ser optimizado para usar múltiples recursos del procesador, resultando en un mayor rendimiento de la aplicación [20].

Entre el conjunto de alternativas para realizar optimización en sistemas multicore, podríamos destacar PLASMA [21]. Este proyecto tiene por objeto afrontar la situación crítica a la que se enfrenta la comunidad de álgebra lineal y computación de alto rendimiento debido a la introducción de este tipo de arquitecturas. Su principal objetivo es abordar las deficiencias de rendimiento de las librerías LAPACK y ScaLAPACK en procesadores multicore y sistemas de multiprocesadores multi-socket, realizando, en los casos en que sea posible, una reestructuración del software para lograr una eficiencia mucho mayor y una alta portabilidad sobre un amplio rango de nuevas arquitecturas. No obstante, no puede considerarse un sustituto inmediato de ambas librerías, pues presenta una funcionalidad limitada (resuelve problemas específicos) y, por ahora, sólo es compatible en sistemas de memoria compartida.

Asimismo, también se han desarrollado auto-optimizadores para *stencils* [22], pequeños núcleos computacionales que permiten la resolución de ecuaciones en derivadas parciales aplicando operaciones sobre puntos adyacentes en una malla estructurada; y que puede verse como un caso muy específico, pero eficiente, de la multiplicación matriz-vector disperso. Las técnicas aplicadas están destinadas a la optimización de la asignación de datos en memoria, del ancho de banda de memoria y de la tasa de cálculo producida en las mallas de puntos.

1.3 Plan de trabajo

Tal y como se ha indicado en la introducción de la presente tesis, el interés creciente mostrado por la comunidad científica en la utilización eficiente de rutinas básicas de álgebra lineal sobre sistemas paralelos lleva a pensar que el trabajo que ahora se inicia puede tener una clara proyección futura. Motivados por tal interés, el principal objetivo va a consistir en elaborar un prototipo de librería denominado 2L-BLAS que incluya, inicialmente, la rutina de multiplicación de matrices con dos niveles de paralelismo (OpenMP+BLAS) y estudiar su comportamiento sobre distintos sistemas multicore de memoria compartida aplicando un proceso de auto-optimización que permita determinar el número de threads a emplear en cada nivel con el fin de obtener una ejecución lo más eficiente posible. Para alcanzar dicho objetivo, se propone el siguiente plan de trabajo, que podrá aplicarse sobre otras rutinas matriciales (LU, QR…), enriqueciendo así el prototipo 2L-BLAS:

1. En primer lugar se analizará el comportamiento que experimenta la rutina de multiplicación de matrices al ejecutarla sobre distintos sistemas multicore utilizando un único nivel de paralelismo (varios threads OpenMP o MKL).

2. A partir del análisis realizado en el punto anterior y de [23, 24], se validarán los resultados obtenidos sobre los sistemas multicore considerados y se procederá al diseño del prototipo de librería 2L-BLAS, encargado de implementar la rutina de multiplicación de matrices utilizando dos niveles de paralelismo (OpenMP+BLAS).

3. A continuación se realizará un estudio experimental variando el tamaño y la forma de las matrices y el número de threads en cada uno de los niveles de paralelismo establecidos, con el fin de comprobar el funcionamiento del prototipo diseñado y evaluar sus prestaciones.

4. Finalmente se analizarán distintas técnicas de auto-optimización y se procederá a aplicarlas a la librería sobre distintos sistemas, comprobando su capacidad de auto-adaptación a las características de los mismos y determinando si alcanza las prestaciones esperadas una vez utilizada sobre dichos sistemas.

Puesto que los puntos 2, 3 y 4 van a constituir el grueso del trabajo a realizar, los siguientes apartados están destinados a explicar más detenidamente la metodología a utilizar en cada uno de ellos.

1.3.1 Implementación del prototipo de librería 2L-BLAS

El prototipo a diseñar va a consistir en una librería que implemente inicialmente la rutina de multiplicación de matrices con 2 niveles de paralelismo (OpenMP+BLAS).

Esta rutina se programará a partir del prototipo establecido en las rutinas de la librería BLAS y recibirá, en una primera aproximación, dos parámetros adicionales que permitirán establecer el número de threads en cada uno de los niveles de paralelismo. No obstante, se podría haber indicado un solo parámetro con el número máximo de threads a usar o incluso no haber indicado ninguno, dejando que sea la propia rutina la que determine el número más apropiado de threads a establecer en cada nivel mediante un proceso de toma de decisiones. En el diseño propuesto se ha optado por especificar ambos parámetros para facilitar tanto el diseño como el uso de la propia rutina, pero más adelante será necesario modificarlo, pues se incorporarán en la rutina técnicas de auto-optimización con el fin de que la propia rutina sea capaz de determinar el número más adecuado de threads a emplear en cada nivel de paralelismo.

En cuanto a la funcionalidad de la propia rutina, la operación a realizar consiste en un producto matricial utilizando dos niveles de paralelismo. Esta operación se realiza multiplicando el bloque de filas adyacentes de A asignado a cada uno de los threads OpenMP por una matriz B sobre la que trabajarán tantos threads como se hayan establecido para el segundo nivel, invocando para ello a la rutina `dgemm` de BLAS.

1.3.2 Estudio del paralelismo de 2 niveles en distintos sistemas

Para estudiar el paralelismo de 2 niveles en cada uno de los sistemas multicore indicados, se partirá del estudio realizado en [23] y se completará mediante la realización de diversas pruebas adicionales, variando el número de threads empleados en cada nivel, para distintos tamaños y formas

del problema. A continuación se analizarán y compararán los resultados obtenidos en cada uno de los sistemas y se determinará para cada caso el número adecuado de threads a emplear en cada nivel, de forma que se consiga el menor tiempo de ejecución posible. De la misma forma, se intentará determinar los parámetros del algoritmo que afectan al rendimiento de la aplicación, tomando como objetivo futuro su ajuste automático para conseguir acelerar de forma óptima la ejecución de la rutina.

1.3.3 Ajuste, auto-ajuste y validación de la librería en distintos sistemas

En esta última fase se analizarán distintas técnicas de optimización con el fin de comprobar si el tiempo de ejecución obtenido y el número de threads establecido en cada nivel de paralelismo por la librería 2L-BLAS corresponde con los estimados en el estudio teórico-experimental, es decir, si el ajuste automático realizado por la librería obtiene la combinación adecuada de threads OpenMP y MKL que permite conseguir el menor tiempo de ejecución obtenido en el estudio realizado en el apartado anterior, procediendo, en tal caso, al ajuste de dichos parámetros en función de las características del sistema donde se vaya a ejecutar la rutina dgemm2L.

Finalmente, se realizarán diversas pruebas sobre cada uno de los sistemas multicore con el fin de validar las técnicas propuestas, dejando abierta la posible ampliación del prototipo de librería añadiendo nuevas rutinas matriciales (LU, QR…)

1.4 Entorno de trabajo

En esta sección se describe el marco de trabajo que se ha utilizado para la realización y evaluación de las pruebas, esto es, las plataformas paralelas de propósito general y los compiladores y librerías de software de álgebra lineal utilizados.

En primer lugar se muestran las características de los sistemas multicore que se han empleado para llevar a cabo los experimentos, indicando para cada uno de ellos el entorno software de trabajo (compiladores y librerías) utilizado:

- **Ben**: HP Superdome con 1.5 TB de memoria compartida con procesadores Intel-Itanium-2 Dual-Core Montvale a 1.6 GHz y un total de 128 cores. Se encuentra alojado en el Centro de Supercomputación de la Fundación Parque Científico de Murcia (FPCMUR). Para la realización de las pruebas se utiliza la librería MKL de Intel, versión 10.2 y el compilador icc de Intel, versión 11.1.

- **Pirineus**: SGI Altix UV 1000 con 6.14 TB de memoria compartida con procesadores Intel Xeon X7542 (hexa-core) a 2.67 GHz y un total de 1344 cores. Se halla en el Centro de Supercomputación de Cataluña (CESCA) y se ha tenido acceso limitado al mismo, pudiendo usar como máximo un total de 256 cores. Para la realización de las pruebas se emplea la librería MKL de Intel en su versión 10.2 y el compilador icc de Intel, versión 11.1.

- **Saturno**: multiprocesador con 32 GB de memoria compartida con procesadores Intel Xeon E7530 a 1.87GHz y un total de 24 cores (4 hexa-cores). Se encuentra situado en el laboratorio de Computación Científica y Programación Paralela de la Universidad de Murcia. Para la realización de los experimentos se ha utilizado la librería MKL de Intel, versión 10.3.2 y el compilador icc de Intel, versión 12.0.2.

En los anexos 1 y 2 se muestran las arquitecturas de los sistemas Saturno y Ben, respectivamente, con el fin de tener una visión general de los mismos que sirva de referencia en aquellos casos en que sea necesario para tomar decisiones u obtener conclusiones al realizar y evaluar los experimentos.

Finalmente, cabe citar en este marco de trabajo un interfaz software auxiliar que ha sido de gran utilidad para el desarrollo del presente trabajo. Se trata de un interfaz en C llamado CBLAS [25], que permite invocar a cualquier rutina de BLAS como si de una función de la librería C se tratase, ocultando de esta forma determinados aspectos relativos a la programación en Fortran, como el almacenamiento por columnas de las matrices en memoria o el paso de parámetros por referencia. Este interfaz se ha usado para invocar a la función `dgemm` de BLAS desde la rutina `dgemm2L` del prototipo de librería 2L-BLAS, siendo necesario, para ello, tener en cuenta los siguientes aspectos:

- Al comienzo de la cabecera de la rutina de BLAS invocada, se ha de especificar el valor del parámetro `const enum CBLAS_ORDER Order`, donde `CBLAS_ORDER` es un tipo enumerado compuesto por los valores `CblasRowMajor=101` o `CblasColMajor=102`, e indica si la matriz está almacenada por filas o columnas.

- Se ha de indicar si las matrices con las que se va a trabajar han sido transpuestas, utilizando para ello sendos argumentos a continuación de `Order` con uno de los siguientes valores: `CblasNoTrans=111` o `CblasTrans=112`.

1.5 Estructura de la tesis

En el capítulo 2 se describe el diseño, funcionalidad y uso de la rutina `dgemm2L` de multiplicación de matrices implementada utilizando dos niveles de paralelismo (OpenMP+BLAS), que constituye el punto de partida en la elaboración del prototipo de librería 2L-BLAS. Asimismo, se indican posibles extensiones a realizar, como la creación de la librería estática `lib2lblas.a`, la adición de nuevas rutinas al prototipo propuesto o el uso de la rutina `dgemm2L` desde programas escritos en Fortran.

En el capítulo 3 se analiza el comportamiento de la rutina `dgemm2L` y se evalúan las prestaciones obtenidas al usar 1 y 2 niveles de paralelismo sobre distintos sistemas multicore de memoria compartida. Se analizan los resultados obtenidos tras habilitar/deshabilitar el paralelismo dinámico en la realización de las pruebas, así como los obtenidos al considerar matrices con distinto tamaño y forma (rectangulares/cuadradas), indicando, a su vez, los detalles de la arquitectura del sistema que afectan al rendimiento de la aplicación y que van a justificar, en un capítulo posterior, el uso de técnicas de auto-optimización para conseguir una ejecución de la rutina lo más eficiente posible

En el capítulo 4 se aborda el proceso de auto-optimización de la rutina de multiplicación de matrices implementada en el capítulo 2. El objetivo de este capítulo es, por tanto, justificar la conveniencia de incluir un proceso de auto-optimización previo a la ejecución de la rutina que permita determinar el número de threads a establecer en cada nivel así como los parámetros del algoritmo a utilizar. Se describe detalladamente el diseño propuesto para la rutina de multiplicación de matrices con capacidad de adaptación a las condiciones del sistema, que se manifestará mediante la elección de los valores apropiados para una serie de parámetros configurables de la rutina, que en este caso son el número de threads OpenMP y MKL a utilizar. Se describirá la estructura y funcionamiento del sistema software construido y se mostrarán algunos resultados experimentales obtenidos con dicha rutina sobre diferentes sistemas multicore.

En el capítulo 5, en primer lugar se destacan las principales conclusiones y resultados obtenidos con la realización de esta tesis de máster y, finalmente, se relacionan algunas posibles líneas futuras de trabajo.

Capítulo 2. Diseño de la rutina dgemm2L

2.1 Introducción

El objetivo del presente capítulo es explicar cómo se ha llevado a cabo la codificación de la rutina de multiplicación de matrices `dgemm2L` usando dos niveles de paralelismo (OpenMP+BLAS). Esta rutina constituye, a su vez, el punto de partida en la elaboración del prototipo de librería 2L-BLAS.

En primer lugar se explicarán los pasos que se han seguido en el diseño de la rutina así como la funcionalidad que presenta. A continuación se indicará el programa de prueba utilizado para comprobar su funcionamiento y, finalmente, se expondrá cómo se puede extender la librería con nuevas rutinas, así como determinados aspectos a tener en cuenta al compilar la rutina `dgemm2L` desde C e invocarla desde Fortran.

2.2 Diseño de la rutina

El diseño de la rutina se ha realizado usando el lenguaje de programación C y se ha definido siguiendo la filosofía de las rutinas ofrecidas por BLAS, en concreto, por la función `dgemm` de BLAS:

```
dgemm(transA,transB,m,n,k,alpha,A,lda,B,ldb,beta,C,ldc)
```

donde `transA` y `transB` indican la forma de las matrices A y B a ser usada en la multiplicación ('N' = normal, 'T' = transpuesta); `m` es el número de filas de las matrices A y C; `n` es el número de columnas de las matrices B y C; `k` es el número de columnas de la matriz A y el número de filas de la matriz B; `alpha` y `beta` son escalares utilizados para la realización del producto matricial (C = alpha*A*B + beta*C); `A`, `B` y `C` son las matrices a utilizar en la realización del producto matricial y `lda`, `ldb`, `ldc` indican el *leading-dimension* de las matrices A, B y C, respectivamente.

El Anexo 3 muestra el código de la rutina implementada, cuya cabecera incorpora dos parámetros adicionales: `thrOMP` y `thrMKL`, para permitir especificar el número de threads a usar en cada nivel de paralelismo:

```
dgemm2L(char transA, char transB, int m, int n, int k, double alpha,
        double *A, int lda, double *B, int ldb, double beta,
        double *C,  int ldc, int thrOMP, int thrMKL)
```

Respecto al cuerpo de la función, la rutina se ha codificado siguiendo el esquema general de paralelismo anidado mostrado a continuación:

```
omp_set_nested(1);                          //Habilita paralelismo anidado
omp_set_num_threads(nthomp);                //Establece threads OpenMP
mkl_set_dynamic(0);                         //Deshabilita paralelismo dinámico
mkl_set_num_threads(nthmkl);                //Establece threads MKL

#pragma omp parallel
{
    obtener el tamaño y la posición inicial
    de la submatriz de A a ser multiplicada

    invocar a la rutina dgemm para multiplicar
    la submatriz de A por la matriz B
}
```

Como se puede apreciar en el Anexo 3, la codificación de la rutina se ha estructurado en dos partes claramente diferenciadas: comprobación de parámetros y realización del cálculo.

Parte 1: En la parte de comprobación de parámetros se chequean los tamaños especificados para las matrices A y B y se ajusta el *leading-dimension* de cada una de ellas al valor adecuado en función del tipo de matriz especificado (Normal, Transpuesta).

Puesto que se van a emplear dos niveles de paralelismo, a continuación se establecen el número de threads OpenMP y MKL a usar en el producto matricial y seguidamente se pasa a la realización del mismo.

Parte 2: En la fase de cálculo, para cada thread OpenMP, se halla el número de filas de la matriz A que le corresponden y la posición de comienzo. A continuación, se invoca a la rutina `dgemm` de BLAS para realizar el producto de la porción de A por la matriz B, utilizando en esta multiplicación el número de threads MKL especificado.

2.3 Uso de la rutina

La funcionalidad de la rutina `dgemm2L` implementada en el prototipo de librería 2L-BLAS se ha comprobado mediante la creación del programa de prueba `dgemm2Lx.c`, siguiendo así el estándar ofrecido por BLAS para los programas de prueba. El Anexo 4 muestra el código de dicho programa. Como se puede apreciar en este anexo, la finalidad del programa es probar que el resultado obtenido al realizar la multiplicación de matrices utilizando 2 niveles de paralelismo es el mismo que el que se obtendría al realizar el producto utilizando la rutina estándar de multiplicación de matrices de BLAS con diferentes combinaciones de los parámetros de configuración de entrada. Para ello, en primer lugar se solicita al usuario las dimensiones de las matrices A y B, los valores de los escalares alpha y beta, el número de threads OpenMP y MKL a usar y si quiere trasponer alguna de las matrices. A continuación, se crean e inicializan las matrices A, B y C necesarias para el cálculo del producto matricial y se realiza dicha operación tanto en secuencial como en paralelo, imprimiendo el tiempo de ejecución empleado en ambos casos y un mensaje de comprobación de los resultados obtenidos.

2.4 Creación de la librería

En las secciones anteriores se ha explicado el diseño de la rutina `dgemm2L` y cómo se puede probar su funcionalidad. Sin embargo, para que pueda ser utilizada desde cualquier directorio de trabajo por cualquier miembro de la comunidad científica, se ha de crear el fichero de librería correspondiente. Por esta razón, a continuación se muestra un ejemplo de creación de la librería estática `lib2lblas.a` en la máquina Saturno. Su creación en otros sistemas se realizaría de forma similar:

```
icc  -c –o dgemm2L.o dgemm2L.c -O3 -I$MKLINCLUDE -I$MKLINCLUDE/lp64
     -L$MKLPATH-lmkl_blas95_lp64 -lmkl_intel_lp64 -lmkl_intel_thread
     -lmkl_solver_lp64 -lmkl_core -liomp5 -openmp -lpthread -lm

ar   rcs lib2lblas.a dgemm2L.o
```

A su vez, durante la fase de compilación se han de especificar los siguientes flags para que el programa que invoque a la rutina `dgemm2L` encuentre tanto la propia librería como los ficheros de cabecera necesarios:

```
-I/opt/intel/include –L/opt/intel/lib –l2lblas
```

Nótese que se ha indicado el directorio `/opt/intel` como referencia, pues es donde se encuentran instalados en Saturno las librerías y el compilador `icc` de Intel. Por tanto, este parámetro puede variar en cada sistema.

Otra posible opción, puesto que la librería a desarrollar es de libre distribución y código abierto, consistiría en proporcionar los ficheros de código fuente al usuario y dejar que sea él mismo el encargado de realizar esta tarea.

2.5 Extensiones

En esta sección se exponen algunas extensiones que se pueden aplicar al prototipo de librería propuesto con el fin de ampliar la funcionalidad ofrecida por el mismo. Se presentará, por un lado, cómo añadir nuevas funciones al prototipo 2L-BLAS y, por otro, cómo estas rutinas pueden ser utilizadas desde programas escritos en Fortran.

2.5.1 Ampliación del prototipo 2L-BLAS

Una vez diseñada la rutina `dgemm2L` y probado su funcionamiento, se podría aplicar el mismo esquema de diseño-implementación-chequeo en el diseño de las rutinas `sgemm2L` (para números reales en simple precisión) y `cgemm2L` (para números complejos de 8 bytes) o `zgemm2l` (para números complejos de 16 bytes), ampliando así la funcionalidad del prototipo de librería 2L-BLAS.

En el caso de rutinas que trabajan con números complejos, se ha de tener en cuenta que C no implementa directamente los tipos `COMPLEX(4)` y `COMPLEX(8)` de Fortran para números complejos de 4 y 8 bytes, respectivamente. Sin embargo, se pueden utilizar estructuras como las proporcionadas por la librería MKL de Intel. Esta librería proporciona los tipos de datos `MKL_Complex8` y `MKL_Complex16`, que se encuentran definidos en el fichero de cabecera `mkl_types.h` como estructuras de datos equivalentes a los tipos `COMPLEX(4)` y `COMPLEX(8)` de Fortran:

```
/* Complex type (single precision) */
typedef struct _MKL_Complex8 {
    float    real;
    float    imag;
} MKL_Complex8;


/* Complex type (double precision) */
typedef struct _MKL_Complex16 {
    double    real;
    double    imag;
} MKL_Complex16;
```

Por tanto, para trabajar con rutinas que manejen este tipo de datos, tan solo es preciso declarar las variables y punteros de tipo float/double a usar, con los tipos `MKL_Complex8` o `MKL_Complex16`, inicializándolos adecuadamente accediendo a los campos de la estructura definida por el tipo e invocando a la rutina siguiendo el prototipo definido por la librería BLAS.

2.5.2 Uso de las funciones del prototipo 2L-BLAS desde Fortran

Por otra parte, dado que una gran cantidad de aplicaciones científicas se desarrollan utilizando el lenguaje de programación Fortran, se ha de proporcionar un interfaz que permita invocar a las rutinas en C de la librería 2L-BLAS desde dichos programas.

Para ello, el nombre de las funciones de la librería 2L-BLAS ha de estar escrito completamente en minúsculas seguido de un guión bajo ('_'), que en el código Fortran no se indicará al realizar la llamada. Además, como en Fortran las variables siempre se pasan por referencia, las funciones de 2L-BLAS que sean invocadas desde Fortran deben esperar que todos sus argumentos sean punteros, lo que requeriría diseñar una versión específica de la rutina de multiplicación de matrices `dgemm2L` para su uso desde Fortran.

El Anexo 5 muestra un ejemplo de un programa escrito en el lenguaje Fortran que hace uso de la función `dgemm2L` implementada en el prototipo de librería 2L-BLAS.

Capítulo 3. Evaluación de prestaciones

3.1 Introducción

Una vez codificada la rutina `dgemm2L` y comprobado su funcionamiento, utilizando para ello distintas configuraciones de entrada, el objetivo de este capítulo va a consistir en llevar a cabo un estudio empírico del comportamiento de dicha rutina al variar el tamaño y forma de las matrices y el número de threads puestos a funcionar en cada nivel de paralelismo, evaluando sus prestaciones en distintos sistemas y determinando los parámetros que van a influir en el tiempo de ejecución. Todo ello justificará el aplicar un proceso de auto-optimización con el fin de obtener una ejecución lo más eficiente posible.

En un principio, es posible pensar que la mejor opción sería usar directamente paralelismo MKL con determinación dinámica de threads y sin paralelismo OpenMP, o usar incluso un número de threads igual al número de cores disponibles, pero como se verá en los experimentos, dependiendo del tamaño y forma de las matrices y del sistema computacional, se pueden obtener tiempos de ejecución menores usando dos niveles de paralelismo, con diferente número de threads en cada nivel.

Otra opción sería usar solamente paralelismo OpenMP. Esto queda reflejado en la Figura 1, donde se muestra, para cada número de threads n, la ganancia obtenida al emplear únicamente paralelismo OpenMP respecto al uso exclusivo de paralelismo MKL, es decir, el tiempo de ejecución obtenido con 1 thread OpenMP y n threads MKL dividido entre el tiempo obtenido con n threads OpenMP y 1 thread MKL. Como se puede observar, los valores obtenidos para los sistemas mostrados se mantienen en torno a 1, y a pesar de ser ligeramente superiores en Ben que en Saturno, en ningún caso permiten obtener una mejora significativa en el speed-up, por lo que dicha opción no será tenida en cuenta a la hora de realizar los experimentos.

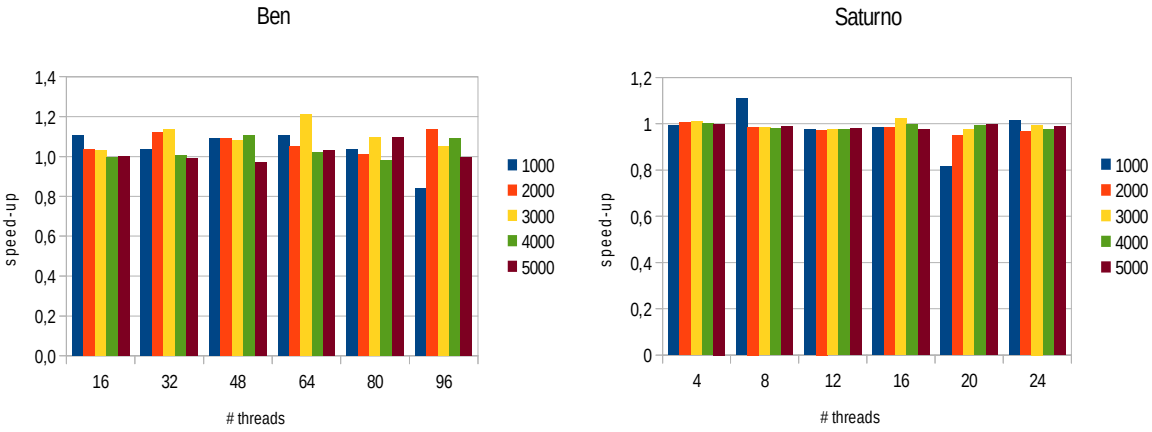


Figura 1. Speed-up obtenido al emplear paralelismo OpenMP respecto al uso de paralelismo MKL en Ben y Saturno, variando el número de threads y el tamaño de las matrices

Así pues, el presente capítulo comenzará con el estudio del comportamiento de la rutina `dgemm2L` usando únicamente el paralelismo interno de la rutina MKL sobre distintos sistemas, al usar distintos tamaños y formas de matrices y habilitando/deshabilitando el paralelismo dinámico. A continuación se analizarán los resultados y se pasará a la realización del mismo tipo de experimentos usando dos niveles de paralelismo (OpenMP+MKL), comparando los resultados obtenidos e indicando los factores que intervienen en la mejora o caída del rendimiento.

Las figuras mostradas en los experimentos realizados, representan el speed-up alcanzado respecto a la ejecución secuencial (1 thread OpenMP - 1 thread MKL) para distintos tamaños de las matrices. Asimismo, el número de threads establecido en cada nivel de paralelismo varía en cada una de las ejecuciones hasta alcanzar el máximo número de cores disponibles en cada sistema, de forma que se pueda estudiar la tendencia seguida en cada uno de ellos al variar el tamaño y forma de las matrices para cada combinación de threads establecida.

Respecto a Saturno, indicar que para poder evaluar correctamente el comportamiento de la rutina `dgemm2L` se han utilizado matrices de tamaño *nxn* y dimensión inferior a 3000, pues a partir de ese valor el sistema comienza a utilizar la memoria secundaria como respaldo de almacenamiento de las matrices y el rendimiento cae en picado. Este fenómeno se manifiesta de igual forma tanto al utilizar paralelismo MKL como al usar paralelismo OpenMP+MKL, tal como muestra la Figura 2.

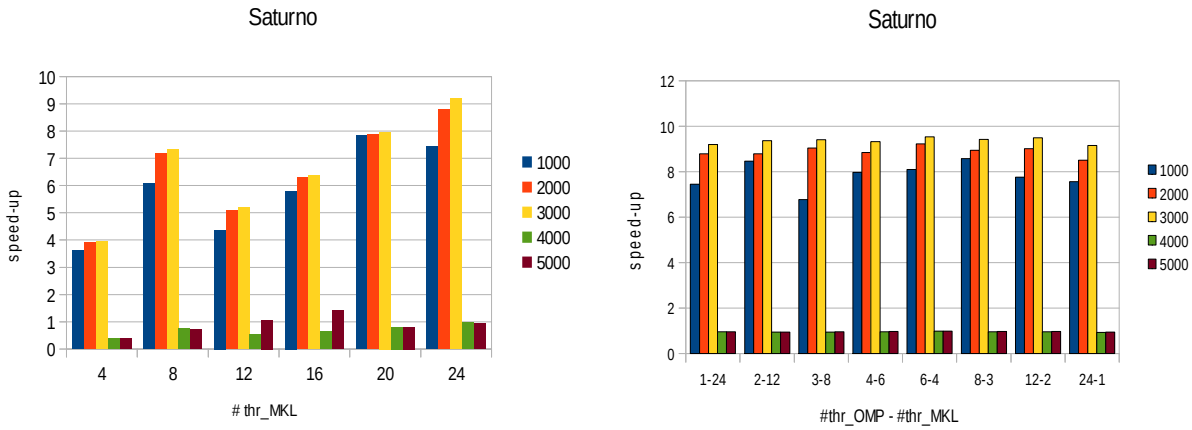


Figura 2. Caída del speed-up experimentada en Saturno cuando la rutina `dgemm2L` se ejecuta con matrices de tamaño nxn y dimensión superior a 3000

En cuanto a Pirineus, comentar que solo se mostrarán gráficos para aquellos casos en que fue posible realizar pruebas durante el acceso temporal que el Centro de Supercomputación de Cataluña (CESCA) ofreció para el uso de dicho sistema.

3.2 Paralelismo MKL

En esta sección se estudia el comportamiento que experimenta la rutina `dgemm2L` sobre distintos sistemas al emplear un solo nivel de paralelismo. Esto se consigue fijando a 1 el número de threads OpenMP e invocando a la función `mkl_set_num_threads(n)` de la librería MKL de Intel con el número de threads MKL a usar (indicado por la variable `n`) al invocar a la función `dgemm` de BLAS, tal y como quedó explicado en el apartado 2.2.

Así pues, en primer lugar se analizan y comparan los resultados obtenidos tras ejecutar las pruebas según se lleven a cabo con y sin selección dinámica de threads y a continuación se realiza lo propio para matrices cuadradas y rectangulares.

3.2.1 Dinámico vs. No dinámico

Tal y como se comentó en la introducción de la presente tesis de máster, la existencia de librerías multithreading de álgebra lineal como ATLAS y MKL, que ofrecen en su implementación la posibilidad de habilitar el paralelismo dinámico, va a permitir decidir en tiempo de ejecución (en función del número de cores disponibles o del tamaño del problema) el número más apropiado de threads a usar de entre los establecidos al invocar a la función correspondiente de la librería encargada de habilitar dicho tipo de paralelismo. Para el caso que nos ocupa, el primer nivel de paralelismo se establecerá usando la librería MKL de Intel, por tanto, para habilitar el paralelismo dinámico en dicho nivel habrá que invocar a la función `mkl_set_dynamic(enable)`, donde la variable `enable` toma el valor 0 o 1 en función de si se quiere deshabilitar o habilitar esta característica, respectivamente.

La Figura 3 muestra el speed-up alcanzado por la rutina `dgemm2L` al ejecutarse con y sin selección dinámica de threads en distintos sistemas.

En Ben, el comportamiento experimentado es muy similar en ambos casos. Se observa cómo conforme aumenta el número de threads MKL y el tamaño de las matrices, se produce un incremento del speed-up, alcanzando su máximo valor en la ejecución con 48 threads y tamaño 5000 de las matrices. A partir de este punto, el speed-up comienza a decrecer pero mejora al aumentar el tamaño de las matrices, quedando, eso sí, bastante lejos del óptimo. No obstante, en cualquier caso, la habilitación del paralelismo dinámico no va a suponer una mejora en el rendimiento de la rutina.

En Saturno, en cambio, al habilitar el paralelismo dinámico se observa una mejora sustancial del speed-up en las ejecuciones con 12 y 16 threads para tamaños 2000 y 3000 de las matrices, pero en el resto de ejecuciones su comportamiento es muy similar al obtenido con el paralelismo dinámico deshabilitado. Asimismo, se aprecia cómo para un número reducido de threads MKL y tamaños grandes de las matrices, el speed-up alcanzado se aproxima bastante al óptimo, pero este valor se aleja conforme aumenta el número de threads, lo que lleva a pensar que el sistema dedica bastante tiempo a la gestión y comunicación de los threads en memoria.

Respecto a Pirineus, el gráfico mostrado refleja un comportamiento muy similar al habilitar y deshabilitar el paralelismo dinámico, aunque con pequeñas fluctuaciones en las ejecuciones con matrices de tamaño 5000. No obstante, se observa que en ambos casos el máximo valor del speed-up se obtiene en la ejecución con 60 threads y a partir de ahí comienza a decrecer. Por tanto, la habilitación del paralelismo dinámico en este sistema no va a ser un factor determinante en la mejora del rendimiento de la aplicación.

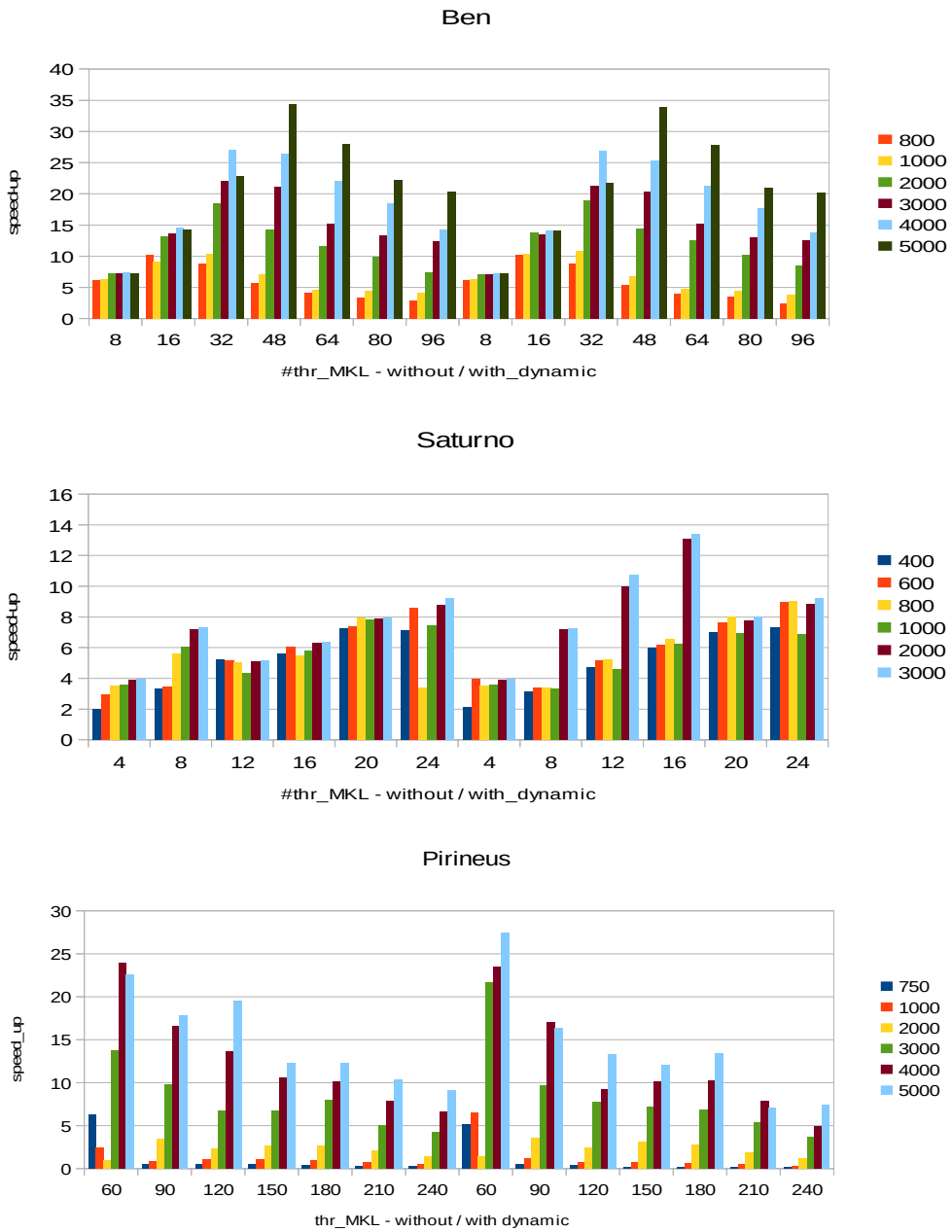


Figura 3. Speed-up obtenido en distintos sistemas al habilitar y deshabilitar el paralelismo dinámico en la ejecución de la rutina `dgemm2L`, variando el número de threads MKL y el tamaño de las matrices

A la vista de los resultados obtenidos en cada uno de los sistemas, se puede concluir que la habilitación del paralelismo dinámico utilizando únicamente paralelismo MKL no constituye una opción que aporte mejoras significativas en el rendimiento de la rutina cuando aumenta el número de threads y el tamaño de las matrices. Esto queda reflejado en la Figura 4, donde se muestra el speed-up obtenido con selección dinámica de threads respecto a la ejecución estática. Como se puede observar, la ganancia alcanzada se mantiene en torno a 1 en prácticamente todas las ejecuciones realizadas con cada uno de los tamaños. No obstante, también se pueden apreciar pequeñas fluctuaciones, que pueden ser debidas a que los experimentos solo se han realizado una vez con el fin de no incrementar excesivamente el tiempo dedicado a la experimentación.

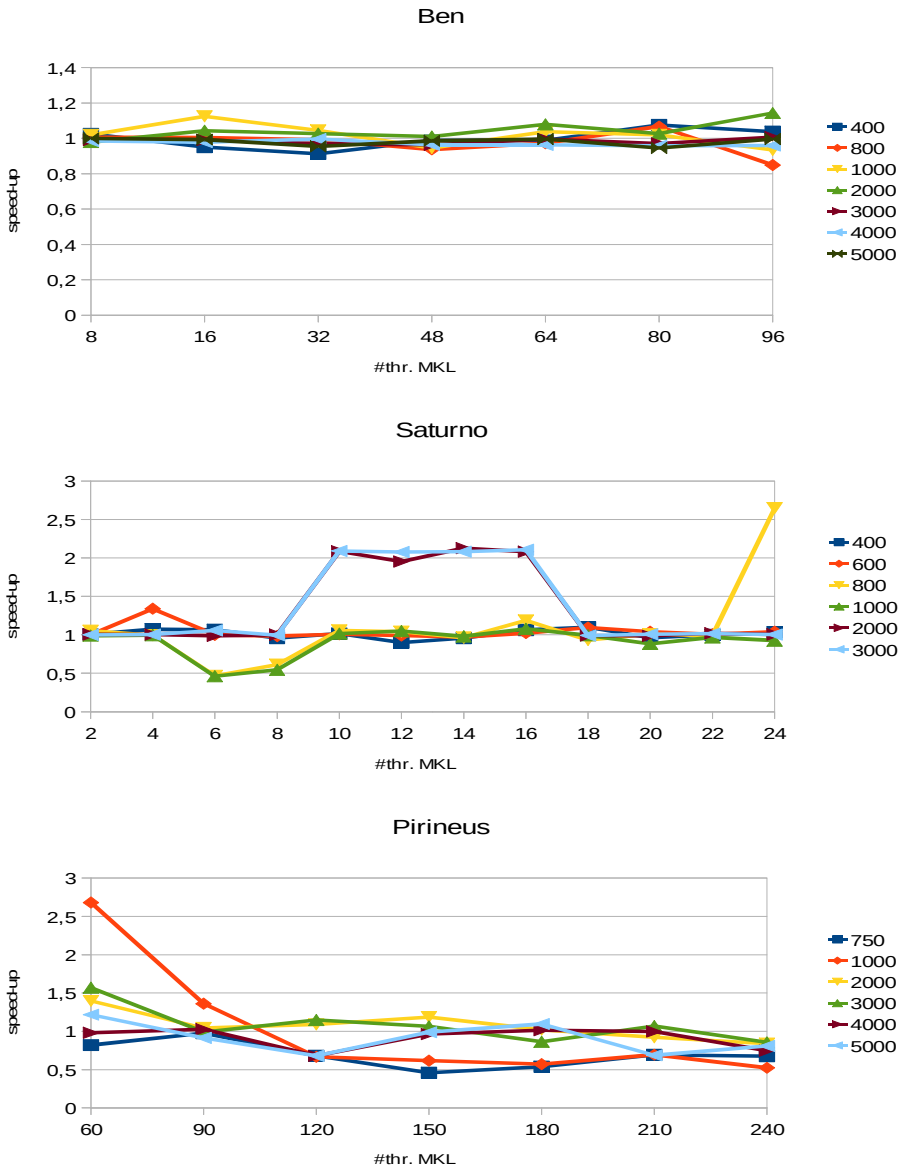


Figura 4. Speed-up obtenido al emplear paralelismo dinámico respecto al uso de paralelismo estático MKL en distintos sistemas, variando el número de threads y el tamaño de las matrices

3.2.2 Matrices cuadradas vs. Matrices rectangulares

Una vez estudiado el comportamiento de distintos sistemas al ejecutar la rutina `dgemm2L` habilitando y deshabilitando el paralelismo dinámico MKL, se va a realizar un análisis orientado a determinar si estos sistemas experimentan grandes cambios en su comportamiento cuando las matrices usadas por la rutina `dgemm2L` presentan distintas formas (cuadradas, rectangulares) y tamaños. Los resultados de dicho análisis quedan reflejados en las Figuras 5 y 6, donde se muestran los gráficos correspondientes a los sistemas Saturno y Ben, respectivamente, con el speed-up alcanzado por la rutina `dgemm2L` al ejecutarse con matrices cuadradas y rectangulares.

En ambos sistemas se observa que cuando la matriz B es rectangular, se obtiene un speed-up superior respecto al alcanzado cuando A es rectangular, siendo además en Saturno superior al conseguido con matrices cuadradas. Posiblemente, esto sea debido al uso de paralelismo MKL sobre la matriz B y al menor número de fallos producidos en caché por el menor tamaño de los bloques de B. Por otro lado, cuando en ambos sistemas las matrices son cuadradas, el comportamiento en Saturno se asemeja al experimentado cuando A es rectangular, mientras que en Ben sucede lo contrario, es decir, su comportamiento es muy similar al mostrado cuando B es rectangular. Por tanto, se puede afirmar que el empleo de matrices rectangulares en Saturno no va a suponer en ningún caso una reducción del speed-up alcanzado por la rutina `dgemm2L`. En Ben, además, la rutina MKL no obtiene el mismo nivel de eficiencia paralela cuando la matriz A no tiene forma cuadrada, por lo que en cualquier caso, como las diferencias no son muy significativas, la mayoría de los experimentos se realizarán utilizando matrices cuadradas.

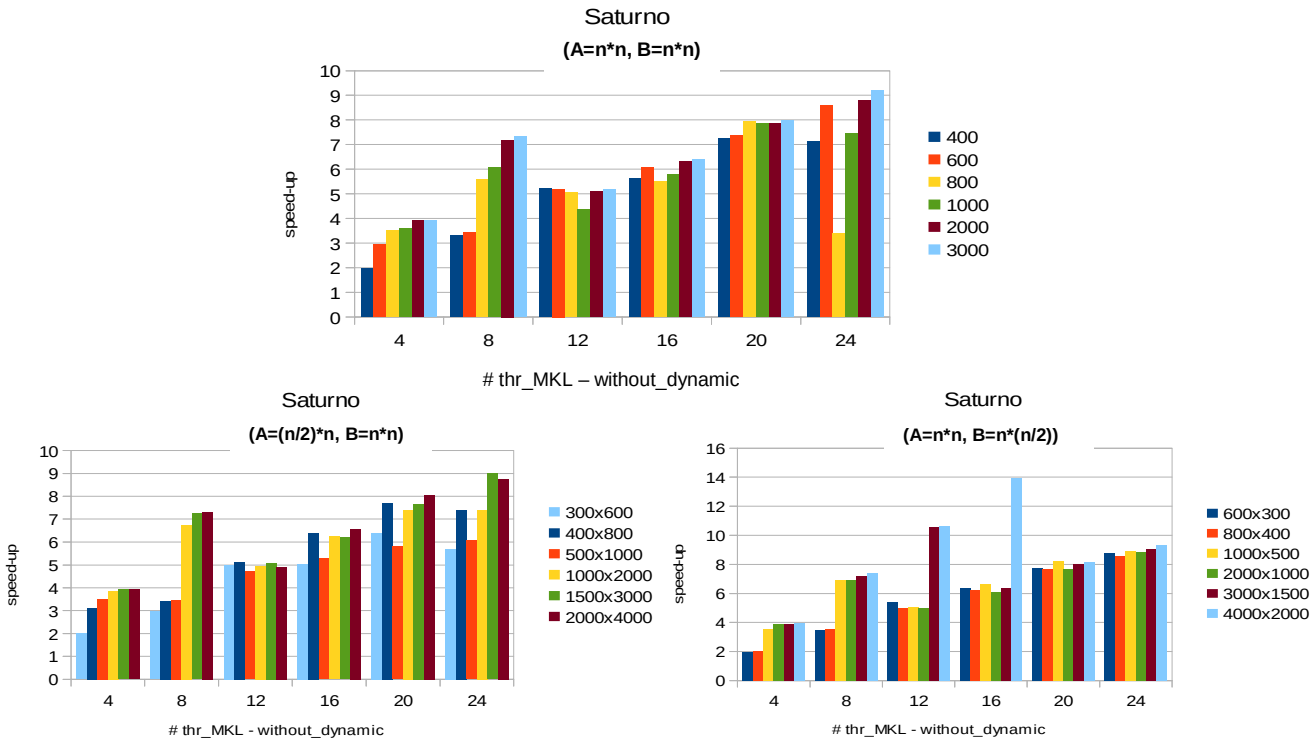


Figura 5. Speed-up obtenido en Saturno al ejecutar la rutina `dgemm2L` con matrices cuadradas y rectangulares de distintos tamaños

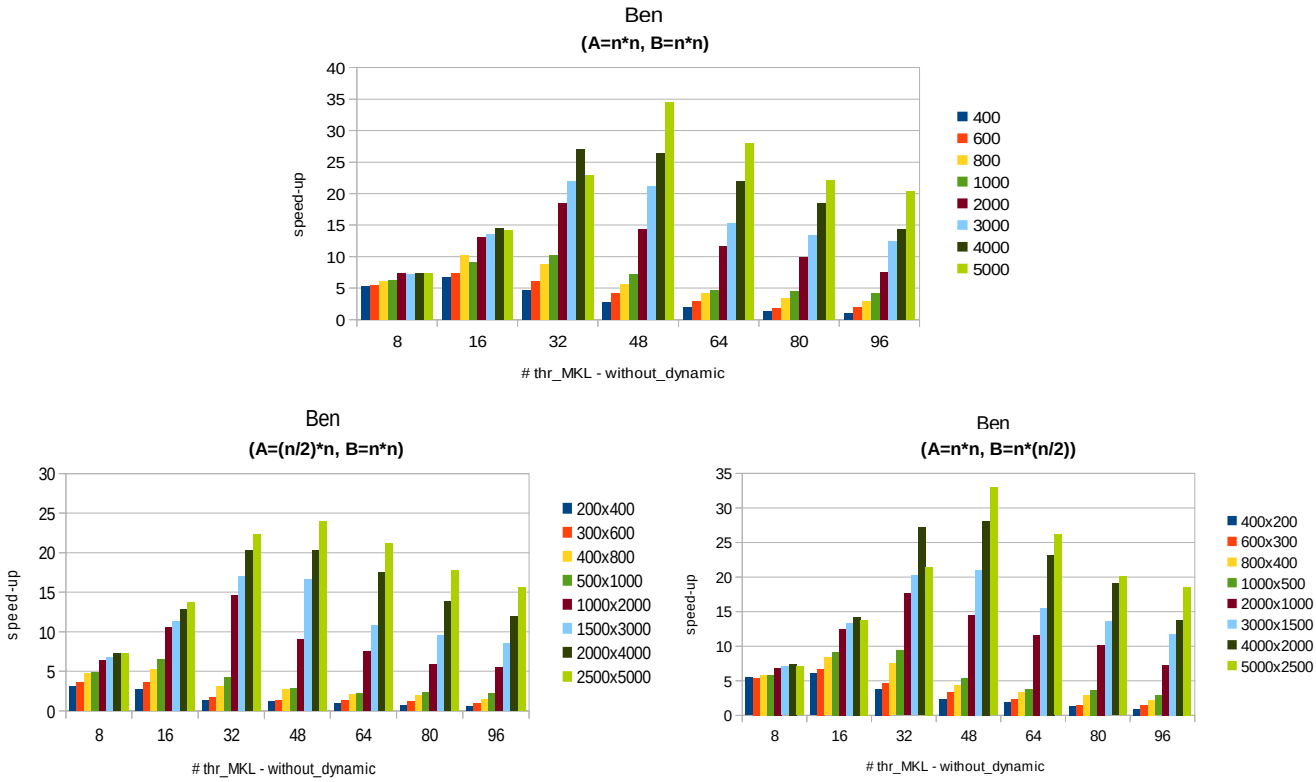


Figura 6. Speed-up obtenido en Ben al ejecutar la rutina `dgemm2L` utilizando matrices cuadradas y rectangulares de distintos tamaños

3.2.3 Resumen y conclusiones

Esta primera parte del capítulo 3 se ha centrado en el estudio del paralelismo MKL que experimenta la rutina `dgemm2L` al ejecutarse con 1 thread OpenMP y varios threads MKL sobre distintos sistemas, considerando, a su vez, el empleo de paralelismo dinámico y matrices rectangulares.

Las tablas mostradas en las páginas siguientes ofrecen un resumen de los resultados presentados en los gráficos de las secciones anteriores, con el fin de aportar una visión más clara acerca de los mismos.

En primer lugar figura la Tabla 1, donde se compara, en cada uno de los sistemas considerados y para el caso de matrices rectangulares, el tiempo de ejecución secuencial (Seq.) con el obtenido al utilizar el máximo número de cores (Max_Cores) y con el menor tiempo obtenido (Low) al variar el número de threads MKL, mostrando en una última columna el speed-up obtenido por la rutina `dgemm2L` al ejecutarse con el número óptimo de threads respecto a su ejecución utilizando el máximo número de cores.

La Tabla 2 hace lo propio para el caso de matrices cuadradas. Asimismo, indicar que en cada sistema se ha utilizado un número de cores igual al máximo disponible en el momento de realización de las pruebas. Así pues, en Ben se han utilizado 96 cores de un total de 128, en Saturno los 24 cores del sistema y en Pirineus 240 (de un máximo permitido de 256). En ambas tablas, los tiempos de ejecución obtenidos se muestran en segundos y los números entre paréntesis representan el número de threads con los que se obtiene el menor tiempo de ejecución utilizando únicamente paralelismo MKL.

n	Seq.	Max_Cores	Low.	Speed-Up. (Max_Cores/Low)
Ben (96 cores)				
200x400	0.0112	0.0207	0.0035 (10)	5.91
300x600	0.0330	0.0366	0.0084 (10)	4.36
400x800	0.0786	0.0561	0.0122 (14)	4.60
500x1000	0.1537	0.0712	0.0219 (18)	3.25
1000x2000	1.2335	0.2250	0.0841 (32)	2.68
1500x3000	4.1969	0.4946	0.2244 (40)	2.20
2000x4000	9.7901	0.8180	0.4459 (38)	1.83
2500x5000	19.1146	1.2223	0.7474 (40)	1.64
400x200	0.0112	0.0116	0.0018 (12)	6.44
600x300	0.0331	0.0228	0.0045 (20)	5.07
800x400	0.0790	0.0354	0.0079 (22)	4.48
1000x500	0.1552	0.0543	0.0143 (22)	3.80
2000x1000	1.2385	0.1696	0.0661 (30)	2.57
3000x1500	4.1423	0.3531	0.1688 (34)	2.09
4000x2000	9.7827	0.7066	0.3156 (42)	2.24
5000x2500	19.1049	1.0304	0.5800 (48)	1.78
Saturno (24 cores)				
300x600	0.0302	0.0042	0.0041 (20)	1.02
400x800	0.0693	0.0094	0.0090 (20)	1.04
500x1000	0.1335	0.0220	0.0215 (22)	1.02
1000x2000	1.0009	0.1354	0.1349 (20)	1.00
1500x3000	3.3588	0.3737	0.3737 (24)	1.00
2000x4000	7.9263	0.9054	0.9054 (24)	1.00
600x300	0.0300	0.0034	0.0034 (24)	1.00
800x400	0.0668	0.0078	0.0078 (24)	1.00
1000x500	0.1336	0.0150	0.0150 (24)	1.00
2000x1000	1.0009	0.1133	0.1133 (24)	1.00
3000x1500	3.3588	0.3704	0.2787 (14)	1.33
4000x2000	7.9263	0.8484	0.5679 (16)	1.49

Tabla 1. Tiempos de ejecución (en segundos) obtenidos al ejecutar la rutina `dgemm2L` en los sistemas Ben, Saturno y Pirineus utilizando matrices rectangulares. Seq (tiempo secuencial), Max_Cores (tiempo con el máximo número de cores), Low (menor tiempo con paralelismo MKL), Speed-Up (aceleración conseguida respecto a la ejecución con el máximo número de cores)

n	Seq.	Max_Cores	Low.	Speed-Up (Max_Cores/Low)
Ben (96 cores)				
400	0.0223	0.0209	0.0033 (16)	6.33
800	0.1573	0.0543	0.0154 (16)	3.52
1000	0.3144	0.0759	0.0269 (22)	2.82
2000	2.4626	0.3306	0.1294 (36)	2.55
3000	8.2604	0.6640	0.3076 (40)	2.16
4000	19.5511	1.3624	0.6387 (40)	2.13
5000	38.1964	1.8791	1.1098 (48)	1.69
Saturno (24 cores)				
400	0.0178	0.0025	0.0024 (21)	1.04
600	0.0577	0.0067	0.0067 (24)	1.00
800	0.1360	0.0399	0.0170 (21)	2.35
1000	0.2498	0.0335	0.0318 (20)	1.05
2000	1.9849	0.2257	0.2257 (24)	1.00
3000	6.6704	0.7255	0.5205 (17)	1.39
Pirineus (240 cores)				
750	0.0956	0.3139	0.0139 (24)	22.58
1000	0.2199	0.4547	0.0322 (12)	14.12
2000	1.6815	1.1569	0.4796 (16)	2.41
3000	5.4696	1.2903	0.3955 (60)	3.26
4000	12.9175	1.9495	0.5397 (60)	3.61
5000	25.1401	2.7449	1.1149 (60)	2.46

Tabla 2. Tiempos de ejecución (en segundos) obtenidos al ejecutar la rutina `dgemm2L` en los sistemas Ben y Saturno utilizando matrices rectangulares. Seq (tiempo secuencial), Max_Cores (tiempo con el máximo número de cores), Low (menor tiempo con paralelismo MKL), Speed-Up (aceleración conseguida respecto a la ejecución con el máximo número de cores)

A la vista de los datos mostrados en las tablas, se puede apreciar que en sistemas de gran dimensión (Ben y Pirineus) el menor tiempo de ejecución no se alcanza al emplear un número de threads coincidente con el máximo número de cores. Esto queda reflejado en el speed-up, donde se muestra claramente la mejora que supone utilizar el óptimo número de threads respecto al máximo. En Saturno, en cambio, como el número de cores es mucho más reducido, en prácticamente todos los tamaños considerados el menor tiempo de ejecución se obtiene al utilizar un número de threads igual al de cores, de ahí que el speed-up alcanzado sea 1 en la mayoría de casos. Todo esto nos lleva a la conclusión de que usar un único nivel de paralelismo, aunque se usen un gran número de cores, no termina de ser una buena opción en sistemas de gran dimensión a la hora de obtener tiempos de ejecución cercanos al óptimo, incluso usando ajuste dinámico de threads, tanto en matrices cuadradas como rectangulares. Por tanto, el uso de dos niveles de paralelismo (OpenMP+MKL) podría ser una buena alternativa para obtener mejores prestaciones. El siguiente apartado trata de justificar esta decisión realizando un conjunto similar de experimentos sobre los mismos sistemas empleando los dos niveles de paralelismo mencionados.

3.3 Paralelismo de 2 niveles (OpenMP+MKL)

Tras analizar el comportamiento que experimenta la rutina `dgemm2L` al emplear solamente paralelismo MKL, se va a llevar a cabo un estudio de dicha rutina al usar dos niveles de paralelismo (OpenMP+MKL). Estos niveles de paralelismo se establecerán de la forma indicada en el apartado 2.2

De la misma forma que en el apartado anterior, se analizarán y compararán los resultados obtenidos tras ejecutar las pruebas con y sin selección dinámica de threads en el nivel de paralelismo MKL. También se realizarán pruebas para matrices cuadradas y rectangulares, mostrando los resultados para las combinaciones de threads que reflejen de forma más precisa el comportamiento de la rutina `dgemm2L` en cada uno de los sistemas de prueba empleados. El objetivo final de esta sección es, por tanto, justificar el empleo de un nivel adicional de paralelismo de cara a obtener una ejecución más eficiente de la rutina.

3.3.1 Dinámico vs. No dinámico

En la sección 3.2.1 se estudió el efecto que tenía habilitar el paralelismo dinámico MKL sobre los sistemas Ben, Saturno y Pirineus al ejecutar la rutina `dgemm2L` con el nivel de paralelismo establecido por MKL. En esta sección se va a llevar a cabo el mismo estudio pero utilizando dos niveles de paralelismo (OpenMP+MKL). Se seguirá la misma estructura que en el apartado mencionado, mostrando en este caso para cada sistema el speed-up alcanzado con diferentes combinaciones de threads OpenMP y MKL, habilitando/deshabilitando la selección dinámica de threads MKL.

La Figura 7 muestra el speed-up alcanzado por la rutina `dgemm2L` al ejecutarse en distintos sistemas con y sin paralelismo dinámico. El número máximo de threads utilizado en cada sistema corresponde con el máximo número de cores disponibles en el momento de realización de las pruebas.

En Ben, la habilitación del paralelismo dinámico produce un crecimiento continuo del speed-up conforme aumenta el número de threads OpenMP y el tamaño de las matrices, alcanzando su máximo valor en la combinación 48-2. Este comportamiento lleva a pensar que cuando varios threads OpenMP se están ejecutando el número de threads MKL utilizado es 1. En cambio, cuando se deshabilita el paralelismo dinámico, se obtienen valores mayores para el speed-up con un menor número de threads OpenMP, siendo la curva de crecimiento mucho menos pronunciada y dando lugar a la obtención del máximo valor en la combinación 24-4, a partir de la cual comienza a decrecer, lo que viene a decir que se están empleando los threads MKL indicados, a diferencia de lo que ocurre en el caso de que se seleccionen dinámicamente los threads MKL.

En Saturno, se repite un comportamiento semejante al experimentado en Ben, lo que lleva a pensar de nuevo que la combinación de paralelismo OpenMP y MKL va a desactivar la selección dinámica de threads MKL. Este fenómeno se puede apreciar en la parte izquierda del gráfico relativo a dicho sistema, donde se observa cómo la rutina ofrece un mejor speed-up para todas las combinaciones de threads OpenMP+MKL y se mantiene prácticamente constante conforme aumenta el tamaño de las matrices, lo que demuestra que se están utilizando todos los threads MKL establecidos.

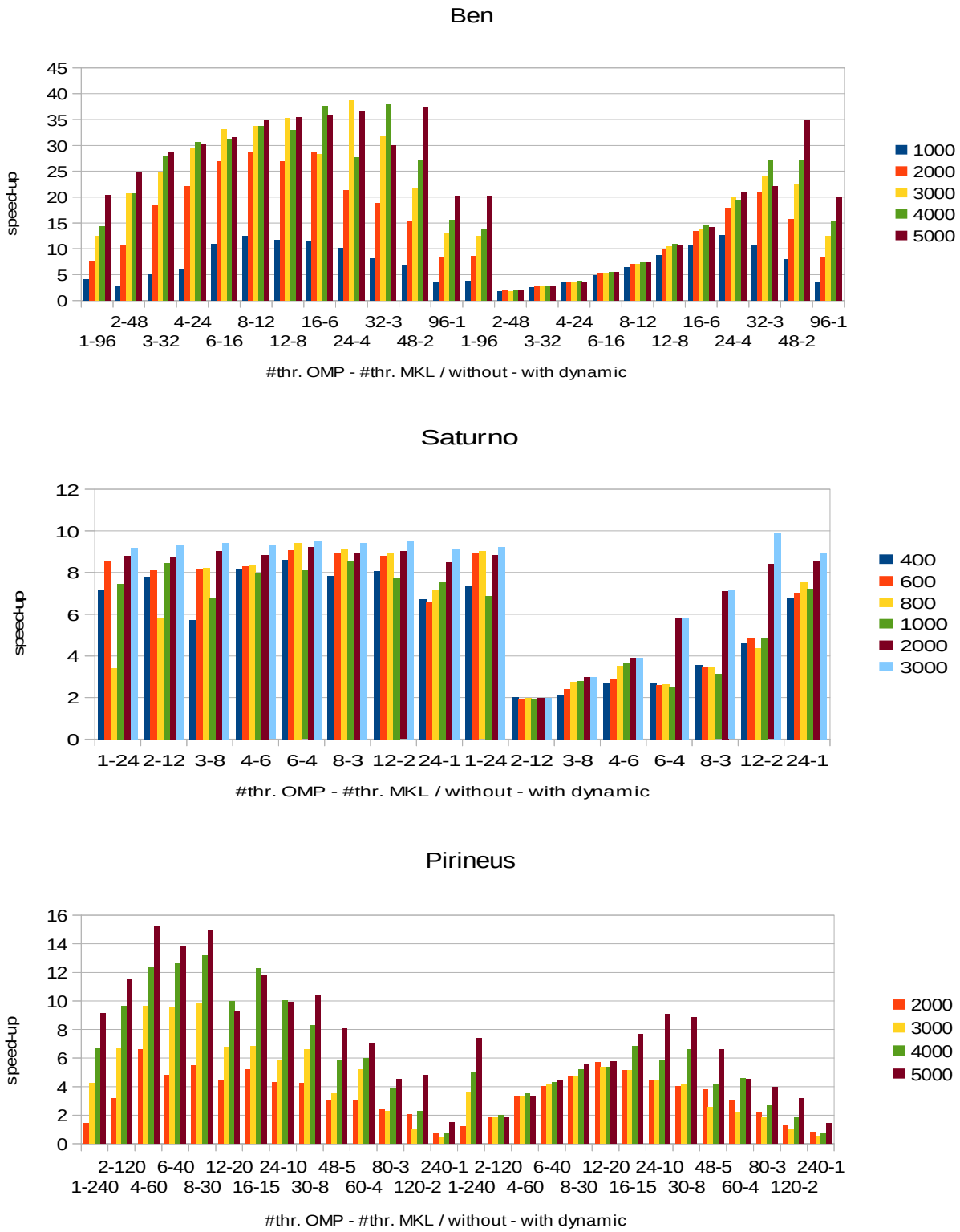


Figura 7. Speed-up obtenido en distintos sistemas, variando el tamaño de las matrices, el número de threads OpenMP y MKL y deshabilitando/habilitando el paralelismo dinámico

En Pirineus, en cambio, el comportamiento experimentado difiere bastante del anterior sistema, pues al habilitar el paralelismo dinámico el mejor speed-up se obtiene en la zona intermedia del gráfico (combinaciones 24-10 y 30-8) y decrece conforme aumenta el número de threads OpenMP y el tamaño de las matrices, lo que indica que el sistema está estableciendo un número de threads MKL más próximo al indicado, es decir, está funcionando mejor la selección dinámica de threads MKL. No obstante, al deshabilitar la ejecución dinámica de threads MKL, el speed-up vuelve a alcanzar valores mayores, al igual que en Ben y Saturno, obteniéndose el máximo en las combinaciones de threads en las que interviene un mayor número de threads MKL (4-60 y 8-30), demostrando, una vez más, que se están empleando todos los threads MKL especificados.

Una vez analizados los resultados obtenidos para cada uno de los sistemas, se puede concluir que el empleo de paralelismo anidado deshabilita el paralelismo dinámico en la librería MKL, al menos en las versiones utilizadas en Ben, Saturno y Pirineus. Esto queda reflejado en los gráficos de la Figura 8, donde se observa cómo al aumentar el tamaño del problema, el speed-up alcanzado con selección dinámica de threads MKL respecto a la ejecución estática (cociente entre el tiempo de ejecución obtenido sin paralelismo dinámico respecto al obtenido con paralelismo dinámico habilitado) es inferior a 1 en todas las ejecuciones realizadas en dichos sistemas. Se nota principalmente con un número alto de threads MKL, lo que confirma que la selección dinámica de threads MKL no es una buena opción a utilizar.

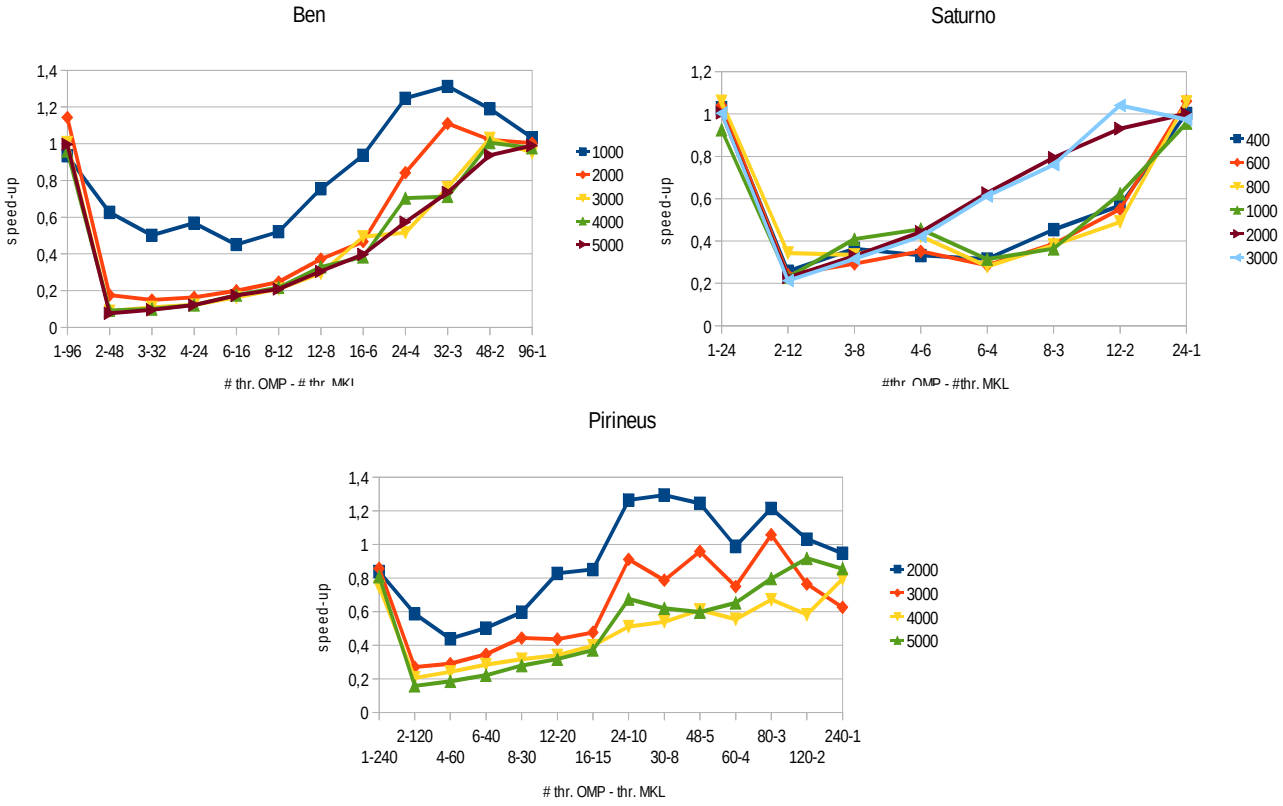


Figura 8. Speed-up obtenido en los distintos sistemas con selección dinámica respecto a selección estática de threads MKL

3.3.2 Matrices cuadradas vs. Matrices rectangulares

En este apartado se va a llevar a cabo un estudio similar al realizado en 3.2.2, estableciendo en este caso los dos niveles de paralelismo utilizados hasta el momento (OpenMP+MKL) y comprobando a su vez cómo varía el comportamiento de la rutina `dgemm2L` en cada uno de los sistemas al trabajar con matrices cuadradas y rectangulares.

Las Figuras 9 y 10 muestran los gráficos correspondientes a los sistemas Saturno y Ben, respectivamente, con el speed-up alcanzado al trabajar con este tipo de matrices.

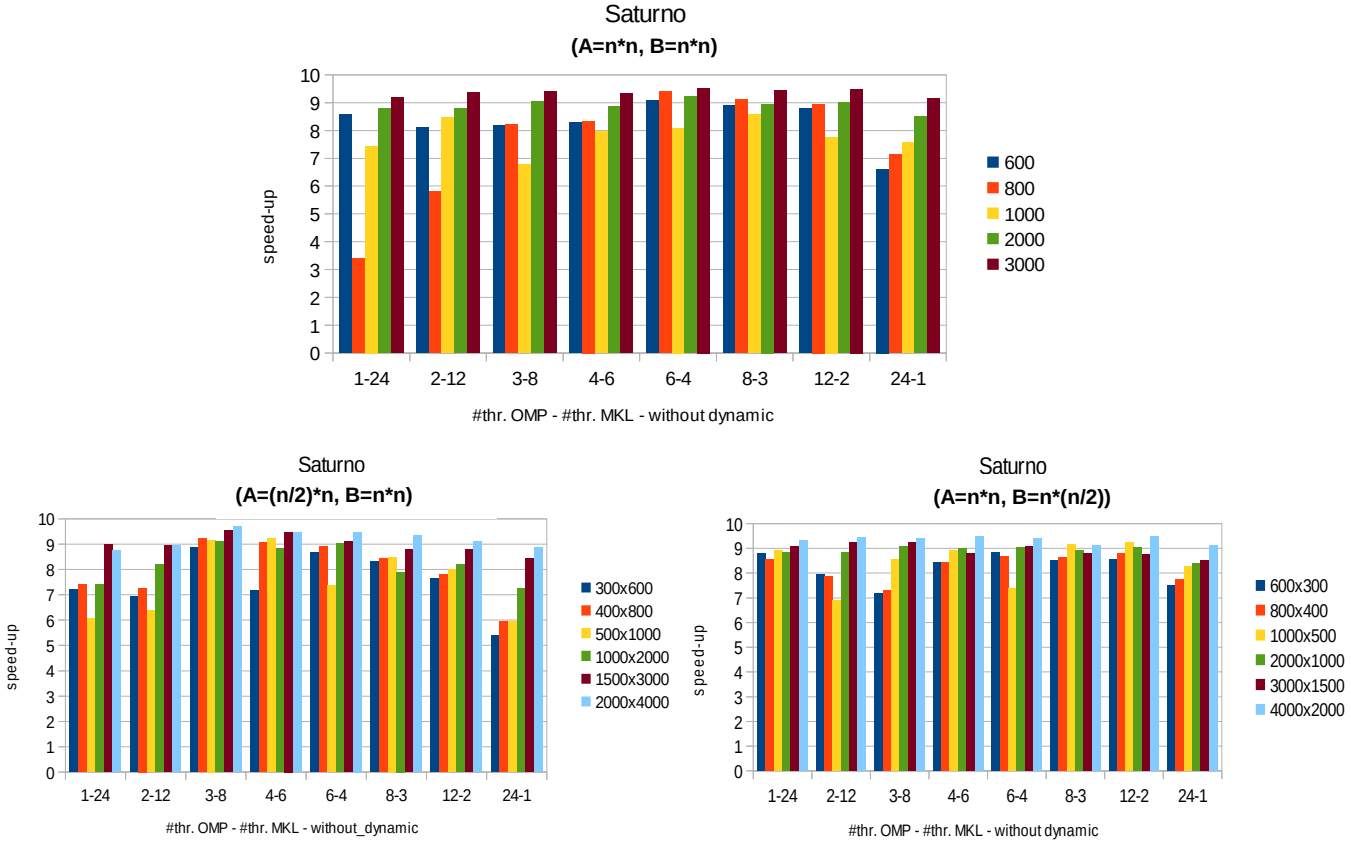


Figura 9. Speed-up obtenido en Saturno al ejecutar la rutina `dgemm2L`, con dos niveles de paralelismo utilizando matrices cuadradas y rectangulares de distintos tamaños

Como se puede observar, los dos sistemas experimentan un comportamiento muy similar al trabajar con matrices cuadradas y rectangulares. Además, ambos obtienen los mejores speed-ups cuando trabajan con matrices cuadradas. No obstante, se pueden apreciar pequeñas diferencias cuando las matrices A o B son rectangulares. En Saturno, si la matriz B es rectangular se obtienen speed-ups ligeramente superiores a los obtenidos cuando A es rectangular. En Ben, sin embargo, sucede lo contrario, pero en ningún caso se obtienen mejoras significativas. Además, se observa que Saturno presenta un speed-up más uniforme tanto en matrices cuadradas como rectangulares para las distintas combinaciones de threads OpenMP y MKL, pues se utilizan menos threads debido al menor número

de cores del sistema; en cambio, en Ben, el speed-up crece de forma continua conforme aumenta el número de threads OpenMP y el tamaño de las matrices hasta la ejecución con 24-4 threads, momento en que empieza a decrecer al aumentar el número de cores. Por tanto, se puede concluir que no se observan diferencias significativas en el speed-up al utilizar dos niveles de paralelismo con matrices cuadradas o rectangulares.

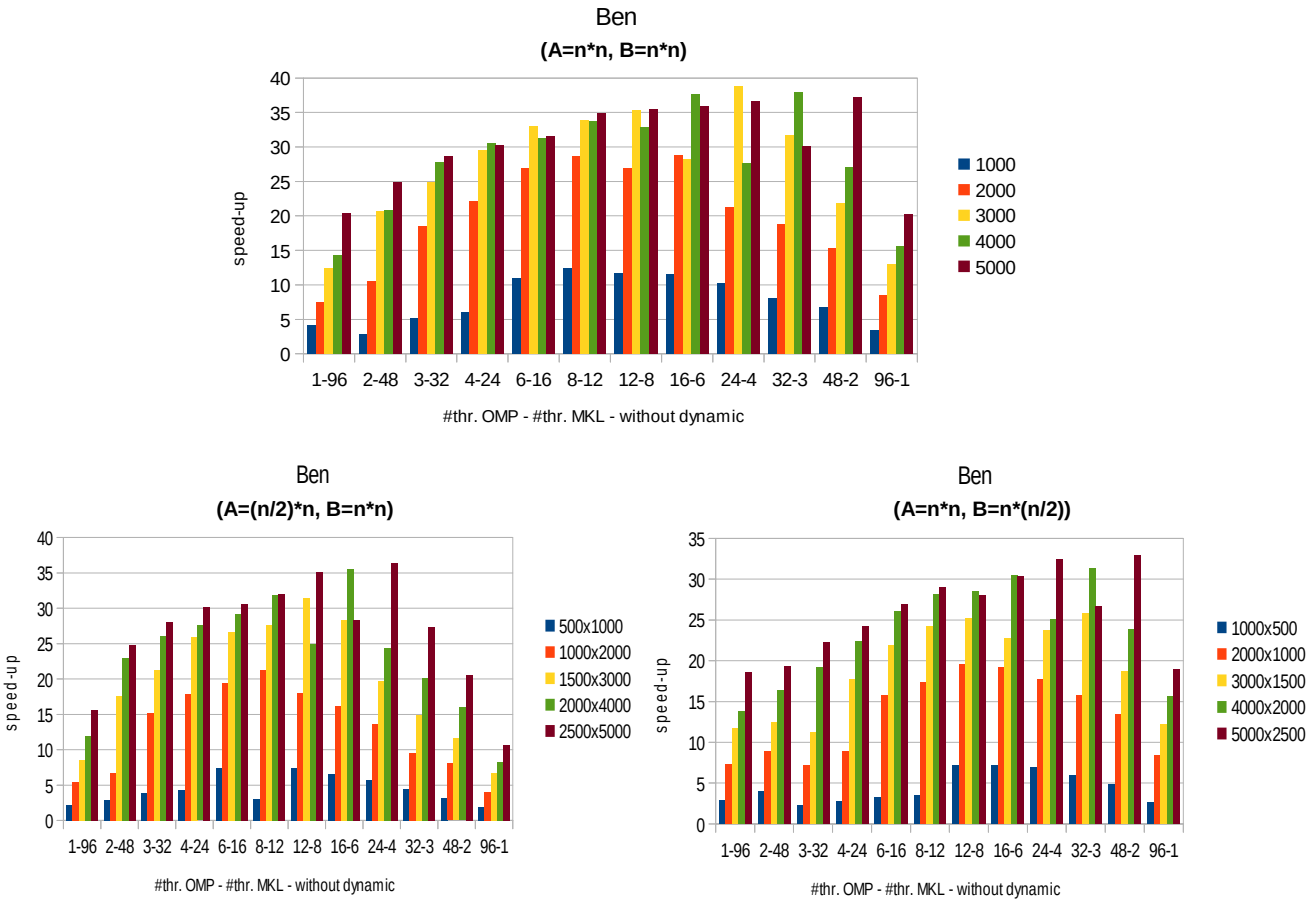


Figura 10. Speed-up obtenido en Ben al ejecutar la rutina `dgemm2L`, con dos niveles de paralelismo utilizando matrices cuadradas y rectangulares de distintos tamaños

3.3.3 Resumen y conclusiones

En esta segunda parte del capítulo 3 se ha analizado el comportamiento que experimenta la rutina `dgemm2L` al ejecutarse con dos niveles de paralelismo (OpenMP+MKL) sobre distintos sistemas, considerando además el empleo de paralelismo dinámico y matrices rectangulares.

Las Tablas 3 y 4 muestran un resumen de los resultados presentados en los gráficos analizados en las secciones anteriores, con el fin de proporcionar una visión más clara acerca de los mismos.

n	Seq.	Max_Cores	Low. MKL	Low. OMP+MKL	Speed-Up (MKL / OMP+MKL)
Ben (96 cores)					
500x1000	0.1537	0.0712	0.0219 (18)	0.0124 (4-7)	1.76
1000x2000	1.2335	0.2250	0.0841 (32)	0.0452 (8-6)	1.86
1500x3000	4.1969	0.4946	0.2244 (40)	0.1251 (12-5)	1.79
2000x4000	9.7901	0.8180	0.4459 (38)	0.2584 (16-4)	1.72
2500x5000	19.1146	1.2223	0.7474 (40)	0.4491 (20-3)	1.66
1000x500	0.1552	0.0543	0.0143 (22)	0,0126 (26-1)	1.13
2000x1000	1.2385	0.1696	0.0661 (30)	0.0482 (13-4)	1.37
3000x1500	4.1423	0.3531	0.1688 (34)	0.1284 (24-3)	1.31
4000x2000	9.7827	0.7066	0.3156 (42)	0.2765 (32-2)	1.14
5000x2500	19.1049	1.0304	0.5800 (48)	0.5068 (20-3)	1.14
Saturno (24 cores)					
300x600	0.030216	0.0042	0.0041 (20)	0.0034 (3-8)	1.21
400x800	0.069301	0.0094	0.0090 (20)	0.0075 (3-8)	1.20
500x1000	0.133496	0.0220	0.0215 (22)	0.0145 (4-6)	1.48
1000x2000	1.000864	0.1354	0.1349 (20)	0.1100 (3-8)	1.23
1500x3000	3.358805	0.3737	0.3737 (24)	0.3520 (3-8)	1.06
2000x4000	7.926316	0.9054	0.9054 (24)	0.8185 (3-8)	1.11
600x300	0.029990	0.0034	0.0034 (24)	0.0034 (6-4)	1.00
800x400	0.066823	0.0078	0.0078 (24)	0.0076 (12-2)	1.03
1000x500	0.133586	0.0150	0.0150 (24)	0.0144 (12-2)	1.04
2000x1000	1.000864	0.1133	0.1133 (24)	0.1103 (3-8)	1.03
3000x1500	3.358805	0.3704	0.2787 (14)	0.2601 (16-1)	1.07
4000x2000	7.926316	0.8484	0.5679 (16)	0.5650 (2-8)	1.01

Tabla 3. Tiempos de ejecución (en segundos) obtenidos al ejecutar la rutina `dgemm2L` en Ben y Saturno utilizando matrices rectangulares. Seq (tiempo en secuencial), Max_Cores (tiempo con máximo número de cores), Low_MKL (menor tiempo con paralelismo MKL), Low_OMP+MKL (menor tiempo con 2 niveles de paralelismo), Speed-Up (ganancia obtenida al utilizar OpenMP+MKL respecto a la ejecución utilizando únicamente paralelismo MKL)

La Tabla 3 presenta, para cada uno de los sistemas considerados, el tiempo de ejecución secuencial (Seq.), el tiempo obtenido al utilizar el máximo número de cores (Max_Cores), el menor tiempo de ejecución utilizando paralelismo MKL (Low. MKL) y dos niveles de paralelismo (Low. OMP+MKL) y una última columna con el speed-up obtenido al usar dos niveles de paralelismo respecto al menor tiempo de ejecución utilizando únicamente paralelismo MKL. La Tabla 4 hace lo propio para el caso de matrices cuadradas. Todos los tiempos se muestran en segundos y corresponden a los obtenidos al ejecutar la rutina en los distintos sistemas utilizando el máximo número de cores disponibles (indicado en ambas tablas para cada sistema). Los números entre paréntesis mostrados en las tablas, representan la combinación de threads OpenMP+MKL con los que se obtiene el menor tiempo de ejecución.

n	Seq.	Max_Cores	Low. MKL	Low. OMP+MKL	Speed-Up (MKL/OMP+MKL)
Ben (96 cores)					
400	0.0223	0.0209	0.0033 (16)	0.0027 (4-4)	1.22
800	0.1573	0.0543	0.0154 (16)	0.0116 (10-3)	1.33
1000	0.3144	0.0759	0.0269 (22)	0.0181 (5-8)	1.49
2000	2.4626	0.3306	0.1294 (36)	0.0749 (10-6)	1.73
3000	8.2604	0.6640	0.3076 (40)	0.2131 (24-4)	1.44
4000	19.5511	1.3624	0.6387 (40)	0.4900 (32-2)	1.30
5000	38.1964	1.8791	1.1098 (48)	0.8964 (20-4)	1.24
Saturno (24 cores)					
400	0.0178	0.0025	0.0024 (21)	0.0021 (6-4)	1.19
600	0.0577	0.0067	0.0067 (24)	0.0064 (6-4)	1.05
800	0.1360	0.0399	0.0170 (21)	0.0146 (6-4)	1.17
1000	0.2498	0.0335	0.0318 (20)	0.0291 (8-3)	1.09
2000	1.9849	0.2257	0.2257 (24)	0.2151 (6-4)	1.05
3000	6.6704	0.7255	0.5205 (17)	0.5205 (1-17)	1.00
Pirineus (240 cores)					
750	0.0956	0.3139	0.0139 (24)	0.0134 (2-16)	1.04
1000	0.2199	0.4547	0.0322 (12)	0.0235 (2-16)	1.37
2000	1.6815	1.1569	0.4796 (16)	0.0797 (5-12)	6.01
3000	5.4696	1.2903	0.3955 (60)	0.2752 (4-15)	1.44
4000	12.9175	1.9495	0.5397 (60)	0.4670 (5-12)	1.16
5000	25.1401	2.7449	1.1149 (60)	0.8598 (10-9)	1.30

Tabla 4. Tiempos de ejecución (en segundos) obtenidos al ejecutar la rutina `dgemm2L` en distintos sistemas utilizando matrices rectangulares. Seq (tiempo en secuencial), Max_Cores (tiempo con máximo número de cores), Low_MKL (menor tiempo con paralelismo MKL), Low_OMP+MKL (menor tiempo con 2 niveles de paralelismo), Speed-Up (ganancia obtenida al utilizar OpenMP+MKL respecto a la ejecución utilizando únicamente paralelismo MKL)

A la vista de los datos reflejados en estas tablas, se puede apreciar cómo en los sistemas indicados se obtiene una mejora en el speed-up respecto a la utilización de paralelismo MKL, siendo más significativa en Ben y Pirineus que en Saturno, dado que este sistema se comporta mejor con matrices de menor dimensión. La Tabla 3 confirma, además, la conclusión obtenida en la sección 3.3.2, esto es, que utilizar matrices rectangulares no presenta diferencias significativas en el speed-up. Asimismo, se observa que los mejores tiempos de ejecución con dos niveles de paralelismo se obtienen para combinaciones intermedias de threads OpenMP y MKL, lo que lleva a afirmar que usar dos niveles de paralelismo va a permitir acelerar la ejecución de la rutina `dgemm2L`, especialmente con matrices de gran tamaño. Así pues, surge la necesidad de establecer un proceso de auto-optimización en dicha rutina que permita determinar de forma automática la combinación más adecuada de threads OpenMP y MKL a emplear en cada nivel de paralelismo, para lo cual será necesario aplicar determinadas técnicas, cuya descripción y análisis será objeto de estudio del siguiente capítulo.

Capítulo 4. Técnicas de auto-optimización

4.1 Introducción

En el presente capítulo se va a llevar a cabo el estudio de distintas técnicas de auto-optimización. Estas técnicas se van a aplicar a la rutina `dgemm2L` y están encaminadas a determinar la combinación más adecuada de threads a establecer en cada nivel de paralelismo OpenMP y MKL con el objetivo de conseguir una ejecución de la rutina lo más eficiente posible.

El capítulo comienza realizando una breve descripción de las fases de que consta una metodología de diseño, instalación y ejecución de una rutina y a continuación da paso al estudio de varias técnicas de auto-optimización. La primera de estas técnicas es bastante sencilla y va a permitir, en una primera instancia, dotar a la rutina `dgemm2L` de la capacidad suficiente como para decidir el número de threads a establecer en cada nivel de paralelismo OpenMP y MKL en función de un parámetro especificado en la propia cabecera de la rutina. Una vez analizados y validados los resultados obtenidos de aplicar esta técnica, se llevará a cabo el estudio de otras técnicas más avanzadas, con el fin de obtener de forma más precisa y en el menor tiempo posible, dicha combinación de threads. Finalmente, se llevará a cabo la inclusión de alguna de estas técnicas en la propia rutina (tal como se detalla en el Anexo 6, donde se muestra su implementación), evaluando y comparando los resultados obtenidos con los del resto de técnicas propuestas y con los obtenidos en los propios experimentos, determinando de esta forma su proximidad respecto al óptimo y si mejoran, por tanto, el rendimiento de la rutina.

4.2 Metodología de diseño, instalación y ejecución

Antes de dar paso a las técnicas de auto-optimización objeto de estudio del presente capítulo, se van a explicar brevemente las fases de que consta una metodología general de diseño, instalación y ejecución de una rutina en un sistema (Figura 11), pues algunas de estas fases serán referenciadas más adelante en las técnicas presentadas.

La metodología propuesta consta de tres fases claramente diferenciadas:

- **Fase de diseño**: se implementa la rutina si es de nueva creación y, a continuación, se programa el gestor de toda la información de optimización.

- **Fase de instalación**: en esta fase, llevada a cabo por el gestor de optimización de la rutina, se ejecuta de manera dirigida la rutina para algunas combinaciones concretas de parámetros de entrada aplicando alguna de las técnicas de auto-optimización propuestas y se almacena la información generada. Esta información será utilizada posteriormente por la rutina durante la fase de ejecución. Para el caso de la rutina `dgemm2L`,

la información a almacenar podría consistir en un conjunto determinado de tamaños de matriz empleados durante la fase de instalación y la combinación de threads OpenMP+MKL con la que se obtiene el menor tiempo de ejecución.

- **Fase de ejecución**: en esta última fase se utiliza la información almacenada en la fase de instalación para ejecutar la rutina ajustando el valor de sus parámetros algorítmicos de acuerdo a esa información.

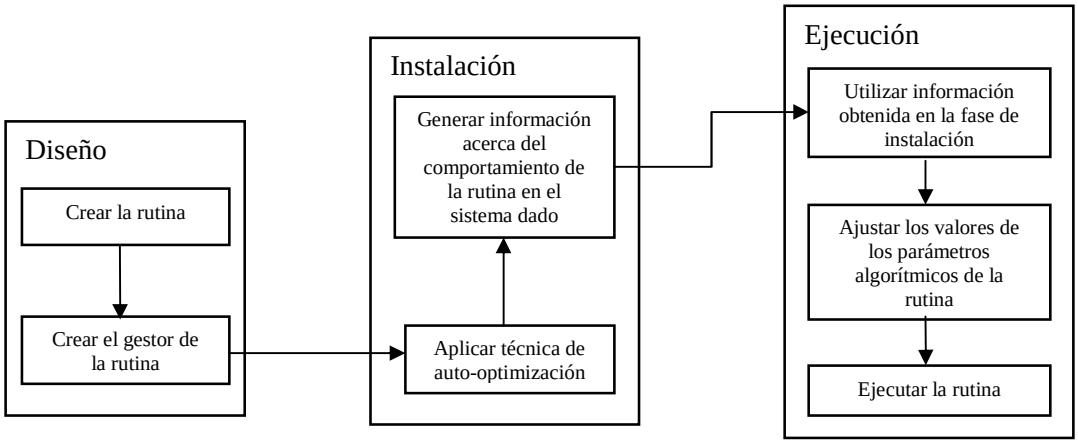


Figura 11. Metodología general de instalación y ejecución de una rutina en un sistema con capacidad de ajuste automático

El estudio a realizar con cada una de las técnicas de auto-optimización se centrará principalmente en la fase de la metodología relativa a la instalación de la rutina, pues como ya se ha comentado, el objetivo principal de estas técnicas va a consistir en obtener la combinación más apropiada de threads a establecer en cada nivel de paralelismo OpenMP y MKL que permita realizar futuras ejecuciones de la rutina con un tiempo de ejecución cercano al óptimo.

4.3 Auto-optimización por aproximación

Esta sección comienza con el estudio de una técnica de auto-optimización que va a permitir a la rutina `dgemm2L` establecer de forma automática el número de threads a utilizar en cada uno de los dos niveles de paralelismo (OpenMP y MKL). Esta decisión se realiza a partir de un nuevo parámetro de tipo entero incluido en la cabecera de la rutina (`maxCores`) que contiene el número máximo de cores a utilizar durante la ejecución y que sustituye, por tanto, a los parámetros `thrOMP` y `thrMKL` hasta ahora utilizados.

Teniendo en cuenta los resultados experimentales obtenidos en el capítulo anterior en los distintos sistemas considerados, se utiliza inicialmente una técnica sencilla consistente en aplicar la raíz cuadrada al máximo número de cores establecido. Si el resultado no es exacto, se utiliza la parte entera para establecer el número de threads OpenMP y la parte entera más 1 para establecer el número de threads MKL. En otro caso, se asigna directamente el resultado obtenido de aplicar la raíz cuadrada tanto al número de threads OpenMP como al número de threads MKL.

La Tabla 5 muestra los tiempos de ejecución obtenidos al aplicar esta técnica en los sistemas Ben y Saturno cuando el máximo número de cores utilizado es 90 y 20, respectivamente, con el fin de poder determinar si dichos valores distan mucho respecto al óptimo alcanzado en los experimentos. En Ben, se establecerán 9 threads OpenMP y 10 MKL, pues $\sqrt{90} = 9{,}4868$ y en Saturno, 4 threads OpenMP y 5 MKL, pues $\sqrt{20} = 4{,}4721$

n	Seq.	Max_Cores	Opt.	Auto-Opt.	Sp. (Opt/Auto-Opt)
		(90) **Ben**			
400	0.0223	0.1203	0.0062 (18-5)	0.0099	0.6271
600	0.0660	0.1620	0.0104 (10-9)	0.0122	0.8554
800	0.1573	0.1830	0.0168 (10-9)	0.0175	0.9600
1000	0.3144	0.1634	0.0255 (10-9)	0.0257	0.9895
1500	1.0518	0.2050	0.0538 (10-9)	0.0551	0.9753
2000	2.4626	0.2863	0.0847 (10-9)	0.0893	0.9482
2500	4.7925	0.5943	0.1638 (10-9)	0.1714	0.9554
3000	8.2604	0.6504	0.2645 (30-3)	0.2738	0.9660
3500	13.1160	1.1473	0.3718 (30-3)	0.4641	0.8011
4000	19.5511	1.3204	0.6092 (18-5)	0.6487	0.9392
4500	27.8241	1.5552	0.7393 (18-5)	0.8376	0.8825
5000	38.1964	1.9356	0.9939 (10-9)	1.1134	0.8926
		(20) **Saturno**			
400	0.0178	0.0025	0.0024 (5-4)	0.0025	0.9501
600	0.0577	0.0078	0.0072 (5-4)	0.0076	0.9561
800	0.1360	0.0171	0.0167 (5-4)	0.0174	0.9592
1000	0.2498	0.0489	0.0332 (4-5)	0.0332	1.0000
1500	0.8409	0.1177	0.1077 (5-4)	0.1112	0.9683
2000	1.9849	0.2521	0.2509 (5-4)	0.2519	0.9962
2500	3.8662	0.4931	0.4834 (5-4)	0.4926	0.9814
3000	6.6704	0.8369	0.8195 (5-4)	0.8345	0.9821

Tabla 5. Tiempos de ejecución (en segundos) obtenidos al ejecutar la rutina `dgemm2L` en Ben y Saturno, variando el tamaño de las matrices. Seq (tiempo secuencial), Max_Cores (tiempo con máximo número de cores en nivel MKL), Opt (óptimo obtenido con paralelismo OpenMP+MKL), Auto-Opt (tiempo con threads establecidos por la auto-optimización) y Sp (mejora obtenida al auto-optimizar)

Se puede observar en la Tabla 5, que el speed-up obtenido al auto-optimizar respecto al óptimo absoluto, se aproxima bastante a 1 para todos los tamaños de matrices, pero dichos resultados se obtienen con una combinación de threads distinta a la calculada por la técnica de auto-optimización para ambos sistemas (9-10 y 4-5), lo que lleva a pensar que aunque esta técnica va a permitir obtener resultados satisfactorios y muy próximos al óptimo, en muy pocos casos va a conseguir hallar la combinación óptima de threads. Por ello, las siguientes secciones del capítulo se van a destinar al estudio de nuevas técnicas de auto-optimización que permitan obtener de forma más precisa la combinación de threads OpenMP+MKL a utilizar en futuras ejecuciones de la rutina `dgemm2L` y que lleve a la consecución de tiempos de ejecución más cercanos al óptimo.

4.4 Auto-optimización MKL

El presente apartado se va a centrar en analizar dos técnicas de auto-optimización que van a permitir establecer el número de threads MKL que optimice el tiempo de ejecución cuando sólo se usa este tipo de paralelismo. Para conseguir este objetivo, será necesario realizar un proceso previo de instalación automática de la rutina en el sistema donde se vaya a ejecutar. En esta sección se proponen y analizan dos técnicas con las que obtener el número de threads MKL con el que se ejecutará dicha rutina con el fin de obtener un tiempo de ejecución próximo al óptimo.

La primera de estas técnicas permitirá obtener el número óptimo de threads MKL llevando a cabo una búsqueda exhaustiva, en tiempo de instalación, entre las ejecuciones realizadas con los distintos tamaños de problema especificados. La otra técnica, en cambio, realizará una búsqueda local utilizando un incremento variable basado en un valor porcentual, con el fin de hallar el número óptimo de threads MKL evitando realizar todas las ejecuciones, reduciendo, por tanto, el tiempo dedicado a la instalación.

4.4.1 Instalación de MKL mediante búsqueda exhaustiva

Como se ha comentado anteriormente, con esta técnica se instala la rutina en el sistema donde vaya a ejecutarse y al finalizar dicho proceso se obtiene el número de threads MKL a utilizar en futuras llamadas a esta rutina.

Para validar este procedimiento de instalación se especifican dos conjuntos de tamaños para las matrices. El primero está compuesto por los tamaños utilizados durante la fase de instalación y el segundo contiene un conjunto de tamaños intermedios para validar si el número de threads y el tiempo de ejecución que se obtendría utilizando la información generada durante la instalación difiere mucho respecto al óptimo obtenido con este conjunto.

Una vez establecido el conjunto de tamaños para la instalación, se procede a ejecutar la rutina `dgemm2L` con cada uno de ellos utilizando el máximo número de cores disponibles en el sistema. Durante esta fase y para cada tamaño en el conjunto de tamaños de instalación, un algoritmo (cuyo esquema en pseudo-código se muestra en la Figura 12) lleva a cabo una búsqueda exhaustiva entre los posibles valores para el parámetro que indica el número de threads a utilizar. Esta búsqueda se realiza entre los valores 1 y el máximo número de cores disponibles en el sistema, y como resultado, se obtiene el número de threads con el que se consigue el menor tiempo de ejecución. Por otro lado, no hay que perder de vista que estamos también interesados en el tiempo total de instalación, pues nos interesa tener una predicción precisa pero si es posible con un tiempo de instalación reducido.

Los resultados obtenidos se validan utilizando un conjunto de tamaños diferentes a los utilizados en la fase de instalación, concretamente serán valores intermedios a aquellos. Para ello se compara el tiempo de ejecución que se obtiene utilizando la información generada durante la instalación con el menor tiempo de ejecución variando el número de threads.

La Tabla 6 muestra el resultado de aplicar esta técnica en los sistemas Ben, Saturno y Pirineus utilizando un total de 96, 24 y 240 cores, respectivamente. Para la instalación de la rutina se ha utilizado el conjunto de tamaños de instalación {500, 1000, 3000, 5000} y para validar se han usado los tamaños {700, 2000, 4000}. Los números entre paréntesis mostrados en la tabla, representan el número de threads MKL con el que se obtiene el tiempo de ejecución indicado.

n	Opt.	Auto. Opt	Sp.	Tiempo Inst.
		Ben		
500	0.0056 (20)			1.40
700	0.0121 (24)	0.0142 (20)	0.85	
1000	0.0270 (20)			5.50
2000	0.1294 (36)	0.1790 (30)	0.72	
3000	0.3076 (40)			70.72
4000	0.6387 (40)	0.8121 (44)	0.79	
5000	1.1098 (48)			275.06
			TOTAL:	352.69
		Saturno		
500	0.0045 (24)			0.22
700	0.0099 (24)	0.0163 (22)	0.61	
1000	0.0318 (20)			1.43
2000	0.2257 (24)	0.2532 (22)	0.89	
3000	0.7255 (24)			26.44
4000	1.6461 (24)	1.9459 (20)	0.85	
5000	2.0901 (18)			77.67
			TOTAL:	105.77
		Pirineus		
500	0.0059 (24)			1.8
750	0.0139 (24)	0.0597 (16)	0.23	
1000	0.0322 (12)			2.74
2000	0.4796 (16)	0.7659 (32)	0.63	
3000	0.3955 (60)			24.61
4000	0.5397 (60)	0.5397 (60)	1.00	
5000	1.1149 (60)			81.41
			TOTAL:	110.56

Tabla 6. Tiempos de ejecución (en segundos) obtenidos en distintos sistemas al aplicar la técnica de auto-optimización basada en búsqueda exhaustiva sobre el nivel de paralelismo MKL. Opt (tiempo óptimo obtenido), Auto-Opt (tiempo obtenido al auto-optimizar), Sp (cociente del tiempo óptimo respecto al obtenido con auto-optimización), Tiempo_Inst (tiempo de instalación para cada tamaño)

Como se puede apreciar en la Tabla 6, el método de auto-optimización utilizado proporciona ejecuciones satisfactorias especialmente para tamaños grandes, pues el cociente Sp. no está muy alejado de 1. También se observa que para tamaños pequeños, el speed-up alcanzado se aleja de dicho valor. Hay que tener en cuenta que para cada tamaño y número de threads se ha realizado

una única ejecución, pues realizar varias ejecuciones para tomar la media incrementaría sustancialmente el tiempo de instalación. Asimismo, indicar que alcanzar un Sp igual a 1 es un ideal muy difícil, pues el óptimo con el que se compara es el que se obtendría con un oráculo ideal perfecto que puede tomar siempre la decisión óptima, ya que posee una visión completa del campo de muestras

Por otro lado, podría ser conveniente reducir los tiempos de instalación, que aumentan drásticamente a partir de tamaño 5000. Además, en una librería con varias funciones y varias implementaciones de la misma función (para tipos float, double, complex…) habría que realizar un gran número de instalaciones, aumentando proporcionalmente el tiempo de instalación.

```
Establecer conjunto C de tamaños de instalación;
Inicializar mín;
PARA cada tamaño t en C HACER
        PARA n=1 hasta el máximo de cores HACER
                Ejecutar rutina dgemm2L con tamaño t, 1 thread OMP y n threads MKL;
                SI (tiempo de ejecución obtenido < mín) ENTONCES
                        Actualizar mín;
                        Actualizar óptimo número de threads MKL (optMKL);
                FIN_SI
        FIN_PARA
        Guardar t, mín y optMKL;
        Inicializar mín;
FIN  PARA
```

Figura 12. Algoritmo de instalación de MKL mediante búsqueda exhaustiva

4.4.2 Instalación de MKL mediante búsqueda local con incremento variable

En esta sección se va a presentar una variación de la técnica anterior. Con el empleo de esta nueva técnica de auto-optimización se pretende reducir el tiempo de instalación utilizado por la anterior, manteniendo, si es posible, la precisión en la selección del número de threads MKL con el que ejecutar la rutina `dgemm2L`.

Esta técnica parte del mismo conjunto de tamaños de matrices empleado en la búsqueda exhaustiva, pero en la instalación no se llevan a cabo todas las ejecuciones para cada uno de los tamaños de problema, sino que se realiza, en tiempo de instalación, una búsqueda local en función de un incremento variable determinado por el valor porcentual establecido para la instalación de la rutina. Inicialmente, el algoritmo de búsqueda local (cuyo esquema en pseudo-código se puede observar en la Figura 13) selecciona el primer tamaño del conjunto de tamaños para la instalación y ejecuta la rutina con 1, 2, 3… threads mientras el tiempo de ejecución obtenido no exceda al mínimo actual en un valor superior al porcentaje establecido. Una vez alcanzado dicho valor del número de threads óptimo para ese tamaño de problema, ejecuta la rutina para el siguiente tamaño de problema con el número de threads MKL con el que se ha obtenido el menor tiempo de ejecución para el tamaño previo, continuando la búsqueda con los valores inmediatamente inferior y superior. Este proceso de búsqueda continúa hasta alcanzar de nuevo un valor para el tiempo de ejecución que exceda el mínimo alcanzado hasta el momento en una cantidad igual al porcentaje especificado. El proceso se repite hasta acabar con los tamaños especificados en el conjunto de instalación,

obteniendo al final del mismo el número de threads MKL a usar con cada tamaño sin necesidad de haber realizado todas las ejecuciones que sí se llevaban a cabo en la técnica de búsqueda exhaustiva durante la instalación de la rutina.

```
Establecer conjunto C de tamaños de instalación;
Establecer tolerancia;
Inicializar mín;
optMKL=1;

PARA cada tamaño t en C HACER
        PARA n=optMKL hasta el máximo de cores HACER
                Ejecutar rutina dgemm2L con tamaño t, 1 thread OpenMP y n threads MKL
                SI (tiempo de ejecución obtenido < mín) ENTONCES
                        Actualizar mín y optMKL;
                SINO SI (tiempo obtenido supera mín en porcentaje > tolerancia) ENTONCES
                        Pasar al siguiente tamaño t en C y realizar ejecuciones a partir de optMKL;
                FIN_SI
        FIN_PARA
        Guardar t, mín y optMKL;
        Inicializar mín;
FIN_PARA
```

Figura 13. Algoritmo de instalación de MKL mediante búsqueda local con incremento variable

La precisión con la que se obtiene el número óptimo de threads MKL a establecer en la rutina depende del incremento porcentual que se escoja al instalar la rutina, pues valores mayores van a permitir acercarse más al tiempo de ejecución óptimo debido a que se van a realizar un mayor número de ejecuciones, ignorando así tiempos de ejecución que correspondan a óptimos locales. Esta idea se puede observar en la Figura 14, donde se ve que para tamaño 5000 se producen mínimos locales que con un incremento de un 1% van a detener la búsqueda realizada por el algoritmo en 36 threads, pero al aumentar el porcentaje se van a seguir realizando ejecuciones, obteniendo así el tiempo de ejecución óptimo, que se alcanza con 48 threads.

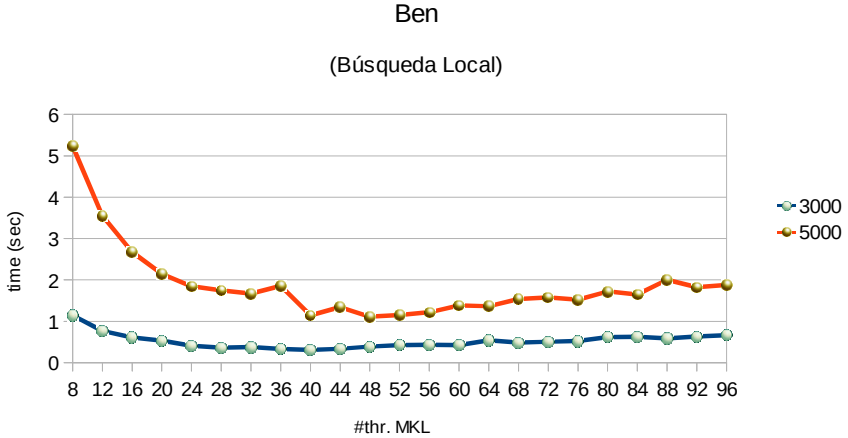


Figura 14. Máximos locales obtenidos en el tiempo de ejecución al variar el número de threads MKL

Por tanto, se ha de llegar a un compromiso entre el número de ejecuciones que se desean realizar durante la instalación y el incremento porcentual establecido, de forma que se realice un número de ejecuciones reducido pero se obtenga una buena aproximación al número óptimo de threads MKL.

La Tabla 7 muestra los resultados obtenidos al aplicar esta técnica en los sistemas Ben, Saturno y Pirineus utilizando un total de 96, 24 y 240 cores, respectivamente. Los números entre paréntesis representan los threads MKL con los que se obtiene el tiempo de ejecución indicado.

n	Opt.	1.00%	10.00%	20.00%	50.00%
			Ben		
500	0.0056 (20)	*0.0056* (20)	*0.0056* (20)	*0.0056* (20)	*0.0056* (20)
700	0.0121 (24)	0.0142 (20)	0.0142 (20)	0.0142 (20)	0.0142 (20)
1000	0.0270 (20)	*0.0270* (20)	*0.0270* (20)	*0.0270* (20)	*0.0270* (20)
2000	0.1294 (36)	0.1398 (24)	0.1790 (30)	0.1790 (30)	0.1790 (30)
3000	0.3076 (40)	0.3621 (28)	*0.3076* (40)	*0.3076* (40)	*0.3076* (40)
4000	0.6387 (40)	1.0979 (30)	0.6387 (40)	0.8121 (44)	0.8121 (44)
5000	1.1098 (48)	1.6701 (32)	1.1424 (40)	*1.1098* (48)	*1.1098* (48)
			Saturno		
500	0.0045 (24)	0.0135 (4)	0.0064 (12)	0.0056 (16)	*0.0045* (24)
700	0.0099 (24)	0.0196 (6)	0.0149 (16)	0.0157 (18)	0.0163 (22)
1000	0.0318 (20)	0.0411 (8)	*0.0318* (20)	*0.0318* (20)	*0.0318* (20)
2000	0.2257 (24)	0.2761 (8)	0.2532 (22)	0.2532 (22)	0.2532 (22)
3000	0.7255 (24)	0.9118 (8)	*0.7255* (24)	*0.7255* (24)	*0.7255* (24)
4000	1.6461 (24)	2.7364 (14)	1.6461 (24)	1.9210 (22)	1.9210 (22)
5000	2.0901 (18)	*2.0901* (18)	*2.0901* (18)	*2.0901* (18)	*2.0901* (18)
			Pirineus		
500	0.0059 (24)	0.0162 (4)	0.0162 (4)	0.0162 (4)	0.0162 (4)
750	0.0139 (24)	0.0774 (8)	0.0774 (8)	0.0774 (8)	0.0774 (8)
1000	0.0322 (12)	*0.0322* (12)	*0.0322* (12)	*0.0322* (12)	*0.0322* (12)
2000	0.4796 (16)	0.4796 (16)	0.4796 (16)	0.7659 (32)	0.7659 (32)
3000	0.3955 (60)	0.8269 (16)	0.8269 (16)	*0.3955* (60)	*0.3955* (60)
4000	0.5397 (60)	1.6802 (16)	1.6802 (16)	0.5397 (60)	0.5397 (60)
5000	1.1149 (60)	2.0365 (16)	2.0365 (16)	*1.1149* (60)	*1.1149* (60)

Tabla 7. Tiempos de ejecución (en segundos) obtenidos en distintos sistemas al aplicar la técnica de auto-optimización basada en búsqueda local con incremento variable sobre paralelismo MKL utilizando el conjunto de tamaños de instalación={500,1000,3000,5000} y el conjunto de validación={700,2000,4000}. Opt (óptimo tiempo de ejecución obtenido). 1%, 10%, 20%, 50% (tiempos obtenidos con auto-optimización, al utilizar el incremento porcentual indicado)

Como se puede observar en la Tabla 7, el número de threads MKL obtenido en cada una de las ejecuciones se va aproximando al óptimo conforme se incrementa el valor porcentual establecido, constatando de esta forma el comportamiento mostrado en el gráfico de la Figura 14. Además, estos valores óptimos se alcanzan con porcentajes comprendidos entre un 10% y un 20% para la mayoría de tamaños de instalación y validación utilizados en cada uno de los sistemas, de forma que utilizar porcentajes mayores no va a aportar mejoras significativas. Asimismo, en el caso de Ben y Pirineus

se aprecia que para determinados tamaños de validación, el tiempo de ejecución obtenido al incrementar el valor porcentual es superior que el obtenido con porcentajes inferiores. Esto es debido a que el número de threads MKL utilizado para cada tamaño de validación se obtiene aplicando un proceso de interpolación sobre el número óptimo de threads obtenido por los tamaños de instalación inmediatamente anterior y posterior, de ahí que pueda darse el caso de que en esa ejecución se obtenga un tiempo de ejecución superior.

Por otra parte, la Tabla 8 refleja los tiempos de instalación obtenidos en los distintos sistemas para cada uno de los incrementos porcentuales considerados.

Ben					
	Exhaustivo	1.00%	10.00%	20.00%	50.00%
t_inst (seg):	352.69	9.5378	8.1567	13.0834	22.7303

Saturno					
	Exhaustivo	1.00%	10.00%	20.00%	50.00%
t_inst (seg):	105.77	30.6451	14.2651	25.0115	29.5082

Pirineus					
	Exhaustivo	1.00%	10.00%	20.00%	50.00%
t_inst (seg):	110.56	13.2863	14.2516	12.9708	17.1192

Tabla 8. Tiempos de instalación (en segundos) con la técnica de auto-optimización basada en búsqueda local con incremento variable sobre paralelismo MKL, en distintos sistemas para cada uno de los incrementos porcentuales indicados durante la fase de instalación realizada

Se observa cómo los mejores tiempos de instalación se obtienen con un incremento de un 10%, seguidos a continuación por los obtenidos con un 20%. Cabe destacar, a su vez, la reducción producida en el tiempo de instalación al pasar de un 1% a un 10%, cuando lo normal sería que aumentase. Esto es así debido a que el algoritmo se sitúa antes en valores próximos al óptimo, evitando de esta forma realizar ejecuciones adicionales en el resto de tamaños del conjunto de instalación. También se observa que estos tiempos son muy inferiores a los obtenidos con la técnica de instalación mediante búsqueda exhaustiva. Por tanto, se puede afirmar que esta técnica de auto-optimización va a permitir obtener el número óptimo de threads MKL a establecer en ejecución para la rutina `dgemm2L` con un tiempo de instalación reducido.

4.5 Auto-optimización OpenMP+MKL

Las técnicas de auto-optimización estudiadas en la sección 4.4 se han centrado en obtener el número óptimo de threads MKL a usar en la ejecución de la rutina `dgemm2L` minimizando, a su vez, el tiempo dedicado a la instalación, con un valor para el número de threads OpenMP fijado en todo momento a 1.

En el presente apartado se va a ampliar dicho estudio introduciendo el nivel adicional de paralelismo ofrecido por OpenMP. El objetivo final, por tanto, será obtener el número de threads OpenMP y MKL a establecer en cada nivel de paralelismo que permita conseguir, en futuras ejecuciones de la rutina, un tiempo de ejecución próximo al óptimo.

En el desarrollo de este apartado se seguirá la estructura propuesta en la citada sección anterior. Así pues, se analizarán las mismas técnicas de auto-optimización descritas, pero teniendo en cuenta el uso de este segundo nivel de paralelismo. Se comenzará el estudio aplicando la técnica de instalación basada en búsqueda exhaustiva sobre el conjunto de ejecuciones realizadas para cada uno de los tamaños de matrices especificados y a continuación se aplicará la técnica basada en búsqueda local con incremento variable, con el fin de evitar realizar todo el conjunto de ejecuciones para las distintas combinaciones de threads OpenMP+MKL y reducir, por tanto, el tiempo dedicado a la fase de instalación de la rutina.

4.5.1 Instalación de OpenMP+MKL mediante búsqueda exhaustiva

En esta sección se va a aplicar la técnica de instalación basada en búsqueda exhaustiva sobre dos niveles de paralelismo, con el fin de obtener la combinación de threads OpenMP+MKL con la que conseguir el mínimo tiempo de ejecución, utilizando la información generada del conjunto de ejecuciones realizadas durante la fase de instalación de la rutina para cada uno de los tamaños de matriz especificados.

Puesto que se ha introducido un nivel adicional de paralelismo, la rutina se ejecutará durante la fase de instalación para todas las combinaciones posibles de threads OpenMP+MKL hasta alcanzar el máximo número de cores disponibles en el sistema. A su vez, tal y como se explicó en la sección 4.4.1, hará uso de dos conjuntos de tamaños para las matrices: el primero de ellos compuesto por los tamaños utilizados durante la fase de instalación y el otro constituido por un conjunto de tamaños intermedios para validar si el número de threads OpenMP+MKL obtenido para cada tamaño a partir de la información generada tras la fase de instalación difiere mucho respecto al óptimo obtenido con este conjunto. Esta información será generada por un algoritmo encargado de determinar dinámicamente y para cada tamaño, la combinación de threads con la que se obtiene el mínimo tiempo de ejecución. Estamos interesados también en el tiempo total empleado en la instalación. La Figura 15 muestra un esquema en pseudo-código de dicho algoritmo.

```
Establecer conjunto C de tamaños de instalación;
Inicializar optOMP y optMKL a 1;
Inicializar mín;
PARA cada tamaño t en C HACER
        PARA omp=1 hasta maxCores HACER
                PARA mkl=1 hasta (maxCores/omp ) HACER
                        Ejecutar rutina dgemm2L con tamaño t y combinación de threads (omp, mkl);
                        SI (tiempo de ejecución obtenido < mín) ENTONCES
                                Actualizar mín, optOMP y optMKL;
                        FIN_SI
                FIN_PARA
        FIN_PARA
        Guardar t, mín, optOMP y optMKL;
        Inicializar mín;
FIN_PARA
```

Figura 15. Algoritmo de instalación de OpenMP+MKL mediante búsqueda exhaustiva

La Tabla 9 muestra el resultado de aplicar esta técnica en los sistemas Ben, Saturno y Pirineus utilizando un total de 96, 24 y 240 cores, respectivamente. En todos ellos se ha utilizado el conjunto de tamaños de instalación {500, 1000, 3000, 5000} y el conjunto de tamaños de validación {700, 2000, 4000}. Los números entre paréntesis mostrados en la tabla representan la combinación de threads OpenMP+MKL con la que se obtiene el tiempo de ejecución indicado.

n	Opt.	Auto. Opt	Sp.	Tiempo Inst.
		Ben		
500	0,0051 (23-1)			6,25
700	0,0102 (7-4)	0,0122 (16-2)	0,83	
1000	0,0177 (10-4)			19,38
2000	0,0795 (10-5)	0,0810 (17-3)	0,98	
3000	0,2191 (25-3)			228,36
4000	0,5088 (32-2)	0,6614 (22-3)	0,77	
5000	0,9207 (20-3)			902,47
			TOTAL:	1156,45
		Saturno		
500	0,0038 (6-4)			0,63
700	0,0098 (6-4)	0,0110 (7-3)	0,89	
1000	0,0291 (8-3)			4,13
2000	0,2151 (6-4)	0,2519 (4-5)	0,85	
3000	0,5205 (1-17)			81,61
4000	1,2580 (1-17)	1,9443 (4-5)	0,65	
5000	1,8915 (7-3)			267,21
			TOTAL:	353,58
		Pirineus		
500	0,0059 (1-24)			13,36
750	0,0134 (2-16)	0,0139 (1-24)	0,96	
1000	0,0235 (2-16)			19,56
2000	0,0797 (5-12)	0,0869 (3-20)	0,92	
3000	0,2752 (4-15)			191,15
4000	0,4670 (5-12)	0,5502 (6-10)	0,85	
5000	0,8598 (10-9)			452,42
			TOTAL:	676,49

Tabla 9. Tiempos de ejecución (en segundos) obtenidos en distintos sistemas al aplicar la técnica de auto-optimización basada en búsqueda exhaustiva sobre los niveles de paralelismo OpenMP+MKL. Opt (tiempo óptimo obtenido), Auto-Opt (tiempo obtenido con auto-optimización), Sp (cociente del tiempo óptimo respecto al obtenido con auto-optimización) y Tiempo_Inst (tiempo de instalación para cada tamaño del conjunto de instalación)

Se puede apreciar en la Tabla 9, cómo el método de auto-optimización utilizado consigue obtener unos tiempos de ejecución cercanos al óptimo y, consecuentemente, un speed-up próximo a 1 respecto a este óptimo. Asimismo, se observa un incremento notable en el tiempo dedicado a la instalación respecto al obtenido al usar únicamente paralelismo MKL debido a que ahora se realizan un mayor número de ejecuciones debido a las posibles combinaciones de threads OpenMP+MKL. Finalmente, indicar que para cada tamaño se ha realizado una única ejecución, pues realizar varias ejecuciones para tomar la media daría lugar a un incremento sustancial del tiempo de instalación. No obstante, el número de experimentos a realizar para cada tamaño podría ser especificado por el administrador del sistema.

4.5.2 Instalación de OpenMP+MKL mediante búsqueda local con incremento variable

Esta técnica de auto-optimización pretende reducir el tiempo de instalación empleado por la búsqueda exhaustiva, manteniendo, si es posible, la precisión en la selección de la combinación de threads OpenMP y MKL con los que realizar futuras ejecuciones de la rutina `dgemm2L`.

De la misma forma que en la técnica anterior, toma como punto de partida los conjuntos de tamaños de instalación y validación especificados para las matrices, pero en la instalación no se llevan a cabo todas las posibles ejecuciones para todas las combinaciones de threads OpenMP y MKL del conjunto dado, sino que se realiza, en tiempo de instalación, una búsqueda local en función de un incremento variable determinado por el valor porcentual establecido para la instalación de la rutina.

Respecto al algoritmo de búsqueda local utilizado, indicar que difiere un poco respecto al empleado en la sección 4.4.2. En la Figura 16 se puede observar un esquema en pseudo-código del mismo. Inicialmente, tras especificar los tamaños de las matrices y el valor porcentual a utilizar, se comienza a ejecutar la rutina con el primer tamaño del conjunto de instalación, comenzando con la combinación de threads 1-1 e incrementando de uno en uno el número de threads MKL mientras no se obtenga una ejecución con un tiempo que supere en el porcentaje establecido al mínimo alcanzado hasta el momento. Una vez alcanzado dicho valor, si se ha obtenido un tiempo de ejecución que mejore el mínimo actual, se incrementa en 1 el número de threads OpenMP y se ejecuta la rutina con la combinación de threads que más se ajuste a la optima obtenida en el nivel anterior (en cuanto a número total de threads y sin exceder el máximo número de cores que se estén usando en el sistema), comparando el mínimo actual con los tiempos obtenidos al ejecutar la rutina incrementando y decrementando el número de threads MKL en el nuevo nivel OpenMP hasta alcanzar de nuevo un valor para el tiempo de ejecución que exceda de nuevo el mínimo alcanzado hasta el momento en una cantidad igual al porcentaje especificado. Tras alcanzar la combinación óptima de threads para el primer tamaño, se pasa al siguiente tamaño del conjunto comenzando con dicha combinación de threads y se repite el proceso hasta acabar con los tamaños especificados en el conjunto de instalación. El resultado final de este proceso es la obtención de las combinaciones de threads OpenMP y MKL a usar con cada tamaño, sin necesidad de realizar todas las ejecuciones que se hacían con la búsqueda exhaustiva durante la instalación de la rutina.

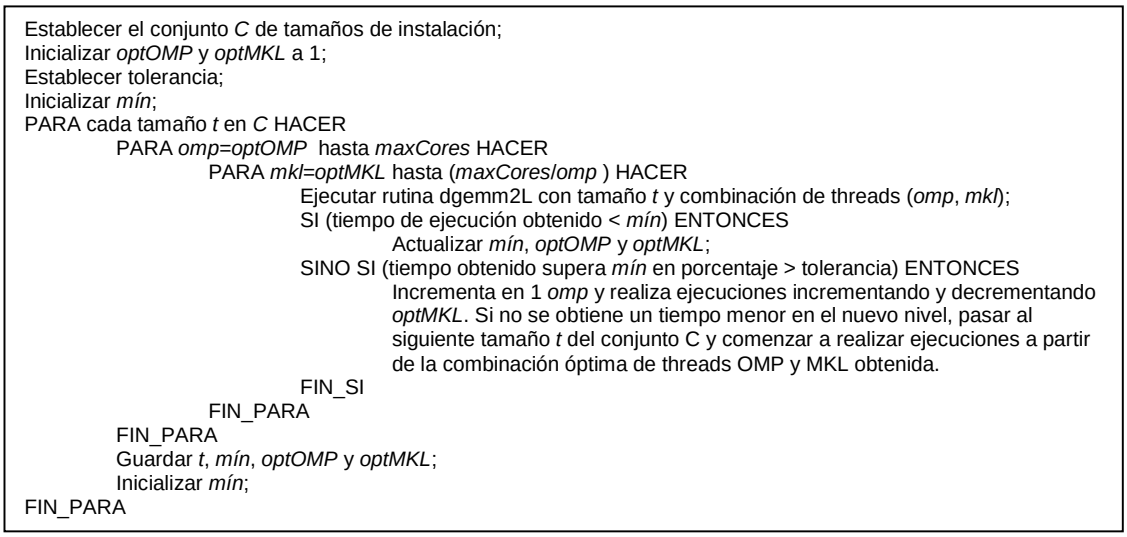


Figura 16. Algoritmo de instalación de OpenMP+MKL mediante búsqueda local con incremento variable

La Tabla 10 muestra los tiempos de ejecución obtenidos durante la instalación en cada uno de estos sistemas con el incremento porcentual indicado. La Tabla 11, por su parte, muestra los resultados obtenidos de aplicar esta técnica en los sistemas Ben, Saturno y Pirineus utilizando un total de 96, 24 y 240 cores, respectivamente. Los números entre paréntesis mostrados representan la combinación de threads OpenMP y MKL con la que se obtiene el tiempo de ejecución indicado.

Ben					
	Exhaustivo	1.00%	10.00%	20.00%	50.00%
t_inst (seg):	1156.45	38.81	33.94	46.74	125.48

Saturno					
	Exhaustivo	1.00%	10.00%	20.00%	50.00%
t_inst (seg):	353.58	36.83	36.83	39.74	44.13

Pirineus					
	Exhaustivo	1.00%	10.00%	20.00%	50.00%
t_inst (seg):	676.49	20.51	18.30	15.22	28.45

Tabla 10. Tiempos de instalación (en segundos) en la técnica de auto-optimización basada en búsqueda local con incremento variable sobre paralelismo OpenMP+MKL, en distintos sistemas para cada uno de los incrementos porcentuales indicados durante la fase de instalación realizada

En cuanto a los tiempos de instalación obtenidos, se puede observar en la Tabla 10 la notable reducción conseguida respecto al empleo de la búsqueda exhaustiva (mostrado en la Tabla 9) lo que demuestra la ventaja de utilizar la búsqueda local en detrimento de no obtener la combinación óptima de threads, pero permitiendo ejecutar la rutina en un tiempo muy próximo al óptimo.

n	Opt.	1.00%	10.00%	20.00%	50.00%
			Ben		
500	0.0050 (23-1)	0.0134 (1-4)	0.0051 (4-6)	0.0051 (4-6)	0.0055 (1-20)
700	0.0102 (7-4)	0.0148 (1-10)	0.0119 (5-6)	0.0119 (5-6)	0.0111 (1-19)
1000	0.0177 (10-4)	0.0297 (1-16)	0.0183 (6-7)	0.0183 (6-7)	0.0246 (1-19)
2000	0.0795 (10-5)	0.0984 (3-15)	0.0826 (6-8)	0.0826 (6-8)	0.0963 (3-16)
3000	0.2191 (25-3)	0.2303 (5-14)	0.2380 (6-10)	0.2380 (6-10)	0.2303 (5-14)
4000	0.5088 (32-2)	0.6291 (6-10)	0.6150 (7-8)	0.6150 (7-8)	1.0728 (6-10)
5000	0.9207 (20-3)	0.9612 (8-7)	0.9612 (8-7)	0.9612 (8-7)	0.9612 (8-7)
			Saturno		
500	0.0038 (6-4)	0.0085 (2-2)	0.0085 (2-2)	0.0085 (2-2)	0.0042 (2-12)
700	0.0098 (6-4)	0.0227 (2-2)	0.0227 (2-2)	0.0227 (2-2)	0.0113 (2-12)
1000	0.0291 (8-3)	0.0325 (3-3)	0.0325 (3-3)	0.0325 (3-3)	0.0295 (2-12)
2000	0.2151 (6-4)	0.2604 (3-3)	0.2604 (3-3)	0.2604 (3-3)	0.2581 (2-10)
3000	0.5205 (1-17)	0.8338 (3-3)	0.8338 (3-3)	0.8338 (3-3)	0.7089 (3-8)
4000	1.2580 (1-17)	2.1135 (3-6)	2.1135 (3-6)	2.1135 (3-6)	1.9754 (3-7)
5000	1.8915 (7-3)	1.9567 (3-7)	1.9567 (3-7)	1.9567 (3-7)	1.9567 (3-7)
			Pirineus		
500	0.0059 (1-24)	0.0075 (4-4)	0.0075 (4-4)	0.0075 (4-4)	0.0075 (4-4)
750	0.0134 (2-16)	0.0160 (4-4)	0.0251 (4-6)	0.0251 (4-6)	0.0251 (4-6)
1000	0.0235 (2-16)	0.0334 (4-3)	0.0264 (4-8)	0.0264 (4-8)	0.0264 (4-8)
2000	0.0797 (5-12)	0.2319 (4-8)	0.0813 (4-15)	0.0813 (4-15)	0.0813 (4-15)
3000	0.2752 (4-15)	0.2752 (4-15)	0.2752 (4-15)	0.2752 (4-15)	0.2752 (4-15)
4000	0.4670 (5-12)	0.4670 (5-12)	0.4670 (5-12)	0.4670 (5-12)	0.4670 (5-12)
5000	0.8598 (10-9)	0.8949 (5-18)	0.8949 (5-18)	0.8949 (5-18)	0.8949 (5-18)

Tabla 11. Tiempos de ejecución (en segundos) obtenidos en distintos sistemas al aplicar la técnica de auto-optimización basada en búsqueda local con incremento variable sobre dos niveles de paralelismo, utilizando el conjunto de tamaños de instalación {500,1000,3000,5000} y el conjunto de tamaños de validación {700,2000,4000}. Opt (óptimo tiempo de ejecución obtenido). 1%, 10%, 20%, 50% (tiempos obtenidos con auto-optimización, al utilizar el incremento porcentual indicado)

Como se puede apreciar en la Tabla 11, en ningún sistema se consigue obtener la combinación óptima de threads OpenMP+MKL, pero sí se observa cómo generalmente los tiempos obtenidos al incrementar el valor porcentual disminuyen o se mantienen más cercanos al óptimo. Asimismo, se puede apreciar cómo incrementar el valor porcentual para un mismo tamaño de problema no va a suponer en todos los casos una reducción en el tiempo de ejecución. Tal es el caso de Ben, donde para tamaño 3000 y porcentaje entre un 10% y un 20% o tamaño 1000 y porcentaje 50%, se observa cómo empeora el tiempo de ejecución alcanzado respecto al obtenido con un 1%. Esto es debido al camino seguido por el algoritmo al incrementar el valor porcentual, pues realiza más ejecuciones y esto lleva a combinaciones de threads diferentes en las que se obtiene un peor tiempo de ejecución. No ocurre lo mismo en Saturno, donde se observa cómo con un 50% se obtiene, para todos los tamaños, tiempos de ejecución más cercanos al óptimo. Por tanto, en función del sistema empleado, el comportamiento observado puede variar. No obstante, tal como demuestran los resultados, utilizar un incremento entre un 10% y un 20% va a permitir obtener, en general, tiempos de ejecución muy cercanos al óptimo.

Capítulo 5. Conclusiones y trabajo futuro

5.1 Conclusiones

La presente tesis de máster comenzó destacando la importancia de aplicar varios niveles de paralelismo sobre rutinas matriciales ofrecidas en librerías de álgebra lineal utilizadas por la comunidad científica, así como indicando la ventaja de aplicar un proceso de auto-optimización destinado a obtener la combinación adecuada de threads a poner en ejecución en cada uno de los niveles de paralelismo establecidos con el fin de obtener una mejora sustancial en el tiempo de ejecución.

El estudio realizado ha tomado como punto de partida la rutina de multiplicación de matrices, pero la metodología y técnicas empleadas se pueden aplicar de la misma forma a rutinas matriciales que trabajen con otros tipos de datos (float, complex...) o realicen otro tipo de operaciones (LU, QR...).

Motivados por el deseo de disponer de una librería compuesta por rutinas matriciales que utilicen dos niveles de paralelismo en su implementación, se ha comenzado su creación con la implementación de la rutina `dgemm2L`, en la que se han incorporado los niveles de paralelismo OpenMP y MKL. Una vez diseñada y codificada, se ha probado y evaluado su funcionamiento sobre diferentes sistemas multicore variando el tamaño de las matrices y el número de threads utilizados en cada nivel y se ha llegado a la conclusión esperada, es decir, que realmente se puede obtener una mejora significativa en el tiempo de ejecución respecto a la ejecución secuencial o el empleo de un solo nivel de paralelismo, pero, para ello, se ha de establecer el número adecuado de threads en cada nivel, razón por la que se han propuesto distintas técnicas de auto-optimización encaminadas a determinar automáticamente la combinación más adecuada de threads OpenMP y MKL a utilizar en la ejecución de la rutina, cuyos resultados también han sido satisfactorios. Por tanto, se puede concluir que el empleo de varios niveles de paralelismo en este tipo de rutinas, unido a un proceso de auto-optimización, va a permitir reducir sustancialmente el tiempo de ejecución, traduciéndose en un aumento de las prestaciones y del rendimiento de la aplicación.

Respecto a los resultados obtenidos en esta tesis, indicar que los experimentos solo se han realizado sobre los sistemas indicados y bajo las condiciones especificadas, por lo que pueden variar ligeramente si se realizan nuevas pruebas o se utilizan otros sistemas. No obstante, constatar que la metodología y las técnicas empleadas para la obtención de los mismos es aplicable sobre otro tipo de rutinas, por lo que creemos que se garantiza la obtención de resultados fiables independientemente del sistema utilizado.

5.2 Resultados de la tesis

Una parte de los resultados obtenidos en el capítulo 3 de la presente tesis de máster fueron expuestos en una reunión del Grupo de Computación Científica y Programación Paralela celebrada el pasado mes de diciembre de 2010. En esa reunión, se analizaron resultados parciales en distintos sistemas sobre el comportamiento experimentado por la rutina de multiplicación de matrices al utilizar dos niveles de paralelismo (OpenMP+MKL).

Asimismo, está prevista la preparación de un artículo de investigación con las técnicas de auto-optimización presentadas en el capítulo 4, así como su inclusión en un workshop de algún congreso de auto-optimización (como el iWAPT: http://iwapt.org/2011/index.html) o de programación paralela (como el iCCS: http://www.iccs-meeting.org/), cuya celebración tenga lugar durante el próximo curso académico.

5.3 Trabajo futuro

En esta tesis se ha mostrado que se pueden utilizar técnicas empíricas para la obtención del número óptimo de threads en rutinas de álgebra lineal utilizando dos niveles de paralelismo. Los resultados obtenidos son satisfactorios, pero es necesario estudiar modificaciones de las técnicas propuestas para mejorar el proceso de toma de decisiones.

A partir del trabajo realizado, se pueden derivar nuevas líneas de investigación que permitan ampliar la funcionalidad alcanzada, bien mediante el empleo de niveles adicionales de paralelismo o mediante la adición de otras rutinas de álgebra lineal al prototipo de librería diseñado. A continuación se enumera cada una de ellas en orden cronológico:

- Estudiar la ganancia obtenida al utilizar la rutina `dgemm2L` en problemas reales donde se utilice la multiplicación de matrices.

- Ampliar el prototipo implementado con nuevas rutinas (LU, QR…).

- Incluir un tercer nivel de paralelismo (MPI+OpenMP+BLAS), analizando su comportamiento en sistemas de gran dimensión, como clusters de computadores y sistemas heterogéneos.

- Desarrollar un prototipo que integre varios niveles de paralelismo y permita su utilización en sistemas híbridos/heterogéneos compuestos por GPUs y multicores.

- Desarrollar una versión con 3 niveles de paralelismo utilizando ScaLAPACK, que determine el número de procesadores y threads OpenMP y BLAS mediante la utilización de algoritmos adaptativos o modelado y análisis empírico, así como el número de procesos por procesador a usar en plataformas heterogéneas.

- Implementar una versión distribuida usando los 3 niveles de paralelismo mencionados (MPI+OpenMP+BLAS), incorporando en ella auto-optimización.

- Refinar la implementación del algoritmo de búsqueda local utilizado en la técnica de auto-optimización con dos niveles de paralelismo, con el fin de evitar recorrer caminos que lleven a la obtención de tiempos de ejecución alejados del óptimo.

- Analizar la combinación de las técnicas empíricas de auto-optimización aquí estudiadas con técnicas basadas en el modelado de rutinas [18].

Bibliografía

[1] ATLAS web page, http://www.netlib.org/atlas/index.html

[2] Intel MKL web page, http://software.intel.com/en-us/intel-mkl/

[3] BLAS web page, http://netlib2.cs.utk.edu/blas/

[4] OpenMP web page, http://openmp.org/wp/

[5] Parallel Computing Group, Universidad de Murcia: http://www.um.es/pcgum/

[6] David A. Bader. *PetaScale Computing: Algorithms and Applications*. Chapman & Hall/CRC. Computational Science Series. 2007.

[7] Michael A. Heroux, Padma Raghavan, Horst D. Simon. *Parallel Processing for Scientific Computing*. SIAM. 2006.

[8] International Conference on Parallel Computing web page, http://www.parco.org/

[9] Parallel Processing and Applied Mathematics web page, http://www.ppam.pl/

[10] International Parallel & Distributed Processing Symposium: http://www.ipdps.org/

[11] LAPACK web page, http://www.netlib.org/lapack/

[12] ScaLAPACK web page, http://www.netlib.org/scalapack/scalapack_home.html

[13] PLAPACK web page, http://www.cs.utexas.edu/~plapack/

[14] Matteo Frigo and Steven G. Johnson. *FFTW: An adaptive software architecture for the FFT*. In Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing.

[15] J. Cuenca. *Optimización automática de software paralelo de álgebra lineal*. Ph. D. Thesis, Junio 2004.

[16] BeBOP web page, http://bebop.cs.berkeley.edu/

[17] J. Cuenca, L. P. García, D. Giménez. *A proposal for autotuning linear algebra routines on multicore platforms*. ICCS 2010, Procedia Computer Science.

[18] J. Cuenca, D. Giménez. *Optimisation of dense linear algebra computation in large NUMA systems through auto-tuned nested parallelism*. (en revisión)

[19] MAGMA web page, http://icl.cs.utk.edu/magma/index.html

[20] S. Akhter and J. Roberts. Multicore Programming. Intel Press. 2006.

[21] PLASMA web page, http://icl.cs.utk.edu/plasma/

[22] Kaushik Datta. Ph.D. Thesis: *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms* (Computer Science Division, U.C. Berkeley, Diciembre 2009).

[23] D. Giménez, A. Lastovetsky. *On the behaviour of the MKL library in multicore shared- memory systems*. XXI Jornadas de Paralelismo, Septiembre 2010.

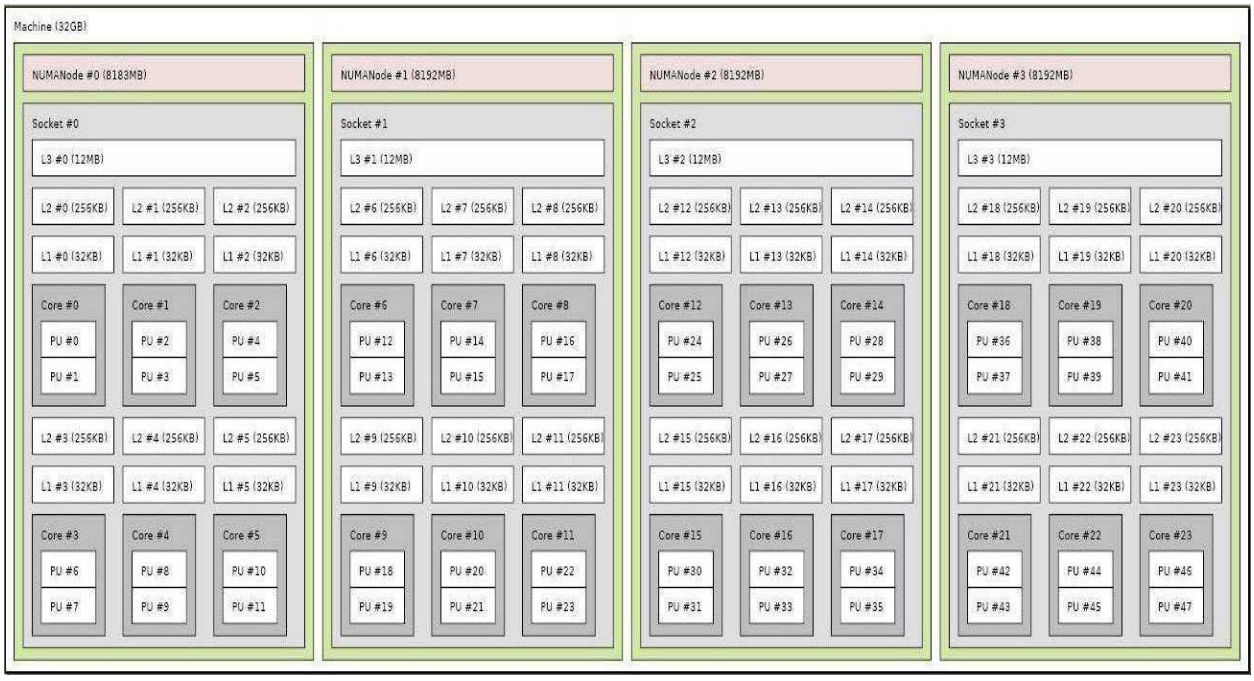
[24] J. Cámara. Multiplicación de Matrices en Sistemas cc-NUMA multicore. Reunión del grupo de investigación de Computación Científica y Programación Paralela, Universidad de Murcia, Diciembre 2010.

[25] CBLAS web page, http://www.mathkeisan.com/usersguide/e/cblas.html

Anexo 1: Arquitectura de Saturno

La siguiente figura muestra de forma esquemática la arquitectura de Saturno. Ha sido generada con la herramienta *hwloc* (http://www.open-mpi.org/projects/hwloc/), un paquete de software que proporciona un modelo abstracto de la topología de arquitecturas modernas que incluyen nodos de memoria NUMA, sockets, cores, cachés compartidas...

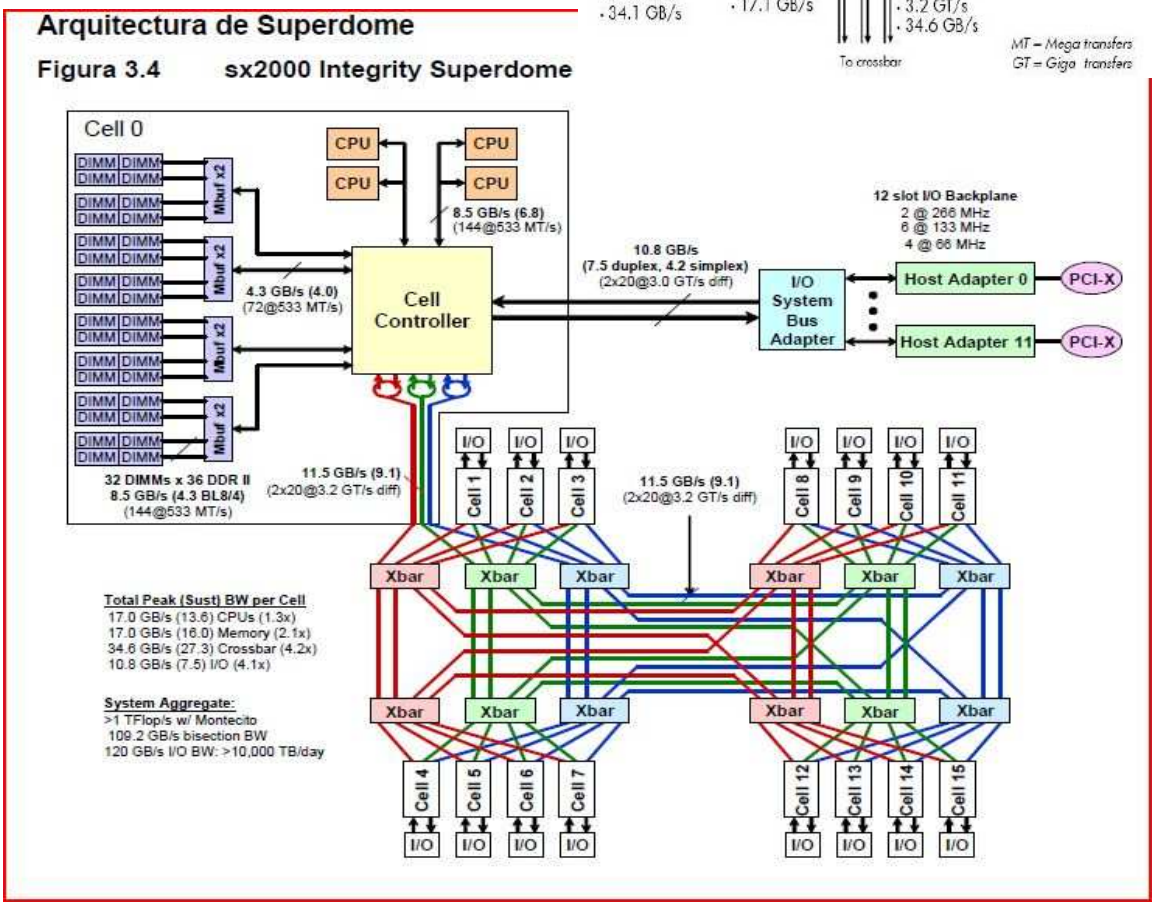
La arquitectura consta de 4 nodos o procesadores hexacore, donde cada core consta teóricamente a su vez de 2 unidades de procesamiento debido a la posibilidad de hyperthreading. Respecto a la jerarquía de memoria, todos los nodos comparten la memoria principal de 32 GB y cada nodo dispone de 3 niveles de caché (L1, L2 y L3), los dos primeros (L1 y L2) dedicados para cada core y el tercero (L3) compartido por todos ellos. Esta estructuración jerárquica de la memoria hace que el sistema se comporte en algunos casos como un sistema NUMA, tal como se refleja en los resultados experimentales obtenidos a lo largo de la tesis.



Anexo 2: Arquitectura de Ben

Ben es un nodo HP Integrity Superdome con una arquitectura cc-NUMA basada en una composición jerárquica de celdas con una interconexión crossbar. Hay dos componentes básicos en la arquitectura: la celda y dos backplane de crossbar.

La celda (derecha) es un SMP con 4 CPUs dual core Itanium-2 y con un controlador de celda ASIC (circuito integrado de aplicación específica) que conecta las CPUs con la memoria local y con los conmutadores crossbar. El ancho de banda de memoria máximo en una celda es de 17.1 GB/s y con los conmutadores crossbar es de 34.5 GB/s.



Los dos backplane crossbar contienen dos conjuntos de tres crossbar ASICs con 8 puertos cada uno, que proporcionan una conexión no bloqueante a las celdas, a otro crossbar o al otro backplane crossbar. Ofrece una malla crossbar completamente conectada con 12 crossbar, donde cada celda se interconecta con 3 crossbar distintos. La interconexión de 16 celdas da lugar a los 128 cores.

Anexo 3: Rutina dgemm2L

```
void dgemm2L(char transA, char transB, int m, int n, int k, double alpha,
             double *A, int lda, double *B, int ldb, double beta, double *C,
             int ldc, int thrOMP, int thrMKL) {

/*    .. Scalar Arguments ..
      double precision alpha, beta
      integer m, n, k, lda, ldb, ldc, thrOMP, thrMKL
      character transA, transB
*     ..
*     .. Array Arguments ..
      double precision A(lda,*), B(ldb,*), C(ldc,*)
*
*  Purpose
*  =======
*
*  dgemm2L performs one of the matrix-matrix operations
*
*     C := alpha*op( A )*op( B ) + beta*C,
*
*  using two levels of parallelism, where  op( X ) is one of
*
*     op( X ) = X  or  op( X ) = X**T,
*
*  alpha and beta are scalars, and A, B and C are matrices, with op( A )
*  an m by k matrix, op( B ) a  k by n matrix and C an m by n matrix.
*
*  Arguments
*  ==========
*
*  transA - char*1
*           On entry, transA specifies the form of op( A ) to be used in
*           the matrix multiplication as follows:
*
*              transA = 'N' or 'n',  op( A ) = A
*
*              transA = 'T' or 't',  op( A ) = A**T
*
*              transA = 'C' or 'c',  op( A ) = A**T
*
*           Unchanged on exit.
*
*  transB - char*1
*           On entry, transB specifies the form of op( B ) to be used in
*           the matrix multiplication as follows:
*
*              transB = 'N' or 'n',  op( B ) = B.
*
*              transB = 'T' or 't',  op( B ) = B**T.
*
*              transB = 'C' or 'c',  op( B ) = B**T.
*
*           Unchanged on exit.
*
*  m      - int
*           On entry,  m  specifies  the number  of rows  of the  matrix
*           op( A )  and of the  matrix  C.  m  must  be at least  zero.
*
*           Unchanged on exit.
```

```
*  n      - int
*           On entry,  n  specifies the number  of columns of the matrix
*           op( B ) and the number of columns of the matrix C. n must be
*           at least zero.
*
*           Unchanged on exit.
*
*  k      - int
*           On entry,  k  specifies  the number of columns of the matrix
*           op( A ) and the number of rows of the matrix op( B ). k must
*           be at least  zero.
*
*           Unchanged on exit.
*
*  alpha  - double precision
*           On entry, alpha specifies the scalar alpha.
*
*           Unchanged on exit.
*
*  A      - double precision array of dimension ( lda, ka ), where ka is
*           k  when  transA = 'N' or 'n',  and is  m  otherwise.
*           Before entry with  transA = 'N' or 'n',  the leading  m by k
*           part of the array  A  must contain the matrix  A,  otherwise
*           the leading  k by m  part of the array  A  must contain  the
*           matrix A.
*
*           Unchanged on exit.
*
*  lda    - int
*           On entry, lda specifies the first dimension of A as declared
*           in the calling (sub) program. When  transA = 'N' or 'n' then
*           lda must be at least  max( 1, m ), otherwise  lda must be at
*           least  max( 1, k ).
*
*           Unchanged on exit.
*
*  B      - double precision array of dimension ( ldb, kb ), where kb is
*           n  when  transB = 'N' or 'n',  and is  k  otherwise.
*           Before entry with  transB = 'N' or 'n',  the leading  k by n
*           part of the array  B  must contain the matrix  B,  otherwise
*           the leading  n by k  part of the array  B  must contain  the
*           matrix B.
*
*           Unchanged on exit.
*
*  ldb    - int
*           On entry, ldb specifies the first dimension of B as declared
*           in the calling (sub) program. When  transB = 'N' or 'n' then
*           ldb must be at least  max( 1, k ), otherwise  ldb must be at
*           least  max( 1, n ).
*
*           Unchanged on exit.
*
*  beta   - double precision
*           On entry,  beta  specifies the scalar  beta.  When  beta  is
*           supplied as zero then C need not be set on input.
*
*           Unchanged on exit.
*
*  C      - double precision array of dimension ( ldc, n )
*           Before entry, the leading  m by n  part of the array  C must
*           contain the matrix  C,  except when  beta  is zero, in which
*           case C need not be set on entry.
```

```
*           On exit, the array  C  is overwritten by the  m by n  matrix
*           ( alpha*op( A )*op( B ) + beta*C ).
*
*  ldc    - int
*           On entry, ldc specifies the first dimension of C as declared
*           in  the  calling  (sub)  program.   ldc  must  be  at  least
*           max( 1, m ).
*
*           Unchanged on exit.
*
*  thrOMP – int
*           On entry, thrOMP specifies the number of threads OpenMP necessaries
*           to  establish the  second level of  parallelism. thrOMP must be at
*          least one.
*
*           Unchanged on exit.
*
*  thrMKL - int
*           On entry, thrMKL specifies the number of threads MKL necessaries
*           to establish  the first level of  parallelism. thrMKL must be at
*          least one.
*
*           Unchanged on exit.
*
*  Further Details
*  ===============
*
*  Two Level Blas routine.
*
*  -- Written on 07-September-2011.
*     Jesús Cámara, Parallel Computing Group University of Murcia
*
*/

   // Local variables
   int i, j, nthr, idthr, inicio, numFilas;

  // Test the input parameters and quick return if possible.
   if ((m==0 || n==0) || ((alpha==0 || k==0) && beta==1)) return;

  // And if alpha equals zero…
   if (alpha == 0) {
      if (beta == 0) {
         for (i=0; i<m; i++)
            for (j=0; j<n; j++)
               C[i*ldc+j] = 0.;
      }
      else {
         for (i=0; i<m; i++)
            for (j=0; j<n; j++)
               C[i*ldc+j] = beta*C[i*ldc+j];
      }
      return;
   }

   // Adjust the leading-dimension of A and B according to transA and transB
   lda = ((transA == 'N') ? k : m);
   ldb = ((transB == 'N') ? n : k);

   // Set the number of threads OMP and the number of threads MKL to establish
   // the two levels of parallelism
   omp_set_num_threads(thrOMP);
   mkl_set_num_threads(thrMKL);
```

```
   // Start the operations
   #pragma omp parallel private(idthr,nthr,numFilas,inicio)
   {
      // Get the number of threads OpenMP established and its id
      nthr=omp_get_num_threads();
      idthr=omp_get_thread_num();

      // If the amount of rows of A not is a multiple of the number of threads,
      // assign remaining rows between threads
      if (idthr < m%nthr)
         numFilas=m/nthr+1;
      else
         numFilas=m/nthr;

      // Indicates the beginning of the block of rows assigned to each thread
      if (idthr <= m%nthr)
         inicio=(m/nthr+1)*idthr;
      else
         inicio=(m/nthr*idthr + m%nthr);

      // Performs the matrix multiplication using the CBLAS interface
      if (numFilas != 0)
      {
         if ((transA == 'N') && (transB == 'N'))
              cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, numFilas, n,
                          k, alpha, &A[inicio*lda], lda, B, ldb, beta,
                          &C[inicio*ldc], ldc);
         else if ((transA == 'T') && (transB == 'N'))
              cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, numFilas, n,
                          k, alpha, &A[inicio], lda, B, ldb, beta,
                          &C[inicio*ldc], ldc);
         else if ((transA == 'N') && (transB == 'T'))
              cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, numFilas, n,
                          k, alpha, &A[inicio*lda], lda, B, ldb, beta,
                          &C[inicio*ldc], ldc);
         else
              cblas_dgemm(CblasRowMajor, CblasTrans, CblasTrans, numFilas, n, k,
                          alpha, &A[inicio], lda, B, ldb, beta,
                          &C[inicio*ldc], ldc);
     }
   }
 }

/* End of dgemm2L */
```

Anexo 4: Programa de Prueba dgemm2Lx

```
#include "io.h"
#include "2LBlas.h"

double mseconds() {
  struct timeval tv;
  struct timezone tz;

  gettimeofday(&tv, &tz);
  return (tv.tv_sec*1000000 + tv.tv_usec);
}

main()
{
  char cad[2],transA,transB;
  double *a, *b, *c, *copia;
  double start,end,time,alpha, beta;
  int fa,ca,lda,fb,cb,ldb,fc,cc,ldc,res,nCores,thrOMP,thrMKL;

  printf("\n Insert the number of rows and columns of the matrix A: ");
  res=scanf("%d",&fa);
  res=scanf("%d",&ca);
  lda=ca; fb=ca;

  printf("\n Insert the number of columns of the matrix B: ");
  res=scanf("%d",&cb);
  ldb=cb; fc=fa;
  cc=cb; ldc=cc;

  printf("\n Insert the value of alpha (double): ");
  res=scanf("%lf",&alpha);

  printf("\n Insert the value of beta (double):  ");
  res=scanf("%lf",&beta);

  printf("\n Insert the number of threads OMP: ");
  res=scanf("%d",&thrOMP);

  printf("\n Insert the number of threads MKL: ");
  res=scanf("%d",&thrMKL);

  nCores=omp_get_max_threads()/2;

  if ((thrOMP + thrMKL) > nCores) {
     printf("Threads_OMP + Threads_MKL > Num_Cores\n");
     exit(-1);
  }

  omp_set_nested(1);
  omp_set_dynamic(0);
  mkl_set_dynamic(0);

  a=(double *) malloc(sizeof(double)*fa*ca);
  b=(double *) malloc(sizeof(double)*fb*cb);
  c=(double *) malloc(sizeof(double)*fc*cc);

  copia=(double *) malloc(sizeof(double)*fc*cc);
  memset(copia, 0, sizeof(double)*fc*cc);
  memset(c, 0, sizeof(double)*fc*cc);
```

```
generar_matriz_ld(copia,fc,cc,ldc,-1.,1.,1);
generar_matriz_ld(c,fc,cc,ldc,-1.,1.,1);

generar_matriz_ld(a,fa,ca,lda,-9.,9.,0);
#ifdef DEBUG
   printf("\n");
   escribir_matriz_ld(a,fa,ca,lda);
#endif

generar_matriz_ld(b,fb,cb,ldb,-9.,9.,0);
#ifdef DEBUG
   printf("\n");
   escribir_matriz_ld(b,fb,cb,ldb);
#endif

printf("\n Do you want to transponse matrix A? (y/n): ");
res=scanf("%s",cad);

if (strcmp(cad, "y") == 0) {
  trasponer(a,fa,ca,lda);
  printf("\n Matriz A transponse!\n");
  escribir_matriz_ld(a,ca,fa,fa);
  transA = 'T';
}
else
  transA = 'N';

printf("\n Do you want to transponse matrix B? (y/n): ");
res=scanf("%s",cad);

if (strcmp(cad, "y") == 0) {
  trasponer(b,fb,cb,ldb);
  printf("\n Matriz B transponse!\n");
  escribir_matriz_ld(b,cb,fb,fb);
  transB = 'T';
}
else
  transB = 'N';

/* ============================== MM_SECUENCIAL ============================== */

start = mseconds();

if (transA == 'N')
{
   if (transB == 'N')
     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, fa, cb, ca, alpha,
                 a, lda, b, ldb, beta, copia, ldc);
   else
     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, fa, cb, ca, alpha, a,
                 lda, b, fb, beta, copia, ldc);
}
else {
   if (transB == 'N')
     cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, fa, cb, ca, alpha, a,
                 fa, b, ldb, beta, copia, ldc);
   else
     cblas_dgemm(CblasRowMajor, CblasTrans, CblasTrans, fa, cb, ca, alpha, a,
                 fa, b, fb, beta, copia, ldc);
}

end = mseconds();
time = (end - start)/1000000.;
```

```
  #ifdef TIME
     printf("\n dgemm: size = %d x %d , time = %.6f sec\n",fa,cb,time);
  #endif

  #ifdef DEBUG
     printf("\n");
     escribir_vector(copia,fc*cc);
  #endif

  /* ============================== DGEMM_2L ============================== */

  start = mseconds();

  dgemm2L(transA,transB,fa,cb,ca,alpha,a,lda,b,ldb,beta,c,ldc,thrOMP,thrMKL);

  end = mseconds();
  time = (end - start)/1000000.;

  #ifdef TIME
     printf("\n dgemm2L: size = %d x %d , time = %.6f sec\n",fa,cb,time);
  #endif

  #ifdef DEBUG
     printf("\n");
     escribir_vector(c,fc*cc);
  #endif

  comparar_vectores(c,copia,fc*cc);

  free(a);
  free(b);
  free(c);
  free(copia);
}
```

Anexo 5: Uso de dgemm2L desde Fortran

```
!*****************************************************************************
!      D G E M M _ 2L  Example Program Text
!*****************************************************************************

      program   DGEMM_2L_MAIN
*
      character*1      transa, transb
      integer          m, n, k
      integer          lda, ldb, ldc
      integer          thrOMP, thrMKL
      double precision alpha, beta
      integer          rmaxa, cmaxa, rmaxb, cmaxb, rmaxc, cmaxc
      parameter        (rmaxa=4, cmaxa=4, rmaxb=5, cmaxb=5, rmaxc=4, cmaxc=5)
      parameter        (lda=rmaxa, ldb=rmaxb, ldc=rmaxc)
      double precision a(rmaxa,cmaxa), b(rmaxb,cmaxb), c(rmaxc,cmaxc)

*     External Subroutines
      external dgemm2L, PrintArrayD
*
*     Executable Statements
*
      print*
      print*,'DGEMM_2L  EXAMPLE PROGRAM'
*
*     Read input data
      read*
      read*, m, n, k
      read*, alpha, beta
      read*, thrOMP, thrMKL
      read 100, transa, transb

      if ((transa.eq.'N').or.(transa.eq.'n')) then
        if (m.gt.rmaxa.or.k.gt.cmaxa) then
          print*, ' Insufficient memory for arrays'
          goto 999
        end if
        read*, ((a(i,j),j=1,k),i=1,m)
      else
        if (k.gt.rmaxa.or.m.gt.cmaxa) then
          print*, ' Insufficient memory for arrays'
          goto 999
        end if
        read*, ((a(i,j),j=1,m),i=1,k)
      end if

      if ((transb.eq.'N').or.(transb.eq.'n')) then
        if (k.gt.rmaxb.or.n.gt.cmaxb) then
          print*, ' Insufficient memory for arrays'
          goto 999
        end if
        read*, ((b(i,j),j=1,n),i=1,k)
      else
        if (n.gt.rmaxb.or.k.gt.cmaxb) then
          print*, ' Insufficient memory for arrays'
          goto 999
        end if
        read*, ((b(i,j),j=1,k),i=1,n)
      end if
```

```
      if (m.gt.rmaxc.or.n.gt.cmaxc) then
          print*, ' Insufficient memory for arrays'
          goto 999
      end if
      read*, ((c(i,j),j=1,n),i=1,m)
*
*     Print input data
      print*
      print*, 'INPUT DATA'
      print 101, m, n, k
      print 102, alpha, beta
      print 103, transa, transb
      print 104, thrOMP, thrMKL

      if ((transa.eq.'N').or.(transa.eq.'n')) then
        call PrintArrayD(0,0,m,k,a,lda,'A')
      else
        call PrintArrayD(0,0,k,m,a,lda,'A')
      end if

      if ((transb.eq.'N').or.(transb.eq.'n')) then
        call PrintArrayD(0,0,k,n,b,ldb,'B')
      else
        call PrintArrayD(0,0,n,k,b,ldb,'B')
      end if

      call PrintArrayD(0,0,m,n,c,ldc,'C')
*
*    Call dgemm2L subroutine
     call dgemm2L(transa,transb,m,n,k,alpha,a,lda,b,ldb,beta,c,ldc,thrOMP,thrMKL)
*
      print*
      print*, 'OUTPUT DATA'

      call PrintArrayD(1,0,m,n,c,ldc,'C')
      stop

 100  format(2(a1,1x))
 101  format(7x,'M=',i1,'  N=',i1,'  K=',i1)
 102  format(7x,'ALPHA=',f5.2,' BETA=',f5.2)
 103  format(7x,'TRANSA=',a1, ' TRANSB=',a1)
 104  format(7x,'thrOMP=',i1,'  thrMKL=',i1)

 999  stop 1
      end
```

Anexo 6: Implementación de la toma de decisiones en la rutina dgemm2L

```
// Header file with auto-tuning information
#include "dgemm2L.h"

void dgemm2L(char transA, char transB, int m, int n, int k, double alpha,
             double *A, int lda, double *B, int ldb, double beta, double *C,
             int ldc, int maxCores) {

   int thrOMP, thrMKL;
   int i, j, nthr, idthr, inicio, numFilas;

   // Test the input parameters and quick return if possible.
   if ((m==0 || n==0) || ((alpha==0 || k==0) && beta==1)) return;

   // And if alpha equals zero…
   if (alpha == 0) {
      if (beta == 0) {
         for (i=0; i<m; i++)
            for (j=0; j<n; j++)
               C[i*ldc+j] = 0.;
      }
      else {
         for (i=0; i<m; i++)
            for (j=0; j<n; j++)
               C[i*ldc+j] = beta*C[i*ldc+j];
      }
      return;
   }

   // Select the most appropriate number of threads OpenMP and MKL using the
   // information saved in the header file and depending of the specified size
   if (m < tamInst[0])
   {
      thrOMP=(1+optOMP[0])/2;
      thrMKL=(1+optMKL[0])/2;
   }
   else if (m > tamInst[TAM_CJTO-1])
   {
      thrOMP=(maxCores+optOMP[TAM_CJTO-1])/2;
      thrMKL=(maxCores+optMKL[TAM_CJTO-1])/2;
   }
   else {
      for (i=0; i<TAM_CJTO-1; i++)
      {
         if (m == tamInst[i])
         {
            thrOMP=optOMP[i];
            thrMKL=optMKL[i];
         }
         else if ((m > tamInst[i]) && (m < tamInst[i+1]))
         {
            thrOMP=(optOMP[i]+optOMP[i+1])/2;
            thrMKL=(optMKL[i]+optMKL[i+1])/2;
         }
       }
   }
```

```
  // Set the number of threads OMP and the number of threads MKL to establish
  // the two levels of parallelism
  omp_set_num_threads(thrOMP);
  mkl_set_num_threads(thrMKL);

  // Adjust the leading-dimension of A and B according to transA and transB
  lda = ((transA == 'N') ? k : m);
  ldb = ((transB == 'N') ? n : k);

  // Start the operations
  #pragma omp parallel private(idthr,nthr,numFilas,inicio)
  {
     // Get the number of threads OpenMP established and its id
     nthr=omp_get_num_threads();
     idthr=omp_get_thread_num();

     // If the amount of rows of A not is a multiple of the number of threads,
     // assign remaining rows between threads
     if (idthr < m%nthr)
        numFilas=m/nthr+1;
     else
        numFilas=m/nthr;

     // Indicates the beginning of the block of rows assigned to each thread
     if (idthr <= m%nthr)
        inicio=(m/nthr+1)*idthr;
     else
        inicio=(m/nthr*idthr + m%nthr);

     // Performs the matrix multiplication using the CBLAS interface
     if (numFilas != 0)
     {
        if ((transA == 'N') && (transB == 'N'))
             cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, numFilas, n,
                         k, alpha, &A[inicio*lda], lda, B, ldb, beta,
                         &C[inicio*ldc], ldc);
        else if ((transA == 'T') && (transB == 'N'))
             cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, numFilas, n,
                         k, alpha, &A[inicio], lda, B, ldb, beta,
                         &C[inicio*ldc], ldc);
        else if ((transA == 'N') && (transB == 'T'))
             cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, numFilas, n,
                         k, alpha, &A[inicio*lda], lda, B, ldb, beta,
                         &C[inicio*ldc], ldc);
        else
             cblas_dgemm(CblasRowMajor, CblasTrans, CblasTrans, numFilas, n, k,
                         alpha, &A[inicio], lda, B, ldb, beta,
                         &C[inicio*ldc], ldc);
    }
  }
}
```