UNIVERSIDAD JAIME I DE CASTELLÓN
E. S. DE TECNOLOGÍA Y CIENCIAS EXPERIMENTALES

Resolución de Sistemas de Ecuaciones Lineales Banda Sobre Procesadores Actuales y Arquitecturas Multihebra. Aplicaciones en Control

Castellón, Octubre de 2008

Tesis Doctoral presentada por: Alfredo Remón Gómez
Dirigida por: Enrique S. Quintana Ortí
Gregorio Quintana Ortí

UNIVERSIDAD JAIME I DE CASTELLÓN
E. S. DE TECNOLOGÍA Y CIENCIAS EXPERIMENTALES

Resolución de Sistemas de Ecuaciones Lineales Banda Sobre Procesadores Actuales y Arquitecturas Multihebra. Aplicaciones en Control

Alfredo Remón Gómez

Índice general

Lista de Figuras

Lista de Tablas

Capítulo 1

Resolución de problemas de álgebra lineal con estructura banda

1.1. Introducción

1.1.1. Motivación y objetivos

Problemas de álgebra lineal como, por ejemplo, sistemas de ecuaciones lineales o problemas de mínimos cuadrados, aparecen en un amplio abanico de aplicaciones de casi todas las áreas científicas y tecnológicas. En particular, problemas de este tipo surgen en el análisis de la resistencia de estructuras de hormigón, la estimación de la órbita de los electrones, la evaluación del campo gravitatorio terrestre, la simulación de circuitos VLSI, el estudio de las propiedades de nanocristales semiconductores, la detección de oclusiones en vasos sanguíneos mediante resonancia magnética, o la simulación del comportamiento de componentes estructurales de aviación. Esta relación no pretende ser más que una breve muestra de la variedad de aplicaciones en las que aparecen problemas de álgebra lineal. En todas ellas, la resolución de estos problemas supone la parte computacionalmente más costosa en la obtención de una respuesta.

En ocasiones, la matriz ligada al problema de álgebra lineal presenta un gran número de elementos nulos. En estos casos, el aprovechamiento de esta propiedad puede reducir considerablemente el coste computacional y de almacenamiento de resolución del problema. Cuando los elementos no nulos de la matriz se distribuyen en ésta de forma "aleatoria" (es decir, sin un patrón evidente), entonces nos encontramos ante un problema de álgebra lineal *dispersa*. Si, en cambio, estos elementos no nulos tienden a agruparse en unas pocas entradas alrededor de la diagonal principal de la matriz, entonces se trata de un problema de álgebra lineal con una estructura *banda*. La figura 1.1 ilustra la estructura diferente de una matriz dispersa y otra banda. La matriz dispersa de la figura surge en un problema de cinética química correspondiente a un estudio de la polución atmosférica (matriz FS_183_1 de la colección *Matrix Market* [65]). La matriz banda aparece en un problema ligado al cálculo de la estructura de una presa (matriz BCSSTK16 de esa misma colección de matrices).

Los problemas con estructura banda se dan en aplicaciones científico-tecnológicas de forma natural y, a menudo, responden a la propia estructura del sistema subyacente. Así, por ejemplo, la colección *Matrix Market* incluye 17 casos con estructura banda que aparecen ligados a la resolución de sistemas de ecuaciones lineales y problemas de mínimos cuadrados. Otra fuente importante de problemas con estructura banda se obtiene al reorganizar las matrices de problemas de álgebra lineal dispersa mediante algoritmos de reetiquetado de grafos. Un ejemplo es el algoritmo *reverse CutHill-McKee* [37], que tiende a agrupar las entradas no nulas de la matriz alrededor de la diagonal.

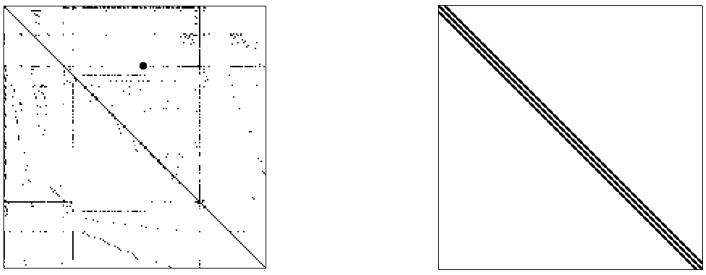


Figura 1.1: Patrón de dispersidad (elementos no nulos) de una matriz dispersa (izquierda) y otra banda (derecha).

Esta reorganización previa es una estrategia utilizada comúnmente por paquetes de resolución de problemas de álgebra lineal dispersa pues, a diferencia de los métodos más habituales para resolver problemas dispersos, los costes computacional y de almacenamiento de resolver los problemas con estructura banda están acotados y son conocidos *a priori*.

A la vista de estas aplicaciones, queda claro el interés que se ha generado y que sigue existiendo en la actualidad por desarrollar bibliotecas de rutinas eficientes para resolver problemas de álgebra lineal (con estructura densa, dispersa o banda) sobre las arquitecturas de computadores del momento. LINPACK, EISPACK [31, 84] y *BLAS*-1 son ejemplos de bibliotecas pioneras en esta línea, desarrolladas en la década de los 70 para los procesadores vectoriales de la época. Más adelante, con la aparición de una organización jerárquica de la memoria (con la inclusión de memorias caché), para compensar la creciente disparidad entre las velocidades del procesador y de la memoria, estas bibliotecas tuvieron que reescribirse por completo, dando lugar a *BLAS*-3 y *LAPACK* [67, 4], las bibliotecas más utilizadas en la actualidad.

LAPACK es una biblioteca de rutinas para *operaciones avanzadas* de álgebra lineal como la resolución de sistemas de ecuaciones lineales, el cálculo del rango numérico, problemas de valores propios, etc. En cambio, *BLAS* define únicamente la especificación (interfaz y funcionalidad) de una colección de rutinas para *operaciones básicas* de álgebra lineal, quedando la implementación de las mismas en manos de los fabricantes de procesadores. El propósito perseguido por este diseño es doble: aislar a los algoritmos avanzados incluidos en *LAPACK* de los cambios en la arquitectura de destino pero, al mismo tiempo, permitir una ejecución eficiente de las rutinas de *LAPACK*, basando la mayor parte de los cálculos realizados en las rutinas de *BLAS*, y haciendo uso de una implementación optimizada de *BLAS* para la arquitectura destino.

Los enormes beneficios que pueden obtenerse al explotar la estructura de las matrices banda provocaron que, desde un principio, *BLAS* y *LAPACK* incluyesen en su diseño rutinas para la resolución de sistemas de ecuaciones lineales de este tipo. Sin embargo, las implementaciones de *BLAS* banda que ponen en práctica estas buenas intenciones están, en la mayoría de los casos, poco optimizadas y la funcionalidad dista de ser completa.

En respuesta a las dos carencias apuntadas (escasa optimización y falta de funcionalidad), el *objetivo general de esta tesis es el diseño, desarrollo y evaluación de una biblioteca de rutinas para la resolución de sistemas de ecuaciones lineales y problemas de mínimos cuadrados con estructura banda sobre las arquitecturas de altas prestaciones actuales.*

El objetivo general de la tesis se plasma en la siguiente *lista inicial de objetivos específicos:*

- Evaluar la eficiencia de las rutinas de *BLAS* banda en las implementaciones más difundidas actualmente sobre procesadores de altas prestaciones.
- En aquellos casos donde se detecte una falta de optimización en las rutinas de *BLAS* banda,

estudiar las causas y proponer nuevas implementaciones más eficientes.

- Completar la funcionalidad de *BLAS* banda con el desarrollo de nuevas rutinas eficientes para operaciones básicas como el producto de matrices o la resolución de múltiples sistemas triangulares de ecuaciones lineales.

- Evaluar la eficiencia de las rutinas de *LAPACK*banda para la resolución de sistemas de ecuaciones lineales y proponer nuevas rutinas con funcionalidad no cubierta actualmente por la biblioteca para este tipo de problemas.

- Completar la funcionalidad de *LAPACK* banda con el desarrollo de nuevas rutinas eficientes para la resolución de problemas de mínimos cuadrados.

En operaciones con matrices banda como el producto de matrices o la factorización por un método directo el coste es proporcional a la dimensión de la matriz y al cuadrado del tamaño de la banda. A medida que estos factores crecen, este coste se incrementa notablemente y se vuelve interesante la utilización de varios procesadores para llevar a cabo estas operaciones. Siguiendo la proliferación de las arquitecturas paralelas con memoria distribuida de principios de la década de los 90 y, más recientemente, la generalización de los *clusters* de computadores personales, se desarrollaron las bibliotecas PBLAS y ScaLAPACK [18], versiones de *BLAS* y *LAPACK* basadas en el paradigma de programación de paso de mensajes, con rutinas específicas que operaban con matrices banda. Otros intentos de diseño de rutinas para operaciones con matrices banda sobre arquitecturas paralelas con memoria distribuida pueden encontrarse en [77, 43, 23].

Durante muchos años, la presunta falta de escalabilidad de las arquitecturas paralelas con memoria compartida ha relegado a un segundo plano el desarrollo de bibliotecas paralelas de álgebra lineal para este tipo de plataformas. Sin embargo, con la eclosión de los procesadores multinúcleo experimentada en estos últimos cuatro años, este planteamiento ya no es válido. Actualmente existen procesadores comerciales con distinto número de núcleos: 2 (Intel Dual Core y AMD Athlon 64 X2), 4 (Intel Quad-Core y AMD quad-core), 6 (SiCortex SC5832), 8 (SUN UltraSparc T1), 1+8 (Cell BE), etc. Las previsiones apuntan a que el número de núcleos se doblará con cada nueva generación de procesadores (cada año y medio, aproximadamente) [2, 9, 19]. Así, existe la previsión realista de que en el año 2014 los procesadores incluyan del orden de 128 núcleos. Este tipo de sistemas es lo que hemos caracterizado en la Tesis como *Arquitecturas Multihebra*.

En línea con el objetivo principal de la tesis, y motivado por la necesidad de una respuesta eficiente en el marco de los futuros procesadores multinúcleo, se plantea una lista complementaria de objetivos específicos para la tesis:

- Evaluar la eficiencia de las rutinas de *BLAS* y *LAPACK* para operaciones de alto coste computacional con matrices banda sobre arquitecturas paralelas con memoria compartida.

- En aquellos casos donde se detecte un rendimiento escaso, estudiar las causas y proponer alternativas más eficientes, tomando en consideración las prestaciones y, habida cuenta del grado de paralelismo que se plantea para los futuros procesadores multinúcleo, la escalabilidad de las soluciones.

Además, con el propósito de demostrar los beneficios obtenidos como fruto de la consecución de los objetivos anteriores, se plantea un último objetivo para la tesis doctoral, consistente en:

- Evaluar el impacto de los resultados anteriores sobre aplicaciones reales que involucren problemas con estructura banda.

1.1.2. Estructura de la memoria

A continuación se expone la organización de la memoria de la tesis. En lo que resta de este capítulo se revisa el estado actual de las bibliotecas de álgebra lineal (densa y banda) *BLAS* y *LAPACK*; seguidamente, se introduce una notación de alto nivel, tomada del proyecto FLAME [41, 15], que se usará para la presentación de los algoritmos que operan sobre matrices banda; en tercer lugar, se repasan brevemente los detalles principales de la arquitectura y la organización de los sistemas de computación de altas prestaciones considerados en la tesis, que incluyen instancias de procesadores superescalares (Intel Xeon) y VLIW (Intel Itanium2), y sistemas multiprocesador (de memoria compartida) construidos tomando como bloques básicos a los procesadores Xeon, Itanium2 y Opteron. Para concluir este capítulo, se ofrecen unos pequeños apuntes sobre evaluación de prestaciones adaptados al cálculo de operaciones de álgebra lineal.

En el Capítulo 2 se aborda el diseño, desarrollo y evaluación de rutinas para el cálculo de operaciones básicas del álgebra lineal banda incluidas en la especificación de *BLAS*-2 sobre procesadores superescalares y VLIW. Entre estas operaciones se considera el producto de una matriz banda por un vector (donde la matriz banda puede, además, presentar una estructura simétrica o triangular), y la resolución de sistemas de ecuaciones lineales donde la matriz de coeficientes presenta simultáneamente una estructura triangular y banda.

En el Capítulo 3 se propone la especificación (interfaz y funcionalidad) de un conjunto de nuevas rutinas para el cálculo de operaciones básicas del álgebra lineal banda con una funcionalidad perteneciente al nivel 3 de *BLAS*. Entre las rutinas consideradas, destacamos el producto de dos matrices (una general y otra banda, o las dos con estructura banda), y la resolución de múltiples sistemas de ecuaciones lineales que comparten la misma matriz de coeficientes con estructura triangular banda. Además, estas especificaciones se plasman en un conjunto de rutinas para operaciones con matrices banda, que se evalúan sobre procesadores superescalares/VLIW y sistemas multiprocesador.

En el Capítulo 4 se analiza el rendimiento de las implementaciones de las factorizaciones banda de Cholesky y LU (con pivotamiento parcial de filas) incluidas en la biblioteca *LAPACK* sobre procesadores superescalares/VLIW y sistemas multiprocesador. Fruto de la evaluación sobre el segundo tipo de plataformas, se proponen nuevas rutinas de factorización que, aumentando el grano de las operaciones realizadas en cada iteración, consiguen mejorar el rendimiento paralelo. Además, para matrices con ancho de banda grande se propone una nueva estrategia para extraer el paralelismo que permite solapar operaciones de diferentes iteraciones de forma dinámica y que ofrece un mayor rendimiento en arquitecturas paralelas con un número de procesadores elevado.

En el Capítulo 5 se diseñan y desarrollan nuevas rutinas para el cálculo de la factorización LU sin pivotamiento, de especial interés en la resolución de ciertos tipos de sistemas de ecuaciones lineales, y la factorización QR, utilizada en la resolución de problemas lineales de mínimos cuadrados. Las nuevas rutinas se evalúan tanto sobre procesadores superescalares y VLIW como sobre sistemas multiprocesador. En este segundo tipo de plataformas, se aplican las mismas estrategias de aumento de la granularidad empleadas en las factorizaciones de Cholesky y LU con pivotamiento para mejorar el rendimiento de las rutinas paralelas.

En el Capítulo 6 se muestran los beneficios que se obtienen al aplicar las nuevas rutinas banda en la resolución de problemas de reducción de modelos y, más en concreto, en la solución de sistemas de ecuaciones lineales con estructura banda que aparecen en aplicaciones reales.

Finalmente, en el Capítulo 7 se ofrecen las conclusiones generales del trabajo desarrollado en esta tesis, se relacionan los principales resultados obtenidos, en forma de publicaciones, y se apuntan las líneas abiertas de investigación.

1.2. BLAS

Los problemas fundamentales del álgebra lineal, como la resolución de sistemas de ecuaciones lineales o el cálculo de valores propios, están presentes en gran número de aplicaciones de ciencia e ingeniería, en ámbitos tan dispares como el cálculo de estructuras, el control automático, el diseño de circuitos integrados o la simulación de reacciones químicas. Así mismo, durante la resolución de estos problemas aparecen repetidamente un conjunto reducido de *operaciones básicas* como, por ejemplo, el cálculo del producto escalar de dos vectores, la resolución de un sistema triangular de ecuaciones lineales o el producto de dos matrices. *BLAS* (*Basic Linear Algebra Subprograms*) es una especificación de una colección de rutinas para operaciones básicas de álgebra lineal densa, que define la funcionalidad y la interfaz que deben presentar las rutinas de la biblioteca. Durante el desarrollo de *BLAS* se contó con la colaboración de especialistas de diferentes áreas de conocimiento [33], circunstancia que le da un especial valor, pues de esta forma se consiguió una especificación que cubre las necesidades requeridas en diversos campos [83].

La especificación *BLAS* comprende un compromiso entre funcionalidad y simplicidad. Por un lado, trata de mantener el número de rutinas y de sus parámetros dentro de unos márgenes razonables y, por otro, intenta ofrecer una funcionalidad amplia. Un ejemplo de la versatilidad de esta biblioteca es la representación de un vector: los elementos no tienen forzosamente que estar contiguos en memoria, sino que habitualmente se incluye un parámetro en la interfaz de la rutina que define la separación entre dos elementos consecutivos del vector.

Existe una implementación de *BLAS* genérica publicada en internet [67], pero lo que realmente hace útil a *BLAS* son las implementaciones específicas para arquitecturas *hardware*. Desde un principio, la implementación optimizada de las rutinas de *BLAS* ha sido una tarea desarrollada por los fabricantes de procesadores. Así, existen bibliotecas propias de AMD (ACML) [1], Intel (MKL) [50], IBM (ESSL) [88] y SUN (Sun Performance Library) [47], pero también existen otras implementaciones como *GotoBLAS* [89] y *ATLAS* [93] que son desarrolladas por investigadores a título propio. En general, el código de cada rutina de estas implementaciones está diseñado con el objetivo de aprovechar eficientemente los recursos de una arquitectura específica, optimizando su rendimiento.

Desde sus inicios en los años 70, *BLAS* ha tenido gran relevancia en la resolución de problemas de álgebra lineal. Su gran fiabilidad, y sobre todo su eficiencia, propiciaron que otras bibliotecas fueran diseñadas para hacer uso internamente de las rutinas de *BLAS*. Además de la fiabilidad y la eficiencia, el uso de *BLAS* aporta otras ventajas adicionales:

- Legibilidad de código: los nombres de las rutinas expresan unívocamente su funcionalidad.
- Portabilidad y eficiencia: al migrar el código a otra máquina, si se utiliza la versión *BLAS* específica para la nueva arquitectura, se seguirá contando con un código muy eficiente.
- Documentación: existe información precisa para todas las rutinas de *BLAS*.

A menudo *BLAS* está implementado en C y Fortran y, en ocasiones, en ensamblador, puesto que de esta forma se puede generar un código más eficiente.

1.2.1. Niveles de BLAS

Cuando se inició el desarrollo de *BLAS*, a principios de la década de los 70, los computadores más potentes utilizaban procesadores vectoriales. Tomando como base este tipo de procesador, *BLAS* se diseñó inicialmente con un grupo reducido de operaciones sobre vectores (*BLAS* de nivel 1 o, simplemente, *BLAS*-1). El principal objetivo de la especificación *BLAS* era que se generaran

implementaciones específicas para cada arquitectura. Así, los creadores de *BLAS* defendieron desde un principio las ventajas de que operaciones básicas, como el producto escalar de dos vectores, fueran implementadas por los diseñadores de la arquitectura *hardware*, ya que ellos pueden generar código más robusto y eficiente [58].

Este conjunto de rutinas proporcionaba una funcionalidad reducida, ya que tan sólo comprendía un número limitado de operaciones sobre vectores, de modo que en 1987 se amplió *BLAS* con un conjunto de rutinas que comprenden el segundo nivel de *BLAS* (*BLAS*-2). Las nuevas rutinas *BLAS* implementan operaciones de tipo matriz-vector, por lo que tanto el número de operaciones aritméticas como la cantidad de datos involucrados es de orden cuadrático. Al igual que para las primeras rutinas *BLAS*, se publicó una primera implementación genérica de *BLAS*-2 [35]. De dicha implementación, los autores destacan que los accesos a la matriz se realizan por columnas, puesto que es así como se encuentra almacenada en Fortran, y además se reduce el número de accesos a memoria reaprovechando la información dentro del procesador. Ambas características redundan en una mayor eficiencia. Del mismo modo, se invitaba a la generación de implementaciones específicas para cada arquitectura, y se ofrecían una serie de consejos, como buscar la opción que mejor adecúe el bucle interior de la rutina a la arquitectura, utilizar código ensamblador o directivas del compilador específicas para la arquitectura, etc.

Eventualmente, la creciente disparidad entre la velocidad del procesador y el ritmo al que la memoria podía servir los datos al mismo provocó la aparición de arquitecturas con múltiples niveles de memoria caché (memoria jerarquizada). Rápidamente se reconoció que las bibliotecas construidas sobre las rutinas de *BLAS*-1 y *BLAS*-2 nunca podrían obtener un rendimiento óptimo sobre estas nuevas arquitecturas: la memoria representa un auténtico cuello de botella para las rutinas *BLAS*-1 y 2, ya que el ratio entre número de operaciones y de datos es $O(1)$ (orden 1), mientras que el ratio entre las velocidades del procesador y de la memoria es mucho mayor. En definitiva, la memoria necesita más tiempo para proporcionar datos al procesador que éste para procesarlos. En estas condiciones, el rendimiento de las rutinas *BLAS*-1 y *BLAS*-2 está limitado por la velocidad de transferencia de datos desde la memoria, pues no hay reutilización de los datos.

Para resolver el problema, en 1989 se definió la especificación de un tercer conjunto de rutinas, el nivel 3 de *BLAS* (*BLAS*-3), que implementaba operaciones con un número de cálculos de orden cúbico frente a un número de datos de orden cuadrático. Es precisamente esta diferencia entre cantidad de cálculos y datos la que, con un buen diseño del algoritmo, permite explotar eficientemente el principio de localidad de referencia en las arquitecturas con memoria jerarquizada, enmascarando la latencia de acceso a memoria y ofreciendo prestaciones más próximas a la máxima velocidad de cómputo del procesador. Desde el punto de vista algorítmico, esto se consigue con los denominados algoritmos por bloques. Estos algoritmos dividen la matriz en submatrices (o bloques) y, al operar con éstas, agrupan los accesos a memoria incrementando la probabilidad de que los datos se encuentren en los niveles de memoria más cercanos al procesador, y por tanto más rápidos. En definiva, los algoritmos por bloques extraen un mayor beneficio de la arquitectura jerarquizada. El tamaño óptimo de las submatrices en este tipo de algoritmos es dependiente de la arquitectura, especialmente del tamaño de la caché. Dentro de cada submatriz, se continúa recomendando el acceso a los elementos por columnas [34].

Al igual que en los niveles anteriores de *BLAS*, en este tercer nivel existe un compromiso entre complejidad y funcionalidad. Un ejemplo de simplicidad es que no se incluyen rutinas para matrices trapezoidales, ya que esto aumentaría el número de parámetros y estas matrices siempre pueden ser tratadas como una matriz triangular y una matriz rectangular. Por otro lado, para aumentar la funcionalidad, se trata con matrices traspuestas, ya que de otro modo el usuario debería transponer la matriz previamente y ésta puede ser una operación costosa por el número de accesos a memoria.

En resumen, las rutinas de *BLAS* se dividen en tres niveles, que deben su nombre al orden de

operaciones a realizar en cada uno:

- Nivel 1: Operaciones sobre vectores (número de operaciones de orden lineal).
- Nivel 2: Operaciones matriz-vector (número de operaciones de orden cuadrático).
- Nivel 3: Operaciones matriz-matriz (número de operaciones de orden cúbico).

A nivel de prestaciones, el motivo de la diferenciación entre niveles parte de la relación entre el número de operaciones y el número de datos implicados. Este ratio es crucial en las máquinas de memoria jerarquizada, ya que si el número de operaciones es mayor que el de datos, es posible realizar varias operaciones por cada acceso a memoria y, de esta forma, aumentar la productividad. Por lo tanto, se definen los niveles en función del ratio entre el número de operaciones y el de datos implicados:

- Nivel 1: El número de operaciones y de datos crece linealmente con el tamaño del problema.
- Nivel 2: El número de operaciones y de datos crece cuadráticamente con el tamaño del problema.
- Nivel 3: El número de operaciones crece cúbicamente con el tamaño del problema mientras que el de datos lo hace cuadráticamente.

Mención aparte merecen las versiones de *BLAS* paralelo para multiprocesadores con memoria compartida. Con el fin de paralelizar los cómputos y emplear eficientemente los recursos disponibles, implementan un código multihebra que distribuye las operaciones entre los distintos procesadores. En estas implementaciones es especialmente favorable el uso de rutinas del nivel 3, ya que la eficiencia en los otros dos niveles está, si cabe, más sesgada por la velocidad de la memoria y no por la velocidad de cómputo de los (múltiples) procesadores.

Como los mismos autores subrayan respecto al paralelismo [32], las rutinas *BLAS*-3 basadas en el cómputo por bloques presentan dos ventajas:

- Diferentes procesadores pueden operar con diferentes bloques simultáneamente.
- Dentro de un bloque, las operaciones sobre diferentes escalares o vectores pueden ejecutarse simultáneamente.

Desde el punto de vista de las prestaciones, podemos concluir que:

- Las prestaciones en los niveles de *BLAS*-1 y *BLAS*-2 están limitadas por la velocidad a la que la memoria puede servir datos.
- El nivel 3 de *BLAS* es el más eficiente, puesto que por cada acceso a memoria puede realizar un mayor número de operaciones, consiguiendo prestaciones cercanas a la velocidad de cómputo del procesador. Además, las rutinas *BLAS*-3 ofrecen unas mejores cualidades para su paralelización, pudiendo obtener por lo tanto mejoras más significativas en arquitecturas multiprocesador con memoria compartida.

1.2.2. Nomenclatura

Los nombres de las rutinas de *BLAS* están formados por entre cuatro y seis caracteres. Se fijó el número máximo de caracteres en seis para cumplir con la especificación Fortran 77 (las versiones más recientes de Fortran no tienen esta limitación), siendo el objetivo que el nombre incluya la mayor información posible sobre la rutina. En general, para las rutinas de *BLAS*-2 y *BLAS*-3 estos nombres tienen la forma TXXYYY, donde:

- T: indica el tipo de datos con los que trabaja la rutina. Puede tomar los siguientes valores:
  - S: número real de precisión simple.
  - D: número real de precisión doble.
  - C: número complejo de precisión simple.
  - Z: número complejo de precisión doble.
- XX: indica el tipo de matrices con los que opera la rutina (*BLAS*-2 y *BLAS*-3); entre otros, podemos encontrar:
  - GE: matriz general.
  - SY: matriz simétrica.
  - TR: matriz triangular (inferior o superior).
  - GB: matriz general banda.
  - SB: matriz simétrica banda.
- YYY: indica la operación que realiza. Puede tener una longitud de dos o tres caracteres; por ejemplo, MM para el producto de matrices, MV para el producto matriz-vector y RK para la actualización de rango k.

Las rutinas de *BLAS*-1 no siguen esta notación, puesto que son anteriores a su definición. La definición de la notación es una necesidad que apareció para las rutinas *BLAS*-2, al operar con matrices. Es decir, en *BLAS*-1 no se precisa especificar el tipo de operandos que recibe, sino simplemente la operación. En general estas rutinas tienen un nombre formado por entre cinco y seis caracteres, donde el primero de ellos especifica el tipo de datos empleado (precisión simple, precisión doble, complejos en precisión simple y complejos en precisión doble) y el resto de caracteres identifica la operación.

1.2.3. Esquemas de almacenamiento

BLAS trabaja con cuatro tipos de almacenamiento de matrices con los siguientes objetivos:

- Facilidad de programación: almacenamientos muy elaborados podrían propiciar códigos muy complejos e ininteligibles.
- Economía de almacenamiento: reducir el tamaño de la memoria necesaria.
- Compactación: reducir el tamaño de la memoria necesaria al tiempo que se mejora la eficiencia del acceso desde las rutinas.

Los almacenamientos y los tipos de matrices asociadas son los siguientes:

- Almacenamiento convencional: la matriz se almacena por columnas como es habitual, por ejemplo, en Fortran. Las columnas consecutivas se encuentran en posiciones consecutivas de memoria. Se usa para matrices generales (densas).
- Almacenamiento compacto: almacenamiento por columnas como el anterior pero donde únicamente se mantiene la parte triangular inferior o superior de la matriz (figura 1.2). Se utiliza para matrices simétricas, hermitianas o triangulares. Dado que los accesos a memoria son costosos, es mejor no tener la matriz completamente almacenada. Por ejemplo, escalar una matriz simétrica que se encuentra almacenada en su totalidad precisa de la actualización (lecturas y escrituras en memoria) de prácticamente el doble de elementos que si se almacena simplemente la parte triangular inferior o superior de la misma.

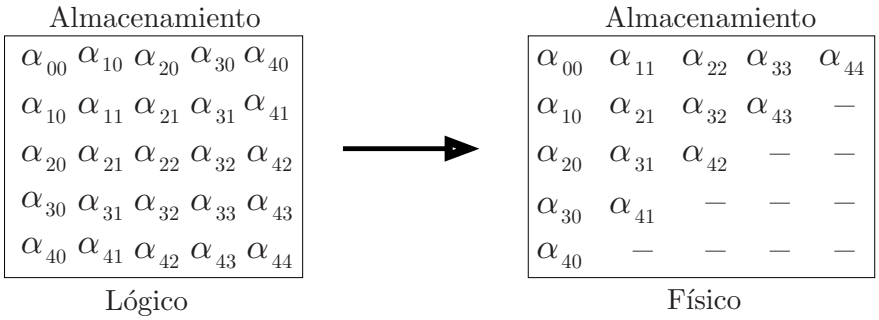


Figura 1.2: Esquema de almacenamiento para matrices simétricas, hermitianas o triangulares de dimensión 5×5.

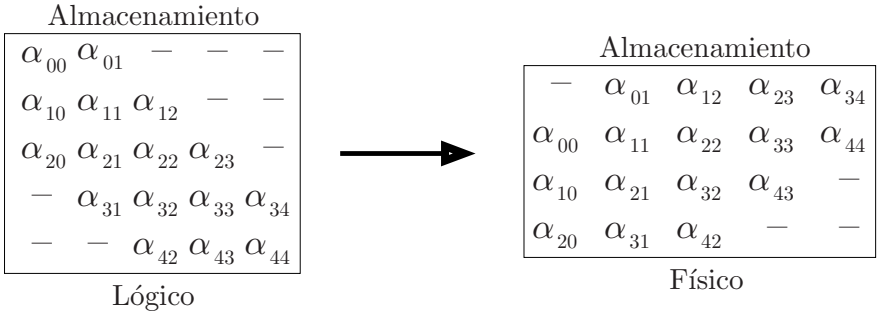


Figura 1.3: Esquema de almacenamiento para matrices banda de dimensión 5×5 y con ancho de banda inferior igual a 2 y superior igual a 1.

- Almacenamiento banda: utilizado para matrices banda, sólo mantiene los elementos que están dentro de la banda (figura 1.3).
- Almacenamiento simétrico banda: utilizado para matrices simétricas o hermitianas banda, almacena únicamente aquellos elementos que están dentro de la banda de la parte triangular superior o inferior de la matriz (figura 1.4).

1.2.4. Argumentos

BLAS especifica diversos parámetros que permiten definir vectores y matrices. Esta definición está ligada a la funcionalidad de la biblioteca y a los formatos de almacenamiento de las diferentes

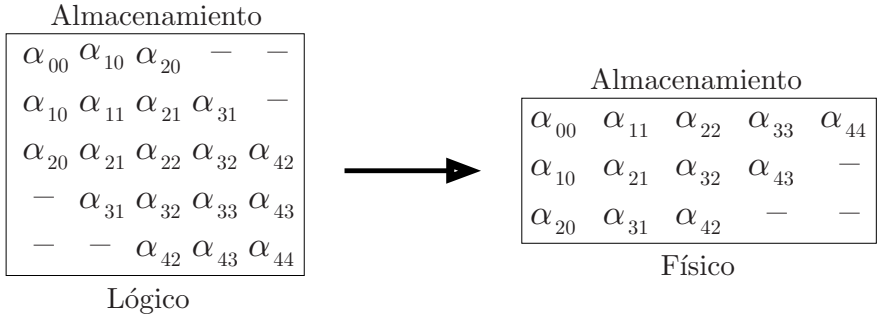


Figura 1.4: Esquema de almacenamiento para matrices simétricas/hermitianas banda de dimensión 5×5 y con ancho de banda inferior y superior igual a 2.

estructuras. Por ejemplo, el parámetro `TRANS` en la especificación de una rutina requiere que la rutina debe ser capaz de ejecutar la operación con la matriz y con su transpuesta. Otros parámetros están ligados al formato de almacenamiento. En este sentido, el parámetro `INCX`, que especifica la distancia física en números de elementos entre las posiciones de memoria que almacenan dos elementos consecutivos del vector de datos, permite que elementos lógicamente consecutivos de un vector no sean vecinos físicos en memoria.

Por lo tanto, la definición de los parámetros marca en muchos casos la funcionalidad y los formatos de almacenamiento soportados por la biblioteca. Durante la definición de los parámetros se buscó un acuerdo entre la funcionalidad de la biblioteca y el número de parámatros. Incrementar el número de parámetros normalmente significa aumentar la funcionalidad de las diferentes rutinas, pero también dificulta su uso y su legibilidad. A continuación se enumeran los parámetros más habituales en la definición de vectores y matrices como argumentos de las rutinas de *BLAS*.

Parámetros que definen un vector

- `X`: posición en memoria del primer elemento del vector.
- `INCX`: distancia, en número de elementos, entre las posiciones de memoria de dos elementos lógicamente consecutivos.
- `N`: número de elementos.

Parámetros que definen una matriz

- `A`: posición de memoria en la que está almacenado el primer elemento de la matriz.
- `LDA`: distancia, en número de elementos, entre las posiciones de memoria que almacenan dos elementos consecutivos de una misma fila.
- `M`: número de filas.
- `N`: número de columnas.
- `TRANS`: define si se debe operar con la matriz o con su transpuesta.
- `UPLO`: para matrices simétricas y triangulares, define si se debe procesar la parte superior o inferior de la matriz.

- `DIAG`: para matrices triangulares, define si se debe asumir que los elementos de la diagonal son unos.
- `KL`: para matrices banda, define el ancho de banda inferior.
- `KU`: para matrices banda, define el ancho de banda superior.
- `KD`: para matrices simétricas banda, define el ancho de banda tanto inferior como superior.

Lógicamente, no todos los parámetros son necesarios para todas las matrices; por ejemplo, para una matriz simétrica sólo es necesario definir una de las dimensiones de la matriz.

1.2.5. BLAS para matrices banda

Desde su inicio, *BLAS* ha dado cierto soporte a las matrices banda por dos motivos principales. En primer lugar, estas matrices aparecen regularmente en algunos problemas de ingeniería y, además, se pueden obtener grandes ventajas tanto espaciales como temporales si se explota su estructura. En consecuencia, existen formatos de almacenamiento y rutinas específicas para este tipo de matrices.

En concreto, la especificación *BLAS* incluye cinco rutinas para el trabajo con matrices banda:

- GBMV: producto de una matriz banda y un vector.
- SBMV: producto de una matriz simétrica banda y un vector.
- TBMV: producto de una matriz triangular banda y un vector.
- HBMV: producto de una matriz hermitiana banda y un vector.
- TBSV: resolución de un sistema de ecuaciones lineales con una matriz de coeficientes con estructura triangular banda.

No obstante, las implementaciones habituales de estas rutinas no están tan optimizadas como las de las rutinas para matrices densas, seguramente debido a que estas últimas tienen un número de usuarios mucho más elevado. Además, el número de operaciones con matrices banda soportado por *BLAS* es reducido. Es por ello que durante la presente tesis se ha buscado definir e incorporar mejoras aplicables al soporte que *BLAS* ofrece para matrices banda.

1.3. LAPACK

LAPACK [68] (*Linear Algebra PACKage*) es una biblioteca que ofrece rutinas para resolver problemas fundamentales de álgebra lineal, y que contiene el estado actual en métodos numéricos. Al igual que *BLAS*, *LAPACK* ofrece soporte tanto a operaciones con matrices densas como con matrices banda, pero mientras que *BLAS* resuelve las operaciones más básicas, *LAPACK* resuelve problemas más complejos, como, por ejemplo, sistemas de ecuaciones lineales, problemas de mínimos cuadrados, y problemas de valores propios y singulares.

LAPACK surgió como resultado de un proyecto iniciado a finales de la década de los 80. El objetivo era obtener una biblioteca que comprendiera las funcionalidades y mejorara el rendimiento de las bibliotecas EISPACK [69] y LINPACK [30]. Dichas bibliotecas, diseñadas en su día para procesadores vectoriales, no ofrecen un rendimiento aceptable sobre los procesadores de altas prestaciones actuales, con cauces segmentados y con memorias jerarquizadas. El principal motivo de

su ineficiencia es que, al estar basadas en operaciones de *BLAS*-1, no hacen un uso óptimo de la jerarquía de memoria. Consecuentemente, sus rutinas pasan más tiempo moviendo datos a/desde memoria que realizando las operaciones pertinentes. La acentuada diferencia entre la velocidad de la memoria y del procesador hace que optimizar el número de accesos a memoria e incluso el patrón de acceso sean cruciales para obtener códigos eficientes [20].

El incremento de prestaciones obtenido por *LAPACK* se basa en:

- Incorporar los nuevos algoritmos surgidos desde las implementaciones de LINPACK y EISPACK.
- Reestructurar los algoritmos para hacer un uso eficiente de *BLAS*.

Respecto a la incorporación de nuevos algoritmos, prácticamente se introdujeron mejoras en la resolución de todos los problemas de álgebra lineal soportados por la biblioteca, siendo especialmente relevantes las mejoras aplicadas a la resolución de problemas de valores propios. Sin embargo, la aportación más importante de *LAPACK* fue la reestructuración de los algoritmos para hacer un uso eficiente de las rutinas *BLAS* y obtener mejores prestaciones.

La implementación genérica de *LAPACK* es de libre acceso [68] e incluye programas de comprobación y temporización. Algunos fabricantes *hardware* han implementado versiones específicas de *LAPACK* para sus arquitecturas, aunque habitualmente se trata de modificaciones menores, como por ejemplo realizar un ajuste del tamaño de bloque utilizado.

Para máquinas multiprocesador, *LAPACK* extrae el paralelismo invocando a una versión paralela del *BLAS*. Es decir, las rutinas de *LAPACK* no incluyen ningún tipo de paralelismo explícito en su código, sino que hacen uso de una implementación paralela (multihebra) de *BLAS*.

1.3.1. LAPACK y BLAS

La biblioteca *BLAS* ofrece una serie de rutinas muy eficientes para operaciones básicas de álgebra lineal. Es por ello que sus rutinas son empleadas por las bibliotecas LINPACK y *LAPACK*. La primera de ellas utiliza internamente rutinas del nivel 1 de *BLAS*, pero las rutinas de este nivel realizan operaciones con vectores y escalares, por lo que no son propicias para la paralelización o el uso eficiente de la jerarquía de memoria. *LAPACK*, por su parte, hace uso de rutinas de todos los niveles de *BLAS*, pero en especial utiliza las rutinas del nivel 3 ya que, como se ha comentado, éstas presentan dos grandes ventajas:

- Gran eficiencia en las máquinas de memoria jerarquizada, ya que sus prestaciones no están limitadas por la velocidad de la memoria.
- Implementan operaciones con más carga de trabajo, con lo que son más convenientes para la paralelización.

Para poder hacer uso de las rutinas de *BLAS*-3, los algoritmos implementados por *LAPACK* fueron reestructurados para trabajar con bloques o submatrices [27, 36]. El uso de algoritmos basados en bloques permite que la mayor parte de las operaciones sean realizadas por las rutinas más eficientes de *BLAS*, y además incrementa las opciones de paralelizar en dos sentidos: paralelizar cada operación con bloques y realizar varias operaciones con distintos bloques en paralelo [29].

No obstante, en muchas ocasiones el algoritmo por bloques requiere para su funcionamiento de una versión sin bloques; es por esto que para diversas operaciones existen dos rutinas diferentes que la implementan, una de ella por bloques basada en rutinas de *BLAS*-3 y otra rutina escalar (sin bloques) basada en rutinas de *BLAS*-2. Un ejemplo es la factorización de Cholesky de una matriz

densa: la rutina por bloques (POTRF) es un algoritmo que opera con tres bloques distintos en cada iteración y sobre uno de ellos la operación a realizar es una factorización de Cholesky. Para este bloque se utiliza la rutina que implementa la factorización de Cholesky sin bloques (POTF2).

La utilización de *BLAS* ofrece otra serie de ventajas:

- Portabilidad y eficiencia: Existen diferentes versiones de *BLAS* específicas para diferentes arquitecturas, por lo que hacer uso de rutinas *BLAS* permite que *LAPACK*, sin modificar su código, se adapte y obtenga altas prestaciones con distintas arquitecturas. Esta eficiencia es de suponer que se trasladará a los procesadores que puedan aparecer en el futuro, ya que los fabricantes de arquitecturas de altas prestaciones implementan versiones de *BLAS* para sus nuevas plataformas.
- Legibilidad: Las rutinas de *BLAS* son ampliamente conocidas y sus nombres son usados como mnemotécnicos de las operaciones que realizan.
- Paralelismo: Existen versiones paralelas de *BLAS*, con lo que al hacer uso de ellas se obtiene directamente una rutina *LAPACK* paralela.

Por lo tanto, la eficiencia de *LAPACK* depende en gran parte del *BLAS* subyacente, pero también de la cantidad de operaciones aritméticas efectuadas por rutinas del nivel 3 de *BLAS*. Las rutinas de *BLAS*-3 son las que tienen una mayor eficiencia, con lo que los algoritmos de *LAPACK* tratan de incrementar la cantidad de operaciones aritméticas realizadas por éstas.

1.3.2. Nomenclatura

El nombre de cada rutina *LAPACK* es un código que especifica su funcionalidad. Al igual que *BLAS*, sigue la especificación de Fortran 77 y tiene una longitud máxima de seis caracteres.

El nombre de una rutina *LAPACK* trata de incluir en él toda la información relevante para el usuario de la misma. El formato utilizado, idéntico al usado por *BLAS*, es XYYZZZ, donde:

- X: indica el tipo de datos con los que trabaja, que puede ser: real de precisión simple (S), real de precisión doble (D), complejo de precisión simple (C) y complejo de precisión doble (Z).
- YY: indica el tipo de matriz (o de la matriz más significativa), por ejemplo GB para matrices banda. En algunos tipos de matriz se permite el uso de formato de almacenamiento general o compacto. Para diferenciar ambas rutinas, cuando el segundo carácter de este código es una P significa que la matriz pasada a la rutina está almacenada en formato compacto.
- XXX: indica la operación que realiza, por ejemplo QRF para la factorización QR.

Para más detalles sobre los nombres de las rutinas *LAPACK*, consultar [17].

1.3.3. Esquemas de almacenamiento

La biblioteca *LAPACK* basa su funcionamiento en el uso de rutinas *BLAS*, por lo que los esquemas de almacenamiento soportados son los mismos que utiliza *BLAS* (ver el apartado 1.2.3).

1.3.4. Argumentos

A pesar de que *LAPACK* soporta hasta 24 tipos diferentes de matrices (bidiagonal, general densa, banda, hermitiana banda, simétrica,...), utiliza prácticamente los mismos parámetros para definir matrices que *BLAS* (ver el apartado 1.2.4).

1.3.5. LAPACK para la factorización de matrices banda

LAPACK incluye una serie de rutinas específicas para matrices banda, como por ejemplo GBSV, que calcula la solución de un sistema de ecuaciones lineales banda, o GBTRF, que obtiene la factorización LU con pivotamiento de una matriz banda.

Este trabajo se centra, a nivel de *LAPACK*, en las rutinas para la resolución de sistemas de ecuaciones lineales, especialmente en la factorización de matrices. Las rutinas para obtener la factorización de una matriz banda son:

- Matriz general banda:
  - GBTF2: implementa un algoritmo que calcula la factorización LU con pivotamiento utilizando rutinas *BLAS*-1 y *BLAS*-2.
  - GBTRF: obtiene la factorización LU con pivotamiento mediante un algoritmo por bloques y llamadas a rutinas *BLAS*-3.
- Matriz simétrica definida positiva banda:
  - PBTF2: calcula la factorización de Cholesky utilizando rutinas *BLAS*-1 y *BLAS*-2.
  - PBTRF: implementa un algoritmo por bloques para obtener la factorización de Cholesky mediante llamadas a rutinas *BLAS*-3.

Por lo tanto la funcionalidad de *LAPACK* para factorizar matrices banda es reducida y no permite, por ejemplo:

- Obtener la factorización QR de una matriz banda. Esta factorización presenta mejores propiedades numéricas que la factorización LU con pivotamiento a cambio de un mayor coste computacional. Además, éste es el tipo de factorización necesario en la resolución de problemas lineales de mínimos cuadrados con estructura banda.
- Obtener la factorización LU sin pivotamiento. Ciertas matrices no precisan de pivotamiento para obtener su factorización LU con gran precisión, pero dado que *LAPACK* no implementa la factorización LU sin pivotamiento, toda factorización LU debe afrontar el siguiente sobrecoste:
  - Sobrecoste espacial debido al relleno. Este coste es proporcional al producto del ancho de banda inferior de la matriz por su dimensión.
  - Sobrecoste temporal elevado debido al cálculo del pivote y a la aplicación de las permutaciones que, además, han de ser aplicadas de nuevo durante el proceso de resolución con la matriz triangular banda inferior.

1.3.6. Contenido de LAPACK

Desde que se publicó la primera versión de *LAPACK* en 1992, esta biblioteca ha experimentado diversas actualizaciones que han aumentado su funcionalidad. La última versión de *LAPACK*, la 3.1.1, se hizo pública en febrero de 2007.

Además de las rutinas específicas para ejecutar las distintas operaciones del álgebra lineal soportadas, se incluyen una serie de programas para comprobar y medir la eficiencia de *LAPACK* [5].

Rutinas para la comprobación de LAPACK

Se incluyen dos programas para la verificación del funcionamiento de *LAPACK* para cada tipo de datos (reales con precisión simple y doble, y complejos con precisión simple y doble); uno de ellos comprueba las rutinas principales (*drivers*) de resolución de sistemas de ecuaciones lineales y problemas de mínimos cuadrados, mientras que el otro verifica los *drivers* para problemas de valores propios. También se incluye el *software* necesario para generar las matrices que se emplean en el test.

Rutinas de temporización de LAPACK

Al igual que las rutinas de comprobación, se incluyen dos programas para la temporización por cada tipo de datos. Los programas generan datos y ejecutan diversas rutinas obteniendo tiempos para ejecuciones con diversos datos y valores de parámetros. Los resultados que los usuarios obtienen y envían al equipo de desarrollo de *LAPACK* son utilizados para futuras revisiones.

1.3.7. Las rutinas y su organización

LAPACK presenta tres tipos de rutinas:

- *Driver*: resuelven problemas elaborados como por ejemplo sistemas de ecuaciones lineales, problemas de mínimos cuadrados y problemas de valores propios.
- Computacionales: realizan operaciones más sencillas que los *drivers* y son utilizadas por éstos para obtener resultados parciales. Algunos ejemplos son la factorización de matrices o la resolución de un sistema de ecuaciones lineales a partir del resultado de factorización de una matriz.
- Utilidades: rutinas para tareas de apoyo a las rutinas computacionales y a los *drivers*. Dentro de este grupo se encuentran:
  - Rutinas que implementan operaciones de bajo nivel, como el escalado de una matriz.
  - Rutinas que implementan subtareas de los algoritmos por bloques, como versiones sin bloques de la operación.
  - Otras rutinas de apoyo como, por ejemplo, una rutina para conocer el tamaño de bloque óptimo de la máquina sobre la que se está ejecutando o comparar dos cadenas.

Todas las rutinas *LAPACK* son accesibles por el usuario, independientemente de su tipo (*driver*, rutina computacional o de utilidad).

1.4. Notación

1.4.1. Vectores y matrices

Un *vector* (columna) x de dimensión n es una *n-tupla* de números. Los vectores son habitualmente identificados por una letra del alfabeto latino, mientras que sus componentes, escalares todos ellos, son identificados por la correspondiente letra griega:

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix}.$$

Siguiendo la convención del lenguaje C, los elementos de vectores y matrices se numeran empezando con el índice 0.

Un vector fila x^T se define como la transpuesta de un vector columna:

$$x^T = \left(\begin{array}{cccc} \chi_0, & \chi_1, & \ldots, & \chi_{n-1} \end{array} \right)$$

Una *matriz* $A \in \mathbb{R}^{m \times n}$ está formada por n vectores columna o m vectores fila. Las matrices se nominan con una letra latina en mayúsculas, mientras que el elemento (i, j) de la matriz A se denomina con la correspondiente letra griega y subíndices, α_{ij}.

Si $m = n$, entonces A es una *matriz cuadrada*; de otra forma, A es una *matriz rectangular*. Si A es una matriz cuadrada en la que todo elemento (i, j) es igual al elemento (j, i), entonces es *simétrica*.

Una matriz A es *triangular superior* si es cuadrada y para todos sus elementos se cumple que $\alpha_{ij} = 0$ si $i > j$. Así mismo, A es *triangular inferior* si es cuadrada y para todos sus elementos se cumple que $\alpha_{ij} = 0$ si $j > i$. De igual forma, A es una matriz *triangular superior (inferior) estricta* si $\alpha_{ij} = 0$ para $i \geq j$ $(j \geq i)$. Finalmente, una matriz es *triangular superior (inferior) unitaria* si $\alpha_{ij} = 0$ si $i \geq j$ $(j \geq i)$ y $\alpha_{ij} = 1$ si $i = j$.

Una matriz simétrica $A \in \mathbb{R}^{n \times n}$ es *positiva definida* si para todo vector real $x \not\equiv 0$ de dimensión n, se cumple que $x^T Ax > 0$.

Una matriz A es *banda*, o se dice que tiene una estructura banda, con ancho de banda inferior k_l y ancho de banda superior k_u, cuando todo elemento (i, j) de la matriz es cero si $i > j + k_l$ o $j > i + k_u$.

1.4.2. Notación algorítmica FLAME

Los algoritmos incluidos en el presente documento se expresan siguiendo la notación desarrollada en el proyecto FLAME (*Formal Linear Algebra Methods Environment*) [90]. El principal resultado de este proyecto es la definición de una metodología para la derivación formal de algoritmos para operaciones de álgebra lineal. Junto con la metodología, otros frutos del proyecto son un conjunto de interfaces de programación que permiten transformar rápidamente algoritmos en códigos, y una notación para especificar algoritmos para operaciones de álgebra lineal [94, 16]. Esta notación, por su concreción, regularidad y simplicidad, es la que emplearemos en la memoria.

La figura 1.5 muestra un algoritmo en notación FLAME para el cálculo de la descomposición de una matriz A en el producto de dos factores triangulares, $A = LU$, con L triangular inferior unidad (elementos diagonales iguales a 1) y U triangular superior. En este algoritmo, los elementos de las matrices resultado L y U sobreescriben a los elementos de A, a excepción de los elementos de la diagonal principal de la matriz L, que no son almacenados ya que todos ellos toman el valor 1.

En el algoritmo inicialmente se particiona A en cuatro bloques A_{TL}, A_{TR}, A_{BL} y A_{BR}, con los tres primeros vacíos. De este modo, al inicio de la primera iteración del bucle el bloque A_{BR} está formado por todos los elementos de A.

Durante cada iteración, el bloque A_{BR} se divide, tal y como se muestra en la figura 1.6, en cuatro bloques: α_{11}, a_{21}, a_{12}^T y A_{22} (quedando A particionada en 9 bloques). Los elementos de los factores L y U correspondientes a los tres primeros de estos bloques son calculados durante la presente iteración, mientras que A_{22} es actualizado. Al finalizar la iteración actual, se recupera el particionado 2×2 sobre la matriz, quedando el bloque A_{BR} formado únicamente por los elementos de A_{22}. De esta forma, en la siguiente iteración las operaciones sobre la matriz se aplican de nuevo sobre el bloque A_{BR} recién definido.

El algoritmo termina cuando el bloque A_{BR} queda vacío, o lo que es lo mismo, cuando el número de filas del bloque A_{TL} coincide con el número de filas de la matriz A. En este sentido, el bucle del

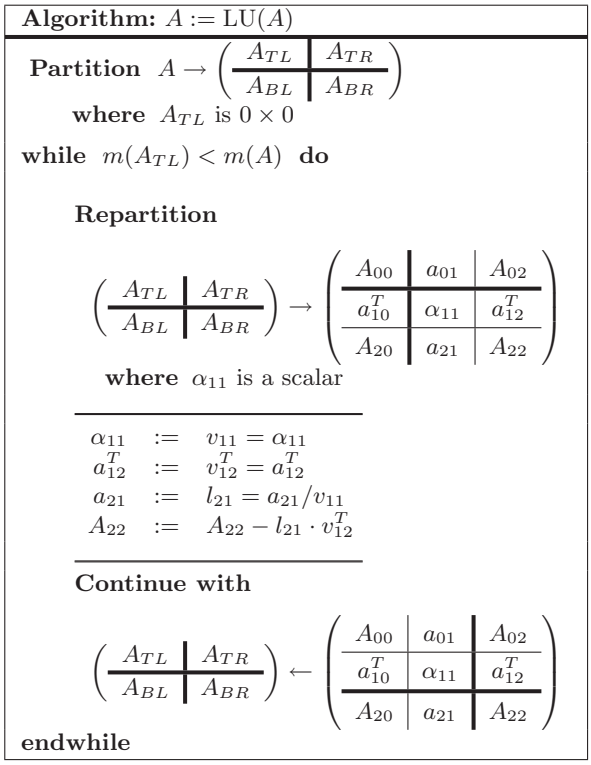


Figura 1.5: Algoritmo para la factorización LU en notación FLAME

algoritmo se repite mientras $m(A_{TL}) < m(A)$, donde $m(\cdot)$ denota el número de filas de una matriz (o el número de elementos de un vector columna). Análogamente, $n(\cdot)$ se utiliza para caracterizar el número de columnas de una matriz (o el número de elementos de un vector fila).

En ocasiones se formulará un *algoritmo por bloques* para el cálculo de una operación. El funcionamiento de este tipo de algoritmos, claves para la obtención de prestaciones elevadas en las arquitecturas actuales, es fácilmente ilustrado mediante la misma notación, en la que sólo es necesario incluir un parámetro más: el tamaño de bloque algorítmico. Éste es un valor escalar que determina el tamaño de los bloques con los que se opera en el algoritmo y, siguiendo la práctica habitual, usaremos la letra b para denotarlo.

La notación FLAME es útil en tanto que permite expresar los algoritmos con un alto nivel de abstracción. Estos algoritmos deben aún plasmarse en una *rutina* o *implementación*, con un mayor nivel de concreción. Así, por ejemplo, en el algoritmo de la figura 1.5 no se indica el modo en que se realizan las operaciones que forman el cuerpo del bucle. Una rutina que, por ejemplo, ejecutase la operación $a_{21} := a_{21}/v_{11}$ mediante una llamada a la rutina SCAL de *BLAS*-1 y otra rutina que calculase este escalado mediante un simple bucle, pese a ser diferentes, responderían al mismo algoritmo. Es importante pues distinguir entre *algoritmo* y *rutina/implementación*.

1.5. Arquitecturas de altas prestaciones

1.5.1. Procesadores vectoriales

Estas arquitecturas están diseñadas para operar sobre vectores, disponiendo de una microarquitectura (por ejemplo, registros y unidades funcionales) orientada al procesamiento de este tipo

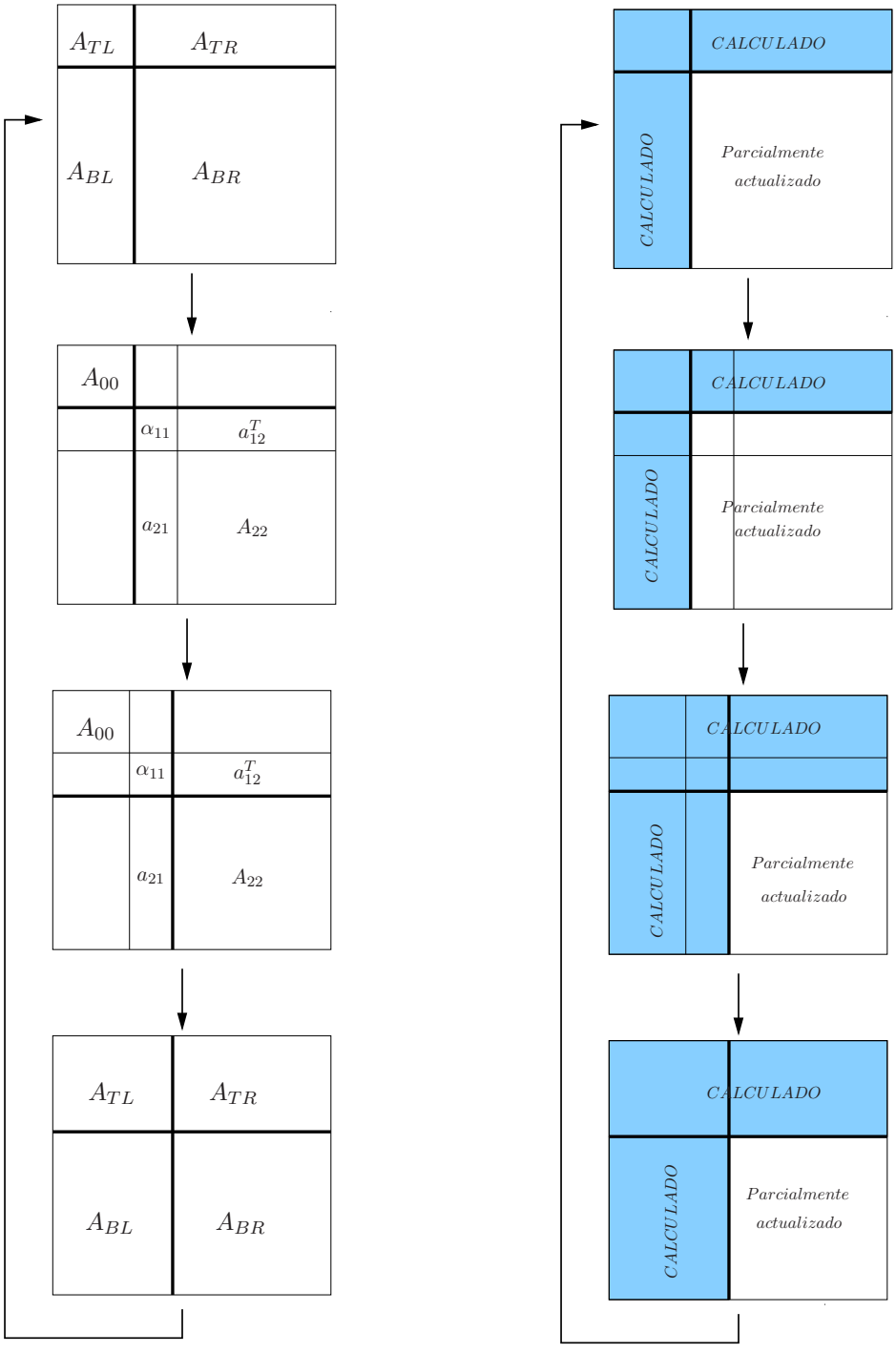


Figura 1.6: Particionado aplicado a la matriz A durante el algoritmo de la figura 1.5.

de datos y de un repertorio con instrucciones máquina que operan con datos y producen resultados que son vectores.

Con la aparición en 1972 del primer procesador vectorial, el Cray-1, estas arquitecturas tomaron gran relevancia y durante más de una década los supercomputadores más potentes estaban basados en este tipo de procesador. Su supremacía en el campo de la supercomputación se extendió hasta inicios de los 90, cuando los sistemas de procesamiento paralelo con memoria distribuida superaron las prestaciones ofrecidas por los sistemas vectoriales. Sin embargo, en el ámbito general, los procesadores vectoriales habían sido desplazados ya antes por los microprocesadores debido al mejor ratio de coste/prestaciones y la mayor versatilidad de estos últimos. Recientemente los procesadores paralelos han vuelto de alguna forma al mercado con las extensiones SSE, SSE2, SSE3 y AVX de los procesadores de Intel.

1.5.2. Microprocesadores

La segmentación como clave para incrementar el rendimiento

La segmentación (*pipelining*) es una técnica que divide la ejecución de una instrucción en varias etapas, solapando la ejecución de etapas diferentes de dos o más instrucciones. El beneficio que se obtiene al aplicar esta técnica es que una instrucción puede comenzar su ejecución antes de que la instrucción anterior haya sido ejecutada completamente. A cambio, precisa una mayor cantidad de recursos *hardware* en forma de circuitería. Teóricamente, el número de instrucciones que se pueden ejecutar simultáneamente con la segmentación es igual al número de etapas en las que se subdivide la ejecución de cada instrucción, si bien la introducción de cada etapa provoca un pequeño retardo de sincronización y complica el funcionamiento real del cauce segmentado.

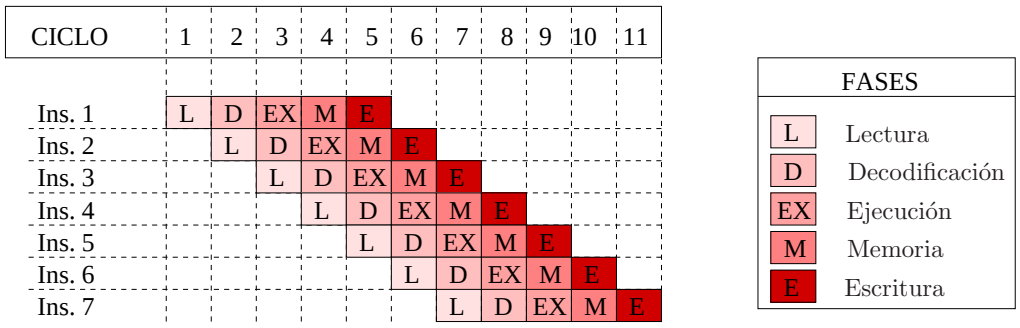


Figura 1.7: Ejemplo de procesamiento segmentado de un conjunto de instrucciones.

En esta técnica, el resultado de una etapa pasa a ser la entrada de la siguiente etapa. Para ello se dispone de una serie de registros entre cada par de etapas consecutivas, de forma que la etapa anterior almacena su salida en un registro del que la etapa siguiente leerá su entrada. Cada una de estas etapas es ejecutada por una circuitería (*hardware*) dedicada. La figura 1.7 muestra la ejecución de diversas instrucciones en un procesador segmentado [51]. La ejecución de cada instrucción se divide en 5 etapas, aunque no todas las instrucciones precisan de todas las etapas. Las tareas a ejecutar en cada etapa varían en función del tipo de instrucción pero, en general, pueden especificarse como:

- Lectura: capta la instrucción de la memoria de instrucciones y calcula el nuevo contador de programa.

- Decodificación: interpreta el tipo de operación a ejecutar y extrae los valores de los operandos del banco de registros.
- Ejecución: si la instrucción es una operación aritmética, se realiza el cálculo en la unidad aritmético-lógica (ALU). Si se trata de una instrucción de acceso a memoria (lectura o escritura), se calcula la dirección de memoria donde leer/escribir.
- Memoria: lee o escribe en memoria si la instrucción es un acceso a ésta.
- Escritura: almacena el resultado de la operación en el banco de registros.

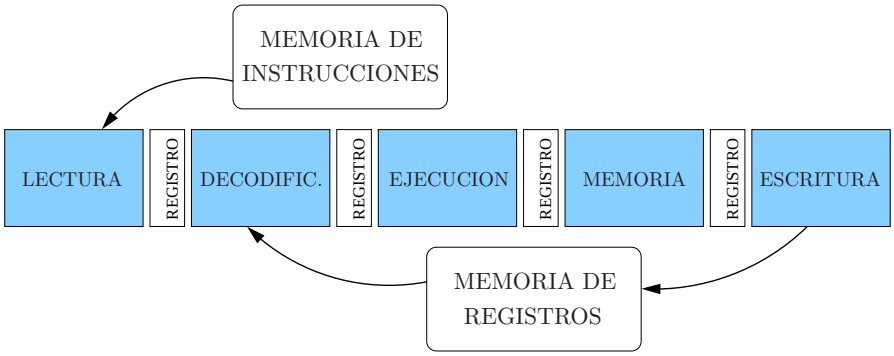


Figura 1.8: Ejemplo de funcionamiento *pipeline*.

La figura 1.8 muestra de forma muy simplificada la organización del *hardware* para una posible estructura *pipeline*.

Procesadores superescalares

Un procesador superescalar es un cauce segmentado capaz de ejecutar varias instrucciones simultáneamente en cada etapa del cauce (lectura, decodificación, ejecución, etc.). El número máximo de instrucciones que se puede gestionar en una etapa concreta se denomina *grado*. Para poder ejecutar varias instrucciones en una misma etapa, se requiere la replicación de la circuitería o bien el empleo de unidades más veloces. Habitualmente, los procesadores superescalares disponen de más de una ALU y una o más unidades funcionales para operaciones en coma flotante.

El potencial de los procesadores superescalares es muy elevado y, en general, su mayor problema consiste en alcanzar una utilización máxima. En este sentido, las intrucciones de salto y las dependencias de datos entre instrucciones resultan especialmente gravosas, pues complican considerablemente su funcionamiento. En las arquitecturas superescalares, los problemas derivados de los saltos en el flujo de ejecución y de las dependencias de datos son resueltos por el *hardware*.

Para reducir los problemas planteados por las operaciones de salto se emplean técnicas de ejecución especulativa y políticas de predicción de saltos. Las dependencias entre instrucciones presentan un problema incluso más importante. Una opción es ampliar el tamaño de la *ventana de instrucciones*, que contiene las operaciones que son susceptibles de ser ejecutadas en paralelo en un instante dado. Cuanto más grande sea esta ventana, mayor es la probabilidad de encontrar instrucciones independientes que puedan ser ejecutadas simultáneamente. Sin embargo, esta solución presenta una dificultad añadida en las arquitecturas superescalares, ya que incrementa la complejidad del *hardware* con lo que, además de dificultar el diseño, aumenta el tiempo requerido para la ejecución de cada instrucción.

En las arquitecturas superescalares la obtención de paralelismo es tarea del *hardware*, con lo que el *software* tiene un papel secundario en la obtención de prestaciones elevadas. La escasa dependencia de los compiladores y del *software* en general es una de las ventajas de estas arquitecturas. A cambio, la planificación dinámica de las instrucciones en tiempo de ejecución, que requiere la detección y el tratamiento correcto de las dependencias de datos y la gestión especulativa de los saltos, consume una gran cantidad de recursos *hardware* en este tipo de procesadores.

Los procesadores Intel Pentium, Xeon, AMD Athlon y AMD Opteron son ejemplos de procesadores superescalares actuales.

Procesadores VLIW

Al igual que los procesadores superescalares, los procesadores VLIW (*Very Long Instruction Word*) son capaces de ejecutar varias instrucciones simultáneamente en cada etapa del cauce. Sin embargo, en estos últimos la responsabilidad de la planificación de las instrucciones queda en manos del *software* y, en concreto, del compilador, que realiza esta tarea de manera previa a la ejecución (planificación estática).

Los procesadores VLIW presentan un juego de instrucciones reducido pero en el que cada instrucción, de gran longitud (de ahí el nombre del procesador), codifica varias operaciones escalares. En particular, las operaciones codificadas en una misma instrucción por el compilador no presentan dependencias entre sí y, por tanto, su ejecución puede proceder en paralelo. Al no tener que detectar dependencias de datos entre las instrucciones en tiempo de ejecución, el *hardware* de este tipo de procesadores se simplifica. Los recursos (transistores) que se liberan de este modo pueden utilizarse, por ejemplo, para incrementar el tamaño de las memorias caché o añadir un mayor número de unidades funcionales. La gestión de saltos en este tipo de procesadores se trata, igual que en los procesadores superescalares, mediante predictores de saltos y ejecución especulativa.

En resumen, los procesadores VLIW tienen como principal ventaja la simplicidad del *hardware* y para ello delegan en manos del compilador la planificación de instrucciones, transfiriendo a éste la responsabilidad de extraer el paralelismo a nivel de instrucción de los códigos. El aprovechamiento del paralelismo en las arquitecturas VLIW requiere pues elaboradas estrategias *software* como el desenrollado de bucles, la segmentación software o la planificación de trazas.

El procesador Intel Itanium2 es una arquitectura VLIW de propósito general.

1.5.3. Otros procesadores paralelos

Extensiones SIMD

Las extensiones SIMD (*Single Instruction Multiple Data*) aparecieron con el fin de optimizar la ejecución de las aplicaciones multimedia, donde es común encontrar una misma operación que debe repetirse sobre múltiples datos. Las extensiones SIMD incorporan un pequeño conjunto de instrucciones y recursos a la microarquitectura del procesador que permiten especificar en una misma intrucción la operación a realizar y varios datos sobre los que realizarla. Por ejemplo, si se precisa sumar una constante a todos los elementos de un vector, es posible utilizar una instrucción SIMD para realizar la suma sobre un grupo de elementos simultáneamente. En este sentido, este modo de funcionamiento está fuertemente relacionado con el procesamiento vectorial.

GPU

Los procesadores gráficos (GPU o *Graphics Processing Units*) son arquitecturas inicialmente diseñadas para el procesamiento de imágenes gráficas. Las operaciones efectuadas sobre gráficos

presentan unas propiedades muy específicas:

- La misma operación se realiza repetidamente sobre un conjunto de datos (vector).
- Presentan un elevado nivel de paralelismo entre instrucciones.
- Trabajan con datos en precisión simple.

Estas características llevaron al desarrollo de procesadores gráficos, ya que es posible implementar *hardware* específico más eficiente, simple y económico. En la actualidad existen en el mercado arquitecturas GPU muy potentes, con gran nivel de segmentación y con acceso paralelo a la memoria. En los últimos años, las limitaciones de la tecnología actual han hecho crecer el interés en utilizar los procesadores gráficos para computación de propósito general.

Cell BE

El procesador Cell BE (abreviatura de Cell Broadband Engine) es una arquitectura de altas prestaciones que incluye diversas unidades funcionales especializadas. Desarrollado por IBM, Sony y Toshiba, tiene una gran capacidad computacional para aplicaciones multimedia, habiendo sido diseñado como núcleo de la videoconsola *PlayStation 3*.

Su diseño le dota de una arquitectura escalable, en la que pueden cooperar múltiples unidades de cómputo y en la que se prioriza el aumento en el ancho de banda sobre la latencia.

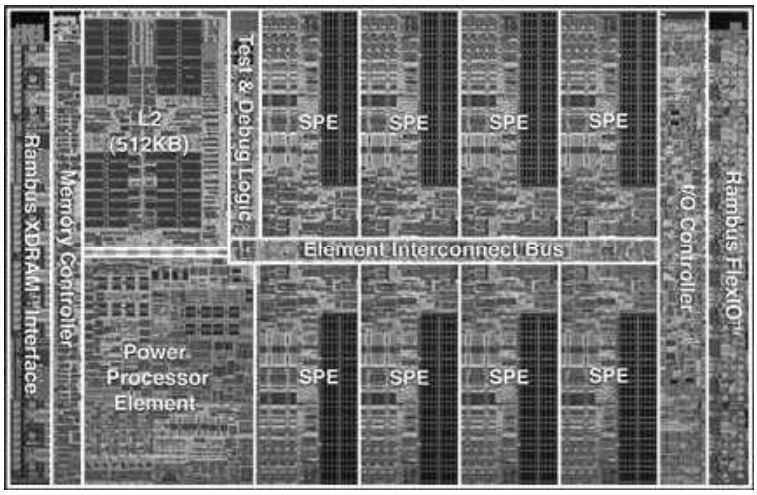


Figura 1.9: Imagen de un procesador Cell BE.

Un procesador Cell BE incluye un procesador principal, llamado PPE (*Power Processing Element*), ocho co-procesadores, llamados SPE (*Synergistic Processing Elements*), y un bus de conexión, EIB (*Element Interconnect Bus*), de altas prestaciones para conectar entre sí los SPE y comunicarlos con los elementos de entrada/salida.

1.5.4. Multiprocesadores y procesadores multinúcleo

Multiprocesadores

Los multiprocesadores con memoria compartida (MMC, a partir de ahora, simplemente multiprocesadores) son plataformas compuestas por varios procesadores completos que disponen de una

memoria común accesible mediante instrucciones *hardware*. Existen dos posibilidades en cuanto a la forma en que se realiza el acceso a la memoria en estas plataformas:

- SMP (*Symmetric Multiprocessor*): El coste de acceso es uniforme a cualquier posición de la memoria.
- NUMA (*Non-Uniform Memory Access*): La memoria está físicamente distribuida entre los procesadores, de modo que cada procesador dispone de una memoria local de acceso más rápido que cuando se accede a la memoria local a otro procesador.

Debido a la simplicidad del principio en el que están fundamentados (replicación completa de procesadores), éstas fueron unas de las primeras arquitecturas paralelas que se diseñaron. La facilidad de su programación, basada en el paradigma de programación de variables compartidas (habitualmente, accesible mediante hebras de ejecución o herramientas de más alto nivel como paralelización de bucles), hace prever que seguirán siendo una plataforma paralela válida en los próximos años. La principal crítica que se hace a los multiprocesadores es la escasa escalabidad desde el punto de vista *hardware* (a medida que aumenta el número de procesadores, la memoria en el caso de los SMP o la red de interconexión en las arquitecturas NUMA se transforman en un cuello de botella). Sin embargo, el número de procesadores que pueden soportar eficientemente estas plataformas es más que suficiente para muchas aplicaciones.

Procesadores multinúcleo

Los procesadores multinúcleo (*multicore processors* o *chip multiprocessors*) incluyen en un sólo chip varias unidades de proceso independientes, denominadas núcleos (o *cores*). Cada núcleo es una unidad operacional completa, es decir, puede incluso ser un procesador con técnicas sofisticadas de paralelismo y/o segmentación. Los problemas de disipación del calor y de consumo de energía de la tecnología actual provocaron que, a partir de 2005, los principales fabricantes de procesadores incorporasen diseños multinúcleo para seguir transformando las mejoras en la escala de integración dictadas por la ley de Moore en un mayor rendimiento de sus productos.

En la actualidad los procesadores multinúcleo incluyen un reducido número de núcleos (entre 4 y 8 en la mayor parte de los casos), pero se espera que en un futuro próximo esta cantidad se vea aumentada considerablemente. Los procesadores multinúcleo, si bien pueden programarse del mismo modo que los multiprocesadores, poseen características propias que los hacen diferentes:

- Las tendencias de diseño apuntan a procesadores multinúcleo heterogéneos, con núcleos de distinta capacidad y/o naturaleza integrados en un mismo sistema [55, 56].
- Las previsiones para los procesadores multinúcleo apuntan a cientos de núcleos en un *chip* [19], frente a los multiprocesadores que, en sus configuraciones comerciales más frecuentes, no superan los 16 ó 32 procesadores.
- La organización habitual de los procesadores multinúcleo (ver figura 1.10) implica comunicaciones entre procesos mucho más eficientes (menor latencia, mayor ancho de banda y consumo más reducido) cuando los datos residen dentro del propio *chip*. Esta economía no es posible en los multiprocesadores y tampoco en los procesadores multinúcleo cuando los datos residen en los niveles de memoria fuera del *chip* [54].

Estas diferencias implican varios *requisitos específicos* sobre la paralelización de bibliotecas de computación de altas prestaciones para procesadores multinúcleo:

- Debido a la variabilidad e incluso heterogeneidad de las soluciones actuales y futuras, las bibliotecas deben ser fácilmente adaptables (flexibles) a diferentes escenarios con la menor merma de eficiencia posible.
- Debido a la mayor concurrencia de estos sistemas, la escalabilidad de las soluciones debe plantearse como criterio fundamental en el desarrollo de bibliotecas.
- Debido a su mayor latencia y elevado consumo, resulta crucial minimizar los accesos a memoria fuera del *chip* que realizan los códigos de las bibliotecas.

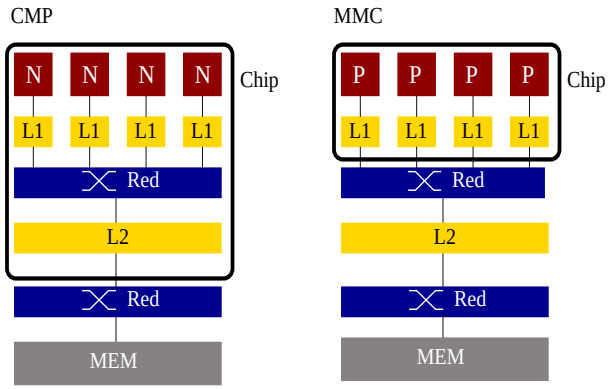


Figura 1.10: Organización habitual de un procesador multinúcleo (CMP, izquierda) y un multiprocesador (MMC, derecha). `N`=núcleo, `P`=procesador, `L1`=caché de primer nivel, `L2`=caché de segundo nivel, `MEM`=memoria principal. La línea gruesa marca los límites del *chip*.

El modo de programación habitual en MMC y procesadores multinúcleo, basado en el uso de hebras, nos hace adoptar el término de arquitecturas o sistemas multihebra para referirnos a ambas plataformas.

1.6. Análisis del Rendimiento

1.6.1. Medidas

La medida fundamental para medir el rendimiento (o eficiencia) de un código es el *tiempo de ejecución*. A pesar de esto, en códigos que principalmente realizan operaciones aritméticas en coma flotante, como es el caso que nos ocupa, a menudo se utiliza como medida de rendimiento la velocidad a la que se efectúan estas operaciones. En concreto, si definimos *flop* como una operación en aritmética de coma flotante (*floating-point arithmetic operation*), la velocidad de ejecución de los códigos que realizan operaciones de álgebra lineal suele medirse en términos de MFLOPs (10^6 flops/seg.) o GFLOPs (10^9 flops/seg.). Pese a ser una medida derivada y no principal como el tiempo de ejecución, la tasa de FLOPs presenta una clara ventaja en la representación gráfica de resultados. Así, mientras que al crecer el tamaño del problema el tiempo de ejecución sigue creciendo proporcionalmente (a menudo en una relación cuadrática o cúbica con éste), la tasa de FLOPs está limitada por la configuración y velocidad del *hardware* (básicamente, el tiempo de ciclo, el número de unidades funcionales del procesador, la tasa de transferencia desde la caché, la

Plataforma	Arquitectura	#Proc.	Frecuencia (GHz)	Cache L2 (KBytes)	Cache L3 (MBytes)	RAM (GBytes)
XEON	INTEL XEON	2	2.4	512	–	1
ITANIUM(SMP)	INTEL ITANIUM2	4	1.5	256	4	4
ITANIUM(NUMA)	INTEL ITANIUM2	16	1.5	256	6	32
OPTERON(NUMA)	AMD OPTERON	16	2.2	2x1024	–	64

Tabla 1.1: Arquitecturas empleadas en la evaluación.

velocidad del bus del sistema, etc.). De este modo las gráficas que representan la tasa de FLOPs tienen una cota superior que hace más clara la representación de los datos.

En cuanto a la ejecución en paralelo de un código, aunque existen medidas específicas muy utilizadas como la aceleración y la eficiencia (que aun así también son derivadas del tiempo de ejecución), en esta memoria optamos por mantener la homogeneidad de los resultados, midiendo el rendimiento de los códigos paralelos mediante la tasa de FLOPs.

1.6.2. Entorno de experimentación

Todos los resultados presentados en este trabajo corresponden a experimentos que se realizaron utilizando aritmética de coma flotante IEEE de precisión doble. En general se establece la dimensión de la matriz en 5,000, y se evalúa el rendimiento variando el ancho de banda entre 1 y 1,250. En este sentido, hay que tener en cuenta que, para problemas donde de tamaño de la matriz es considerablemente mayor que el tamaño de la banda, el rendimiento de las rutinas que implementan operaciones sobre matrices banda está determinado únicamente por el segundo de estos factores. En los algoritmos por bloques se ha experimentado con tamaños de bloque entre 1 y 100, pero únicamente se muestran los resultados correspondientes al tamaño de bloque que ha sido determinado como el óptimo a través de las pruebas.

Las características de las arquitecturas utilizadas en la evaluación figuran en la tabla 1.1. Las arquitecturas XEON e ITANIUM son multiprocesadores simétricos (SMP) mientras que ITANIUM y OPTERON son arquitecturas NUMA con los módulos de memoria RAM distribuidos físicamente entre 8 nodos de forma equitativa, con 2 procesadores por nodo. XEON e ITANIUM(SMP) son *chipsets* de INTEL. ITANIUM(NUMA) es un multiprocesador SGI ALTIX 350 con una red de interconexión en anillo desarrollada por la propia compañía (SGI NUMALINK). OPTERON(AMD) incluye 8 procesadores OPTERON de doble núcleo. Dado que la eficiencia de las rutinas de *BLAS* y la eficacia del compilador es crucial, para cada plataforma se ha utilizado las versiones de la tabla 1.2.

En los códigos paralelos, otro factor de influencia en las prestaciones es el número de *vías de paralelismo* que se utilizan en la ejecución. En nuestros experimentos el paralelismo se extrae ejecutando múltiples hilos o hebras (*threads*), un mecanismo con unos costes de sincronización y comunicación más ligeros que los asociados al uso de procesos, y muy adecuado para entornos con múltiples procesadores que comparten una memoria común. En la evaluación de los códigos paralelos, se utilizan tantos procesadores como número de hebras se determina para el experimento.

Plataforma	*BLAS*	Compilador	*Flags* de Optimización	Sistema Operativo
XEON	*GotoBLAS* 1.00 *MKL* 8.0	`gcc` 3.3.5	`-O3`	Linux 2.4.27
ITANIUM(SMP)	*GotoBLAS* 0.95mt *MKL* 8.0	`icc` 9.0	`-O3`	Linux 2.4.21
ITANIUM(NUMA)	*GotoBLAS* 0.95mt *MKL* 8.0	`icc` 9.0	`-O3`	Linux 2.4.21
OPTERON(SMP)	*MKL* 9.1	`icc`	`-O3`	Linux

Tabla 1.2: *Software* empleado en la evaluación.

Capítulo 2

BLAS 2 banda

En este capítulo se evalúan las rutinas que operan con matrices con estructura banda de las implementaciones de *BLAS* más difundidas actualmente, *BLAS de referencia*, *MKL* y *GotoBLAS*, proponiéndose diversas nuevas implementaciones que, en determinadas condiciones, resultan más eficientes. Las operaciones consideradas en el capítulo incluyen el producto de una matriz banda y un vector, donde la matriz puede además presentar una estructura simétrica o triangular, y la resolución de un sistema triangular banda de ecuaciones lineales. Por el número de datos y operaciones aritméticas que precisan, se trata pues de operaciones del nivel 2 de *BLAS*. En consecuencia, todas ellas se pueden implementar haciendo uso de rutinas de *BLAS*-1 y/o *BLAS*-2 denso. Si bien una implementación basada en *BLAS*-2 denso en general no implica un mejor uso del sistema de memoria, sí presenta otras ventajas sobre las implementaciones basadas en *BLAS*-1, como puede ser un menor número de llamadas a rutinas de *BLAS* o una mejor legilibidad del código. Además, las implementaciones basadas en *BLAS*-2 son un paso intermedio hacia las implementaciones de *BLAS*-3 propuestas para las operaciones consideradas en el siguiente capítulo.

De manera general, para el producto matriz simétrica banda por vector se define el ancho de banda de la matriz como k_d (es decir, el elemento (i, j) de la matriz es cero si $i > j+k_d$ o $j > i+k_d$). En las dos operaciones donde interviene una matriz triangular banda (producto matriz por vector y resolución de sistemas de ecuaciones lineales), únicamente se considera el caso triangular inferior del problema (el elemento (i, j) de la matriz es cero si $i > j + k_l$, con k_l la dimensión de la banda inferior, o $j > i$). Para matrices generales banda, se considera que el ancho de banda inferior es k_l y el ancho de banda superior es k_u (en definitiva, el elemento (i, j) de la matriz es cero si $i > j + k_l$ o $j > i + k_u$). En cada uno de estos casos sólo se almacenan los elementos no nulos de la matriz, dispuestos físicamente tal y como se ilustró en el capítulo 1.

El capítulo está estructurado en 4 secciones, con las 3 primeras de éstas correspondientes a las operaciones de producto matriz por vector con matrices simétrica banda, general banda y triangular banda, y la cuarta y última sección dedicada a la resolución de sistemas triangulares banda. Los resultados experimentales de este capítulo se ofrecen a medida que se introducen las operaciones e incluyen dos procesadores de altas prestaciones actuales: Intel Itanium2 e Intel xeon (a partir de ahora, arquitecturas itanium y xeon, respectivamente). El bajo coste de las operaciones de *BLAS*-2 banda, proporcional a la dimensión de la matriz y el tamaño de la banda, hace poco interesante el estudio de implementaciones paralelas de estas operaciones.

2.1. Producto de una matriz simétrica banda por un vector

La operación de *BLAS* considerada en esta sección corresponde a la expresión

$$y := \beta \cdot y + \alpha \cdot A \cdot x, \tag{2.1}$$

donde $\alpha, \beta \in \mathbb{R}$, los vectores $x, y \in \mathbb{R}^n$, y $A \in \mathbb{R}^{n \times n}$ es una matriz simétrica con estructura banda y ancho de banda k_d. Por simplicidad, asumimos que $\alpha = \beta = 1$ en (2.1), de modo que obtenemos una variante más sencilla de la operación,

$$y := y + A \cdot x. \tag{2.2}$$

En esta sección se evalúan y muestran diferentes rutinas (implementaciones) para la operación (2.2), la mayoría de ellas basadas en el uso de otras rutinas del nivel 1 y 2 de *BLAS* denso.

2.1.1. Algoritmo SBMV$_{\text{UNB}}$

La figura 2.1 muestra, en notación FLAME, el algoritmo SBMV$_{\text{UNB}}$ para el cálculo de la operación (2.2). En cada iteración del algoritmo se completa el cálculo del elemento γ_1 y se actualizan todas las componentes de y_2. La figura 2.2 detalla los elementos accedidos durante una de sus iteraciones. Como puede observarse en ambas figuras, únicamente se referencian los elementos de la parte triangular inferior de la matriz.

2.1.2. Implementación del BLAS de referencia

La rutina SBMV_REF del *BLAS de referencia* para la operación (2.2) implementa el algoritmo de la figura 2.1 con las operaciones

$$\gamma_1 := \gamma_1 + \alpha_{11} \cdot \chi_1 + a_{21}^T \cdot x_2, \tag{2.3}$$

$$y_2 := y_2 + a_{21} \cdot \chi_1, \tag{2.4}$$

calculadas mediante un único bucle que, en cada iteración, acumula sobre γ_1 cada una de las multiplicaciones del producto escalar $a_{21}^T \cdot x_2$ y, simultáneamente, actualiza uno de los elementos del vector y_2[1].

Considerando el almacenamiento compacto de la matriz, la figura 2.3 muestra los elementos de A referenciados durante una iteración de la rutina SBMV_REF, en este caso la segunda, para una matriz 10×10 con ancho de banda $k_d = 4$. Desde el punto de vista de los accesos a memoria, la rutina presenta dos propiedades muy favorables:

- El número de accesos es mínimo, ya que se recorre A una sóla vez.
- El acceso por columnas a los elementos de A, suponiendo que el esquema de almacenamiento de la matriz empaquetada corresponde a Fortran, coincide con la disposición física de la matriz en memoria, favoreciendo la prelectura de datos (*prefecth*) a la *cache*.

Estas dos características son altamente propicias para las arquitecturas actuales, en las que la velocidad de la memoria es muy inferior a la del procesador.

No obstante, los elementos de los vectores x e y son accedidos hasta en k_d iteraciones consecutivas, lo que puede propiciar una pérdida de prestaciones. El uso de rutinas del nivel 2 de *BLAS* puede reducir el número de lecturas.

[1] En realidad, la rutina SBMV_REF del *BLAS de referencia* implementa la operación (2.1). Inicialmente un bucle obtiene el producto $y := \beta \cdot y$, después dos bucles anidados calculan $y := \alpha \cdot A \cdot x + y$.

Algorithm: $[y] := \text{SBMV}_{\text{UNB}}(y, A, x, k_d)$

Partition $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_M \\ \hline y_B \end{array}\right)$, $A \rightarrow \left(\begin{array}{c|c|c} A_{TL} & & \\ \hline A_{ML} & A_{MM} & \\ \hline & A_{BM} & A_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_M \\ \hline x_B \end{array}\right)$

where y_T, x_T have 0 elements; y_M, x_M have $k_d + 1$ elements;
A_{TL} is 0×0; and A_{MM} is $(k_d + 1) \times (k_d + 1)$

while $m(y_T) < m(y)$ **do**

Repartition

$$\left(\begin{array}{c} y_T \\ \hline y_M \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \gamma_1 \\ \hline y_2 \\ \hline \gamma_3 \\ \hline y_4 \end{array}\right), \quad \left(\begin{array}{c|c|c} A_{TL} & \star & \\ \hline A_{ML} & A_{MM} & \star \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & \star & \star & & \\ \hline a_{10}^T & \alpha_{11} & \star & & \\ \hline A_{20} & a_{21} & A_{22} & \star & \star \\ \hline & & a_{32}^T & \alpha_{33} & \star \\ \hline & & A_{42} & a_{43} & A_{44} \end{array}\right),$$

$$\left(\begin{array}{c} x_T \\ \hline x_M \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \\ \hline \chi_3 \\ \hline x_4 \end{array}\right)$$

where γ_1, α_{11}, χ_1 are scalars;
γ_3, α_{33}, χ_3 are scalars if $m(y_B) > 0$ or empty otherwise

---

$\gamma_1 := \gamma_1 + \alpha_{11} \cdot \chi_1 + a_{21}^T \cdot x_2$
$y_2 := y_2 + a_{21} \cdot \chi_1$

Continue with

$$\left(\begin{array}{c} y_T \\ \hline y_M \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \gamma_1 \\ \hline y_2 \\ \hline \gamma_3 \\ \hline y_4 \end{array}\right), \quad \left(\begin{array}{c|c|c} A_{TL} & \star & \\ \hline A_{ML} & A_{MM} & \star \\ \hline & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & \star & \star & & \\ \hline a_{10}^T & \alpha_{11} & \star & & \\ \hline A_{20} & a_{21} & A_{22} & \star & \star \\ \hline & & a_{32}^T & \alpha_{33} & \star \\ \hline & & A_{42} & a_{43} & A_{44} \end{array}\right),$$

$$\left(\begin{array}{c} x_T \\ \hline x_M \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \\ \hline \chi_3 \\ \hline x_4 \end{array}\right)$$

endwhile

Figura 2.1: Algoritmo SBMV_{UNB} para la operación $y := y + A \cdot x$. Sólo la parte triangular inferior de A es accedida, de modo que aquellas partes de la matriz denotadas mediante el símbolo "$\star$" referencian elementos, trozos de vectores o bloques de la matriz no accedidos.

2.1.3. Implementaciones basadas en rutinas de BLAS-1 denso

La implementación de *BLAS de referencia* no hace uso de otras rutinas de *BLAS*. En cambio, es posible obtener nuevas implementaciones del algoritmo SBMV_{UNB} calculando las operaciones (2.3) y (2.4) respectivamente mediante sendas llamadas a las rutinas DOT y AXPY, pertenecientes al nivel 1 de *BLAS* denso. Así, teniendo en cuenta que tanto α_{11} y a_{21} como χ_1 y x_2 están dispuestos en posiciones consecutivas de memoria, una sóla invocación a la rutina DOT bastará para calcular γ_1. Además, la actualización del vector y_2 puede obtenerse mediante una llamada a la rutina AXPY.

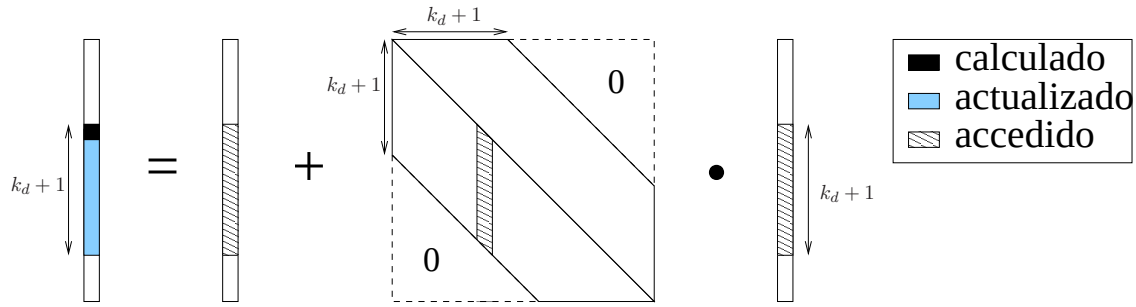


Figura 2.2: Acceso a los elementos durante una iteración del algoritmo SBMV_{UNB}.

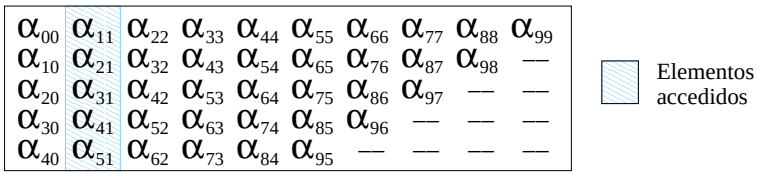


Figura 2.3: Acceso a los elementos en la segunda iteración (del algoritmo SBMV_{UNB} y) de la rutina SBMV_REF considerando el esquema compacto de almacenamiento para matrices simétricas banda.

De esta forma se plantean tres nuevas implementaciones (variantes) de la rutina SBMV:

- La rutina SBMV_B1 completa el cálculo de γ_1 y actualiza y_2 mediante sendas llamadas a DOT y AXPY.
- La rutina SBMV_B1_DOT realiza el cálculo de γ_1 en (2.3) mediante un bucle. Se trata pues de una variante de la rutina SBMV_B1 con el código de la rutina DOT *embebido* (*inline*). El cálculo de y_2 se sigue realizando mediante una llamada a AXPY.
- De manera recíproca, SBMV_B1_AXPY realiza el cálculo de y_2 en (2.4) mediante un bucle (código de la rutina AXPY embebido). El cálculo de γ_1 se sigue realizando mediante una llamada a DOT.

La conveniencia de estas variantes está supeditada al uso de implementaciones optimizadas de las rutinas DOT y AXPY, que hagan un uso eficiente de los recursos. Desde el punto de vista de los accesos a memoria, las tres variantes presentan las siguientes características:

- El modo de operar supone que los elementos de la matriz A son accedidos en dos ocasiones, característica ésta que supone doblar el número de accesos a la matriz que se producían en la rutina SBMV_REF del *BLAS de referencia*.
- Por otro lado, el acceso a los elementos de la matriz sigue coincidiendo con la disposición física de la matriz en memoria.

2.1.4. Algoritmo SBMV_{BLK}

El producto de una matriz (simétrica banda) por un vector es una operación perteneciente al nivel 2 de *BLAS* y, por lo tanto, es posible implementarla mediante llamadas a rutinas de *BLAS*-2 denso. Para poder emplear estas rutinas se precisa de un algoritmo por bloques como, por ejemplo,

Algorithm: $[y] := \text{SBMV}_{\text{BLK}}(y, A, x, k_d)$

Partition $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_M \\ \hline y_B \end{array}\right)$, $A \rightarrow \left(\begin{array}{c|c|c} A_{TL} & \star & \\ \hline A_{ML} & A_{MM} & \star \\ \hline & A_{BM} & A_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_M \\ \hline x_B \end{array}\right)$

where y_T, x_T have 0 elements; y_M, x_M have k_d elements; A_{TL} is 0×0; and A_{MM} is $k_d \times k_d$

while $m(y_T) < m(y)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c} y_T \\ \hline y_M \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline y_3 \\ \hline y_4 \end{array}\right), \quad \left(\begin{array}{c|c|c} A_{TL} & \star & \\ \hline A_{ML} & A_{MM} & \star \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & \star & \star & & \\ \hline A_{10} & A_{11} & \star & \star & \\ \hline A_{20} & A_{21} & A_{22} & \star & \star \\ \hline & A_{31} & A_{32} & A_{33} & \star \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right),$$

$$\left(\begin{array}{c} x_T \\ \hline x_M \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline x_3 \\ \hline x_4 \end{array}\right)$$

where x_1, y_1 have b_M elements; A_{11} is $b_M \times b_M$;
x_3, y_3 have b_B elements; and A_{33} is $b_B \times b_B$,
with $b_M := \text{mín}(b, m(y_M))$ and $b_B := \text{mín}(b, m(y_B))$

$$\begin{array}{lcl} y_1 & := & y_1 + A_{11} \cdot x_1 + A_{21}^T \cdot x_2 + A_{31}^T \cdot x_3 \\ y_2 & := & y_2 + A_{21} \cdot x_1 \\ y_3 & := & y_3 + A_{31} \cdot x_1 \end{array}$$

Continue with

$$\left(\begin{array}{c} y_T \\ \hline y_M \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline y_3 \\ \hline y_4 \end{array}\right), \quad \left(\begin{array}{c|c|c} A_{TL} & \star & \\ \hline A_{ML} & A_{MM} & \star \\ \hline & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & \star & \star & & \\ \hline A_{10} & A_{11} & \star & \star & \\ \hline A_{20} & A_{21} & A_{22} & \star & \star \\ \hline & A_{31} & A_{32} & A_{33} & \star \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right),$$

$$\left(\begin{array}{c} x_T \\ \hline x_M \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline x_3 \\ \hline x_4 \end{array}\right)$$

endwhile

Figura 2.4: Algoritmo por bloques SBMV_{BLK} para la operación $y := y + A \cdot x$. Sólo la parte triangular inferior de A es accedida.

el mostrado en la figura 2.4. En cada iteración de este algoritmo se completa el cálculo de los elementos del vector y_1 y se actualizan los elementos de y_2 e y_3, todo ello mediante operaciones sobre bloques de la matriz. Es importante destacar que, al igual que en el algoritmo mostrado en la figura 2.1, únicamente se accede a los elementos de la parte triangular inferior de A (incluida la operación con A_{11}).

2.1.5. Implementaciones basadas en rutinas de BLAS-2 denso

El algoritmo por bloques en la figura 2.4 puede implementarse mediante invocaciones a rutinas del nivel 2 de *BLAS* denso. A continuación se desglosan las operaciones realizadas en cada iteración

del algoritmo indicando (entre paréntesis), para cada una de éstas, la rutina de *BLAS*-2 denso que la implementa:

1. $A_{11} \cdot x_1$: producto matriz simétrica por vector (SYMV).
2. $A_{21}^T \cdot x_2$: producto matriz general por vector (GEMV).
3. $A_{31}^T \cdot x_3$: producto matriz triangular por vector (TRMV).
4. $A_{21} \cdot x_1$: producto matriz general por vector (GEMV).
5. $A_{31} \cdot x_1$: producto matriz triangular por vector (TRMV).

Todas las rutinas necesarias (SYMV, GEMV y TRMV) pertenecen al nivel 2 de *BLAS* denso. Ahora bien, la funcionalidad de la rutina TRMV no coincide plenamente con las necesidades del algoritmo. En la especificación de *BLAS*, esta rutina recibe como parámetros la matriz y el vector multiplicando, vector que finalmente se sobreescribe con el resultado. En nuestro caso, el resultado del producto se debe almacenar en y_1 en la primera invocación, e y_3 en la segunda invocación, mientras que los vectores multiplicando son x_1 y x_3, respectivamente. Es decir, contrariamente a la interfaz de TRMV, el vector multiplicando y el vector resultado son diferentes. Para solucionar este problema, se puede hacer una copia del vector multiplicando en un espacio de trabajo (w), utilizar este espacio en la invocación a la rutina TRMV, y finalmente acumular el resultado recogido en w sobre el vector resultado mediante una llamada a la rutina AXPY.

Teniendo en cuenta lo anterior, en cada iteración de la rutina SBMV_B2 se realizará la siguiente secuencia de operaciones e invocaciones a rutinas de *BLAS* denso:

$$
\begin{aligned}
&(\text{SYMV}) && y_1 && := y_1 + A_{11} \cdot\ x_1, && (2.5)\\
&(\text{GEMV}) && y_1 && := y_1 + A_{21}^T \cdot\ x_2, && (2.6)\\
& && y_1 && := y_1 + A_{31}^T \cdot\ x_3, && (2.7)\\
&(\text{COPY}) && w && := x_3, && (2.8)\\
&(\text{TRMV}) && w && := A_{31}^T \cdot\ w, && (2.9)\\
&(\text{AXPY}) && y_1 && := y_1 + w, && (2.10)\\
&(\text{GEMV}) && y_2 && := y_2 + A_{21} \cdot\ x_2, && (2.11)\\
& && y_3 && := y_3 + A_{31} \cdot\ x_3, && (2.12)\\
&(\text{COPY}) && w && := x_3, && (2.13)\\
&(\text{TRMV}) && w && := A_{31} \cdot\ w, && (2.14)\\
&(\text{AXPY}) && y_3 && := y_3 + w. && (2.15)
\end{aligned}
$$

Así pues, durante una iteración del algoritmo se realizan un total de 9 invocaciones a rutinas de *BLAS*-1 y 2 denso (SYMV, GEMV, COPY, TRMV y AXPY), seis de ellas (las realizadas al descomponer (2.7) y (2.12)) requeridas como consecuencia de las diferencias entre la funcionalidad ofrecida por TRMV y las necesidades de la rutina SBMV_B2.

En lo sucesivo se presentan diferentes variantes que disminuyen el número de llamadas a rutinas, de manera que se reduzca el sobrecoste debido a las propias invocaciones y, en algunos de estos casos, se realicen operaciones con bloques de mayor dimensión y, por lo tanto, con mayor carga de trabajo.

Implementación con el código COPY embebido (SBMV_B2_COPY)

Si analizamos las llamadas a la rutina COPY en (2.8) y (2.13) encontramos que la primera vez que se invoca a COPY es para el cómputo de y_1 y sólo se realiza la copia de b elementos, y la segunda vez que se utiliza esta rutina es para la actualización de y_3 y de nuevo se emplea únicamente para realizar la copia de b elementos.

Así pues, con b pequeño, el trabajo realizado en cada invocación a la rutina COPY es reducido. Esta circunstancia plantea como posible variación realizar las copias al espacio de trabajo mediante código embebido, dando lugar a la variante SBMV_B2_COPY. Si el número de elementos a copiar es pequeño, como cabe esperar, el tiempo requerido por este código embebido será menor que el utilizado por COPY.

Esta medida está destinada a reducir el tiempo dedicado a las copias de x_1 y x_3. No obstante, hay que tener en cuenta que el tiempo dedicado a la copia no es una parte importante del tiempo total, por lo que la posible mejora introducida por esta implementación será moderada. La conveniencia de esta variante dependerá, en gran medida, de la eficiencia de la rutina COPY, y de los valores de b y k_d. En concreto, a medida que aumenta el ratio k_d/b, los beneficios de esta variante se reducen.

Implementación con código AXPY embebido (SBMV_B2_AXPY)

La rutina AXPY también es invocada en dos ocasiones, en (2.10) y (2.15), operando en cada una de éstas con un vector de talla b, es decir, con un vector de dimensión reducida. Así pues, de nuevo tenemos la posibilidad mejorar las prestaciones sustituyendo cada invocación por código embebido, lo que resulta en la variante SBMV_B2_AXPY.

Sin embargo, una vez más, el tiempo empleado por la rutina AXPY no representa una parte importante del tiempo total, con lo que la posible mejora introducida será moderada. Al igual que en el caso anterior, la conveniencia de esta variante está condicionada por la eficiencia de una rutina de *BLAS*, en este caso AXPY, y por los valores de b y k_d.

Imp lementación con código COPY y AXPY embebido (SBMV_B2_COPY_AXPY)

En esta rutina se unen las modificaciones descritas en las dos variantes anteriores, de forma que tanto las llamadas a la rutina COPY como a la rutina AXPY se reemplazan por código embebido.

Implementación con código TRMV embebido (SBMV_B2_TRMV)

La rutina TRMV devuelve el resultado en el mismo vector en el que recibe los datos. Es por ello que, en las operaciones (2.7) y (2.12), se necesita utilizar un espacio de trabajo w y las rutinas COPY y AXPY. La variante SBMV_B2_TRMV implementa el producto de la matriz triangular mediante código embebido, de manera que no se precisen las llamadas a las rutinas COPY, TRMV y AXPY.

Como resultado, al realizarse las operaciones mediante código embebido, se evitan las llamadas a algunas rutinas de *BLAS* denso, eliminando el sobrecoste de las propias llamadas; a cambio, estas operaciones no son ejecutadas por una rutina *BLAS* optimizada.

Implementación Merge (SBMV_B2_MERGE)

Esta variante, SBMV_B2_MERGE, reduce el número de llamadas a rutinas de *BLAS* y al mismo tiempo opera con bloques de mayor dimensión. El hecho de operar con bloques de mayor tamaño permite a las rutinas de *BLAS* (especialmente de *BLAS* paralelo) obtener mejores prestaciones. Como se ha comentado anteriormente, seis de las invocaciones a rutinas de *BLAS* son realizadas para ejecutar las operaciones (2.7) y (2.12), y cuatro de éstas (las realizadas a COPY y AXPY) se

deben a las diferencias entre la funcionalidad de la rutina TRMV y los requerimientos del algoritmo SBMV_{BLK}. Ahora bien, si todos los elementos del bloque A_{31} se encontraran almacenados (incluyendo aquellos elementos nulos que caen fuera de la banda, es decir, la parte estrictamente triangular inferior del bloque A_{31}), sería posible operar con los bloques A_{21} y A_{31} conjuntamente, puesto que conformarían entre ambos una matriz densa. De esta forma, ambos bloques podrían ser computados simultáneamente con una única llamada a la rutina GEMV, suplantando esta invocación a las que se realizan a las rutinas COPY, TRMV y AXPY en (2.5)–(2.15). Esta variante reduce el número de invocaciones a rutinas por iteración a tan sólo 3 (rutinas SYMV, GEMV y GEMV). A cambio opera con elementos nulos (en concreto, se realizan $(b^2 - 2b)$ operaciones no útiles, cuyo resultado es 0) y se realizan copias de bloques de datos de tamaño reducido.

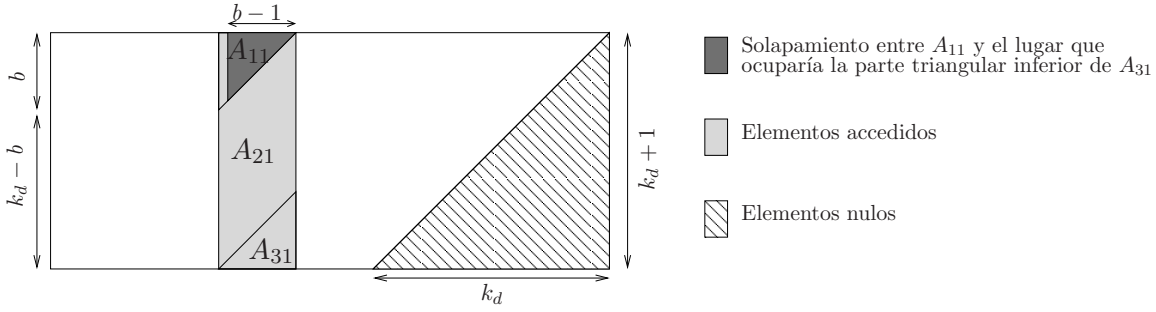


Figura 2.5: Acceso a los elementos en la implementación SBMV_B2_MERGE.

Para simular que A_{31} está almacenada completamente se precisa rellenar con ceros ciertas posiciones de memoria, ocupadas por otros elementos de la matriz banda. Esto es precisamente lo que implementa esta versión: en cada iteración rellena la región de memoria en la que se almacenaría la parte estrictamente triangular inferior de A_{31} (ver figura 2.5) y llama a la función GEMV con la matriz formada por los bloques A_{21} y A_{31} (incluida su parte estrictamente triangular inferior).

Las operaciones a ejecutar en cada iteración de la rutina son las siguientes:

$$
\begin{aligned}
&(\text{SYMV}) && y_1 && := y_1 + A_{11} \cdot\ x_1, && (2.16)\\
& && W && := \text{STRIL}(A_{31}), && (2.17)\\
& && \text{STRIL}(A_{31}) && := 0, && (2.18)\\
&(\text{GEMV}) && y_1 && := y_1 + \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot \begin{bmatrix} x_2 \\ x_3 \end{bmatrix}, && (2.19)\\
&(\text{GEMV}) && \begin{bmatrix} y_2 \\ y_3 \end{bmatrix} && := \begin{bmatrix} y_2 \\ y_3 \end{bmatrix} + \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot\ x_1, && (2.20)\\
& && \text{STRIL}(A_{31}) && := W. && (2.21)
\end{aligned}
$$

La notación $\text{STRIL}(B)$ se refiere a la parte estrictamente triangular inferior de la matriz B.

Hay que recordar que tanto el tamaño del bloque a copiar como el número de elementos nulos con los que operar dependen de b ($\frac{b^2}{2} - b$ en ambos casos) y, dado que b toma valores reducidos, el sobrecoste introducido por esta variante no es muy elevado evitándose, a cambio, el sobrecoste derivado por seis llamadas a rutinas, cuatro de ellas del nivel 1 de *BLAS* denso.

2.1.6. Resultados experimentales

Arquitectura ITANIUM

La figura 2.6 recoge los resultados obtenidos con la implementación de la rutina SBMV del *BLAS de referencia*, compilada utilizando el compilador de INTEL *ifort* y el compilador de GNU *g77* (SBMV_REF+*ifort* y SBMV_REF+*g77* respectivamente), así como las implementaciones de esta rutina en las bibliotecas *MKL* (SBMV_MKL) y *GotoBLAS* (SBMV_GOTO). Como se puede comprobar, la rutina de la biblioteca *BLAS de referencia* compilada con *ifort* obtiene las mejores prestaciones para matrices de banda muy estrecha. Esto se debe a la utilización de código *embebido* y a la correspondiente ausencia de sobrecoste provocado por la invocación de rutinas. Las llamadas a rutinas conllevan un coste computacional que, en situaciones con poca carga de trabajo, adquiere un peso relevante.

A medida que aumenta el tamaño de la banda de la matriz, SBMV_MKL primero y SBMV_GOTO después, consiguen los mejores resultados. No obstante, para matrices de banda media y ancha la mejor respuesta se obtiene de nuevo con la rutina SBMV_REF+*ifort*, aunque en esta ocasión la rutina SBMV_GOTO ofrece prestaciones prácticamente iguales. La línea correspondiente a la rutina SBMV_MKL indica claramente que ésta ha sido especialmente optimizada para matrices de banda estrecha, mientras que para matrices de banda media y ancha revela unas prestaciones muy discretas.

Respecto a los compiladores empleados, podemos decir que *ifort* produce un código más eficiente que *g77*. Dado que la diferencia entre las prestaciones obtenidas por los códigos de ambos compiladores es importante, durante el resto de experimentos sobre ITANIUM correspondientes a esta operación únicamente utilizaremos el compilador *ifort*.

La figura 2.7 muestra los resultados para las diferentes variantes que implementan el algoritmo SBMV_{UNB} haciendo uso de rutinas del nivel 1 de *BLAS* denso (apartado 2.1.3). En los experimentos realizados, tanto para *GotoBLAS* como para *MKL*, la variante con mejores prestaciones es SBMV_B1_AXPY, es decir, la variante que invoca a la rutina *BLAS*-1 DOT pero, en cambio, realiza la operación AXPY mediante un bucle embebido. Para matrices de banda estrecha (ancho de banda menor que 100), los mejores resultados se obtienen con la rutina SBMV_MKL.

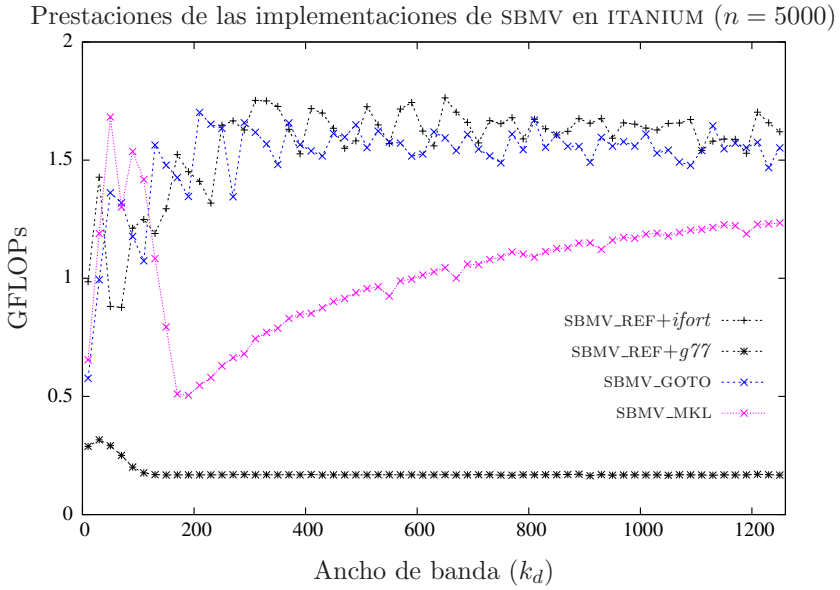


Figura 2.6: Comparativa de las diferentes implementaciones de bibliotecas *BLAS*.

La figura 2.8 muestra la eficiencia de las variantes que implementan el algoritmo SBMV_{BLK} mediante invocaciones a rutinas del nivel 2 de *BLAS* denso (apartado 2.1.5). Como se puede apreciar, las variantes introducidas mejoran las prestaciones de la rutina de *MKL* para anchos de banda mayores a 150, pero no superan en ningún caso las obtenidas por la rutina de *GotoBLAS*. Los resultados muestran que no hay grandes diferencias de prestaciones entre las diferentes implementaciones basadas en rutinas de *BLAS*-2 denso propuestas.

Por último, la figura 2.9 recoge los resultados obtenidos con las rutinas de las bibliotecas *BLAS de referencia* compilado con *ifort* (SBMV_REF+*ifort*), *GotoBLAS* (SBMV_GOTO) y *MKL* (SBMV_MKL), así como de las variantes con mejores prestaciones basadas en *BLAS*-1 denso y *BLAS*-2 denso, SBMV_B1_AXPY y SBMV_B2, respectivamente. La rutina del *BLAS de referencia* se presenta como la mejor opción para matrices con ancho de banda menor que 40. Para anchos de banda entre 40 y 100 las mejores prestaciones se obtienen con la rutina SBMV_MKL. Para anchos de banda entre 100 y 200, SBMV_GOTO, SBMV_REF+*ifort* y SBMV_B1_AXPY+*GotoBLAS* (que invoca a la rutina DOT de *GotoBLAS*) ofrecen prestaciones similares; finalmente, para matrices con ancho de banda mayor que 200, SBMV_B1_AXPY+*GotoBLAS* es la rutina que presenta mejores resultados.

Arquitectura XEON

La figura 2.10 muestra los resultados obtenidos para las dos implementaciones de *BLAS* optimizadas (*MKL* y *GotoBLAS*), y la implementación del *BLAS de referencia* utilizando ambos compiladores (*ifort* y *g77*). Contrariamente a lo que ocurre en la arquitectura ITANIUM, no existen diferencias notables entre las prestaciones obtenidas por ambos compiladores. Por este motivo, en lo sucesivo los experimentos se realizarán únicamente con el compilador *ifort*, aunque igualmente podrían haberse obtenido con el compilador *g77*.

La rutina de la biblioteca *GotoBLAS*, SBMV_GOTO, obtiene las mejores prestaciones para anchos de banda entre 20 y 420, mientras que para anchos de banda superiores es la rutina de *MKL*, SBMV_MKL, la que obtiene los mejores resultados.

En consecuencia, el comportamiento de la rutina SBMV_MKL en esta arquitectura difiere del visto en la arquitectura ITANIUM, ya que en este caso la rutina de *MKL* parece estar optimizada para matrices de banda tanto media como ancha.

Respecto a las variantes basadas en rutinas de *BLAS*-1 denso, los resultados reflejados en la figura 2.11 muestran que, salvo en unos pocos casos escasamente significativos, ninguna de estas variantes mejora las prestaciones obtenidas por las rutinas SBMV_MKL y SBMV_GOTO.

Igualmente, la figura 2.12 muestra que, en el caso de las variantes que implementan el algoritmo SBMV_{BLK} basándose en rutinas del nivel 2 de *BLAS* denso, ninguna de éstas mejora las prestaciones obtenidas por las rutinas de las bibliotecas *GotoBLAS* y *MKL*. Además, al igual que ocurre en la arquitectura ITANIUM, todas estas variantes obtienen resultados similares.

La figura 2.13 regoge los resultados de las rutinas de las tres bibliotecas *BLAS* estudiadas (*GotoBLAS*, *MKL* y *BLAS de referencia*), así como los de las variantes basadas en *BLAS*-1 denso y *BLAS*-2 denso que presentan las mejores prestaciones, SBMV_B1_AXPY y SBMV_B2, respectivamente. Como se puede comprobar, las rutinas de *GotoBLAS* y *MKL* obtienen los mejores resultados. La primera para matrices de banda estrecha y la última para matrices de banda ancha. También podemos observar que, a diferencia de lo visto en la arquitectura ITANIUM, las implementaciones que hacen uso de rutinas de *BLAS*-2 denso obtienen mejores prestaciones que las basadas en rutinas de *BLAS*-1.

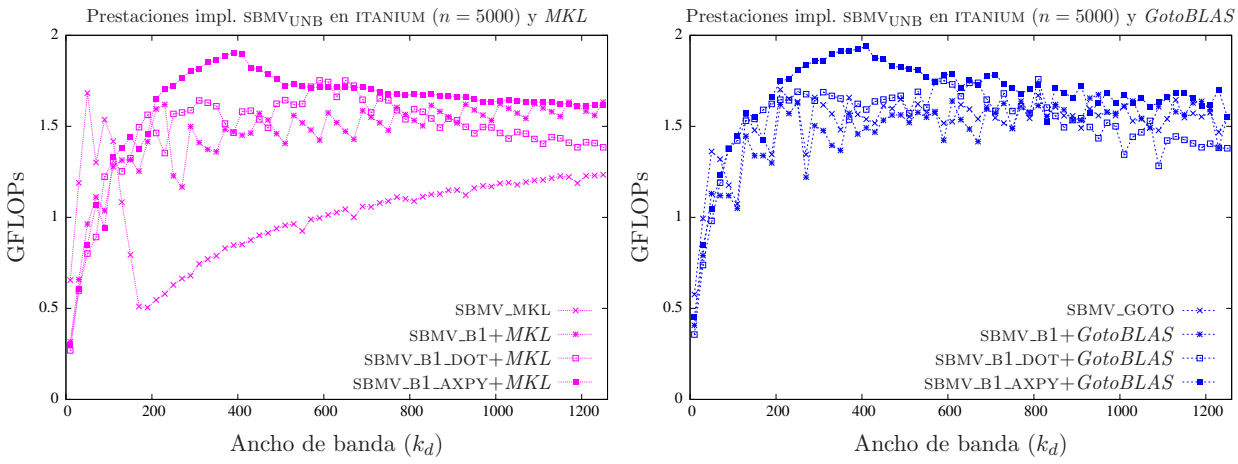


Figura 2.7: Comparativa de las diferentes implementaciones basadas en *BLAS*-1 denso.

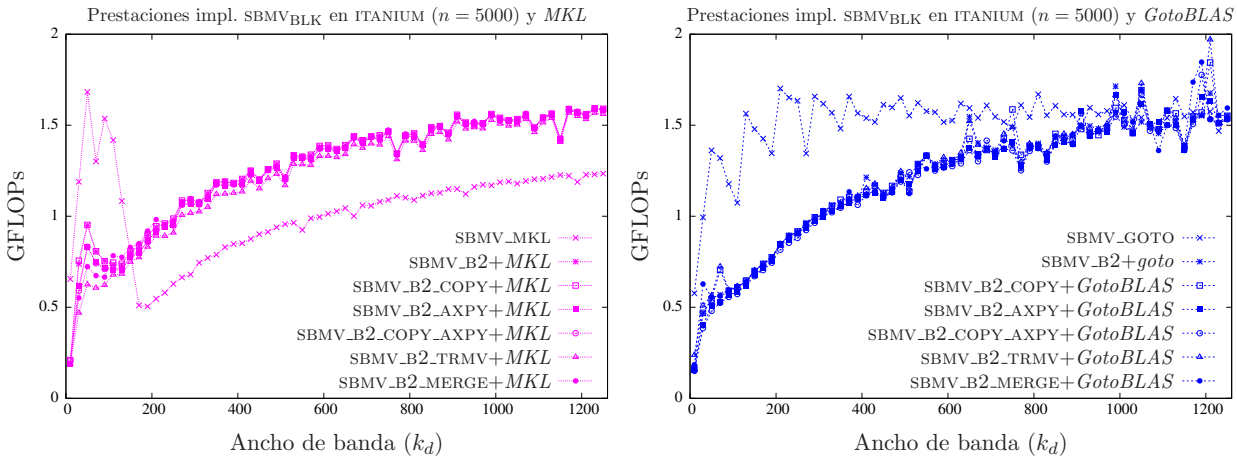


Figura 2.8: Comparativa de las diferentes implementaciones basadas en *BLAS*-2 denso.

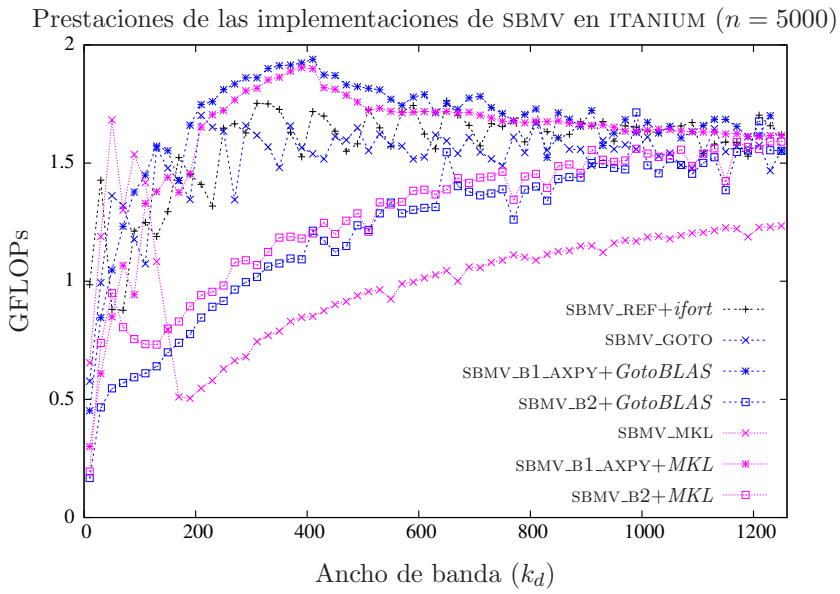


Figura 2.9: Comparativa de las mejores implementaciones.

2.1.7. Conclusiones

La operación (2.1) pertenece al nivel 2 de *BLAS*, por lo que puede ser implementada utilizando rutinas del primer y/o del segundo nivel de *BLAS*. Se han propuesto diferentes implementaciones para ambos niveles y se han comparado los resultados con los obtenidos por el *BLAS de referencia* así como con los de las implementaciones de *BLAS* optimizadas *MKL* y *GotoBLAS*.

En la arquitectura ITANIUM, los resultados demuestran la conveniencia de utilizar, para esta operación, implementaciones basadas en el uso de rutinas de *BLAS*-1 denso. Destaca en particular la eficiencia de la nueva rutina SBMV_B1_AXPY, que basa su funcionamiento en la rutina DOT. Las prestaciones obtenidas por SBMV_B1_AXPY son superiores a las alcanzadas por el resto de rutinas, incluidas las de las bibliotecas *MKL* y *GotoBLAS*. Las gráficas también muestran que la rutina DOT de la biblioteca *GotoBLAS* es ligeramente más eficiente que su análoga en *MKL*.

En lo referente a la arquitectura XEON, SBMV_GOTO es la mejor opción si se opera con matrices de banda estrecha, mientras que para anchos de banda mayores, SBMV_MKL obtiene las mejores prestaciones. Este comportamiento de la rutina de *MKL* difiere del encontrado en la arquitectura ITANIUM, donde estaba claramente optimizada para operar con matrices de banda estrecha. Al igual que en la arquitectura ITANIUM, no se aprecian diferencias significativas entre las prestaciones obtenidas por las distintas implementaciones basadas en rutinas de *BLAS*-2 denso.

Es interesante destacar que en la arquitectura ITANIUM, las implementaciones basadas en rutinas del nivel 1 de *BLAS* obtienen mejores prestaciones que las basadas en rutinas del segundo nivel, mientras que en XEON sucede exactamente lo contrario.

2.2. Producto de una matriz general banda por un vector

La operación estudiada a continuación responde a la expresión

$$y := \beta \cdot y + \alpha \cdot A \cdot x, \tag{2.22}$$

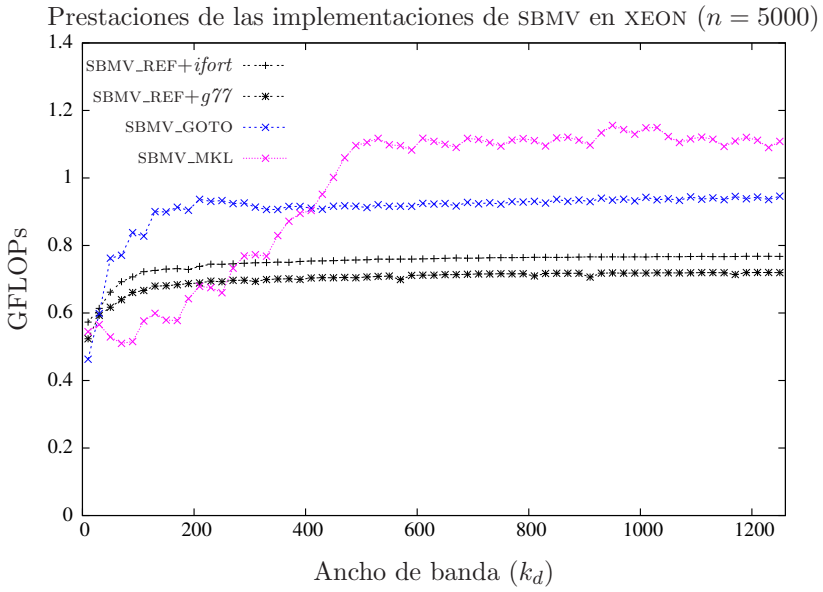


Figura 2.10: Comparativa de las diferentes implementaciones de bibliotecas *BLAS*.

donde $\alpha, \beta \in \mathbb{R}$; $x \in \mathbb{R}^m$ e $y \in \mathbb{R}^n$ son vectores; y A $\in \mathbb{R}^{n \times m}$ es una matriz general con estructura banda y anchos de banda superior e inferior k_u y k_l, respectivamente. Por simplicidad, consideraremos el caso en que $\alpha = \beta = 1$ en (2.22), de modo que obtenemos una variante más sencilla de la operación,

$$y := y + A \cdot x. \tag{2.23}$$

También, aunque la matriz A puede aparecer traspuesta en los productos anteriores (por ejemplo, $y := y + A^T \cdot x$), únicamente consideramos el caso no transpuesto de la operación. El estudio aquí desarrollado puede extenderse fácilmente a este otro caso del producto matriz general banda por vector.

En esta sección se evalúan rutinas para la operación (2.23) en las bibliotecas *BLAS de referencia*, *GotoBLAS* y *MKL*, y se diseñan y desarrollan nuevas implementaciones basadas en el uso de rutinas de los niveles 1 y 2 de *BLAS* denso.

2.2.1. Algoritmo GBMV$_{\text{UNB}}$

El algoritmo mostrado en la figura 2.14 calcula el producto (2.23). Se trata de un algoritmo iterativo que, durante la iteración j, toma la primera columna de A con la que todavía no se ha operado (columna j) y el elemento j-ésimo del vector x (χ_j), calcula su producto, y acumula el resultado sobre el vector y. A lo largo de las primeras k_u iteraciones tanto α_{11} como χ_1 están vacíos y, por lo tanto, el tamaño de y_M aumenta hasta llegar a estar formado por $k_l + k_u$ elementos. Al mismo tiempo el tamaño de y_T se mantiene, siendo un bloque vacío durante estas primeras k_u iteraciones. Algo similar ocurre durante las últimas $k_l + 1$ iteraciones, en las que χ_3 y a_{32} son vectores vacíos. Esto provoca que el tamaño de y_M y A_{MR} decrezca hasta convertirse en vector y bloque vacío, respectivamente.

El vector a_{01} está formado por todos los elementos nulos situados por encima de la banda, mientras que a_{31} incluye los elementos nulos que están situados por debajo de la banda. El resto de los elementos de la columna, aquellos que están dentro de la banda, forman α_{11} y a_{21}.

El algoritmo GBMV$_{\text{UNB}}$ recorre la matriz A por columnas, tal y como se muestra en la figura

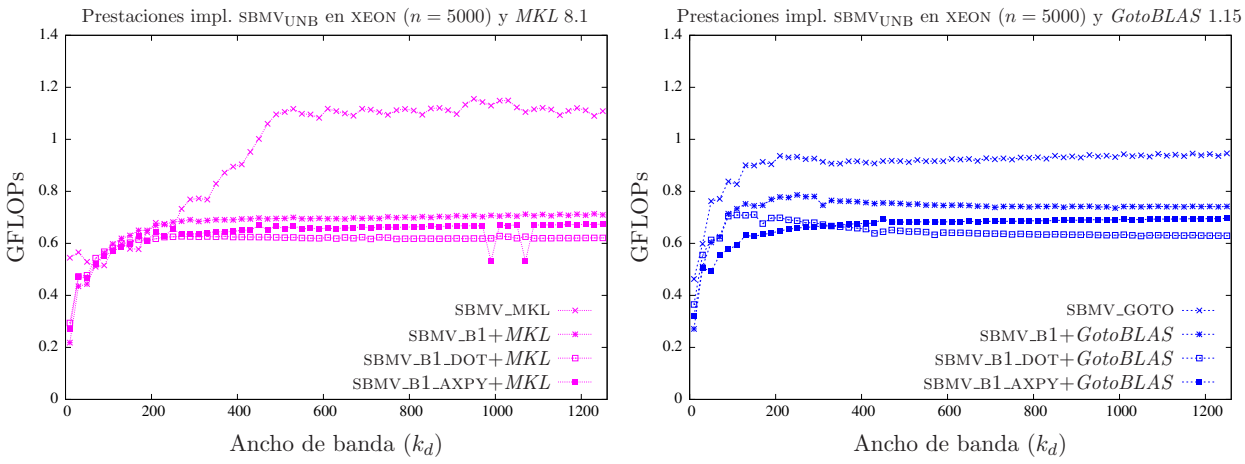


Figura 2.11: Comparativa de las diferentes implementaciones basadas en *BLAS*-1 denso.

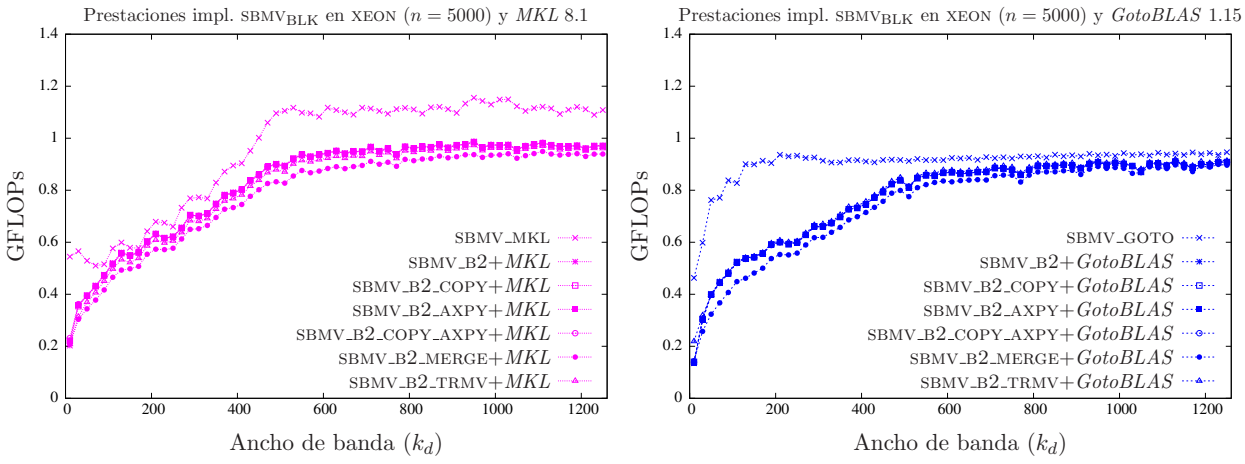


Figura 2.12: Comparativa de las diferentes implementaciones basadas en *BLAS*-2 denso.

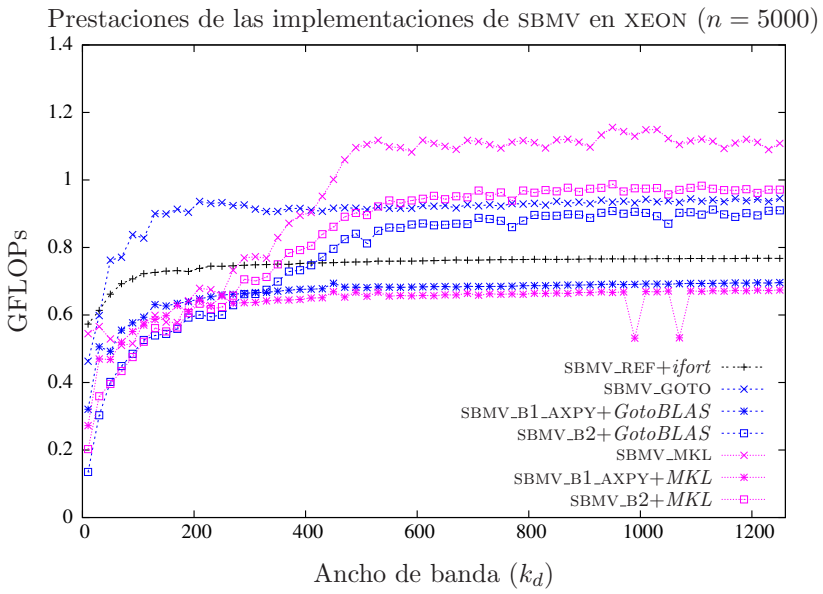


Figura 2.13: Comparativa de las mejores implementaciones.

2.15, de forma que adapta el modo de acceso al esquema físico de almacenamiento, que queda ilustrado por la figura 2.16.

2.2.2. Implementación de BLAS de referencia

La rutina GBMV_REF perteneciente al *BLAS de referencia* ejecuta la operación (2.22) siguiendo el algoritmo GBMV_{UNB}. En esta rutina, la actualización del vector formado por γ_1 e y_2 ($[\gamma_1\ y_2^T]^T$), se ejecuta mediante un bucle que recorre el vector $[\alpha_{11}\ a_{21}^T]^T$; durante la j-ésima iteración de este bucle se acumula en s_j el producto escalar entre la componente j-ésima de $[\alpha_{11}\ a_{21}^T]^T$ y χ_1.

Al igual que sucede con el algoritmo GBMV_{UNB}, la mejor cualidad de GBMV_REF es la forma en la que se accede a los elementos de la matriz A. En primer lugar, el número de accesos sobre la matriz A es minimizado, ya que cada elemento de la matriz es accedido una única vez. En segundo lugar, los accesos a los elementos se realizan por columnas (como muestra la figura 2.16) y, por tanto, se adecúa a la disposición física de los elementos en memoria.

2.2.3. Implementaciones basadas en rutinas de BLAS-1 denso

El algoritmo GBMV_{UNB} puede ser implementado mediante rutinas del nivel 1 de *BLAS* denso, de forma que los cálculos sean realizados por implementaciones de altas prestaciones, y al mismo tiempo se efectúe un acceso óptimo a los elementos de A. Éste es el caso de la rutina GBMV_B1, en la que las operaciones aritméticas precisas para la actualización de $[\gamma_1\ y_2^T]^T$ son ejecutadas por la rutina AXPY.

Como alternativa, la rutina GBMV_B1_AXPY_DOT implementa el algoritmo $\text{GBMV}_{\text{UNB\_DOT}}$, en la figura 2.17, de modo que la matriz A es recorrida a lo largo de su diagonal. La parte triangular inferior de A es accedida por columnas, mientras que la parte estrictamente triangular superior de A es accedida por filas, tal y como se muestra en la figura 2.18.

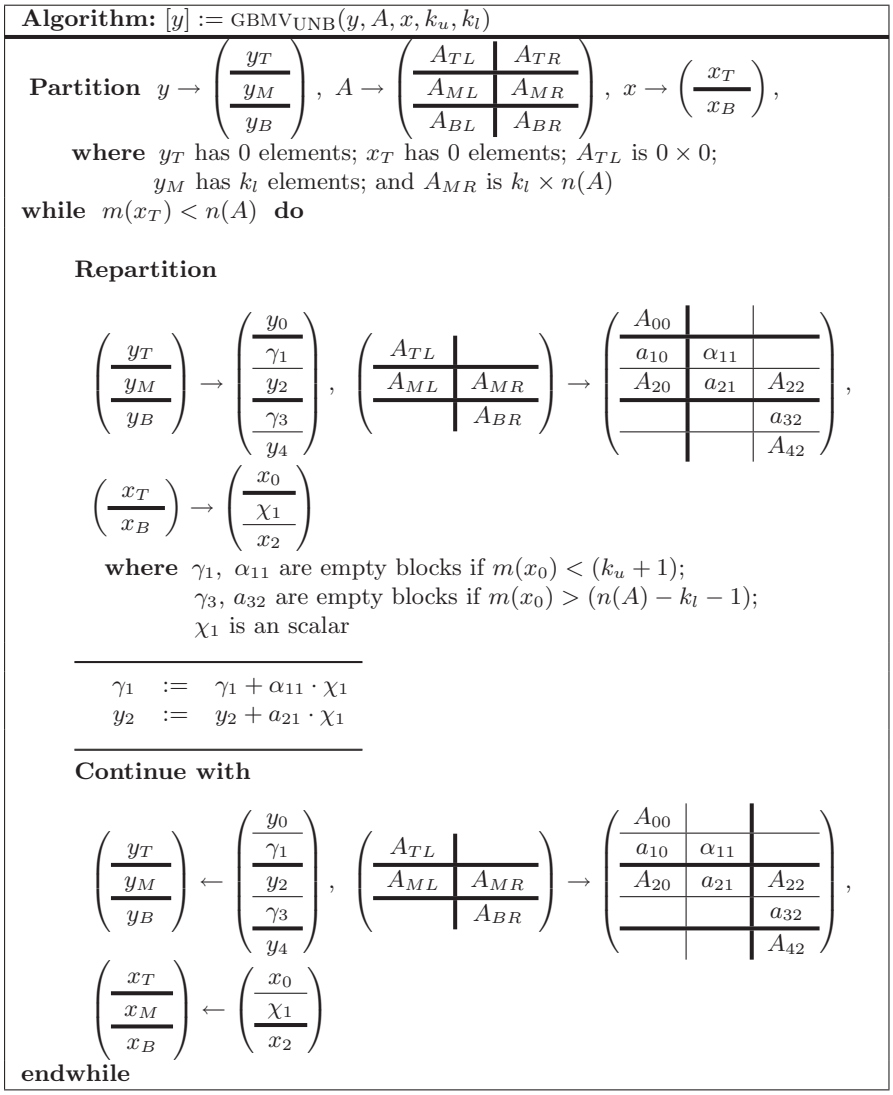


Figura 2.14: Algoritmo GBMV_{UNB} para la operación $y := y + A \cdot x$.

Durante una iteración del algoritmo se realizan las siguientes operaciones:

$$\gamma_1 := \gamma_1 + \alpha_{11} \cdot \chi_1 + a_{12}^T \cdot x_2, \tag{2.24}$$

$$y_2 := y_2 + a_{21} \cdot \chi_1. \tag{2.25}$$

Para la ejecución de (2.24), GBMV_B1_AXPY_DOT invoca a la rutina DOT, perteneciente al *BLAS*-1 denso, mientras que para la operación (2.25) hace uso de la rutina AXPY, también del *BLAS*-1 denso.

2.2.4. Algoritmo GBMV_{BLK}

El algoritmo GBMV_{BLK} (figura 2.19) para la operación (2.23) es un algoritmo por bloques que, a diferencia del algoritmo GBMV_{UNB}, puede ser implementado utilizando rutinas del nivel 2 de *BLAS* denso.

En cada iteración de GBMV_{BLK}, se opera con tres bloques de A:

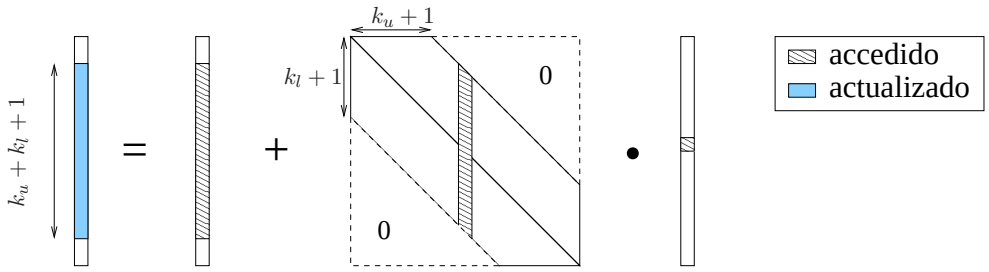


Figura 2.15: Acceso a los elementos durante una iteración del algoritmo GBMV$_{\text{UNB}}$.

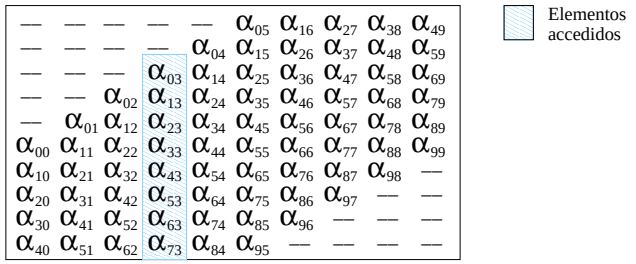


Figura 2.16: Acceso a los elementos durante la cuarta iteración (del algoritmo GBMV$_{\text{UNB}}$ y) de la rutina GBMV_REF considerando el esquema compacto de almacenamiento para matrices generales banda.

- El bloque A_{11} es triangular inferior y, como se muestra en la figura 2.19, su tamaño está en función de la iteración del algoritmo. Durante las iteraciones iniciales, aquellas que trabajan con las primeras k_u columnas de A, el bloque A_{11} está vacío, mientras que el resto de iteraciones es un bloque de dimensión $b \times b$.

- El bloque A_{21} es rectangular denso con b columnas y un número de filas que varía en función de la iteración del algoritmo. En cada iteración este bloque está formado por la mayor región rectangular incluida dentro de la banda en las b columnas activas de A.

- El bloque A_{31} es triangular superior de talla b durante las primeras iteraciones, y un bloque vacío las últimas $\frac{k_l}{b}$ iteraciones, las destinadas a operar con las últimas k_l columnas de A.

  La figura 2.20 muestra el particionado empleado por este algoritmo tanto para la matriz A como para los vectores x e y.

  Las operaciones realizadas en cada iteración del algoritmo son las siguientes:

$$y_1 := y_1 + A_{11} \cdot x_1, \tag{2.26}$$
$$y_2 := y_2 + A_{21} \cdot x_1, \tag{2.27}$$
$$y_3 := y_3 + A_{31} \cdot x_1. \tag{2.28}$$

  En particular, las operaciones (2.26) y (2.28) representan un producto entre una matriz triangular y un vector, en tanto que (2.27) es un producto entre una matriz general densa y un vector. El nivel 2 de *BLAS* denso incluye rutinas para ejecutar ambos tipos de producto.

Algorithm: $[y] := \text{GBMV}_{\text{UNB\_DOT}}(y, A, x, k_u, k_l)$

Partition $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$, $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$

where y_T and x_T have 0 elements; A_{TL} is 0×0

while $m(y_T) < m(y)$ **do**

Repartition

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \gamma_1 \\ \hline y_2 \\ \hline y_3 \end{array}\right), \quad \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c} A_{00} & a_{01} & A_{02} & \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T & \\ \hline A_{20} & a_{21} & A_{22} & A_{23} \\ \hline & & A_{32} & A_{33} \end{array}\right),$$

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \\ \hline x_3 \end{array}\right)$$

where $\gamma_1, \alpha_{11}, \chi_1$ are scalars;
y_2, a_{21} have $\text{mín}(k_l, m(A) - (m(x_0) + 1))$ elements;
x_2, a_{12}^T have $\text{mín}(k_u, n(A) - (m(x_0) + 1))$ elements

$$\begin{array}{lcl} \gamma_1 & := & \gamma_1 + \alpha_{11} \cdot \chi_1 + a_{12}^T \cdot x_2 \\ y_2 & := & y_2 + a_{21} \cdot \chi_1 \end{array}$$

Continue with

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \gamma_1 \\ \hline y_2 \\ \hline y_3 \end{array}\right), \quad \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c} A_{00} & a_{01} & A_{02} & \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T & \\ \hline A_{20} & a_{21} & A_{22} & A_{23} \\ \hline & & A_{32} & A_{33} \end{array}\right),$$

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \\ \hline x_3 \end{array}\right)$$

endwhile

Figura 2.17: Algoritmo $\text{GBMV}_{\text{UNB\_DOT}}$ para la operación $y := y + A \cdot x$.

2.2.5. Implementaciones basadas en rutinas de BLAS-2 denso

La rutina GBMV_B2 implementa el algoritmo GBMV_{BLK} haciendo uso de rutinas *BLAS*-2, de forma que la mayor parte de las operaciones aritméticas son ejecutadas por estos núcleos optimizados de *BLAS*.

La rutina de *BLAS* GEMV implementa el producto entre la matriz general densa y un vector, necesario en (2.27), mientras que TRMV calcula el producto matriz vector cuando la matriz presenta una estructura tringular, operación presente en (2.26) y (2.28). La invocación de esta última rutina no es inmediata, ya que su interfaz no se corresponde con las necesidades del algoritmo GBMV_{BLK}. La rutina TRMV almacena el resultado del producto en el vector operando, mientras que tanto en (2.26) como en (2.28) los vectores resultado y operando difieren. Para solventar este inconveniente, se realiza una copia del vector operando x_1 en un espacio de trabajo, w, se utiliza este espacio de trabajo para invocar a TRMV, y finalmente se acumulan en el vector resultado los elementos de w.

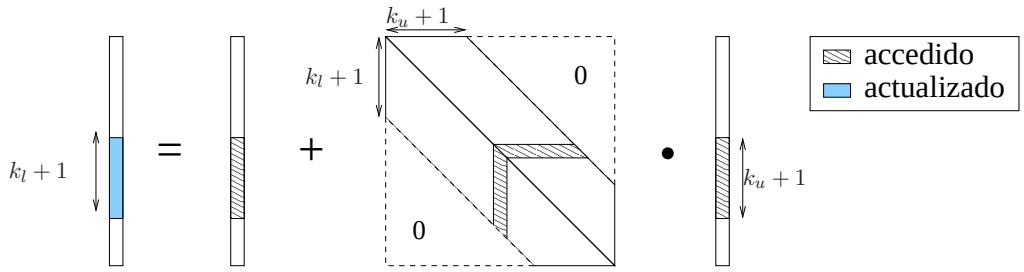


Figura 2.18: Acceso a los elementos durante una iteración de la rutina GBMV_B1_AXPY_DOT.

Algorithm: $[y] := \text{GBMV}_{\text{BLK}}(y, A, x, k_u, k_l)$

Partition $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_M \\ \hline y_B \end{array}\right)$, $A \rightarrow \left(\begin{array}{c|c} A_{TL} & \\ \hline A_{ML} & A_{MR} \\ \hline & A_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$

where y_T, x_T have 0 elements; y_M has k_l elements; A_{TL} is 0×0 and A_{MR} is $k_l \times n(A)$

while $m(y_T) < m(y)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c} y_T \\ \hline y_M \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline y_3 \\ \hline y_4 \end{array}\right), \quad \left(\begin{array}{c|c} A_{TL} & \\ \hline A_{ML} & A_{MR} \\ \hline & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & & \\ \hline A_{10} & A_{11} & \\ \hline A_{20} & A_{21} & A_{22} \\ \hline & A_{31} & A_{32} \\ \hline & & A_{42} \end{array}\right), \quad \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \end{array}\right)$$

where y_1 has 0 elements if $m(x_0) < (k_u + 1)$ and has b elements otherwise;
y_3 has 0 elements if $m(x_0) > (n(A) - k_l - 1)$ and has b elements otherwise;
A_{11} is empty if $m(x_0) < (k_u + 1)$ and is $b \times b$ otherwise;
A_{31} is empty if $m(x_0) > (n(A) - k_l - 1)$ and is $b \times b$ otherwise;
x_1 has b elements

$$\begin{array}{lcl} y_1 & := & y_1 + A_{11} \cdot x_1 \\ y_2 & := & y_2 + A_{21} \cdot x_1 \\ y_3 & := & y_3 + A_{31} \cdot x_1 \end{array}$$

Continue with

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \\ \hline y_3 \\ \hline y_4 \end{array}\right), \quad \left(\begin{array}{c|c} A_{TL} & \\ \hline A_{ML} & A_{MR} \\ \hline & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & & \\ \hline A_{10} & A_{11} & \\ \hline A_{20} & A_{21} & A_{22} \\ \hline & A_{31} & A_{32} \\ \hline & & A_{42} \end{array}\right), \quad \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \end{array}\right)$$

endwhile

Figura 2.19: Algoritmo por bloques GBMV_{BLK} para la operación $y := y + A \cdot x$.

La secuencia de operaciones ejecutadas por GBMV_B2 es pues la siguiente:

$$\begin{array}{lrll} & y_1 & := y_1 + A_{11} \cdot x_1, & (2.29) \\ (\text{COPY}) & w & := x_1, & (2.30) \\ (\text{TRMV}) & w & := w + A_{11} \cdot w, & (2.31) \\ (\text{AXPY}) & y_1 & := y_1 + w, & (2.32) \\ (\text{GEMV}) & y_2 & := y_2 + A_{21} \cdot x_1, & (2.33) \\ & y_3 & := y_3 + A_{31} \cdot x_1, & (2.34) \\ (\text{COPY}) & w & := x_1, & (2.35) \\ (\text{TRMV}) & w & := w + A_{31} \cdot w, & (2.36) \\ (\text{AXPY}) & y_3 & := y_3 + w. & (2.37) \end{array}$$

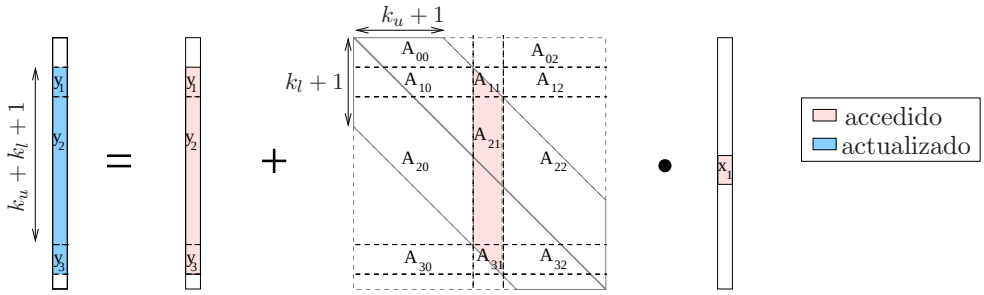


Figura 2.20: Acceso a los elementos durante una iteración de la rutina GBMV_B2.

La rutina GBMV_B2 realiza un total de siete invocaciones a rutinas de *BLAS* denso, cuatro de ellas a rutinas del nivel 1 de *BLAS*. El elevado número de invocaciones a rutinas y en especial a rutinas del primer nivel de *BLAS* denso, propicia la búsqueda de nuevas alternativas a GBMV_B2.

Implementación (GBMV_B2_TRMV_INLINE)

Esta nueva rutina es una modificación de GBMV_B2 en la que se trata de eliminar las invocaciones a rutinas *BLAS* que conllevan un bajo coste computacional: en concreto, las operaciones (2.35), (2.36) y (2.37) que realizan tres invocaciones a rutinas *BLAS*, dos de ellas pertenecientes a su primer nivel. Estos tres cálculos operan con bloques de tamaño reducido.

El código de GBMV_B2_TRMV_INLINE realiza el producto en (2.34) con dos bucles anidados, evitando con ello las tres llamadas a rutinas *BLAS* y las operaciones innecesarias que se realizan para adaptar el algoritmo a la interfaz de TRMV.

$$y_1 := y_1 + A_{11} \cdot x_1, \tag{2.38}$$

$$(\text{COPY}) \quad w := x_1, \tag{2.39}$$

$$(\text{TRMV}) \quad w := w + A_{11} \cdot w, \tag{2.40}$$

$$(\text{AXPY}) \quad y_1 := y_1 + w, \tag{2.41}$$

$$(\text{GEMV}) \quad y_2 := y_2 + A_{21} \cdot x_1, \tag{2.42}$$

$$y_3 := y_3 + A_{31} \cdot x_1. \tag{2.43}$$

Implementación (GBMV_B2_MERGE)

La rutina GBMV_B2_MERGE reduce el número de invocaciones a subrutinas *BLAS* efectuadas por GBMV_B2. Para ello, une los bloques A_{21} y A_{31} en un único macrobloque para, con una sóla invocación a GEMV, actualizar los vectores y_2 e y_3.

En esta rutina, se rellena con ceros la sección de memoria en la que estaría dispuesta la parte triangular inferior estricta de A_{31} siguiendo el esquema de almacenamiento para matrices generales, de forma que este bloque esté almacenado completamente y pueda ser tratado como una matriz general densa. Anteriormente se debe guardar una copia de esta sección de memoria en un espacio de trabajo, para después poder recuperar su valor original. La secuencia de operaciones necesarias es la siguiente:

$$y_1 := y_1 + A_{11} \cdot x_1, \tag{2.44}$$

(COPY) $$w := x_1, \tag{2.45}$$

(TRMV) $$w := w + A_{11} \cdot w, \tag{2.46}$$

(AXPY) $$y_1 := y_1 + w, \tag{2.47}$$

$$\begin{bmatrix} y_2 \\ y_3 \end{bmatrix} := \begin{bmatrix} y_2 \\ y_3 \end{bmatrix} + \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot x_1, \tag{2.48}$$

$$W := \text{STRIL}(A_{31}), \tag{2.49}$$

$$\text{STRIL}(A_{31}) := 0, \tag{2.50}$$

(GEMV) $$\begin{bmatrix} y_2 \\ y_3 \end{bmatrix} := \begin{bmatrix} y_2 \\ y_3 \end{bmatrix} + \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot x_1, \tag{2.51}$$

$$\text{STRIL}(A_{31}) := W. \tag{2.52}$$

La copia de STRIL(A_{31}) y su puesta a ceros se realizan simultáneamente mediante dos bucles anidados, y esta misma codificación es empleada para la restauración de los valores de STRIL(A_{31}) en (2.52). Ejecutando esta secuencia de operaciones, GBMV_B2_MERGE invoca únicamente a cuatro rutinas *BLAS*. A cambio, se realizan dos copias de bloques de tamaño reducido y se rellena con ceros una región de memoria.

2.2.6. Resultados experimentales

Para reducir el número de resultados, en todos los experimentos con matrices generales banda que siguen se establece $k_u = k_l$.

Arquitectura ITANIUM

Las prestaciones alcanzadas por las diferentes rutinas incluidas en las implementaciones de *BLAS* en estudio se presentan en la figura 2.21. La rutina de *BLAS de referencia* ha sido compilada con *ifort* y *g77*. Como se puede observar, el código generado con *ifort* es más eficiente que el generado con *g77*. En adelante, los experimentos se ejecutarán pues con este compilador.

Las implementación más eficiente cuando A es una matriz de banda estrecha es GBMV_MKL, mientras que cuando A es una matriz de banda media o ancha ($k_u = k_l > 300$) GBMV_GOTO es la rutina que alcanza mayores prestaciones.

La figura 2.22 muestra la eficiencia de las implementaciones basadas en rutinas *BLAS*-1 presentadas en esta sección, tanto para *MKL* como para *GotoBLAS*. En el caso de la biblioteca *MKL*, la rutina GBMV_MKL incluida en la biblioteca es en general la más rápida. Respecto a las implementaciones basadas en *GotoBLAS*, las rutinas GBMV_GOTO y GBMV_B1+*GotoBLAS* obtienen prestaciones similares, siendo ambas opciones igualmente eficientes.

Las prestaciones de las nuevas rutinas basadas en operaciones *BLAS*-2 se muestran en la figura 2.23. La gráfica de la izquierda compara las prestaciones de las nuevas implementaciones cuando éstas invocan a rutinas de *MKL*. Como se puede comprobar, GBMV_MKL es más eficiente que las nuevas rutinas cuando se opera con una matriz A con ancho de banda (inferior y superior) menor que 350, mientras que para anchos de banda mayores, GBMV_B2 es la opción más eficiente. Si la implementación de *BLAS* empleada es *GotoBLAS*, la rutina GBMV_GOTO es claramente la más rápida al operar con matrices de banda estrecha, pero al igual que sucede en el caso de *MKL*, para matrices de banda media o ancha las implementaciones *BLAS*-2 consiguen mayores prestaciones, especialmente la implementación GBMV_B2+*GotoBLAS*.

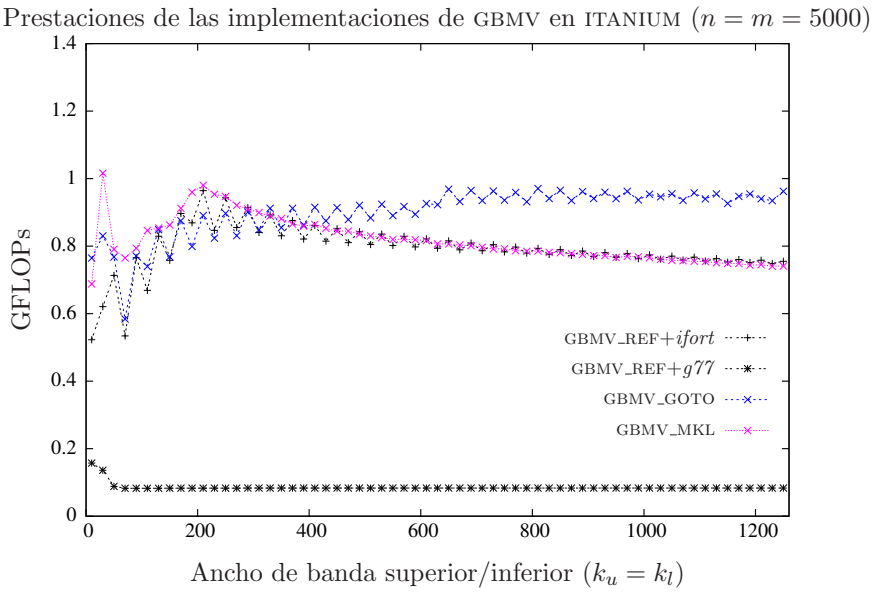


Figura 2.21: Comparativa de las diferentes implementaciones de bibliotecas *BLAS*.

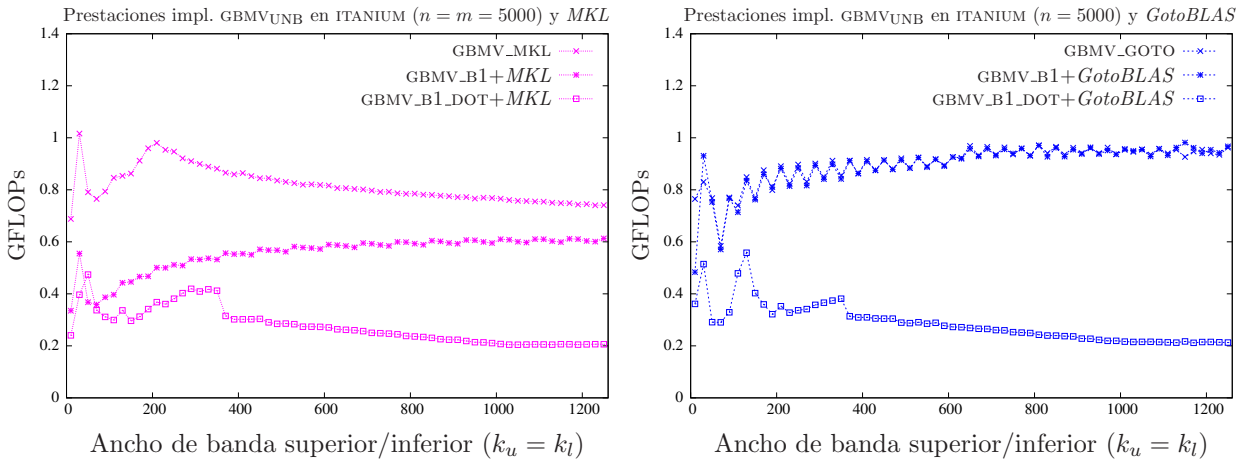


Figura 2.22: Comparativa de las diferentes implementaciones basadas en *BLAS*-1 denso.

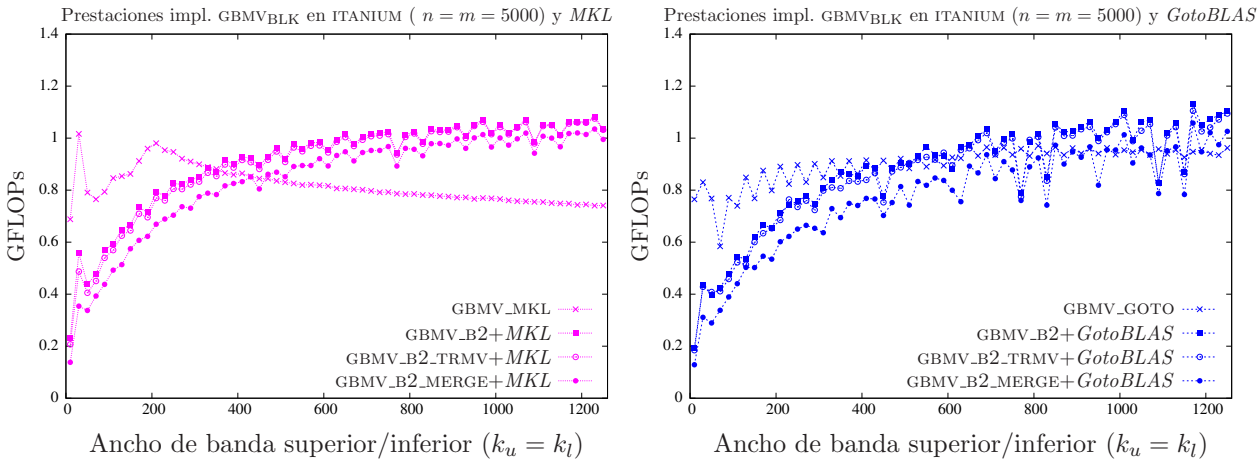


Figura 2.23: Comparativa de las diferentes implementaciones basadas en *BLAS*-2 denso.

La figura 2.24 incluye los resultados obtenidos por las rutinas GBMV. Como muestra la figura, cuando el ancho de banda superior e inferior de la matriz A es menor que 350, las rutinas GBMV_GOTO y GBMV_MKL generan las mejores prestaciones, mientras que para anchos de banda superiores GBMV_B2+*MKL* se revela como la más eficiente de las rutinas.

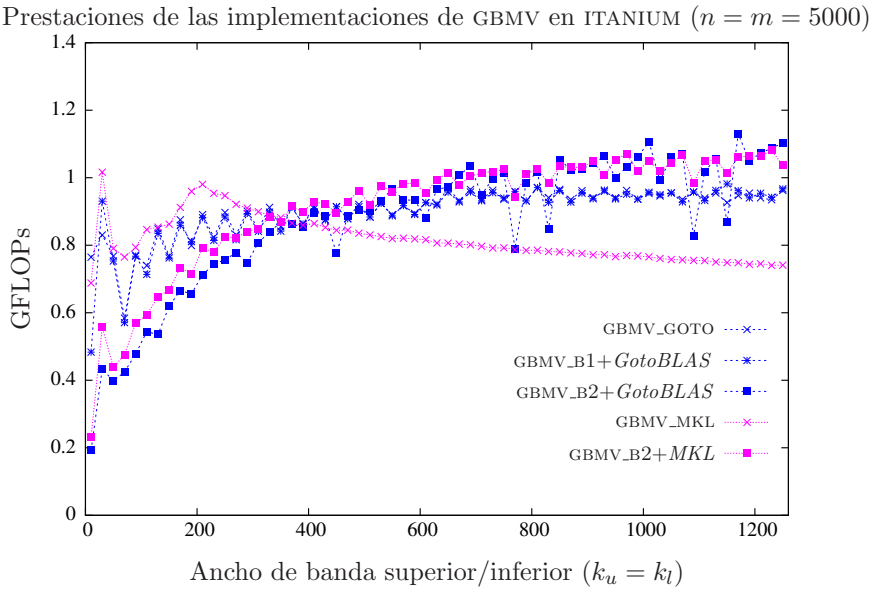


Figura 2.24: Comparativa de las mejores implementaciones.

Arquitectura XEON

En la figura 2.25 se muestran las prestaciones de las rutinas incluidas en *BLAS de referencia*, *GotoBLAS* y *MKL*. La gráfica recopila los resultados de la rutina GBMV_REF utilizando los compiladores *ifort* y *g77*. En esta arquitectura el código obtenido por ambos compiladores presentan idénticas prestaciones. Como se puede observar, la rutina de *GotoBLAS* es muy eficiente cuando

el ancho de banda de la matriz A es pequeño, mientras que la rutina de *MKL* se presenta como la mejor opción cuando el la matriz A es una matriz de banda media o ancha.

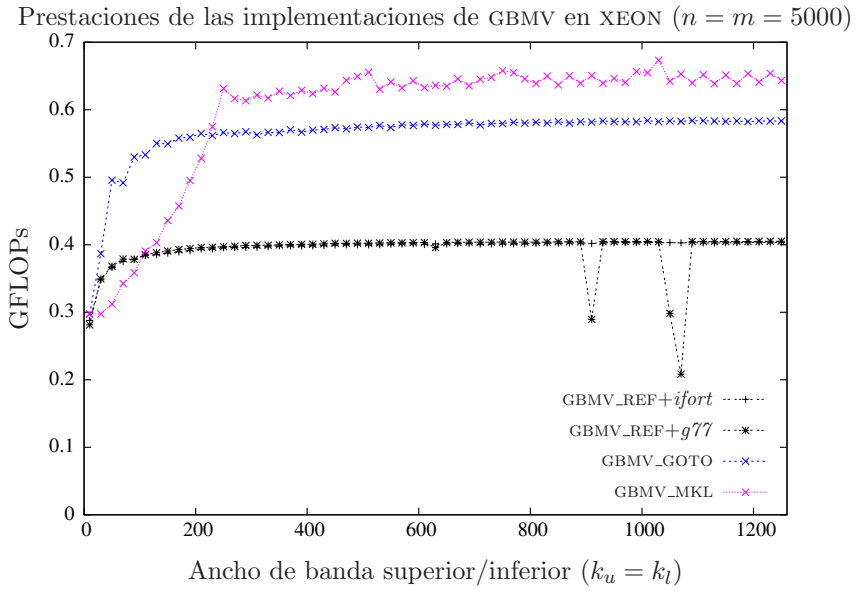


Figura 2.25: Comparativa de las diferentes implementaciones de bibliotecas *BLAS*.

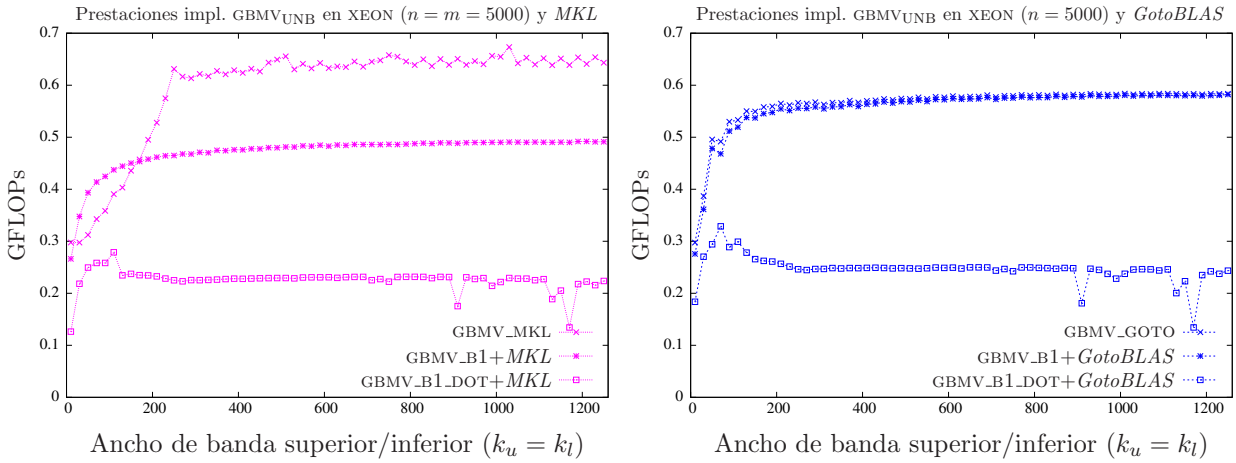


Figura 2.26: Comparativa de las diferentes implementaciones basadas en *BLAS*-1 denso.

La figura 2.26 compara las prestaciones de las nuevas rutinas GBMV basadas en núcleos computacionales *BLAS*-1 con las rutinas incluidas en las bibliotecas *MKL* y *GotoBLAS*. Al emplear la biblioteca *MKL*, la nueva rutina GBMV_B1 mejora las prestaciones de la rutina GBMV_MKL cuando los anchos de banda superior e inferior de la matriz A son inferiores a 200, mientras que para anchos de banda superiores GBMV_MKL es la rutina que obtiene las mejores prestaciones. En el caso de *GotoBLAS*, las rutinas GBMV_B1 y GBMV_GOTO son las más eficientes y obtienen similares prestaciones. La rutina GBMV_B1_DOT presenta bajas prestaciones, ya que parte de los accesos sobre la matriz A se realizan por filas.

Respecto a las implementaciones del algoritmo GBMV$_{BLK}$ y *MKL*, todas las nuevas rutinas obtienen resultados similares a los de GBMV_MKL. En el caso de *GotoBLAS*, la rutina GBMV_GOTO

es la más eficiente cuando los anchos de banda superior e inferior de la matriz A son menores que 250, mientras que para anchos de banda superiores las rutinas GBMV_B2_TRMV y GBMV_B2_MERGE obtienen las mejores prestaciones.

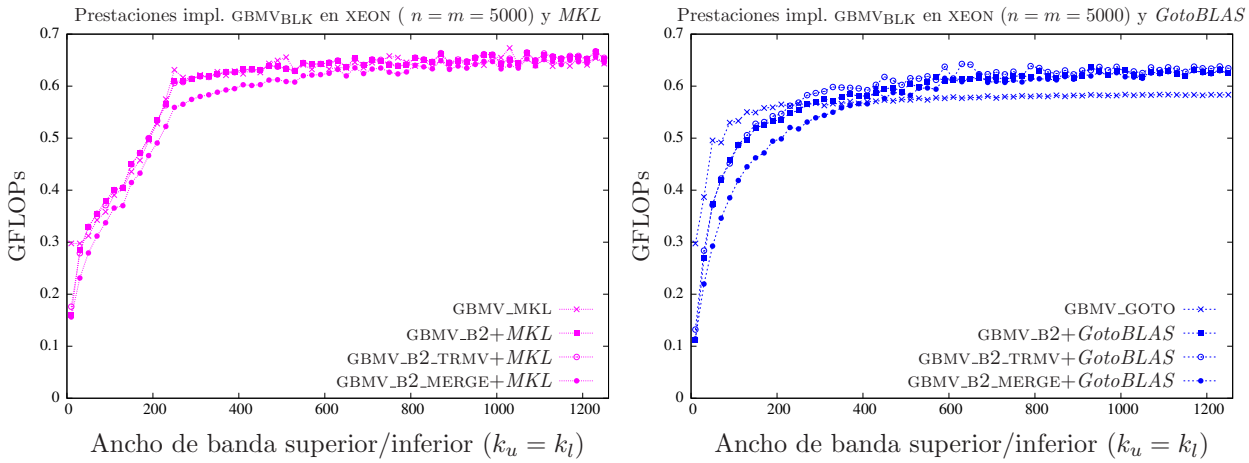


Figura 2.27: Comparativa de las diferentes implementaciones basadas en *BLAS*-2 denso.

La figura 2.28 reune las prestaciones de las mejores rutinas para la arquitectura XEON. La conveniencia de utilizar una rutina depende del ancho de banda de la matriz A. Así, para anchos de banda (tanto superior como inferior) menor que 450, las rutinas GBMV_GOTO y GBMV_B1+*GotoBLAS* son las de mayor eficiencia, mientras que para anchos de banda mayores, GBMV_MKL y GBMV_B2_TRMV +*MKL* son las implementaciones más rápidas. Las nuevas rutinas logran igualar y en ocasiones superar los resultados obtenidos por las rutinas optimizadas de las implementaciones *BLAS* estudiadas.

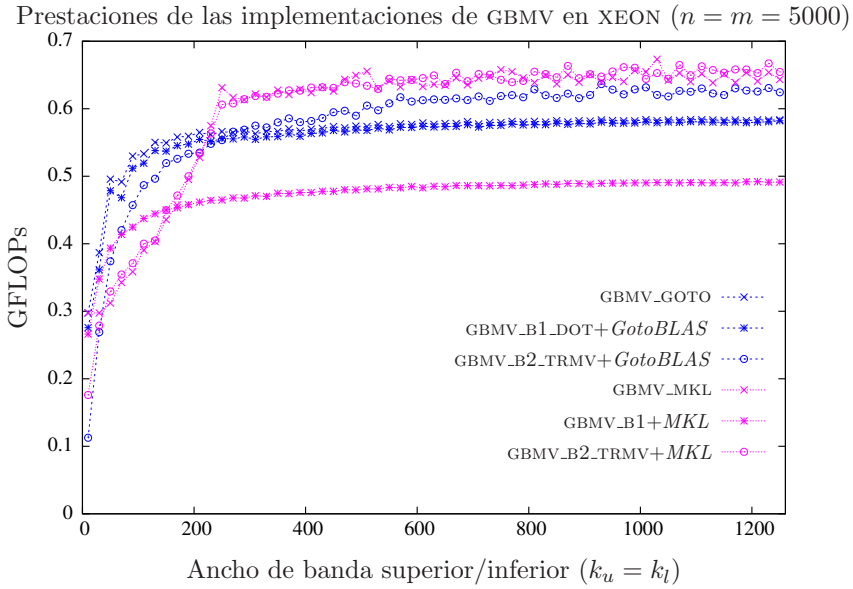


Figura 2.28: Comparativa de las mejores implementaciones.

2.2.7. Conclusiones

Los resultados de los experimentos realizados nos ofrecen una visión detallada de las posibles mejoras a introducir en las rutinas que implementan la operación (2.23).

Como era de esperar, las rutinas basadas en invocaciones a núcleos computacionales del nivel 2 de *BLAS* obtienen las mejores prestaciones cuando la matriz A presenta una banda media o ancha. Únicamente en esta situación el coste computacional requerido en cada iteración del algoritmo GBMV$_{\text{BLK}}$ justifica la utilización de rutinas del nivel 2 de *BLAS*.

Cuando la matriz A es de banda estrecha, la nueva rutina GBMV_B1 obtiene buenas prestaciones, igualando las obtenidas por las rutinas incluidas en las implementaciones de *BLAS* estudiadas.

Las prestaciones obtenidas por la rutina GBMV_B1_DOT son muy inferiores a las alcanzadas por GBMV_B1, ya que realiza parte de sus accesos a los elementos de la matriz A por filas.

Las prestaciones de la rutina GBMV_B2_MERGE son inferiores a las obtenidas por el resto de implementaciones basadas en *BLAS*-2, de entre las cuales destaca GBMV_B2_TRMV, al conseguir reducir ligeramente el sobrecoste introducido por GBMV_B2.

En la arquitectura ITANIUM, el compilador *ifort* genera los códigos más eficientes, mientras que en la arquitectura XEON los códigos generados por ambos compiladores ofrecen idénticas prestaciones.

2.3. Producto de una matriz triangular banda por un vector

En esta sección se evalúa el caso particular del producto matriz por vector

$$x := \alpha \cdot A \cdot x, \tag{2.53}$$

donde $\alpha \in \mathbb{R}$, el vector $x \in \mathbb{R}^n$, y $A \in \mathbb{R}^{n \times n}$ es una matriz triangular (inferior o superior), con estructura banda y ancho de banda k_l, que alternativamente puede aparecer en la expresión anterior como transpuesta. Es importante destacar que, a diferencia de las operaciones comentadas en otras secciones de este capítulo, el vector multiplicando y el vector resultado coinciden.

Por simplicidad, en lo sucesivo consideraremos unicamente el caso en que A es una matriz triangular inferior que aparece sin transponer en (2.53), con ancho de banda k_l, y asimismo tomaremos $\alpha = 1$. Pese a todo, el estudio que sigue puede extenderse directamente al caso en que A es una matriz triangular superior.

2.3.1. Algoritmo TBMV$_{\text{UNB}}$

La figura 2.29 muestra un algoritmo que calcula la operación (2.53). Este algoritmo, al que denominaremos TBMV$_{\text{UNB}}$, recorre la matriz A de derecha a izquierda. Durante la iteración j del mismo, se opera con la columna k-ésima de A, donde $k = n - j + 1$. El resultado de estas operaciones es la obtención del k-ésimo elemento de x y la actualización de los siguientes k_l elementos de este vector.

La matriz A es recorrida de derecha a izquierda con el fin de eliminar los problemas de dependencia de datos entre las operaciones realizadas en sucesivas iteraciones. Además, para cada iteración, se actualiza x_2 antes que χ_1, y de esta manera se resuelve también la dependencia de datos entre estas dos operaciones.

El acceso a los elementos realizado en cada iteración se muestra en la figura 2.30. Teniendo en cuenta cómo se almacenan físicamente las matrices banda triangulares en memoria de forma compacta (ver sección 1.2), en la figura se observa claramente una importante propiedad del algoritmo: el acceso a los elementos se realiza por columnas. Como se ha explicado en secciones anteriores, el

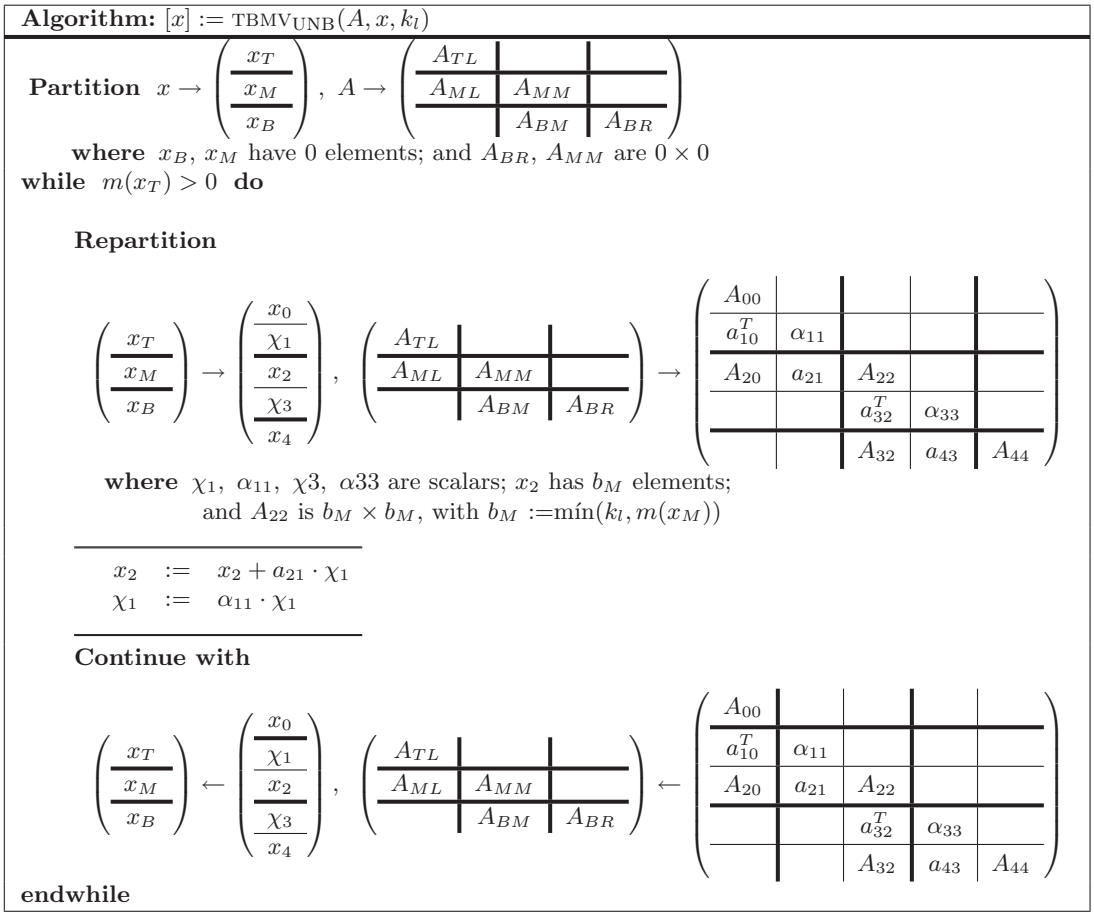


Figura 2.29: Algoritmo TBMV_{UNB} para la operación $x := A \cdot x$.

acceso por columnas es muy eficiente, ya que en Fortran los elementos de una columna se encuentran almacenados de manera consecutiva en memoria.

2.3.2. Implementación del BLAS de referencia

El *BLAS de referencia* implementa la operación (2.53) mediante la rutina TBMV_REF. El algoritmo codificado en esta rutina es precisamente el mostrado en la figura 2.29. En cada iteración del algoritmo, un bucle recorre la correspondiente columna de A, actualizando χ_1 y cada uno de los elementos de x_2.

La implementación del *BLAS de referencia* presenta tres características favorables para su ejecución en arquitecturas con un sistema de memoria jerarquizado:

- Número de operaciones reducido, ya que no opera con los elementos nulos de A que se encuentran fuera de la banda.
- Acceso a memoria por columnas.
- Número de accesos reducido, ya que los elementos de A son recuperados una sóla vez, si bien no sucede lo mismo con los elementos de x.

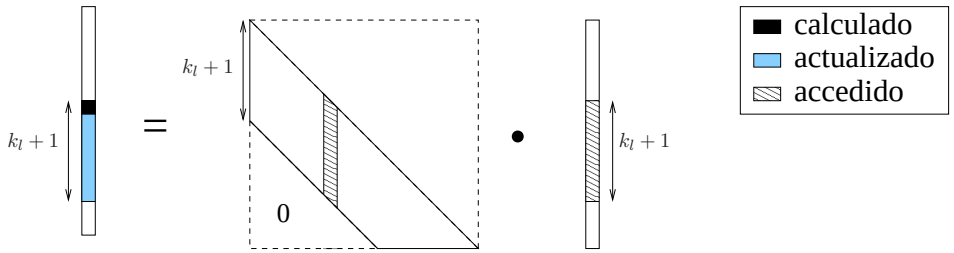


Figura 2.30: Acceso a los elementos durante una iteración del algoritmo TBMV$_{\text{UNB}}$.

Por contra podemos citar como una deficiencia el hecho de no utilizar otros núcleos de *BLAS* optimizados para realizar las operaciones aritméticas.

A continuación se proponen dos nuevas rutinas que implementan la operación en estudio basándose en el uso de núcleos de computación optimizados de *BLAS*.

2.3.3. Implementación basada en rutinas de BLAS-1 denso

La rutina TBMV_B1 para la operación (2.53) computa la mayoría de las operaciones aritméticas mediante llamadas a rutinas del nivel 1 de *BLAS* denso. Concretamente, implementa el algoritmo TBMV$_{\text{UNB}}$ realizando una invocación a la rutina AXPY para la actualización de x_2.

Esta rutina comparte las ventajas de TBMV_REF; es decir, forma de acceso optimizada a los elementos de A y número de operaciones mínimo. A estas ventajas, hay que añadir que prácticamente todas las operaciones aritméticas son realizadas por la rutina optimizada de *BLAS* denso AXPY.

No obstante, dos características, una de la propia operación y otra del algoritmo propuesto, llevan a la búsqueda de nuevas rutinas y algoritmos más eficientes para la operación (2.53):

- La operación en estudio pertenece al nivel 2 de *BLAS* y, en consecuencia, una implementación basada en este nivel podría ser más eficiente. Para ello será preciso un nuevo algoritmo, ya que las propiedades de TBMV$_{\text{UNB}}$ no permiten el uso de rutinas de *BLAS*-2 denso.
- Los elementos del vector x son accedidos repetidamente en el algoritmo.

Esta búsqueda se traduce en el algoritmo y las implementaciones descritas a continuación.

2.3.4. Algoritmo TBMV$_{\text{BLK}}$

El algoritmo TBMV$_{\text{BLK}}$, mostrado en la figura 2.31, aplica sobre la matriz A un particionado que permite su implementación mediante rutinas del nivel 2 de *BLAS*, recorriéndose la matriz de izquierda a derecha.

En cada iteración se opera con b columnas de A. Los elementos no nulos de A dentro de estas columnas recaen en los bloques A_{11}, A_{21} y A_{31}. Como se ilustra en la figura 2.32, A_{11} y A_{31} son bloques triangulares (inferior y superior respectivamente), mientras que A_{21} es un bloque rectangular denso.

En este algoritmo se accede a los elementos de A en una sóla ocasión y se realiza este acceso por columnas, al igual que ocurría en el algoritmo TBMV$_{\text{UNB}}$. En cambio, cada elemento de x es accedido en $\frac{k_l+1}{b}$ iteraciones, frente a las $k_l + 1$ ocasiones en que era accedido en el algoritmo TBMV$_{\text{UNB}}$. Es precisamente esta situación la que se explota en nuestro caso para intentar obtener mejores prestaciones.

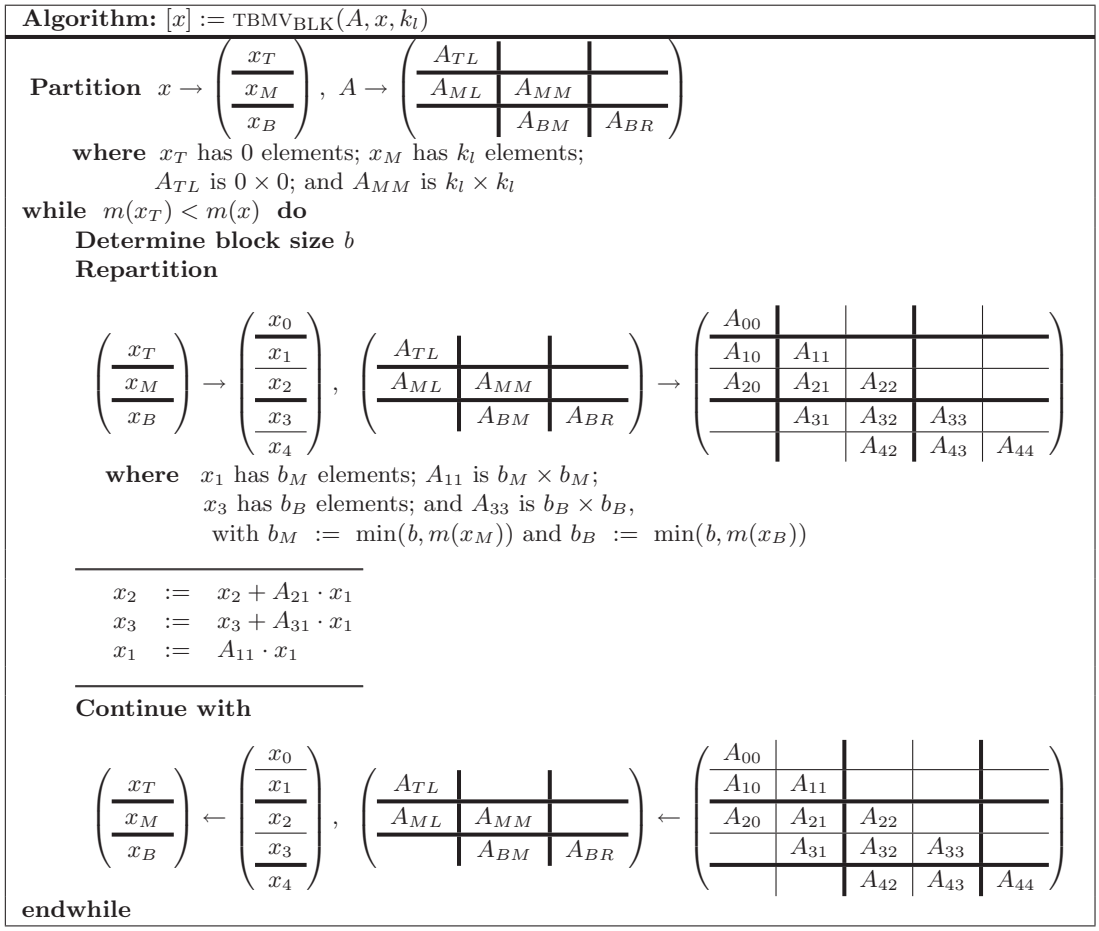


Figura 2.31: Algoritmo por bloques TBMV_{BLK} para la operación $x := A \cdot x$.

2.3.5. Implementaciones basadas en rutinas de BLAS-2 denso

La rutina TBMV_B2 implementa el algoritmo TBMV_{BLK} invocando a rutinas del nivel 2 de *BLAS* denso como se describe a continuación.

Si analizamos las operaciones realizadas en cada iteración del algoritmo TBMV_{BLK}:

$$x_2 := A_{21} \cdot x_1 + x_2, \tag{2.54}$$

$$x_3 := A_{31} \cdot x_1 + x_3, \tag{2.55}$$

$$x_1 := A_{11} \cdot x_1, \tag{2.56}$$

podemos comprobar que las operaciones (2.56) y (2.55) representan sendos productos entre una matriz triangular y un vector, mientras la operación (2.54) corresponde a un producto entre una matriz densa y un vector. Estas operaciones son las implementadas por las rutinas de *BLAS*-2 TRMV y GEMV respectivamente. Es decir, cada una de las operaciones puede ser ejecutada por una rutina del nivel 2 de *BLAS* denso.

Ahora bien, las tres operaciones presentan una dependencia de datos, puesto que todas ellas acceden al bloque x_1 y una de ellas modifica su contenido. Por este motivo, en un principio puede parece suficiente con ejecutar la operación de actualización del bloque x_1 (operación (2.56)) en último lugar.

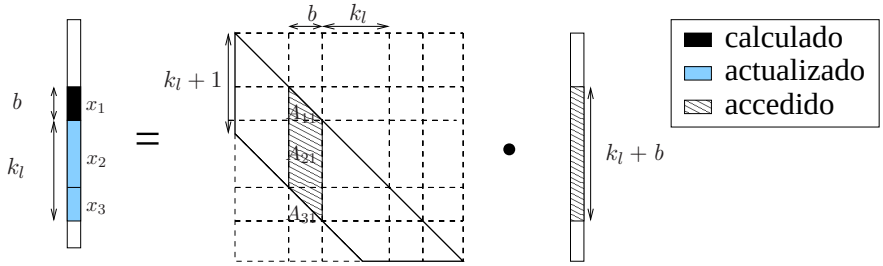


Figura 2.32: Acceso a los elementos durante una iteración del algoritmo TBMV_{BLK}.

Sin embargo, un análisis detallado muestra que existe una dependencia de datos entre las operaciones realizadas en dos iteraciones consecutivas del algoritmo. La primera iteración modifica los vectores x_2 y x_3, mientras que la iteración posterior precisa los contenidos originales de estos bloques (que formarán el bloque x_1 y parte del x_2 de la nueva iteración). Una posible solución consiste en, al inicio del algoritmo, replicar el vector x sobre un espacio de trabajo, w, para después realizar todas las operaciones de lectura sobre w y las de escritura sobre x.

Otro problema aparece en el uso de la rutina TRMV, pues el vector operando y resultado de la rutina TRMV de *BLAS* denso deben coincidir. Esta circunstancia se da en el caso de la actualización de x_1 (operación (2.56)), pero no en la actualización de x_3 (operación (2.55)). Para resolver esta dificultad se emplea un espacio de trabajo auxiliar, v. En este espacio se almacena una copia de x_1, con ella se invoca a la rutina TRMV, y finalmente se actualiza x_3 con el contenido de v.

Además, la operación (2.54), implementada mediante una invocación a GEMV, crea la necesidad de inicializar x_2 a ceros. Esto significa que, tras hacer la copia del vector x en el espacio v y antes de iniciar la primera iteración, se deberá rellenar con ceros el vector x. Esta inicialización genera, en el uso de la rutina TRMV para la operación (2.56), el mismo problema encontrado en (2.55). Aplicaremos para esta operación la misma solución, realizando una copia de x_1 en el mismo espacio de trabajo (v).

Así pues, el espacio de trabajo auxiliar necesario para la rutina TBMV_B2 (suma de las dimensiones de w y v) es de $n + b$ elementos, y en resumen, las operaciones a realizar antes de iniciar las iteraciones son:

$$(\text{COPY}) \qquad w := x, \tag{2.57}$$
$$x := 0. \tag{2.58}$$

en tanto que las operaciones a realizar en cada iteración son:

$$x_1 := A_{11} \cdot x_1 + x_1, \tag{2.59}$$
$$(\text{COPY}) \qquad v := w, \tag{2.60}$$
$$(\text{TRMV}) \qquad v := A_{11} \cdot v, \tag{2.61}$$
$$(\text{AXPY}) \qquad x_1 := x_1 + v, \tag{2.62}$$
$$(\text{GEMV}) \qquad x_2 := A_{21} \cdot w + x_2, \tag{2.63}$$
$$x_3 := A_{31} \cdot w + x_3, \tag{2.64}$$
$$(\text{COPY}) \qquad v := w, \tag{2.65}$$
$$(\text{TRMV}) \qquad v := A_{31} \cdot v, \tag{2.66}$$
$$(\text{AXPY}) \qquad x_3 := x_3 + v. \tag{2.67}$$

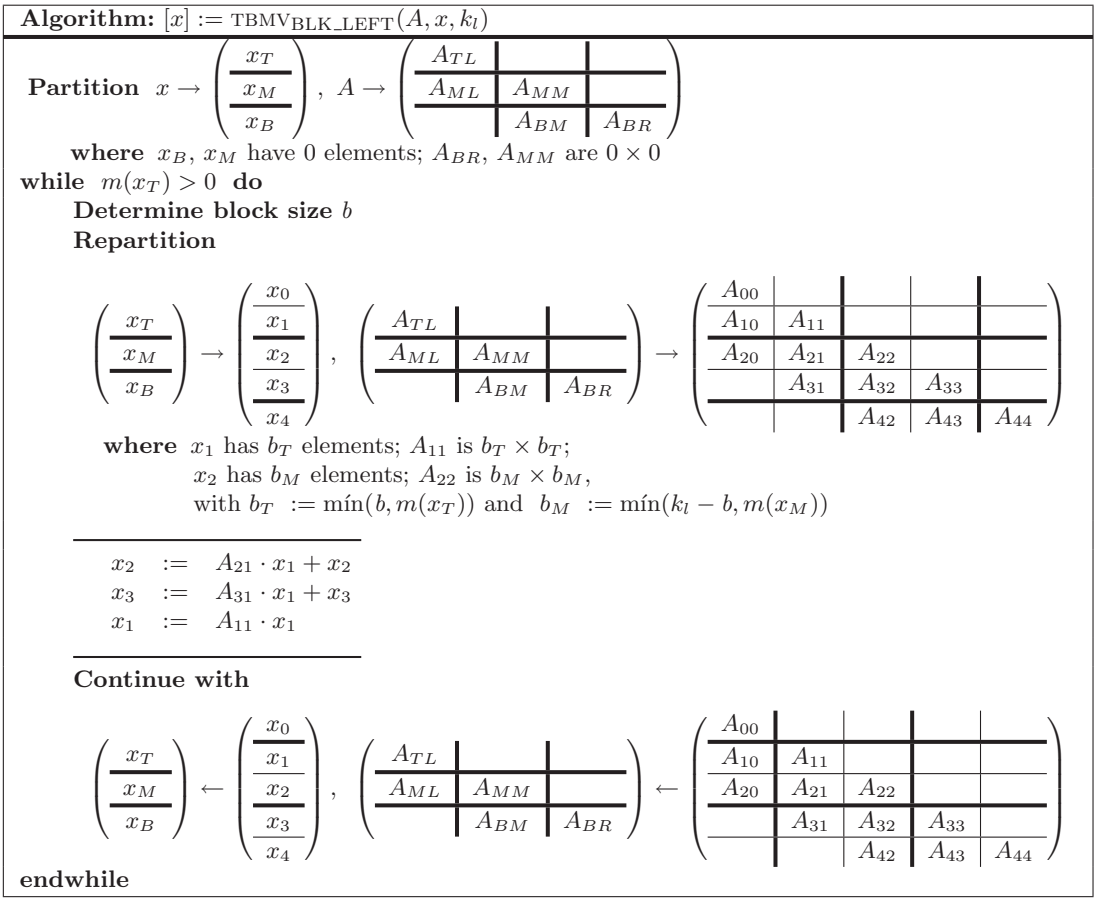


Figura 2.33: Algoritmo por bloques $\text{TBMV}_{\text{BLK\_LEFT}}$ para la operación $x := A \cdot x$.

Esta rutina, junto a las ventajas comentadas anteriormente respecto al acceso de A y número de operaciones, presenta como principales cualidades el que las operaciones aritméticas son ejecutadas por rutinas de *BLAS*-2 denso y los accesos a elementos del vector x se han reducido por un factor de b (por las mismas razones expuestas en el apartado 2.3.4).

Implementación sin espacio de trabajo (TBMV_B2_LEFT)

La rutina TBMV_B2 es insatisfactoria por la cantidad de espacio de trabajo auxiliar empleado y el número de copias requeridas. La rutina TBMV_B2_LEFT reduce sustancialmente ambas deficiencias al mismo tiempo que explota el uso de rutinas del nivel 2 de *BLAS* denso. Tanto el espacio de trabajo auxiliar como la mayoría de las copias realizadas en TBMV_B2 eran la solución aplicada a los problemas de dependencia de datos. La estrategia empleada por TBMV_B2_LEFT para solventar este problema es la reordenación de las operaciones.

La rutina TBMV_B2_LEFT está basada en el algoritmo $\text{TBMV}_{\text{BLK\_LEFT}}$ mostrado en la figura 2.33. En este algoritmo la matriz A es recorrida de derecha a izquierda. De esta forma, al igual que sucede en el algoritmo TBMV_{UNB}, se eliminan los problemas de dependencias de datos entre las operaciones efectuadas en distintas iteraciones. Además, para evitar problemas de dependencia de datos entre las operaciones de una iteración, se realiza la operación de escritura sobre el bloque x_1 en último lugar.

La secuencia de operaciones ejecutada en cada iteración es la siguiente:

$$
\begin{aligned}
&(\text{GEMV}) && x_2 && := A_{21} \cdot\ x_1 + x_2, && (2.68)\\
& && x_3 && := A_{31} \cdot\ x_1 + x_3, && (2.69)\\
&(\text{COPY}) && && v := x_1, && (2.70)\\
&(\text{TRMV}) && && v := A_{31} \cdot\ v, && (2.71)\\
&(\text{AXPY}) && && x_3 := x_3 + v, && (2.72)\\
&(\text{TRMV}) && x_1 && := A_{11} \cdot\ x_1. && (2.73)
\end{aligned}
$$

Si comparamos TBMV_B2 con TBMV_B2_LEFT, podemos observar que al recorrer esta última la matriz de derecha a izquierda, la inicialización a ceros es innecesaria ya que la primera operación que se ejecuta con cada elemento de x es siempre (2.73) (pese a que (2.73) es la última de las operaciones ejecutadas en cada iteración, también es la única operación que se realiza en la primera iteración), por lo que el contenido inicial de cada elemento de x es sobreescrito antes de que se ejecuten sobre él los productos (2.68) o (2.69). Además, dado que las lecturas de x_1 realizadas por las operaciones (2.68) y (2.69) se efectúan antes de que x_1 sea modificado en (2.73), la copia en el espacio de trabajo w es innecesaria.

Por otro lado, al ser la actualización de x_1 la primera operación sobre este bloque de todas las realizadas por el algoritmo, tampoco en esta ocasión se necesita una copia auxiliar.

La única copia necesaria en esta implementación aparece en la actualización de x_3 y se debe a que, en este producto, el vector operando (x_1) y resultado (x_3) son diferentes y, como se ha comentado con anterioridad, la interfaz de la rutina TRMV requiere que ambos vectores coincidan.

Tras lo explicado, podemos destacar cuatro aspectos positivos de TBMV_B2_LEFT:

- Efectúa un acceso optimizado a la matriz A.
- Presenta un acceso al vector x mejorado respecto a las implementaciones TBMV_B1 y del *BLAS de referencia*.
- Requiere un número cuasi óptimo de operaciones aritméticas.
- Utiliza rutinas del nivel 2 de *BLAS*.

Por contra, como características negativas, tenemos que:

- Requiere un espacio de trabajo si bien de tamaño reducido (b elementos).
- Realiza una copia y una operación extra (invocación a la rutina AXPY), ambas sobre un vector de b elementos.

2.3.6. Resultados experimentales

Arquitectura ITANIUM

La figura 2.34 muestra las prestaciones obtenidas mediante las rutinas incluidas en las bibliotecas *BLAS de referencia*, *GotoBLAS* y *MKL* para la operación (2.53). El código de la rutina del *BLAS de referencia* ha sido compilado con *ifort* y *g77*. Como se puede observar, no existe una implementación mejor que las demás, sino que la elección de la mejor rutina está en función del ancho de banda de la matriz A. Para matrices de banda estrecha ($k_l < 70$), *MKL* obtiene las mejores prestaciones; para matrices de banda media ($70 < k_l < 260$) es la rutina del *BLAS de referencia* (junto al compilador

ifort) la más eficiente; y para matrices de banda ancha la implementación de *GotoBLAS* supera al resto.

De este primer experimento, podemos concluir que:

- El compilador *ifort* genera un código más eficiente.
- La rutina de *MKL* está claramente optimizada para matrices de banda estrecha.
- Ninguna implementación es superior al resto, sino que la conveniencia de utilizar una u otra depende del ancho de banda de A.

Dado que *ifort* genera el código más eficiente para esta arquitectura, el resto de experimentos serán realizados con este compilador.

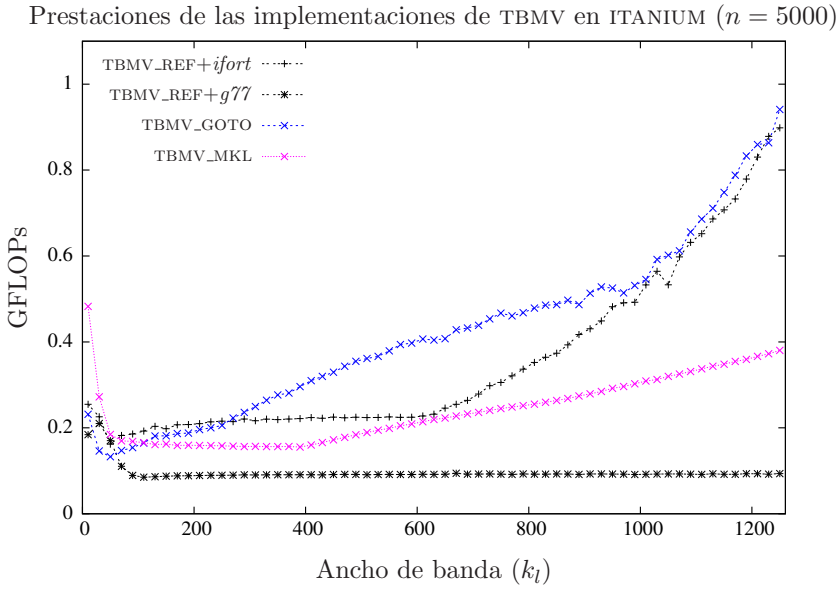


Figura 2.34: Comparativa de las diferentes implementaciones de bibliotecas *BLAS*.

La figura 2.35 compara las prestaciones de las rutinas propuestas en los apartados 2.3.3 y 2.3.5, basadas respectivamente en *BLAS*-1 y *BLAS*-2. Las gráficas de la derecha e izquierda muestran, respectivamente, los resultados obtenidos por estas implementaciones cuando invocan internamente a las rutinas de las bibliotecas *GotoBLAS* y *MKL* para ejecutar las operaciones aritméticas. En ambas gráficas se incluyen las prestaciones de la rutina de la correspondiente biblioteca que implementa la operación en estudio, TBMV.

Para esta operación y arquitectura, las implementaciones basadas en rutinas del nivel 2 de *BLAS* son las que obtienen mejores prestaciones. Pese a que TBMV_B2_LEFT reduce notablemente el número de copias y operaciones sobre vectores realizadas respecto a TBMV_B2, la diferencia entre las prestaciones de ambas rutinas es mínima. Esto quiere decir que el coste computacional requerido por estas copias es reducido. No obstante, TBMV_B2_LEFT, además de obtener unas prestaciones ligeramente superiores, precisa de un espacio de trabajo auxiliar mucho menor.

Tanto en el caso de *GotoBLAS* como en el de *MKL*, TBMV_B1 obtiene buenas prestaciones en relación a las obtenidas por la rutina TBMV de ambas versiones de *BLAS*; de hecho, salvo para matrices de banda muy estrecha en el caso de *MKL*, TBMV_B1 siempre mejora a TBMV. Para matrices con ancho de banda superior a 1200, las prestaciones de todas las rutinas propuestas son similares.

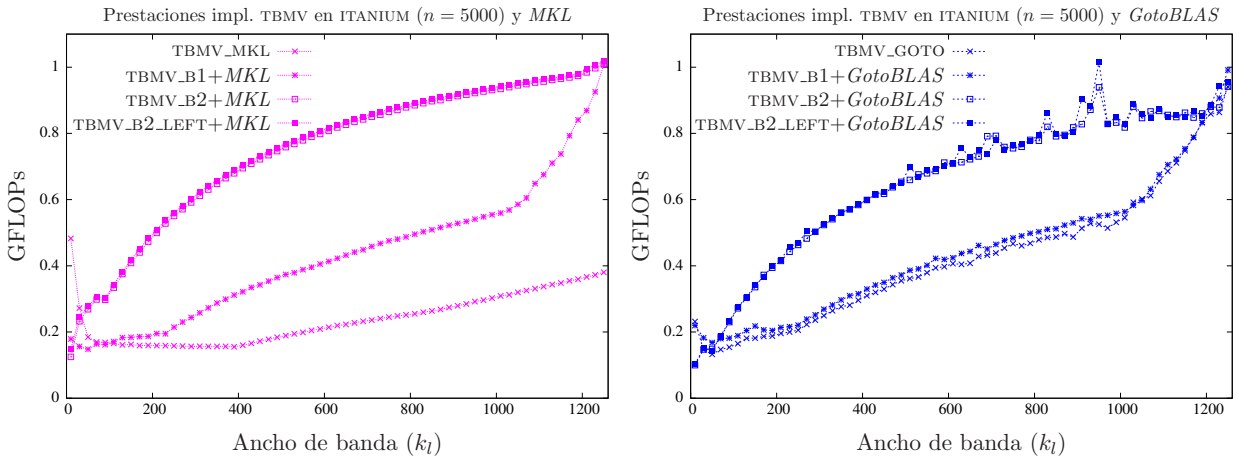


Figura 2.35: Comparativa de las diferentes implementaciones basadas en *BLAS*-1 y 2 denso.

A modo de resumen, la figura 2.36 recoge los resultados de la rutina TBMV de las tres implementaciones de *BLAS* en estudio así como los de la mejor rutina propuesta en esta sección, tanto para el caso de *GotoBLAS* como para *MKL*. En ambos casos, esta rutina es TBMV_B2_LEFT.

Como se puede apreciar, las prestaciones de TBMV_B2_LEFT mejoran claramente las obtenidas por el resto de las rutinas, excepto para matrices con ancho de banda inferior a 40, caso en el que la rutina de *MKL* obtiene unas prestaciones especialmente elevadas.

Por los resultados obtenidos, podemos concluir que las implementaciones de las rutinas del nivel 2 del *BLAS* invocadas por TBMV_B2_LEFT (TRMV y GEMV) son más eficientes en *MKL*.

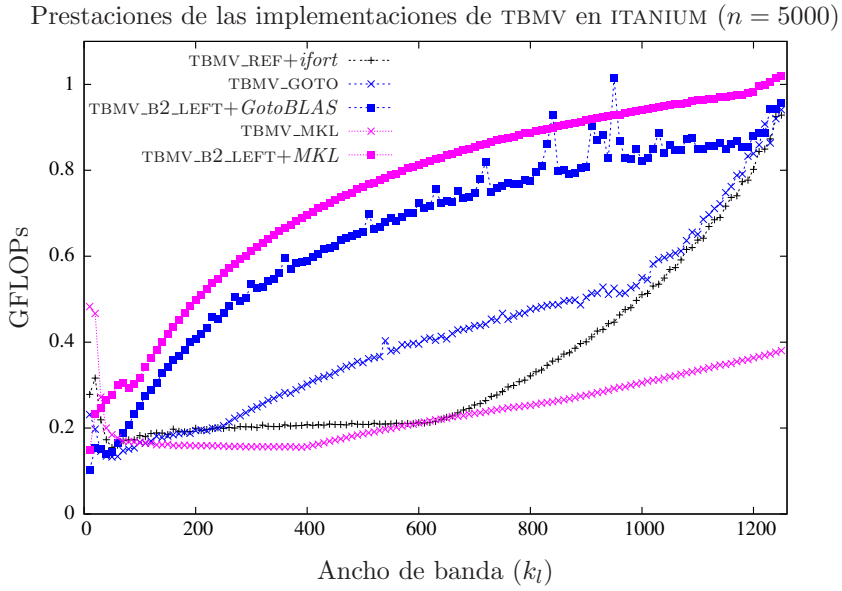


Figura 2.36: Comparativa de las mejores implementaciones.

Arquitectura XEON

La gráfica 2.37 muestra las prestaciones de las rutinas del *BLAS de referencia*, *GotoBLAS* y *MKL* para el producto entre una matriz banda y un vector. En este experimento, para la rutina del *BLAS de referencia* se han empleado los compiladores *ifort* y *g77*, con el objetivo de conocer cuál de estos compiladores genera el código más eficiente.

Este primer experimento muestra que las prestaciones obtenidas por los códigos generados con ambos compiladores son similares. Es por ello que durante el resto de experimentos se empleará únicamente *ifort*.

En esta ocasión, la rutina de *MKL* obtiene las mejores prestaciones para todos los casos de estudio incluidos. No obstante, la rutina de *GotoBLAS* obtiene resultados que tan sólo son ligeramente inferiores.

La figura 2.38 muestra los resultados obtenidos por todas las implementaciones propuestas en este estudio, tanto para el caso en que emplean los núcleos computacionales de la biblioteca *MKL* (derecha) como de *GotoBLAS* (izquierda).

Los resultados de todas las rutinas obtenidos en este experimento son bastante similares, aunque al igual que sucede con la arquitectura ITANIUM, las implementaciones basadas en rutinas del nivel 2 de *BLAS* denso se comportan mejor que las basadas en rutinas de *BLAS*-1 denso. En el caso de la biblioteca *MKL*, las prestaciones de TBMV son ligeramente mejores que las obtenidas por la mejor de las rutinas propuestas, TBMV_B2_LEFT. En el caso de *GotoBLAS*, tanto las nuevas rutinas como la implementación de TBMV propia de esta biblioteca obtienen resultados similares.

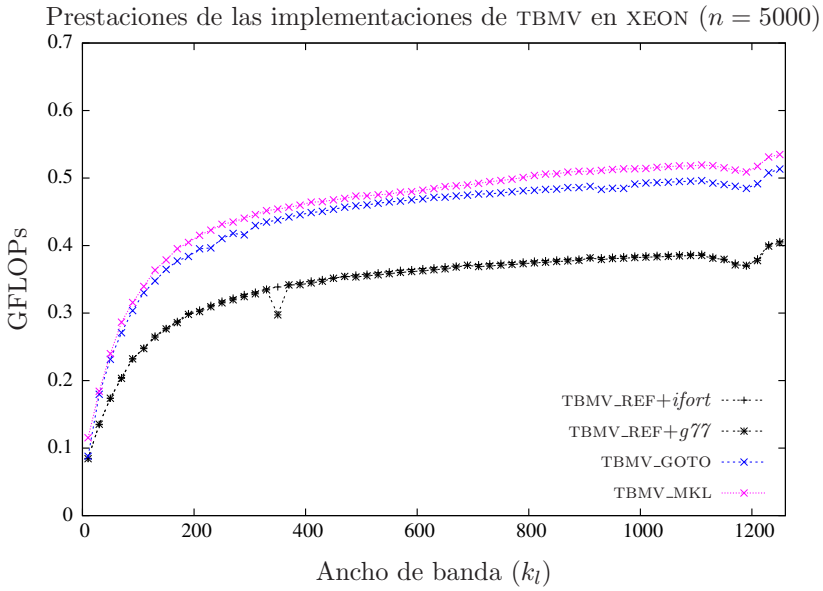


Figura 2.37: Comparativa de las diferentes implementaciones de bibliotecas *BLAS*.

La figura 2.39 compara los resultados de las mejores rutinas propuestas con los obtenidos por las rutinas de las implementaciones de *BLAS* en estudio. Se puede decir que la rutina incluida en *MKL* es la que genera las mejores prestaciones. A pesar de ello, las prestaciones del resto de las rutinas mostradas en la figura, exceptuando la rutina de *BLAS de referencia*, son muy similares.

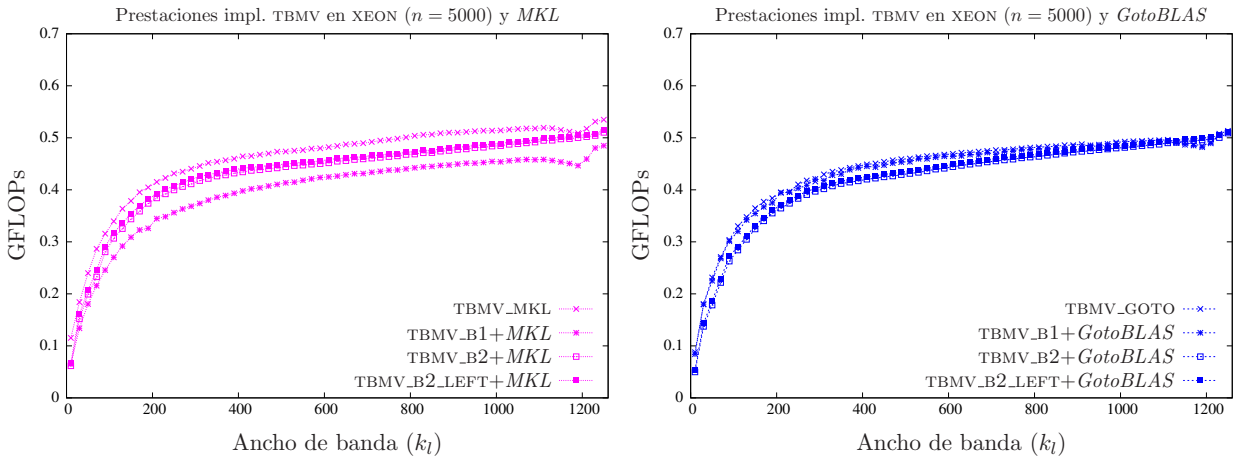


Figura 2.38: Comparativa de las diferentes implementaciones basadas en *BLAS*-1 y 2 denso.

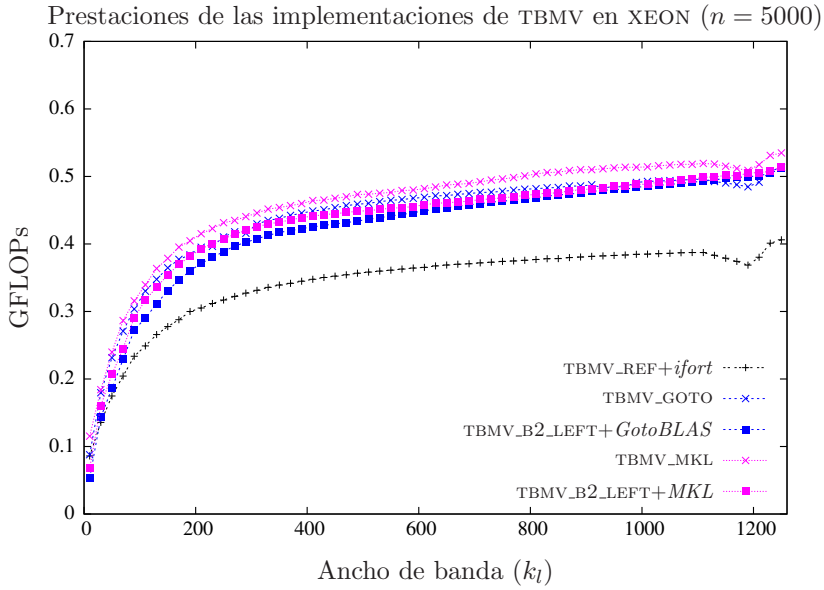


Figura 2.39: Comparativa de las mejores implementaciones.

2.3.7. Conclusiones

De los resultados mostrados se puede inferir que las implementaciones basadas en rutinas de *BLAS*-2 denso obtienen mejores prestaciones que las basadas en rutinas de *BLAS*-1. Las prestaciones de las dos rutinas basadas en rutinas de *BLAS*-2 denso son bastante parecidas, aunque TBMV_B2_LEFT es ligeramente más eficiente, y además presenta un coste espacial menor.

Respecto a las tres implementaciones de *BLAS* estudiadas, podemos concluir que en la arquitectura ITANIUM, la conveniencia de una biblioteca u otra está en función del ancho de banda de la matriz A, mientras que en XEON las implementaciones de *MKL* se muestran como la mejor opción.

El compilador *ifort* produce los códigos más eficientes en ITANIUM. En la arquitectura XEON no se han obtenido diferencias apreciables entre las prestaciones de los códigos generados con ambos compiladores.

Las rutinas propuestas mejoran notablemente las prestaciones ofrecidas por TBMV en la arquitectura ITANIUM. En la arquitectura XEON, estas rutinas prácticamente igualan los resultados ofrecidos por las rutinas de las bibliotecas *MKL* y *GotoBLAS*.

2.4. Solución de un sistema triangular banda

La solución de un sistema de ecuaciones lineales puede expresarse de forma matricial como

$$x := A^{-1} \cdot b, \tag{2.74}$$

donde $x, b \in \mathbb{R}^n$ son los vectores de incógnitas y términos independientes, respectivamente, y $A \in \mathbb{R}^{n \times n}$ es la matriz de coeficientes que, en la operación considerada en esta seccción, presenta una estructura triangular (inferior o superior) banda. En la funcionalidad ofrecida por *BLAS*, la matriz A puede aparecer en la expresión anterior como transpuesta, multiplicada además por un escalar $\alpha \in \mathbb{R}$; por simplicidad, consideramos el caso en que A es una matriz triangular inferior, con ancho de banda k_l, que aparece en (2.74) sin transponer, y asumimos que $\alpha = 1$. No obstante, el estudio puede ser fácilmente extendido al resto de casos. En la práctica x y b son el mismo vector, que a partir de ahora denominaremos x, y que inicialmente contiene b para ser sobreescrito al completar la operación con los elementos de x.

2.4.1. Algoritmo TBSV_{UNB}

El algoritmo TBSV_{UNB}, mostrado en la figura 2.40, obtiene la solución de un sistema triangular banda como el mostrado en (2.74). Durante la iteración j del algoritmo, el elemento j-ésimo de x es calculado y los siguientes k_l elementos de este vector son actualizados, despejando de las correspondientes ecuaciones la j-ésima incógnita. En estas operaciones intervienen, además de los $k_l + 1$ elementos de x citados, los elementos de la columna j-ésima de A. La figura 2.41 muestra los elementos accedidos durante una iteración del algoritmo.

Al igual que en la mayoría de los algoritmos mostrados en secciones anteriores, TBSV_{UNB} realiza los accesos a los elementos de la matriz por columnas. Esta propiedad posibilita un buen rendimiento en máquinas con un sistema de memoria jerarquizado.

2.4.2. Implementación del BLAS de referencia

La rutina TBSV_REF, perteneciente al *BLAS de referencia*, implementa el algoritmo TBSV_{UNB}. En cada iteración, tras obtener el valor de χ_1, recorre mediante un bucle el vector x_2 actualizando cada uno de sus componentes.

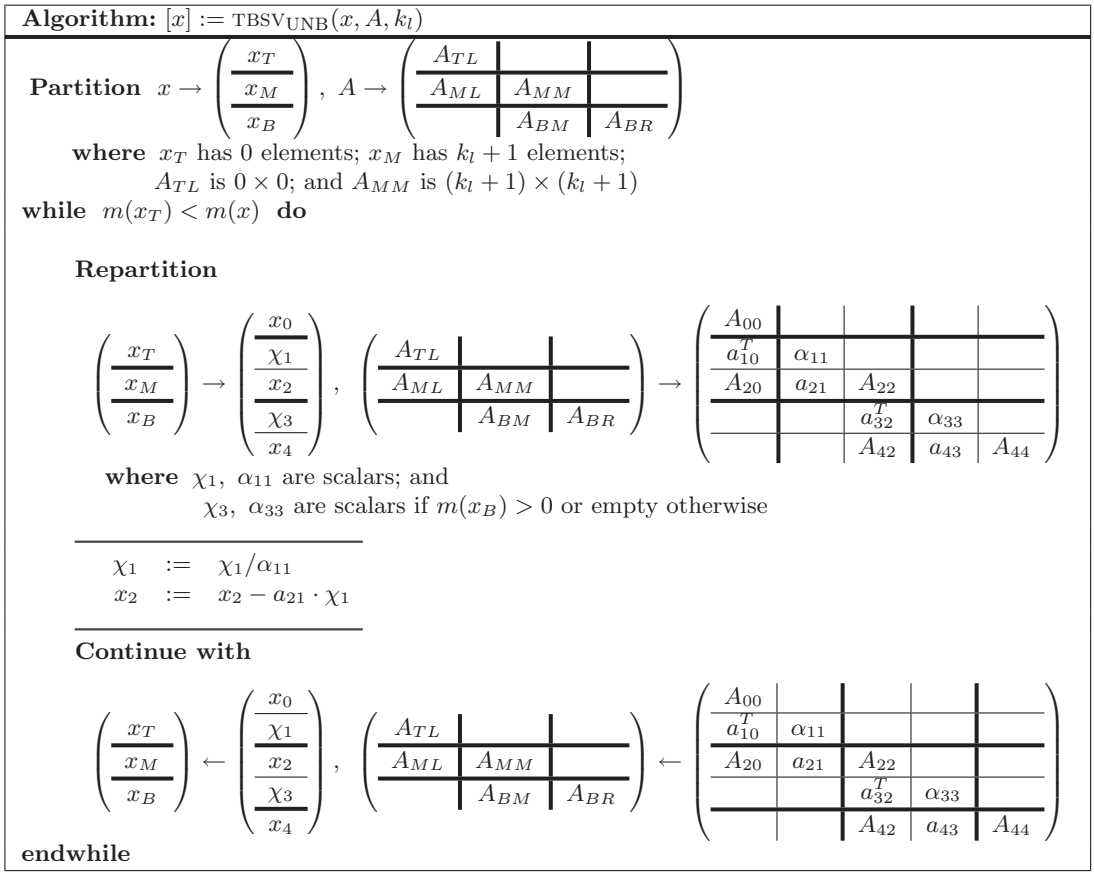


Figura 2.40: Algoritmo TBSV_{UNB} para la operación $x := A^{-1} \cdot x$.

La rutina TBSV_REF realiza un acceso a los elementos idéntico al mostrado en la figura 2.41. Por lo tanto, minimiza el número de accesos a la matriz A, ya que cada uno de sus elementos es accedido en una única ocasión, mientras que cada elemento de x es accedido hasta en k_l iteraciones diferentes. Respecto a la forma en que los elementos son accedidos, tanto en el caso de la matriz como en el del vector, se accede a posiciones consecutivas de memoria (en A se accede por columnas mientras que en x la afirmación anterior será cierta siempre que los elementos del vector se dispongan de manera consecutiva en la memoria). Además, la rutina evita operar con los elementos nulos de A situados fuera de la banda, reduciendo así el número de operaciones aritméticas a ejecutar.

Como aspecto negativo, comentar que las operaciones aritméticas no son ejecutadas por rutinas optimizadas de *BLAS* denso.

2.4.3. Implementación basada en rutinas de BLAS-1 denso

La rutina TBSV_B1 para la operación (2.74), al igual que TBSV_REF, implementa el algoritmo TBSV_{UNB} pero, a diferencia de la implementación de *BLAS de referencia*, ejecuta la mayoría de las operaciones aritméticas mediante núcleos optimizados de *BLAS* denso. En particular la actualización de x_2 corresponde a la operación implementada por la rutina del nivel 1 de *BLAS* AXPY. Así pues, la rutina TBSV_B1 invoca a AXPY para esta actualización, con lo que todas las operaciones aritméticas presentes en cada iteración, salvo una, son ejecutadas mediante esta rutina de *BLAS* denso. Ésta es la única diferencia existente entre TBSV_REF y TBSV_B1, por lo que el análisis reali-

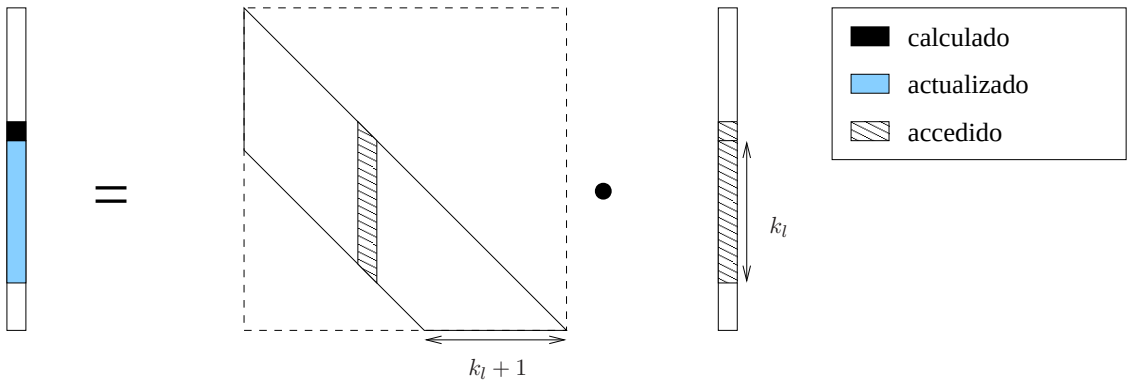


Figura 2.41: Acceso a los elementos durante una iteración del algoritmo TBSV_{UNB}.

zado en la sección 2.4.2 referente al número y modo de acceso es también válido para esta nueva rutina.

En definitiva, y dado que prácticamente la totalidad de las operaciones aritméticas son ejecutadas por un núcleo optimizado de *BLAS* denso, el aspecto con un margen de mejora más notable en esta rutina es el número de accesos a los elementos de x. Esta propiedad es inherente al algoritmo que implementa, por lo que una mejora en este sentido hace necesario el diseño de un nuevo algoritmo.

2.4.4. Algoritmo TBSV_{BLK}

El algoritmo TBSV_{BLK}, en la figura 2.42, resuelve el sistema (2.74) mediante operaciones con matrices y vectores, que pueden ser ejecutadas por rutinas del nivel 2 de *BLAS* denso. El uso de estas rutinas puede reducir el número de accesos sobre el vector x en un factor b (tamaño de bloque del algoritmo). También cabe destacar que TBSV_{BLK} mantiene todas las propiedades favorables de TBSV_{UNB}, a saber, accede a los elementos por columnas, realiza el número de accesos mínimo sobre los elementos de A, y mantiene el número de operaciones aritméticas.

En cada iteración del algoritmo, se opera con b columnas de A. Los bloques A_{11}, A_{21} y A_{31} contienen los elementos de A que se encuentran dentro de la banda en dichas columnas. Como se ilustra en la figura 2.43, los bloques A_{11} y A_{31} son triangulares (inferior y superior respectivamente), mientras que A_{21} es un bloque rectangular denso.

El bloque x_1 es reemplazado por la solución del sistema planteado por la matriz A_{11} y el propio x_1, es decir, se requiere la solución de un sistema triangular. Por otra parte, las actualizaciones de x_2 y de x_3 requieren sendos cálculos de productos matriz por vector (con una matriz densa y una matriz triangular superior, respectivamente).

2.4.5. Implementaciones basadas en rutinas de BLAS-2 denso

La rutina TBSV_B2 codifica el algoritmo TBSV_{BLK}. Esta nueva rutina utiliza núcleos optimizados de *BLAS*-2 denso para ejecutar la mayoría de las operaciones aritméticas. En concreto, las operaciones a ejecutar en cada iteración del algoritmo son las siguientes:

$$x_1 := A_{11}^{-1} \cdot x_1, \tag{2.75}$$

$$x_2 := x_2 - A_{21} \cdot x_1, \tag{2.76}$$

$$x_3 := x_3 - A_{31} \cdot x_1. \tag{2.77}$$

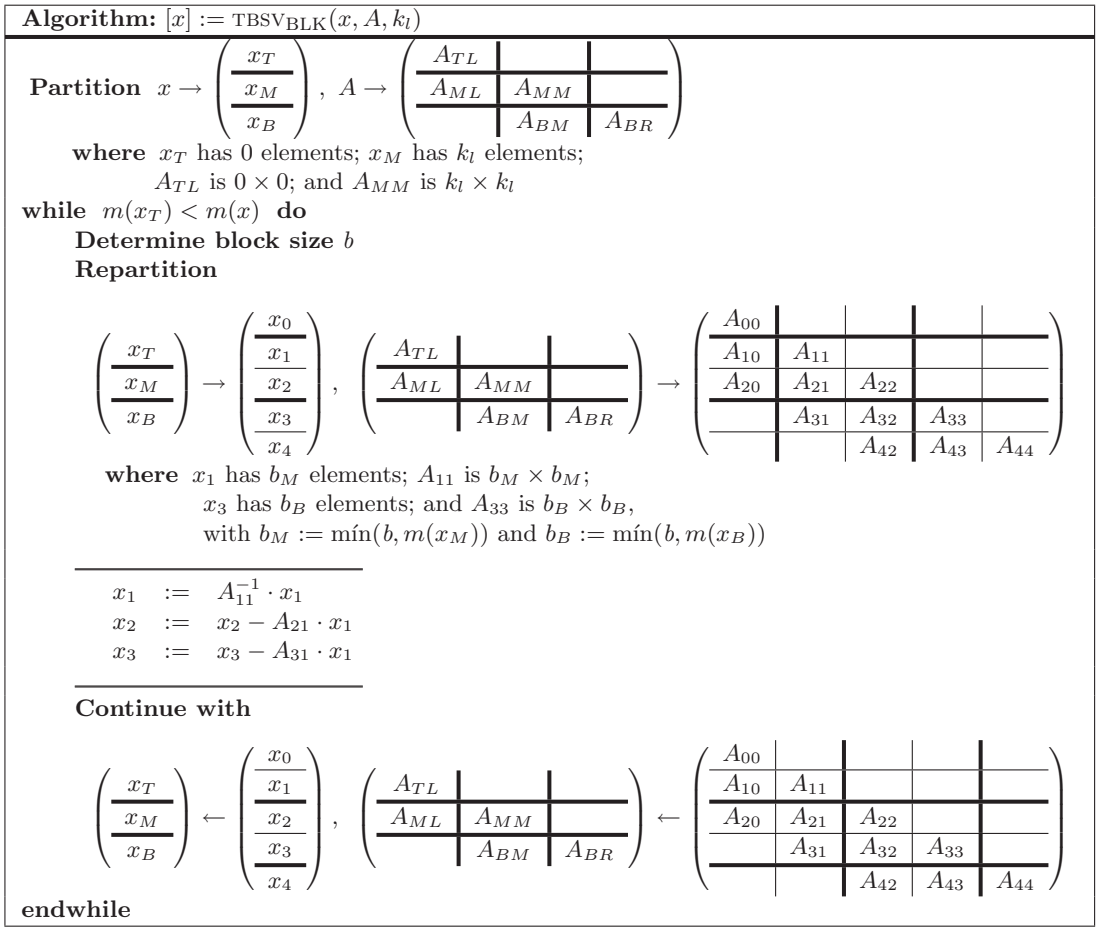


Figura 2.42: Algoritmo por bloques TBSV_{BLK} para la operación $x := A^{-1} \cdot x$.

La operación (2.75) requiere la resolución de un sistema triangular, y está implementada por la rutina de *BLAS*-2 denso TRSV. La rutina de *BLAS* GEMV implementa el producto entre una matriz densa y un vector, que es precisamente la operación necesaria en (2.76). Por último, la actualización de x_3, definida en la expresión (2.77), puede ser calculada mediante una invocación a la rutina TRMV.

Si evaluamos las necesidades de nuestro algoritmo y las interfaces de las rutinas *BLAS* a emplear, comprobamos que de nuevo la rutina TRMV nos plantea un problema ya conocido. A diferencia de lo requerido por TRMV, los vectores solución y multiplicando en (2.77) difieren. Para solventar este problema será necesario utilizar un espacio de trabajo (w). En particular, inicialmente se crea una copia de x_1 en w, con esta copia se ejecuta TRMV, y finalmente se substrae el resultado de TRMV a x_3.

Ésta es la secuencia detallada de las operaciones de una iteración del algoritmo:

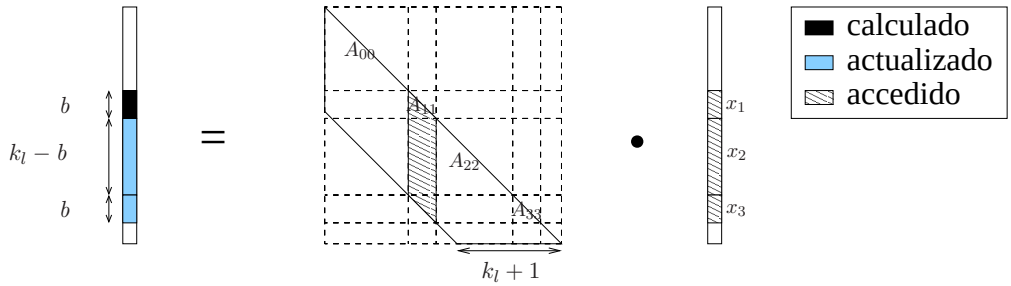


Figura 2.43: Acceso a los elementos durante una iteración del algoritmo TBSV_{BLK}.

$$
\begin{aligned}
(\text{TRSV}) \quad & x_1 := A_{11}^{-1} \cdot x_1, && (2.78)\\
(\text{GEMV}) \quad & x_2 := x_2 - A_{21} \cdot x_1, && (2.79)\\
& x_3 := x_3 - A_{31} \cdot x_1, && (2.80)\\
(\text{COPY}) \quad & w := x_1, && (2.81)\\
(\text{TRMV}) \quad & w := A_{31} \cdot w, && (2.82)\\
(\text{AXPY}) \quad & x_3 := x_3 - w. && (2.83)
\end{aligned}
$$

En TBSV_B2, los accesos sobre la matriz A son óptimos en número y forma, y además el número accesos sobre elementos de x es reducido. Cada elemento de x es accedido en $\frac{k_l+1}{b}$ iteraciones como máximo. Por lo tanto, desde el punto de vista del acceso a memoria, TBSV_B2 es una rutina muy eficiente. No obstante, esta rutina precisa de hasta 5 invocaciones a rutinas *BLAS* por iteración, circunstancia que puede perjudicar su rendimiento en problemas de talla reducida.

2.4.6. Resultados experimentales

Arquitectura ITANIUM

La figura 2.44 muestra las prestaciones de las rutinas para la resolución de sistemas triangulares banda presentes en las bibliotecas *GotoBLAS*, *MKL* y *BLAS de referencia*, esta última compilada con *g77* e *ifort*. Como se puede observar, los códigos generados por ambos compiladores obtienen prestaciones muy dispares, siendo *ifort* el que genera el código más eficiente.

La rutina de *BLAS de referencia* compilada con *ifort* genera las mejores prestaciones para matrices con ancho de banda menor que 600, mientras que para matrices con ancho de banda mayor, es la rutina de *GotoBLAS* la que obtiene las mejores prestaciones.

La eficiencia de las implementaciones propuestas se muestra en la figura 2.45. En las gráficas se reproducen las prestaciones cuando los nuevos códigos invocan a rutinas de *MKL* (izquierda) y *GotoBLAS* (derecha). En el caso de *MKL*, la rutina TBSV_B2 obtiene los mejores resultados para matrices de ancho de banda mayor que 50, mientras que para matrices con banda más estrecha es TBSV la rutina más eficiente. En el caso de *GotoBLAS* los resultados son muy diferentes. Tanto TBSV como TBSV_B1 consiguen las mayores prestaciones independientemente del ancho de banda de la matriz A y, además, ambas implementaciones obtienen rendimientos idénticos. Tan sólo con matrices de banda ancha TBSV_B2 se muestra como una alternativa eficiente.

La figura 2.46 recoge los resultados de las rutinas TBSV de todas las bibliotecas incluidas en el estudio, así como los de la mejor de las rutinas propuestas en esta sección (TBSV_B2 para *MKL* y

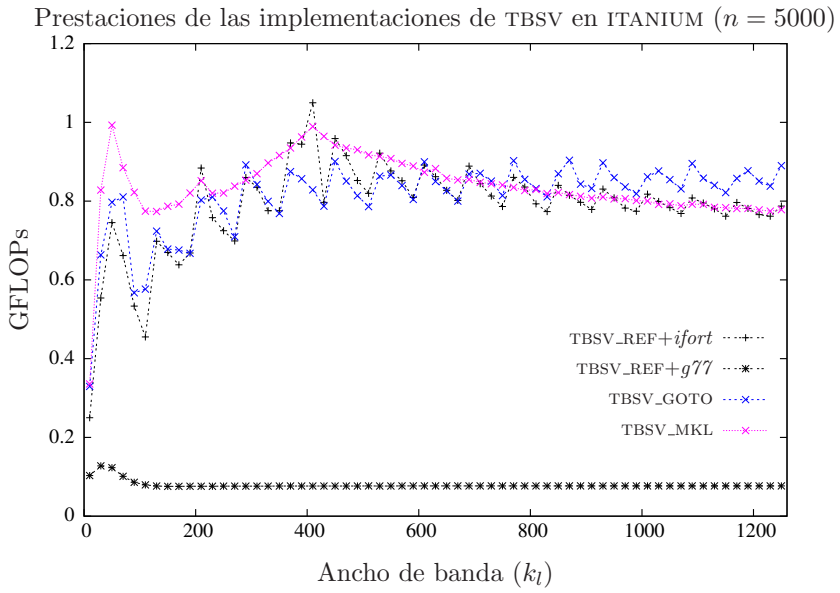


Figura 2.44: Comparativa de las diferentes implementaciones de bibliotecas *BLAS*.

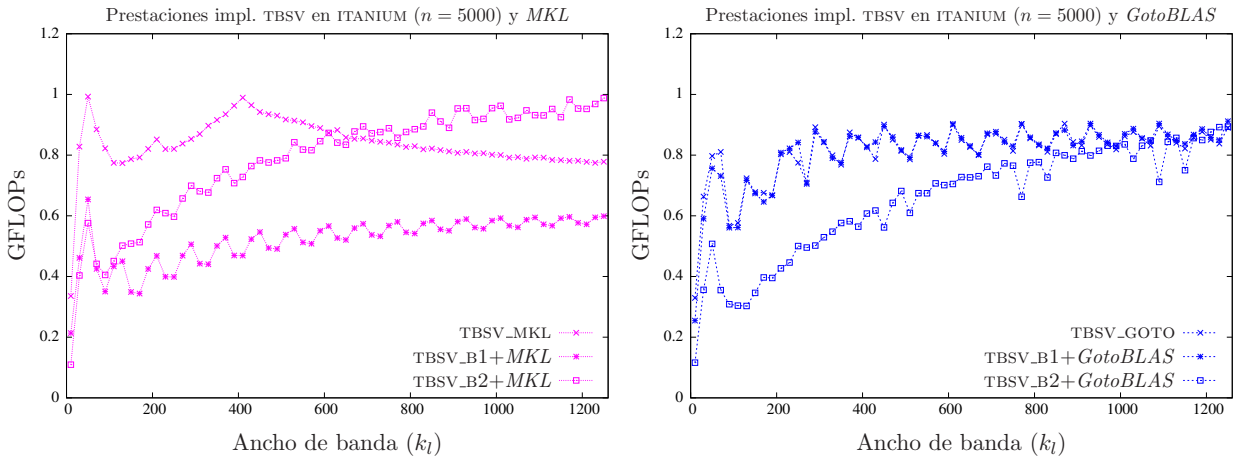


Figura 2.45: Comparativa de las diferentes implementaciones basadas en *BLAS*-1 y 2 denso.

TBSV_B1 para *GotoBLAS*). Como se puede apreciar, la rutina de *MKL* obtiene las mejores prestaciones para matrices con banda media y/o estrecha, mientras TBSV_B2 invocando a rutinas *BLAS*-2 de *MKL* es la opción más eficiente para matrices con k_l superior a 700.

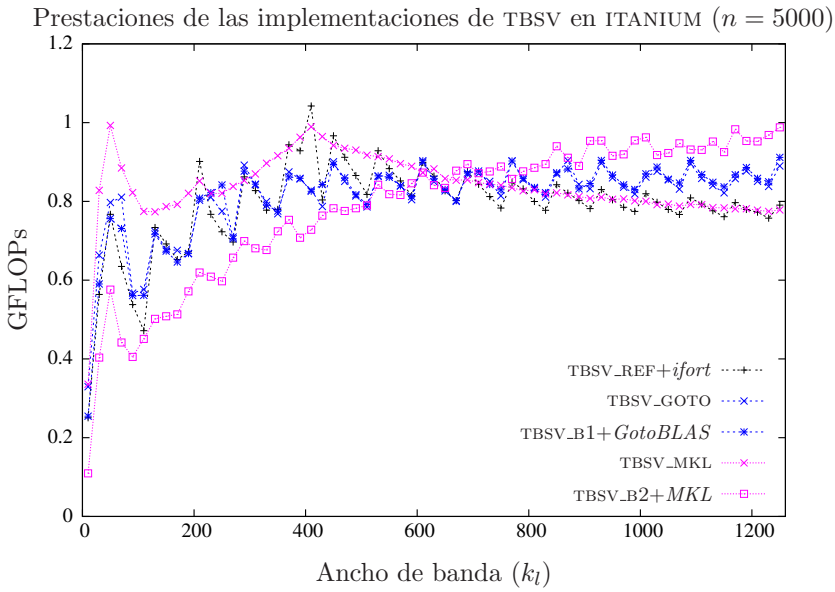


Figura 2.46: Comparativa de las mejores implementaciones.

Arquitectura XEON

Las prestaciones de las rutinas TBSV incluidas en las bibliotecas en estudio para esta operación son las mostradas en la figura 2.47. Para la rutina del *BLAS de referencia* se muestran los resultados al ser compilada con *ifort* y *g77*. A diferencia de lo ocurrido en la arquitectura ITANIUM, ambos compiladores obtienen códigos de similares prestaciones.

Los resultados demuestran que la rutina de *BLAS de referencia* es la más eficiente al operar con matrices de banda estrecha ($k_l < 100$), para matrices con ancho de banda entre 100 y 500 es la rutina incluida en *GotoBLAS* la que calcula el producto más rápidamente, mientras que para matrices con ancho de banda mayor es la rutina de *MKL* la que obtiene mejores resultados. Por lo tanto, podemos decir que la conveniencia de usar una u otra rutina depende fuertemente del ancho de banda de la matriz A.

La figura 2.48 (izquierda) detalla las prestaciones obtenidas por la rutina TBSV de *MKL*, así como los resultados de los códigos propuestos utilizando rutinas *MKL*. Las mejores prestaciones con matrices de banda estrecha ($k_l < 50$) se obtienen con TBSV, mientras que para matrices de ancho de banda superior es la rutina propuesta TBSV_B2 la más eficiente.

La gráfica de la figura 2.48 (derecha) es la análoga utilizando rutinas de *GotoBLAS*. En ella se refleja como las rutinas TBSV_B1y TBSV obtienen las mismas prestaciones. Ambas se comportan de forma más eficiente que TBSV_B2 al operar con matrices con ancho de banda inferior a 450, pero a partir de este ancho de banda, TBSV_B2 se revela como la mejor opción.

La figura 2.49 recoge las prestaciones de las rutinas de las bibliotecas estudiadas, así como de la mejor rutina propuesta, TBSV_B2. De esta gráfica se pueden extraer diferentes conclusiones:

- La elección de la rutina más conveniente está en función del ancho de banda de la matriz.

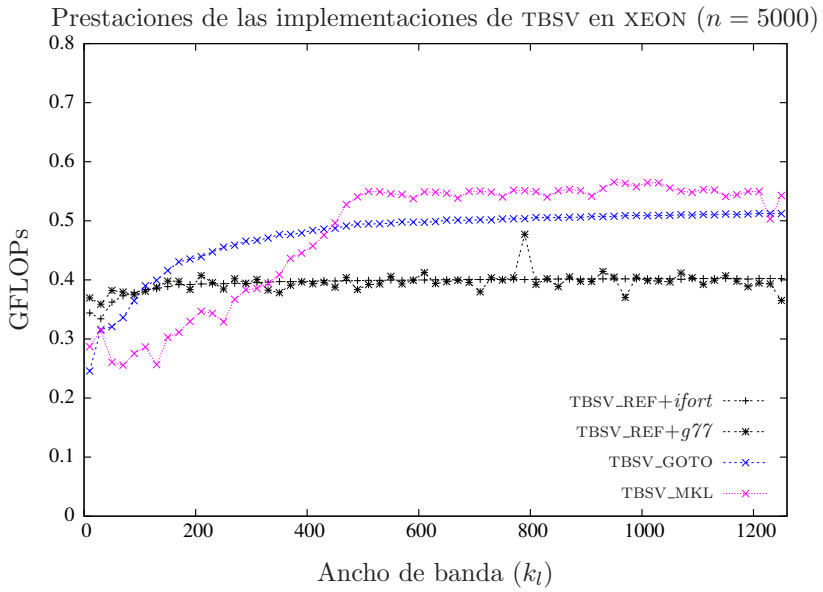


Figura 2.47: Comparativa de las diferentes implementaciones de bibliotecas *BLAS*.

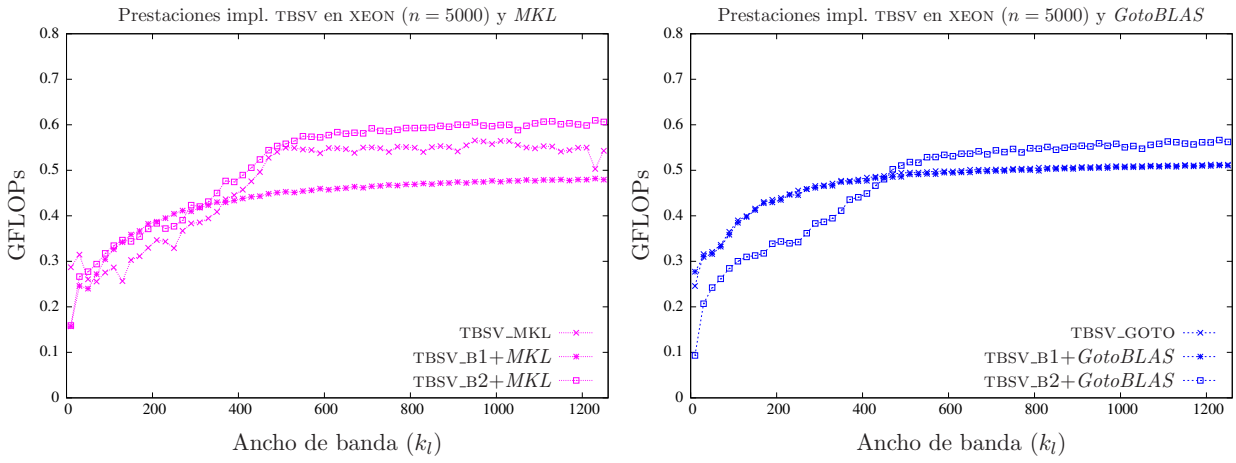


Figura 2.48: Comparativa de las diferentes implementaciones basadas en *BLAS*-1 y 2 denso.

- La nueva rutina TBSV_B2 ofrece las mejores prestaciones para matrices con ancho de banda superior a 450. No obstante, para matrices con ancho de banda inferior, la rutina propuesta TBSV_B1 genera prestaciones similares a las obtenidas por TBSV.
- Para matrices con banda inferior a 100, el esfuerzo computacional es demasiado pequeño como para justificar la invocación de otras rutinas de *BLAS*, y por este motivo la rutina de *BLAS de referencia* es la más eficiente para estas matrices.

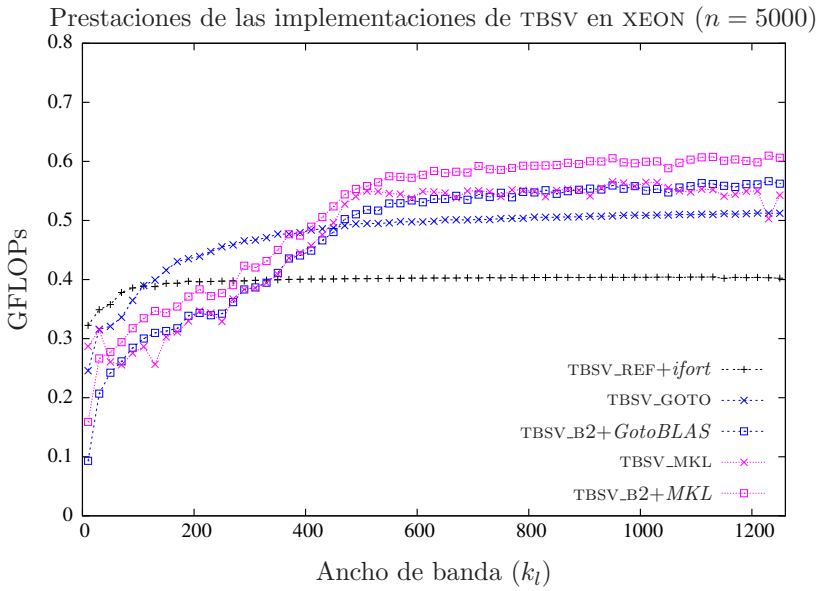


Figura 2.49: Comparativa de las mejores implementaciones.

2.4.7. Conclusiones

Se han probado diferentes implementaciones para la operación (2.74) cuando la matriz A es triangular inferior banda. Las nuevas rutinas generadas han sido comparadas experimentalmente con la incluida en la biblioteca *BLAS de referencia* y las de las implementaciones de *BLAS* afinadas *GotoBLAS* y *MKL*.

La rutina propuesta TBSV_B2 obtiene las mejores prestaciones para matrices con banda ancha, salvo en el caso de la arquitectura ITANIUM y la biblioteca *MKL*. Las prestaciones de TBSV_B1 invocando a rutinas de *GotoBLAS* y *MKL* son similares a las de la correspondiente rutina TBSV.

Cuando la matriz A presenta una banda estrecha, la rutina del *BLAS de referencia* compilada con *ifort* genera las mejores prestaciones.

Podemos afirmar que, en la arquitectura XEON, las rutinas propuestas mejoran las prestaciones de las rutinas TBSV incluidas en *MKL* y *GotoBLAS* independientemente del ancho de banda de la matriz A (con la salvedad de *MKL* y $k_l < 50$).

En el caso de ITANIUM, las rutinas propuestas igualan o mejoran a la rutina de *GotoBLAS* para cualquier ancho de banda, mientras que mejoran a la rutina *MKL* únicamente si la matriz A es una matriz de banda ancha.

Capítulo 3

BLAS 3 banda

BLAS no incluye en su especificación actual ninguna rutina para el cálculo de operaciones con coste computacional cúbico que procesen matrices banda, necesarias en diversas aplicaciones de ciencia e ingeniería. Como resultado, el único modo de efectuar operaciones como, por ejemplo, el producto $C = A \cdot B$ de una matriz banda A y otra densa B, es utilizar repetidamente la rutina de *BLAS*-2 para el producto matriz banda por vector, una vez por cada columna de la matriz B. Si bien esta solución puede ser aceptable cuando el número de columnas de la matriz densa es muy reducido, a medida que este número crece, los repetidos accesos a la matriz banda (uno por columna) degradan rápidamente las prestaciones. Es conveniente, pues, la especificación, diseño e implementación de un conjunto de rutinas que cubran la funcionalidad del *BLAS*-3 banda, reduciendo el número de accesos a los datos y mejorando en consecuencia las prestaciones.

En este capítulo se propone la especificación (interfaz y funcionalidad) de un conjunto de rutinas que corresponderían al nivel 3 de *BLAS* banda. Entre la funcionalidad cubierta por las nuevas rutinas se encuentra el producto de matrices donde una de las matrices que intervienen presenta una estructura banda, que puede ser además simétrica o triangular. Siguiendo el estándar de nomenclatura de *BLAS*, denotaremos a estas rutinas por los nombres SBMM (matriz simétrica banda), GBMM (matriz general banda) y TBMM (matriz triangular banda). Asimismo, también se contempla la resolución de múltiples sistemas triangulares de ecuaciones que comparten una misma matriz de coeficientes con estructura (triangular) banda, TBSM en la notación utilizada. En tercer lugar, se incluye como funcionalidad deseable el producto de dos matrices generales banda. Para esta rutina utilizaremos el nombre GBGBMM que, si bien se aparta del estándar de nomenclatura empleado en *BLAS*, sí especifica claramente su funcionalidad a la par que permite futuras extensiones con otros casos especiales del producto de dos matrices banda. Para cada una de estas nuevas rutinas se proponen diversas implementaciones y se realiza un estudio experimental de las prestaciones ofrecidas.

En lo sucesivo se considera que los anchos de banda inferior y superior de una matriz general banda son, respectivamente, k_l y k_u, mientras que en el caso simétrico el ancho de banda común es k_d. Para la resolución de sistemas triangulares, sólo se considera el caso triangular inferior, con un ancho de banda k_l. En todas estas operaciones se almacenan los elementos no nulos siguiendo el esquema compacto para matrices banda correspondiente.

El capítulo está estructurado en 5 secciones dedicadas, por este orden, al producto matricial con una de las matrices simétrica banda, general banda o triangular banda (tres primeras secciones); la resolución de múltiples sistemas triangulares banda (cuarta sección); y el producto de dos matrices generales banda (quinta sección). Los resultados experimentales de este capítulo se ofrecen a medida que se introducen las operaciones utilizando como plataforma de evaluación los

procesadores INTEL ITANIUM2 e INTEL XEON. El coste computacional de las operaciones de nivel 3 resulta habitualmente más elevado que el de las correspondientes operaciones del nivel 2 (por ejemplo, en el producto $C = A \cdot B$ comentado anteriormente, el aumento del coste respecto al de un simple producto de una matriz banda por un vector es proporcional al número de columnas de la matriz B). En consecuencia, para este tipo de operaciones sí resulta interesante la evaluación de la eficiencia paralela, y se incluye la evaluación en plataformas paralelas con p=4 procesadores ITANIUM y p=2 procesadores XEON, utilizándose 4 hebras de ejecución en la primera y 2 hebras de ejecución en la segunda.

3.1. Producto de una matriz simétrica banda por una matriz

La operación considerada en esta sección efectúa el cálculo

$$C := \alpha \cdot A \cdot B + \beta \cdot C \quad \text{o} \tag{3.1}$$
$$C := \alpha \cdot B \cdot A + \beta \cdot C, \tag{3.2}$$

donde $\alpha, \beta \in \mathbb{R}$ son factores de escalado, $B, C \in \mathbb{R}^{m \times n}$ son matrices generales (densas), y $A \in \mathbb{R}^{m \times m}$ en (3.1) o $A \in \mathbb{R}^{n \times n}$ en (3.2) es una matriz simétrica banda, con ancho de banda k_d, de la que sólo se almacenan los elementos en su parte triangular inferior o superior.

Siguiendo la notación *BLAS*, una rutina que implementara esta operación se denominaría SBMM, nombre que utilizaremos en lo sucesivo. La especificación Fortran-77 que se propone para (la implementación en doble precisión de) esta rutina es la mostrada en la figura 3.1. La rutina calcula el producto (3.1) o (3.2), en función del argumento `SIDE`, teniendo en cuenta que únicamente se almacena la parte triangular inferior o superior de la matriz banda (argumento `UPLO`). La intención de los restantes argumentos es obvia a partir de la definición de la operación.

En esta sección se considera el producto (3.1) en el caso en que únicamente se almacena la parte triangular inferior de A. Aun así, los algoritmos y resultados pueden ser fácilmente adaptados al producto (3.2) o al caso en que es la parte superior la que se encuentra almacenada. Así mismo, por simplicidad, se considera que $\alpha = \beta = 1$ en (3.1), de modo que obtenemos una variante más sencilla de la operación,

$$C := A \cdot B + C. \tag{3.3}$$

3.1.1. Implementaciones basadas en SBMV

Una forma simple de implementar la operación (3.3) es realizar n productos matriz-vector, uno por cada columna de B. De esta forma se pueden crear tantas rutinas para SBMM como implementaciones se tengan de SBMV. Precisamente en la sección 2.1 se estudian diferentes variantes de esta última rutina. Siguiendo este razonamiento se han codificado las variantes de SBMM que hacen uso de las mejores implementaciones de SBMV, concretamente SBMV_B1_AXPY y SBMV_B2. El resultado es un conjunto de rutinas para el producto de una matriz simétrica banda por una matriz general, basadas en *BLAS*-2, con unas prestaciones similares a las de la implementación de SBMV subyacente.

A fin de mejorar la localidad de los accesos, y en consecuencia las prestaciones, a continuación se plantea un algoritmo alternativo, que resulta en implementaciones que realizan sus cálculos mediante núcleos del *BLAS*-3.

```
      SUBROUTINE DSBMM( SIDE, UPLO, M, N, K, ALPHA, A, LDA, B, LDB, BETA,
                        C, LDC )
*     .. Scalar Arguments ..
      DOUBLE PRECISION   ALPHA, BETA
      INTEGER            K, LDA, LDB, LDC, M, N
      CHARACTER          SIDE, UPLO
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, *), B( LDB, *), C( LDC, *)
*
*  Purpose
*  =======
*
*  DSBMM performs one of the matrix-matrix operations
*
*     C := alpha*A*B + beta*C,
*
*  or
*
*     C := alpha*B*A + beta*C,
*
*  where alpha and beta are scalars, A is a symmetric band matrix with
*  k super-diagonals/sub-diagonals and B and C are m by n matrices.
*
```

Figura 3.1: Especificación propuesta para la rutina SBMM.

Algorithm: $[C] := \text{SBMM}_{\text{BLK}}(C, A, B, k_d)$

Partition $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array} \right)$, $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array} \right)$
where C_L, B_L have 0 columns

while $n(C_L) < n(C)$ **do**
Determine block size c
Repartition

$$\left(\begin{array}{c|c} C_L & C_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array} \right), \left(\begin{array}{c|c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right)$$

where C_1, B_1 have c columns

$C_1 := \text{SBMM}_{\text{INNER\_LOOP}}(C_1, A, B_1, k_d)$

Continue with

$$\left(\begin{array}{c|c} C_L & C_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array} \right), \left(\begin{array}{c|c} B_L & B_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right)$$

endwhile

Figura 3.2: Bucle externo del algoritmo por bloques SBMM_{BLK} para la operación $C := A \cdot B + C$.

3.1.2. Algoritmo SBMM_{BLK}

El algoritmo para el producto de una matriz simétrica banda por una matriz general que se propone a continuación se compone de dos bucles, a los que denominaremos interno y externo. El bucle externo, en la figura 3.2, recorre horizontalmente las matrices B y C, operando en cada iteración con la matriz A y los bloques B_1 y C_1, formados ambos por c columnas.

El bucle interno, en la figura 3.3, calcula los elementos de c columnas de C (en $C_1 = E$). Para

Algorithm: $[E] := \text{SBMM}_{\text{INNER\_LOOP}}(E, A, D, k_d)$

Partition $E \rightarrow \begin{pmatrix} E_T \\ \hline E_M \\ \hline E_B \end{pmatrix}, A \rightarrow \left(\begin{array}{c|c|c} A_{TL} & \star & \\ \hline A_{ML} & A_{MM} & \star \\ \hline & A_{BM} & A_{BR} \end{array}\right), D \rightarrow \begin{pmatrix} D_T \\ \hline D_M \\ \hline D_B \end{pmatrix}$

where E_T, D_T have 0 rows; E_M, D_M have k_d rows; A_{TL} is 0×0; and A_{MM} is $k_d \times k_d$

while $m(E_T) < m(E)$ **do**

Determine block size b

Repartition

$$\begin{pmatrix} E_T \\ \hline E_M \\ \hline E_B \end{pmatrix} \rightarrow \begin{pmatrix} E_0 \\ \hline E_1 \\ \hline E_2 \\ \hline E_3 \\ \hline E_4 \end{pmatrix}, \left(\begin{array}{c|c|c} A_{TL} & \star & \\ \hline A_{ML} & A_{MM} & \star \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & & & & \\ \hline A_{10} & A_{11} & & & \\ \hline A_{20} & A_{21} & A_{22} & & \\ \hline & A_{31} & A_{32} & A_{33} & \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right)$$

$$\begin{pmatrix} D_T \\ \hline D_M \\ \hline D_B \end{pmatrix} \rightarrow \begin{pmatrix} D_0 \\ \hline D_1 \\ \hline D_2 \\ \hline D_3 \\ \hline D_4 \end{pmatrix}$$

where E_1, D_1 have b_M rows; E_3, D_3 have b_B rows,
with $b_M := \text{mín}(b, m(E_M))$ and $b_B := \text{mín}(b, m(E_B))$

$$\begin{array}{lcl} E_1 & := & A_{11} \cdot D_1 + E_1 \\ E_1 & := & A_{21}^T \cdot D_2 + E_1 \\ E_1 & := & A_{31}^T \cdot D_3 + E_1 \\ E_2 & := & A_{21} \cdot D_1 + E_2 \\ E_3 & := & A_{31} \cdot D_1 + E_3 \end{array}$$

Continue with

$$\begin{pmatrix} E_T \\ \hline E_M \\ \hline E_B \end{pmatrix} \leftarrow \begin{pmatrix} E_0 \\ \hline E_1 \\ \hline E_2 \\ \hline E_3 \\ \hline E_4 \end{pmatrix}, \left(\begin{array}{c|c|c} A_{TL} & \star & \\ \hline A_{ML} & A_{MM} & \star \\ \hline & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & & & & \\ \hline A_{10} & A_{11} & & & \\ \hline A_{20} & A_{21} & A_{22} & & \\ \hline & A_{31} & A_{32} & A_{33} & \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right)$$

$$\begin{pmatrix} D_T \\ \hline D_M \\ \hline D_B \end{pmatrix} \leftarrow \begin{pmatrix} D_0 \\ \hline D_1 \\ \hline D_2 \\ \hline D_3 \\ \hline D_4 \end{pmatrix}$$

endwhile

Figura 3.3: Bucle interno del algoritmo por bloques SBMM_{BLK} para la operación $C := A \cdot B + C$. Sólo la parte triangular inferior de A es accedida.

ello, recorre la matriz A a lo largo de su diagonal al igual que se hace en el algoritmo SBMV_{BLK}. En cada iteración, opera con los elementos no nulos de b columnas de A y completa el cálculo de b filas de C_1 (en E_1), al tiempo que actualiza las k_d filas siguientes (en E_2 y E_3).

La figura 3.4 muestra cómo se particionan las matrices B y C durante una iteración del bucle externo. El bloque C_1 se calcula durante la iteración actual del bucle. La figura 3.5 ilustra el acceso realizado sobre los elementos de las tres matrices durante una iteración del bucle interno.

A diferencia de los algoritmos introducidos hasta el momento, SBMM_{BLK} incluye dos tamaños de bloque, c para las matrices B y C, y b para las tres matrices. Dos versiones más específicas

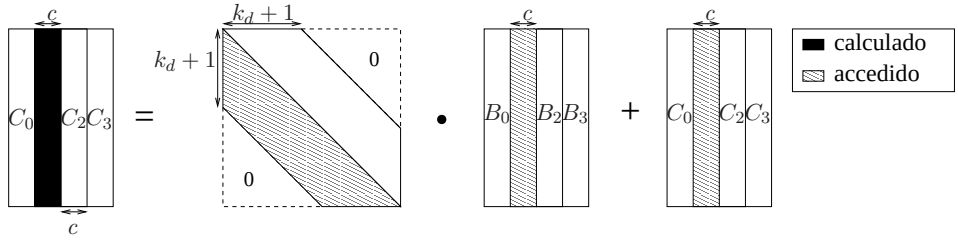


Figura 3.4: Acceso a los elementos en el bucle externo del algoritmo SBMM_{BLK}.

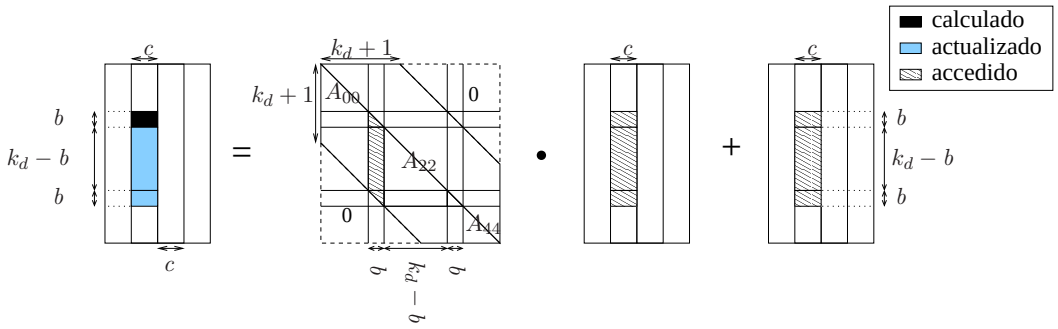


Figura 3.5: Acceso a los elementos en el bucle interno del algoritmo SBMM_{BLK}.

del algoritmo se obtienen al unificar el valor de ambos tamaños de bloque, o al asignar a c el valor de n (caso en el que el bucle externo pierde sentido). El valor de c influye en la eficiencia de la rutina, aportando una mayor flexibilidad a la optimización de este algoritmo. Cada elemento de A es accedido una vez por cada iteración del bucle externo, es decir, en n/c ocasiones a lo largo de todo el algoritmo. Así pues, asignar a c un valor demasiado pequeño aumenta el número de accesos a los elementos de A y puede reducir las prestaciones. Por otro lado, b determina en gran medida la granularidad de las operaciones realizadas en el algoritmo de modo que, un valor demasiado pequeño para esta variable, desemboca en operaciones poco eficientes. En cambio, un valor demasiado elevado para c y b puede propiciar que en cada operación ejecutada en el bucle interno, los datos desborden la capacidad de la memoria caché y se produzca con ello una pérdida de prestaciones.

3.1.3. Implementaciones basadas en rutinas de BLAS-3 denso

Teniendo en cuenta las actualizaciones a realizar sobre los bloques que aparecen en la figura 3.3 y la estructura de cada uno de éstos (ver figura 3.5), la siguiente lista de operaciones especifica los cálculos a ejecutar en cada iteración del bucle interno, así como la rutina de *BLAS* denso que los implementa:

$$
\begin{aligned}
(\text{SYMM}) \quad & E_1 := A_{11} \cdot D_1 + E_1, && (3.4)\\
(\text{GEMM}) \quad & E_1 := A_{21}^T \cdot D_2 + E_1, && (3.5)\\
& E_1 := A_{31}^T \cdot D_3 + E_1, && (3.6)\\
(\text{LACPY}) \quad & W := D_3, && (3.7)\\
(\text{TRMM}) \quad & W := A_{31}^T \cdot W + W, && (3.8)\\
& E_1 := W + E_1, && (3.9)\\
(\text{GEMM}) \quad & E_2 := A_{21} \cdot D_2 + E_2, && (3.10)\\
& E_3 := A_{31} \cdot D_3 + E_3, && (3.11)\\
(\text{LACPY}) \quad & W := D_3, && (3.12)\\
(\text{TRMM}) \quad & W := A_{31} \cdot W + W, && (3.13)\\
& E_3 := W + E_3. && (3.14)
\end{aligned}
$$

No todas las operaciones pueden ser ejecutadas invocando directamente a una única rutina de *BLAS*. Esto sucede con los productos matriz triangular-matriz en (3.6) y (3.11). La rutina de *BLAS* TRMM, que implementa el producto de dos matrices cuando una de éstas es triangular, requiere que la matriz general que interviene como operando y la matriz resultado sean la misma. Esto no sucede en las operaciones (3.6) y (3.11). Para resolver este problema se realiza una copia de la matriz operando (D_3 en ambos casos) sobre un espacio de trabajo denominado W, y tras invocar a TRMM con este espacio de trabajo, se actualiza el contenido de la matriz resultado (E_1 y E_3) con el valor de los elementos de W. La rutina LACPY efectúa la copia elemento a elemento entre dos matrices (operaciones (3.7) y (3.12)). En el caso de las operaciones (3.9) y (3.14), no existe rutina de *BLAS* que ejecute este tipo de actualización, con lo que se efectuará esta copia mediante dos bucles anidados.

La rutina SBMM_B3 implementa el algoritmo SBMM_{BLK} ejecutando las operaciones aritméticas tal y como se ha detallado.

Implementación Merge (SBMM_B3_MERGE)

La implementación del algoritmo SBMM_{BLK} en la rutina SBMM_B3_MERGE es similar en gran medida a la realizada en la rutina SBMM_B3. La principal diferencia estriba en que en esta nueva codificación se reduce el número de invocaciones a rutinas *BLAS* ejecutando las operaciones con los bloques A_{21} y A_{31} con una única llamada a la rutina GEMM. Para ello, durante cada iteración se realiza una copia del sector físico de memoria en el que se almacena la parte triangular inferior estricta del bloque A_{31} (según el esquema de almacenamiento para matrices densas), se fija el valor de cada uno de los elementos de este sector a cero, se invoca a la rutina GEMM en dos ocasiones (la primera para calcular E_1 y la segunda para actualizar E_2 e E_3) y, finalmente, se reestablecen los valores de la parte triangular inferior estricta de A_{31}. De esta forma, la secuencia de operaciones es la siguiente:

$$
\begin{aligned}
(\text{SYMM}) \qquad E_1 &:= A_{11} \cdot\ D_1 + E_1, && (3.15)\\
W &:= \text{STRIL}(A_{31}), && (3.16)\\
\text{STRIL}(A_{31}) &:= 0, && (3.17)\\
(\text{GEMM}) \qquad E_1 &:= \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix}^T \cdot \begin{bmatrix} D_2 \\ D_3 \end{bmatrix} + E_1, && (3.18)\\
(\text{GEMM}) \qquad \begin{bmatrix} E_2 \\ E_3 \end{bmatrix} &:= \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot \begin{bmatrix} D_2 \\ D_3 \end{bmatrix} + \begin{bmatrix} E_2 \\ E_3 \end{bmatrix}, && (3.19)\\
\text{STRIL}(A_{31}) &:= W. && (3.20)
\end{aligned}
$$

Las operaciones (3.16) y (3.17) pueden ejecutarse simultáneamente mediante dos bucles anidados. La operación (3.20) recibe el mismo tipo de codificación.

La variante implementada en esta rutina reduce el número de invocaciones a rutinas de *BLAS* denso. Mientras que SBMM_B3 realiza cinco llamadas a rutinas de diferentes niveles de *BLAS* en cada iteración, la implementación SBMM_B3_MERGE realiza en cada iteración tres invocaciones a rutinas de *BLAS*, todas ellas pertenecientes al nivel 3, incrementando de este modo el tamaño de los bloques que particicipan en las operaciones. El aspecto negativo de esta última rutina es que en cada iteración precisa de dos copias de aproximadamente $b/2$ elementos cada una.

3.1.4. Resultados experimentales

Arquitectura ITANIUM

BLAS secuencial La gráfica de la izquierda de la figura 3.6 recoge los resultados obtenidos por las nuevas rutinas SBMM_B3 y SBMM_MERGE cuando se utilizan los núcleos computacionales del tercer nivel de *BLAS* implementados en la biblioteca *MKL*, así como los de la rutina que invoca repetidamente (n veces) a la implementación de SBMV incluida en la biblioteca *MKL* (identificada como SBMV_*MKL*). Para las implementaciones SBMM_B3 y SBMM_MERGE se ha experimentado con los valores 4 y 20 para la variable n (las prestaciones de SBMV_MKL no varían con el valor de n). Se han seleccionado estos valores para n con el fin de mostrar que las implementaciones *BLAS*-3 obtienen buenas prestaciones incluso cuando n toma valores pequeños. Como se puede observar, las dos rutinas basadas en *BLAS*-3 alcanzan prestaciones claramente superiores a las obtenidas por SBMV, especialmente en el caso de la rutina SBMM_MERGE.

La gráfica a la derecha de la figura 3.6 muestra los resultados obtenidos utilizando rutinas *GotoBLAS* para las diferentes implementaciones presentadas en esta sección. De nuevo, las implementaciones *BLAS*-3 mejoran notablemente las prestaciones obtenidas por SBMV.

Finalmente, a modo de resumen, la figura 3.7 ofrece una comparativa entre las prestaciones obtenidas utilizando las rutinas SBMV incluidas en las implementaciones *BLAS* en estudio y las mejores implementaciones para SBMM presentadas en este trabajo (SBMM_B3 de *GotoBLAS* y SBMM_MERGE de *MKL*) cuando $n = 4$. Como se aprecia en la figura, las rutinas *BLAS*-3 obtienen los mejores tiempos, destacando su rendimiento cuando se enlazan con los núcleos de la biblioteca *MKL*.

BLAS paralelo La figura 3.8 muestra las prestaciones de las nuevas implementaciones basadas en núcleos computacionales de *BLAS*-3 cuando se enlazan con versiones paralelas multihebra de *BLAS*. A diferencia del caso secuencial, las nuevas rutinas empeoran sus prestaciones cuando la matriz A presenta una banda estrecha pero, en cambio, obtienen un rendimiento notablemente mayor cuando el ancho de banda de la matriz aumenta.

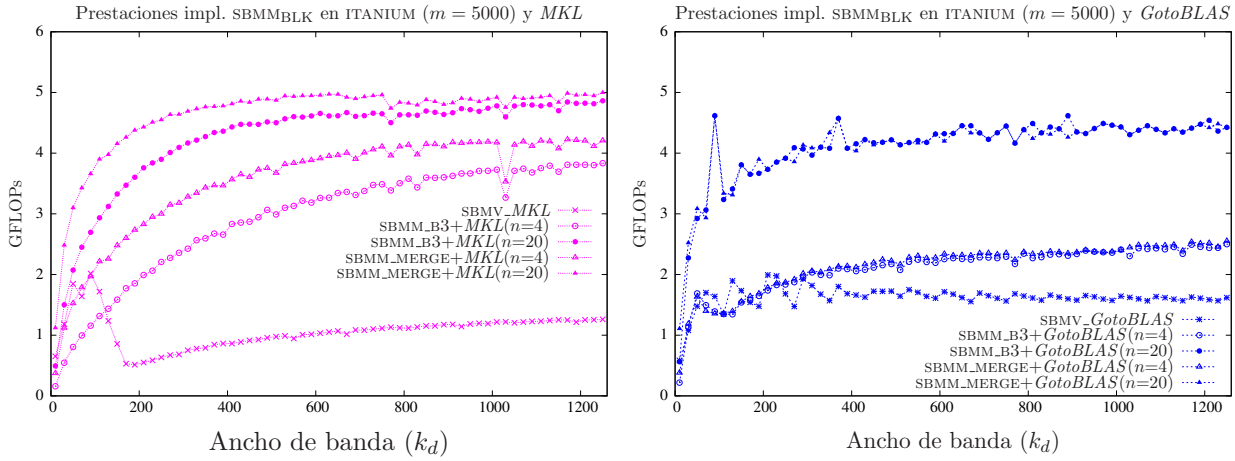


Figura 3.6: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

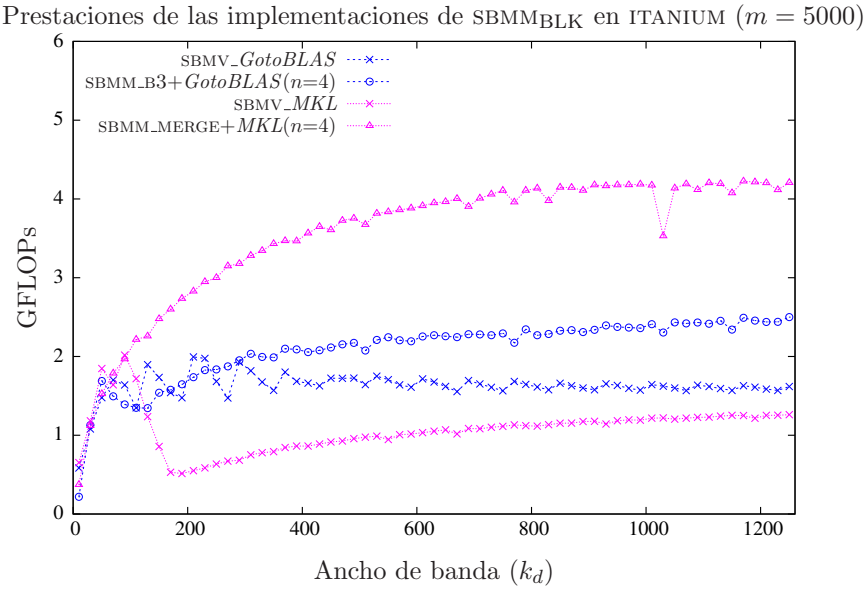


Figura 3.7: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

Al emplear rutinas de la biblioteca *MKL* (gráfica de la izquierda de la figura), tanto SBMM_B3 como SBMM_MERGE obtienen prestaciones similares, siendo SBMM_B3 ligeramente más eficiente.

Cuando las nuevas rutinas invocan a núcleos computacionales de *GotoBLAS* (gráfica de la derecha), SBMM_B3 es ligeramente más eficiente cuando el ancho de banda es reducido. La rutina SBMM_MERGE, sin embargo, se ve más beneficiada por el incremento de k_d de forma que, cuando A es una matriz de banda ancha, SBMM_B3 y SBMM_MERGE obtienen prestaciones muy parecidas. La rutina SBMM_MERGE opera con bloques de mayor tamaño y obtiene más paralelismo, circunstancia especialmente beneficiosa cuando la matriz A no presenta una banda estrecha. No obstante, el sobrecoste que introduce por las distintas copias ejecutadas hace que SBMM_B3 obtenga mejores resultados cuando k_d toma valores pequeños.

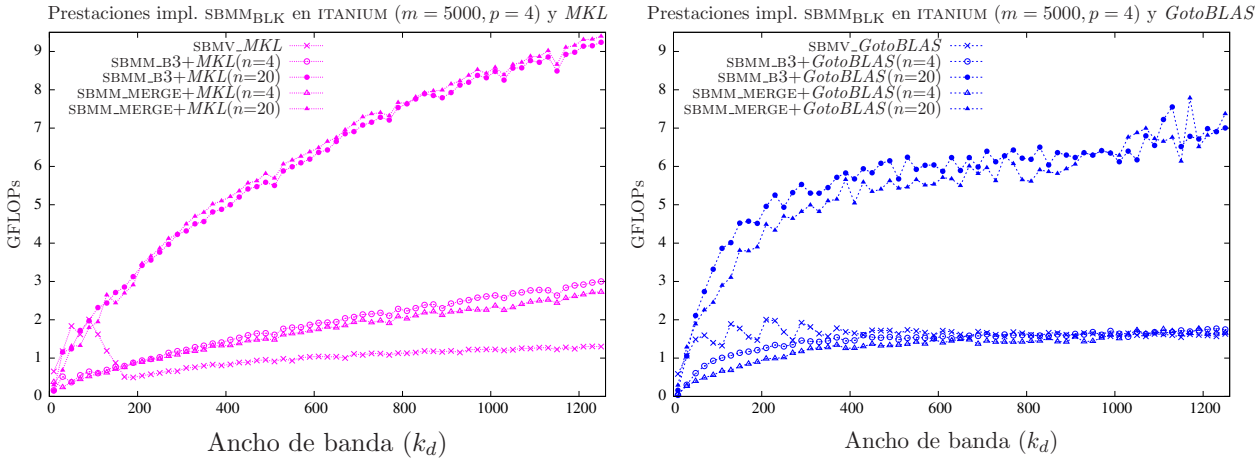


Figura 3.8: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

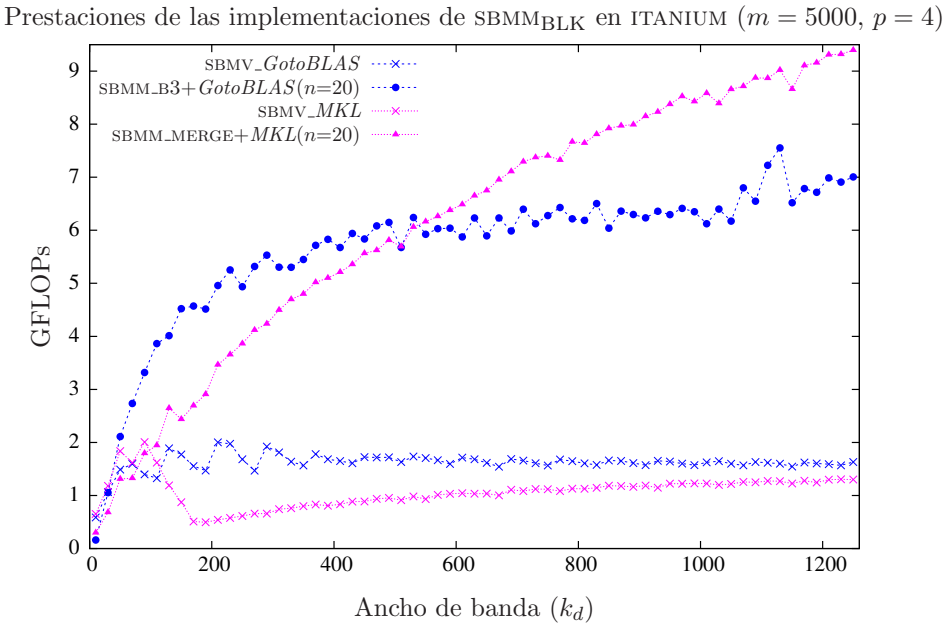


Figura 3.9: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

En la figura 3.9 se recogen los resultados de las mejores implementaciones identificadas en los experimentos paralelos. Los valores pequeños de n no son propicios para el uso de rutinas *BLAS*-3, ni tampoco para las versiones paralelas de *BLAS* utilizadas en este experimento, como queda reflejado en la gráfica. Un comentario similar procede respecto al ancho de banda, especialmente con las versiones que invocan a rutinas de *MKL*. Por contra, con valores grandes de k_d y n las implementaciones paralelas de *BLAS* son hasta un 50 % más rápidas que las secuenciales. La selección de la mejor implementación está en función del ancho de banda y del valor de n. Así, si A es una matriz de banda ancha, las implementaciones basadas en rutinas *MKL* representan la mejor opción.

Arquitectura XEON

BLAS secuencial A continuación se repite el estudio con la arquitectura XEON, comparando las versiones que emplean repetidamente la implementación de SBMV incluida en las bibliotecas *BLAS* con las nuevas implementaciones basadas en rutinas *BLAS*-3.

La figura 3.10 ofrece los resultados de las implementaciones basadas en rutinas SBMV y los de las rutinas SBMM_B3 y SBMM_MERGE. A la izquierda de la figura se disponen los resultados obtenidos utilizando *MKL*, mientras que la gráfica de la derecha muestra los resultados generados con *GotoBLAS*.

Como se puede observar en la gráfica de la izquierda, las implementaciones basadas en rutinas *BLAS*-3 son más eficientes que la basada en la rutina SBMV, incluso cuando el valor de n es pequeño o la banda de la matriz es estrecha. En particular, en este último caso, la versión SBMM_MERGE es ligeramente más rápida que SBMM_B3. Cuando el ancho de banda o el valor de n aumenta, la diferencia entre las prestaciones de las implementaciones *BLAS*-3 y de niveles inferiores de *BLAS* se acentúa.

Respecto a los resultados obtenidos con *GotoBLAS* (gráfica de la derecha), las rutinas basadas en núcleos computacionales de *BLAS*-3 obtienen resultados ligeramente superiores a los obtenidos por la invocación repetida de la rutina SBMV de *GotoBLAS* al trabajar con valores de n muy pequeños (en este caso n=4). Por otro lado, para valores de n mayores, las rutinas SBMM_B3 y SBMM_MERGE son notablemente más rápidas.

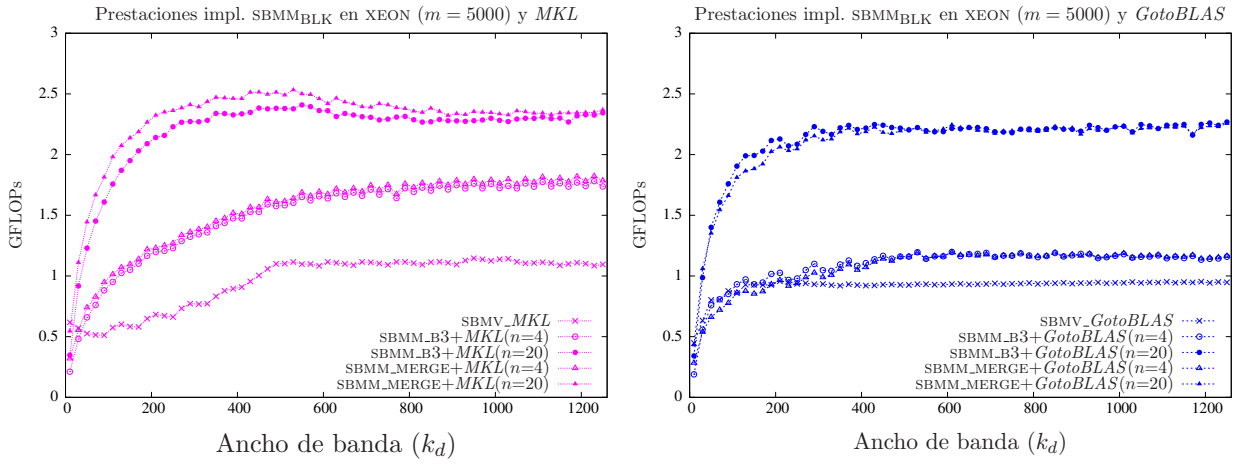


Figura 3.10: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

La figura 3.11 ofrece las prestaciones de las mejores implementaciones de SBMM estudiadas. Entre éstas, SBMM_MERGE+*MKL* destaca por su eficiencia. Incluso en operaciones con una matriz

de banda estrecha y con valores de n reducidos, esta implementación consigue prestaciones más elevadas.

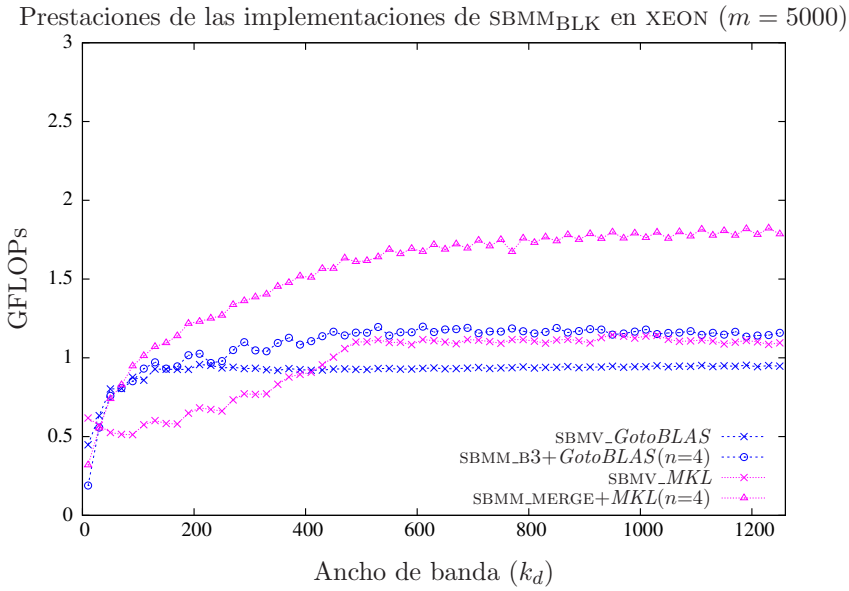


Figura 3.11: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

BLAS paralelo La figura 3.12 muestra los resultados obtenidos con las rutinas *BLAS*-3 y las rutinas incluidas en las implementaciones *BLAS* en estudio. Las prestaciones de las nuevas implementaciones basadas en rutinas *MKL* varían drásticamente con el valor de n. Los resultados obtenidos con $n = 4$ son muy inferiores a los obtenidos con $n = 20$. Naturalmente, el ancho de banda influye en las prestaciones, de forma que al operar con matrices de banda estrecha las prestaciones son reducidas, pero con matrices de banda ancha las nuevas rutinas superan claramente las prestaciones de la rutina SBMV_*MKL*.

Una lectura análoga se realiza de la gráfica que muestra los resultados con *GotoBLAS*. Las nuevas rutinas mejoran claramente a la incluida en la biblioteca *MKL*; así, con $n = 20$ y $k_d = 100$, las rutinas SBMM_B3 y SBMM_MERGE triplican las prestaciones de SBMV_*GotoBLAS*.

La figura 3.13 muestra los resultados de las mejores implementaciones. Las prestaciones más elevadas con $n = 20$ son obtenidas por la rutina SBMM_MERGE+*MKL*. Esta rutina triplica el rendimiento obtenido por la implementación SBMV_*MKL*, que invoca iterativamente a la rutina SBMV incluida en la biblioteca *MKL*.

3.1.5. Conclusiones

De los resultados mostrados en las gráficas anteriores podemos extraer diversas conclusiones comunes para ambas arquitecturas:

- En general las rutinas basadas en la biblioteca *MKL* obtienen mejores prestaciones.
- Si se emplea *GotoBLAS*, la diferencia entre SBMM_B3 y SBMM_MERGE es reducida, pero si se utilizan rutinas de *MKL*, SBMM_MERGE es notablemente más rápida.

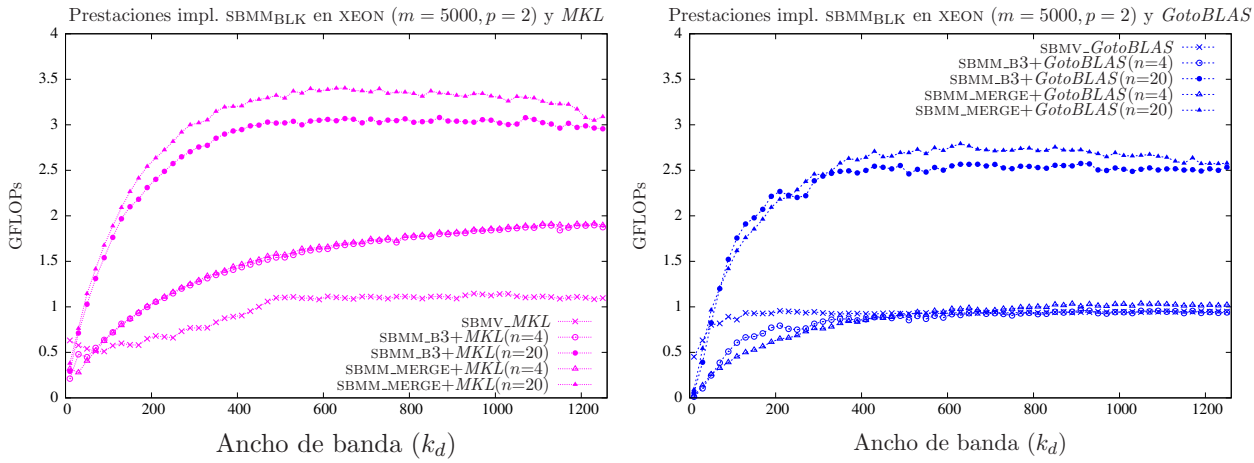


Figura 3.12: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

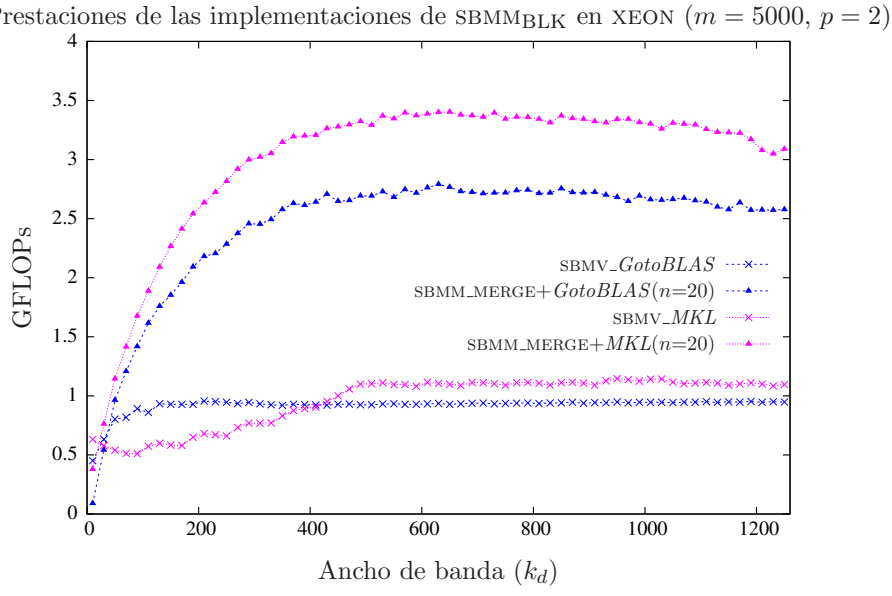


Figura 3.13: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

- Las nuevas rutinas basadas en núcleos computacionales de *BLAS*-3 son más eficientes incluso cuando el valor de n es pequeño. Las nuevas rutinas *BLAS*-3 mejoran notablemente la solución alternativa basada en el uso iterativo de SBMV, logrando fácilmente duplicar la velocidad de cálculo de esta última si se emplean versiones secuenciales de *BLAS*, e incluso triplicar esta velocidad en caso que se emplee una versión paralela.

3.2. Producto de una matriz general banda por una matriz

Consideramos en esta sección el caso general del producto de una matriz banda por una matriz general

$$C := \alpha \cdot op(A) \cdot op(B) + \beta \cdot C \quad \text{o} \tag{3.21}$$
$$C := \alpha \cdot op(B) \cdot op(A) + \beta \cdot C, \tag{3.22}$$

donde $\alpha, \beta \in \mathbb{R}$, $C \in \mathbb{R}^{m \times n}$, y $op(X)$ es un operador que puede transponer o no la matriz sobre la que actúa de modo que $op(X) = X$ o X^T. Además, $op(A) \in \mathbb{R}^{m \times k}$ y $op(B) \in \mathbb{R}^{k \times n}$ en (3.21) mientras que $op(B) \in \mathbb{R}^{m \times k}$ y $op(A) \in \mathbb{R}^{k \times n}$ en (3.22), si bien en ambos casos A presenta una estructura banda con anchos de banda superior e inferior k_u y k_l respectivamente.

Siguiendo la nomenclatura *BLAS*, el nombre de la rutina que implementase el producto de matrices en estudio sería GBMM. En lo sucesivo emplearemos esta nomenclatura. La especificación Fortran-77 que se propone para (la implementación en doble precisión de) esta rutina es la ilustrada en la figura 3.14. El valor del argumento `SIDE` determina cuál de los productos, (3.21) o (3.22), se calcula. Además, los argumentos `TRANSA` y `TRANSB` determinan, respectivamente, si se opera con la transpuesta de las matrices A y B. La intención de los restantes argumentos es la habitual en *BLAS*.

En esta sección se estudia la operación (3.21), con las matrices sin transponer y $\alpha = \beta = 1$, de forma que se obtiene la variante simplificada

$$C := A \cdot B + C, \tag{3.23}$$

donde $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$ y $C \in \mathbb{R}^{m \times n}$; no obstante, el estudio puede ser fácilmente ampliado a los restantes casos.

3.2.1. Implementaciones basadas en GBMV

Una forma simple de implementar el producto especificado en (3.23) es dividir la operación en productos entre la matriz general banda y cada uno de los n vectores columna de B. Esta técnica permite la implementación de GBMM mediante el uso de las rutinas GBMV revisadas en la sección 2.2. La eficiencia de esta solución es equivalente a la de la propia rutina GBMV a la que invoca.

3.2.2. Algoritmo GBMM_{BLK}

La implementación de GBMM mediante la invocación repetida de la rutina GBMV conlleva el uso de núcleos computacionales de *BLAS* de los niveles 1 y/o 2. En cambio, el algoritmo GBMM_{BLK} descrito a continuación calcula la operación (3.23) mediante un proceso iterativo, en el que en cada iteración se realizan operaciones que pertenecen al nivel 3 de *BLAS*.

El algoritmo está formado por dos bucles anidados; el bucle externo (figura 3.15) recorre las matrices B y C de izquierda a derecha, calculando c columnas de C en cada iteración. El bucle

```
      SUBROUTINE DGBMM( SIDE, TRANSA, TRANSB, M, N, K, KL, KU, ALPHA,
                        A, LDA, B, LDB, BETA, C, LDC)
*     .. Scalar Arguments ..
      DOUBLE PRECISION    ALPHA, BETA
      INTEGER             K, KL, KU, LDA, LDB, LDC, M, N
      CHARACTER           SIDE, TRANSA, TRANSB
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, *), B( LDB, *), C( LDC, *)
*
*  Purpose
*  =======
*
*  DGBMM performs one of the matrix-matrix operations
*
*  (1)     C := alpha*op(A)*op(B) + beta*C,
*
*  or
*
*  (2)     C := alpha*op(B)*op(A) + beta*C,
*
*  where  op( X ) is one of
*
*     op( X ) = X   or   op( X ) = X',
*
*  alpha and beta are scalars, A is a band matrix with
*  kl sub-diagonals and ku super-diagonals,
*  and C an m by n matrix.
*
*  In (1) op( A ) is an m by k matrix and op( B ) is a k by n matrix
*  In (2) op( B ) is an m by k matrix and op( A ) is a k by n matrix
*
```

Figura 3.14: Especificación propuesta para la rutina GBMM.

Algorithm: $[C] := \text{GBMM}_{\text{BLK}}(C, A, B, k_u, k_l)$

Partition $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array} \right)$, $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array} \right)$
where C_L, B_L have 0 columns

while $n(C_L) < n(C)$ **do**
Determine block size c
Repartition

$$\left(\begin{array}{c|c} C_L & C_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array} \right), \quad \left(\begin{array}{c|c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right)$$

where C_1, B_1 have c columns

$$C_1 := \text{GBMM}_{\text{INNER\_LOOP}}(C_1, A, B_1, k_u, k_l)$$

Continue with

$$\left(\begin{array}{c|c} C_L & C_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array} \right), \quad \left(\begin{array}{c|c} B_L & B_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right)$$

endwhile

Figura 3.15: Bucle externo del algoritmo por bloques GBMM_{BLK} para la operación $C := A \cdot B + C$.

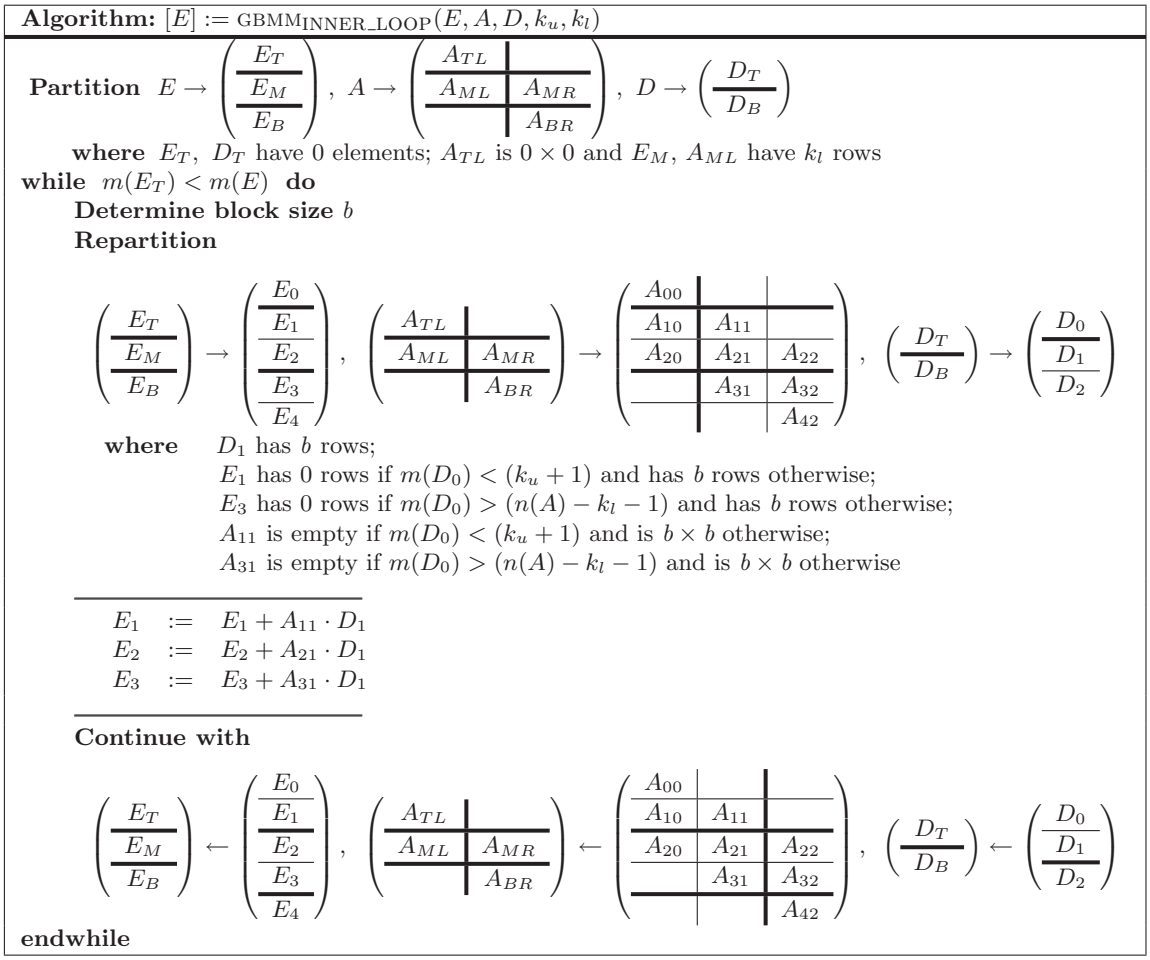


Figura 3.16: Bucle interno del algoritmo por bloques GBMM_{BLK} para la operación $C := A \cdot B + C$.

interno (figura 3.16) recorre la matriz A de izquierda a derecha, operando en cada iteración con los elementos de b columnas de esta matriz y b filas de las matrices C (en E) y B (en D).

Como se puede comprobar, GBMM_{BLK} trabaja con dos tamaños de bloque: c define el número de columnas de C que se computan en cada iteración del bucle externo, y b define el número de columnas de A con las que se opera en cada iteración del bucle interno. La presencia de dos tamaños de bloque presenta dos lecturas; por un lado, el funcionamiento del algoritmo podrá ajustarse mejor a las características de la arquitectura que lo ejecute; por contra, el afinado de los parámetros se dificulta sensiblemente.

La principal ventaja que presenta este algoritmo es que puede ser implementado mediante invocaciones a rutinas del nivel 3 de *BLAS*, circunstancia que facilita un acceso más favorable a los elementos de las matrices y provoca una potencial mejora de las prestaciones.

3.2.3. Implementaciones basadas en rutinas de BLAS-3 denso

La rutina GBMM_B3 implementa el algoritmo GBMM_{BLK} invocando a rutinas de *BLAS*-3 denso. En esta implementación, la mayor parte de las operaciones aritméticas son ejecutadas por núcleos computacionales del nivel 3 de *BLAS*.

En concreto, las operaciones a ejecutar en cada iteración del algoritmo son tres productos de

matrices:

$$E_1 := A_{11} \cdot D_1 + E_1, \tag{3.24}$$
$$E_2 := A_{21} \cdot D_1 + E_2, \tag{3.25}$$
$$E_3 := A_{31} \cdot D_1 + E_3. \tag{3.26}$$

En los productos (3.24) y (3.26) participan sendas matrices triangulares, A_{11} y A_{31} respectivamente, y una matriz densa, en una operación implementada por la rutina TRMM de *BLAS*. La ecuación (3.25) representa un producto entre dos matrices generales, operación soportada por la rutina GEMM. Las dos rutinas aquí comentadas pertenecen al tercer nivel de *BLAS* y son empleadas por GBMM_B3 para codificar el algoritmo GBMM_{BLK}.

El uso de la rutina TRMM precisa de un trabajo extra, ya que en la especificación de *BLAS* ésta almacena el resultado del producto sobre la matriz operando, circunstancia que no se ajusta a lo requerido en (3.24) ni en (3.26). Una solución pasa por almacenar la matriz operando (D_1 en ambos casos) en un espacio de trabajo, W, invocar con este espacio de trabajo a la rutina TRMM, y finalmente acumular en la matriz resultado (E_1 y E_3 respectivamente) el valor de los elementos de W. De esta forma, la secuencia de operaciones a ejecutar (y la rutina *BLAS* asociada a cada una de ellas) será la siguiente:

$$E_1 := A_{11} \cdot D_1 + E_1, \tag{3.27}$$
$$W := D_1, \tag{3.28}$$
(TRMM) $$W := A_{11} \cdot W + W, \tag{3.29}$$
$$E_1 := W + E_1, \tag{3.30}$$
(GEMM) $$E_2 := A_{21} \cdot D_1 + E_2, \tag{3.31}$$
$$E_3 := A_{31} \cdot D_1 + E_3, \tag{3.32}$$
$$W := D_1, \tag{3.33}$$
(TRMM) $$W := A_{31} \cdot W + W, \tag{3.34}$$
$$E_3 := W + E_3. \tag{3.35}$$

Las copias (3.28) y (3.33) son codificadas mediante dos bucles anidados. Estas operaciones podrían ser ejecutadas por la rutina LACPY de *BLAS*, pero el tamaño de los bloques es habitualmente reducido, con lo que el sobrecoste de invocar a LACPY es mayor que la ganancia que esta rutina *BLAS* puede ofrecer. La codificación de (3.30) y (3.35) se realiza igualmente mediante dos bucles anidados.

Estas cuatro operaciones suponen un sobrecoste ligado a la utilización de la rutina *BLAS* TRMM. No obstante, el tamaño de los bloques con los que se operan en ella es reducido y, por lo tanto, también lo es el coste computacional asociado. Por contra, la mayoría de las operaciones aritméticas son ejecutadas por rutinas *BLAS*.

Además, la rutina GBMM_B3, al igual que cualquier rutina que implemente GBMM_{BLK}, exhibe una buena localidad en el acceso a los elementos de las matrices, trabajando con bloques de columnas tanto en la matriz A como en C (E), y con un bloque de elementos de B (D) de tamaño reducido, tal y como se muestra en la figura 3.17.

A pesar de lo comentado, la rutina GBMM presenta ciertas cualidades que animan a la búsqueda de nuevas rutinas más eficientes, en particular, el sobrecoste debido a las operaciones de copia y acumulación ejecutadas sobre los bloques D_1 y D_3, y las invocaciones a TRMM con bloques de dimensión reducida y con bajo coste computacional.

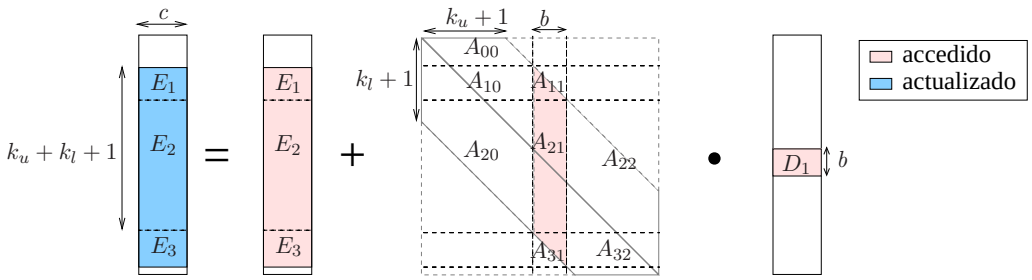


Figura 3.17: Acceso a los elementos en la rutina GBMM_B3.

Implementación merge (GBMM_MERGE)

La rutina GBMM_MERGE trata de eliminar las propiedades negativas de GBMM_B3, al mismo tiempo que mantiene sus mejores características. Esta nueva rutina opera conjuntamente con los bloques A_{21} y A_{31}, reduciendo el número de invocaciones a rutinas *BLAS* a únicamente dos por iteración. Para poder operar de este modo, precisa que el bloque A_{31} se encuentre almacenado completamente según el esquema de almacenamiento para matrices densas y, para ello, toma la región de memoria ocupada por su parte triangular inferior (según este esquema de almacenamiento), guarda una copia de la misma en el espacio de trabajo W, y la rellena con ceros. Tras ello, una llamada a GEMM opera conjuntamente con los bloques A_{21} y A_{31}, y calcula de forma simultánea los bloques E_2 y E_3.

La secuencia de operaciones a ejecutar por GBMM_MERGE es la siguiente:

$$E_1 := A_{11} \cdot D_1 + E_1, \tag{3.36}$$

$$W := D_1, \tag{3.37}$$

(TRMM)
$$W := A_{11} \cdot W + W, \tag{3.38}$$

$$E_1 := W + E_1, \tag{3.39}$$

$$\begin{bmatrix} E_2 \\ E_3 \end{bmatrix} := \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot D_1 + \begin{bmatrix} E_2 \\ E_3 \end{bmatrix}, \tag{3.40}$$

$$W := \text{STRIL}(A_{31}), \tag{3.41}$$

$$\text{STRIL}(A_{31}) := 0, \tag{3.42}$$

(GEMM)
$$\begin{bmatrix} E_2 \\ E_3 \end{bmatrix} := \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot D_1 + \begin{bmatrix} E_2 \\ E_3 \end{bmatrix}, \tag{3.43}$$

$$\text{STRIL}(A_{31}) := W. \tag{3.44}$$

Las operaciones (3.41) y (3.42) se realizan simultáneamente mediante dos bucles anidados, mientras que (3.44) recibe la misma codificación. A diferencia de GBMM_B3, GBMM_MERGE únicamente invoca a dos rutinas *BLAS* por cada iteración y GEMM opera con bloques de mayor dimensión. Lo que consigue con ello es que el grado de paralelismo encontrado por esta rutina sea mayor, pudiendo explotar mejor los beneficios ofrecidos por las versiones paralelas de *BLAS*. Al mismo tiempo se evita invocar a TRMM con un número reducido de datos, circunstancia ésta en la que las rutinas *BLAS*-3 no son capaces de alcanzar altas prestaciones. Además, al igual que hiciera GBMM, esta rutina realiza un acceso a memoria eficiente.

De nuevo, la rutina presenta un sobrecoste ocasionado por las operaciones de copia y acumulación. Las operaciones que provocan este sobrecoste son todas ellas, a excepción de (3.39), copias ejecutadas sobre bloques de pequeño tamaño y, por lo tanto, con un bajo coste computacional.

3.2.4. Resultados experimentales

Arquitectura ITANIUM

BLAS secuencial En esta sección se han presentado diferentes formas de implementación del producto entre una matriz general banda y una matriz densa. Las primeras versiones propuestas invocan repetidamente a la rutina GBMV, que implementa el producto entre una matriz general banda y un vector. En consecuencia, estas implementaciones tienen las mismas prestaciones que la rutina GBMV a la que invocan.

La figura 3.18 muestra las prestaciones de las implementaciones del algoritmo GBMM_{BLK} expuestas en la sección, tanto si se basan en rutinas de *MKL* (gráfica de la izquierda) como si emplean rutinas de *GotoBLAS* (gráfica de la derecha).

Como se muestra en la figura, los resultados de las nuevas implementaciones dependen en gran medida del ancho de banda de la matriz A y del número de columnas de las matrices C y B (n). Al emplear núcleos computacionales de *MKL*, las nuevas rutinas superan los resultados de GBMV_MKL (la rutina que invoca repetidamente a la rutina GBMV incluida en *MKL*) con valores de n y $k_u(= k_l)$ reducidos. Por ejemplo, si n=4, entonces las nuevas rutinas *BLAS*-3 mejoran a GBMV_*MKL* con $k_u = k_l > 150$, mientras que si $n = 20$ la mejoran para cualquier ancho de banda. Sin embargo, los mayores beneficios de las rutinas *BLAS*-3 aparecen cuando los valores del ancho de banda y de n son superiores, de forma que con $n = 20$ y $k_l = k_u = 1250$ las nuevas rutinas son aproximadamente seis veces más rápidas que la invocación repetida de GBMV_MKL.

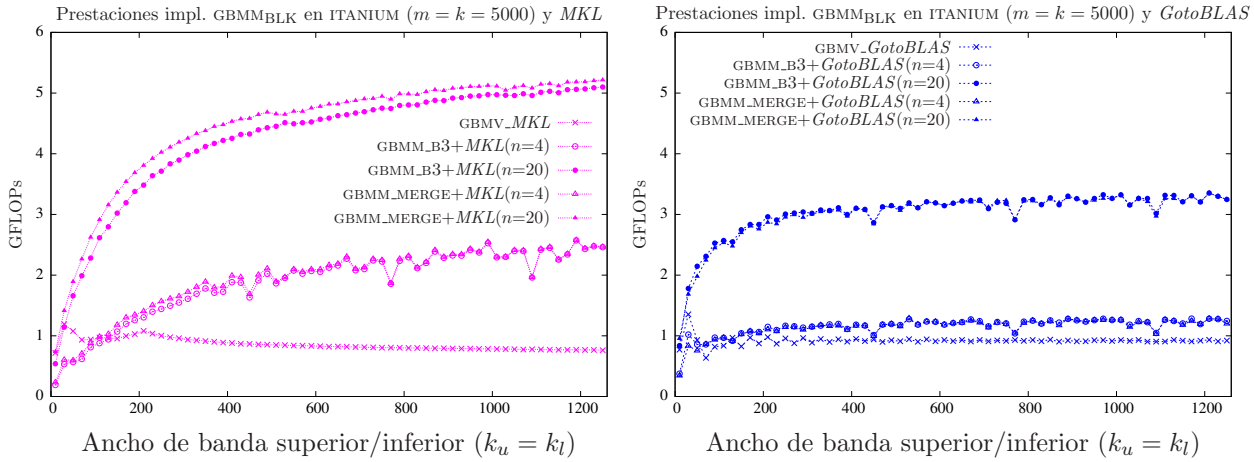


Figura 3.18: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

En el caso de *GotoBLAS*, los resultados muestran una gran variación en función del valor de n y también del ancho de banda de A. Cuando $n = 4$ las nuevas rutinas mejoran a GBMV_GOTO con anchos de banda superiores a 100, mientras que cuando $n = 20$ la superan para cualquier ancho de banda. Las mejores prestaciones se obtienen con $n = 20$ y $k_u = k_l = 1250$, llegando a ser aproximadamente cuatro veces mayores que las de GBMV_GOTO.

Si se comparan las nuevas rutinas enlazadas con *MKL* y *GotoBLAS*, para matrices de banda estrecha GBMM_B3 se revela como la mejor opción, mientras que para matrices con ancho de banda media o ancha, GBMM_MERGE iguala a GBMM_B3 en el caso de *GotoBLAS* y es la rutina más eficiente en el caso de *MKL*.

Finalmente, la figura 3.19 recoge los resultados de las mejores implementaciones *BLAS*-3 para GBMM. Se muestran en ella, además, las implementaciones derivadas del uso iterativo de las rutinas

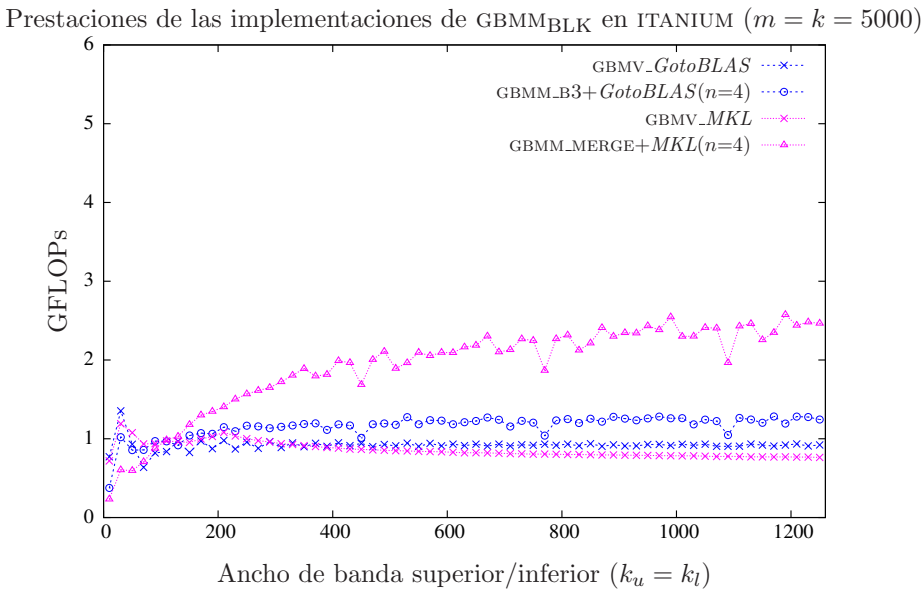


Figura 3.19: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

GBMV incluidas en *MKL* y *GotoBLAS*. Por simplicidad se han reproducido únicamente los resultados para $n = 4$. Como se observa en la figura, las rutinas *BLAS*-3 son notablemente más eficientes cuando A es una matriz de banda media o ancha, mientras que si es una matriz de banda estrecha, las nuevas rutinas no alcanzan buenas prestaciones.

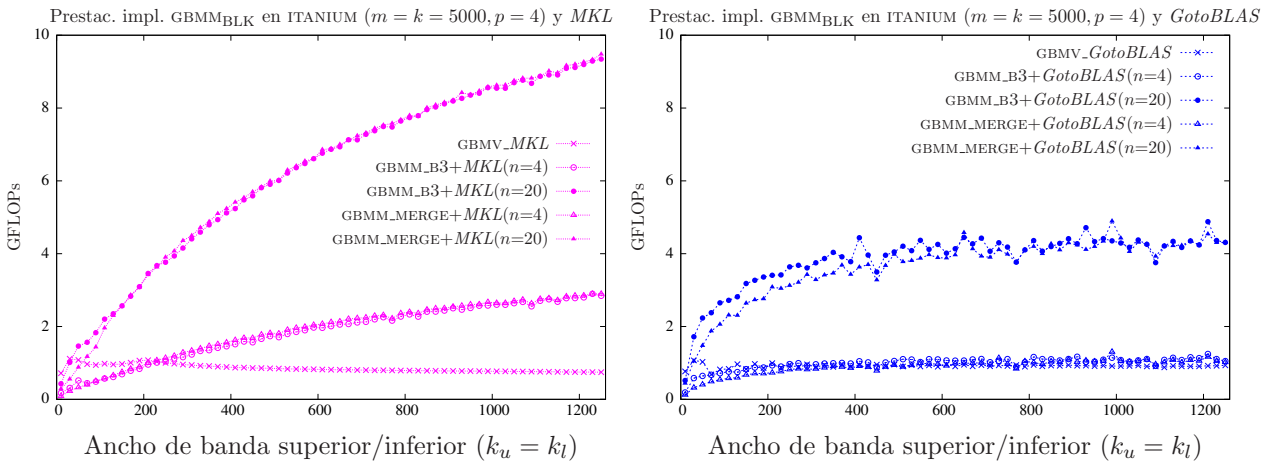


Figura 3.20: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

BLAS paralelo La figura 3.20 muestra los resultados de las diferentes implementaciones basadas en *BLAS*-3 de la rutina GBMM utilizando versiones paralelas de *BLAS*. En este caso, los valores de n y de $k_u + k_l$ influyen más aún si cabe en los resultados obtenidos por las nuevas rutinas, ya que cuanto más grandes son estos valores, mayor es el nivel de paralelismo así como el nivel de utilización de los recursos *hardware*. Por contra, cuando tanto n como $k_u + k_l$ toman valores reducidos, el coste introducido por la creación y sincronización de las hebras de ejecución es mayor

que los beneficios que aporta el paralelismo, de forma que, en estos casos, el uso de un *BLAS* secuencial es más eficiente.

Los resultados para *MKL* (gráfica de la izquierda) muestran que las nuevas rutinas pueden ser hasta once veces más eficientes que el uso repetitivo de GBMV. En el caso de *GotoBLAS* esta diferencia se reduce a cuatro. Las nuevas rutinas presentan comportamientos diferentes al emplear núcleos computacionales de *MKL* o de *GotoBLAS*; mientras que con *GotoBLAS* las prestaciones mejoran rápidamente con el ancho de banda y se estabilizan cuando $k_u = k_l = 500$, con *MKL* las prestaciones crecen de una forma más lenta con el valor de k_u y k_l pero no se detienen como en el caso de *GotoBLAS*. La consecuencia de estas diferencias es que, para matrices de banda estrecha, es preferible emplear *GotoBLAS*, mientras que al operar con matrices de banda media o ancha *MKL* obtiene prestaciones muy superiores.

Para completar este estudio, la figura 3.21 recoge los resultados de las mejores implementaciones *BLAS*-3 y de las implementaciones basadas en las rutinas GBMV incluidas en las implementaciones *BLAS MKL* y *GotoBLAS*. En esta gráfica quedan patentes los variados rendimientos de las implementaciones basadas en *MKL* y en *GotoBLAS*. Esto provoca que, al operar con matrices de banda estrecha, la mejor implementación con *GotoBLAS* duplique la eficiencia de la mejor implementación con *MKL* y que suceda exactamente lo contrario cuando se opera con una matriz de banda ancha.

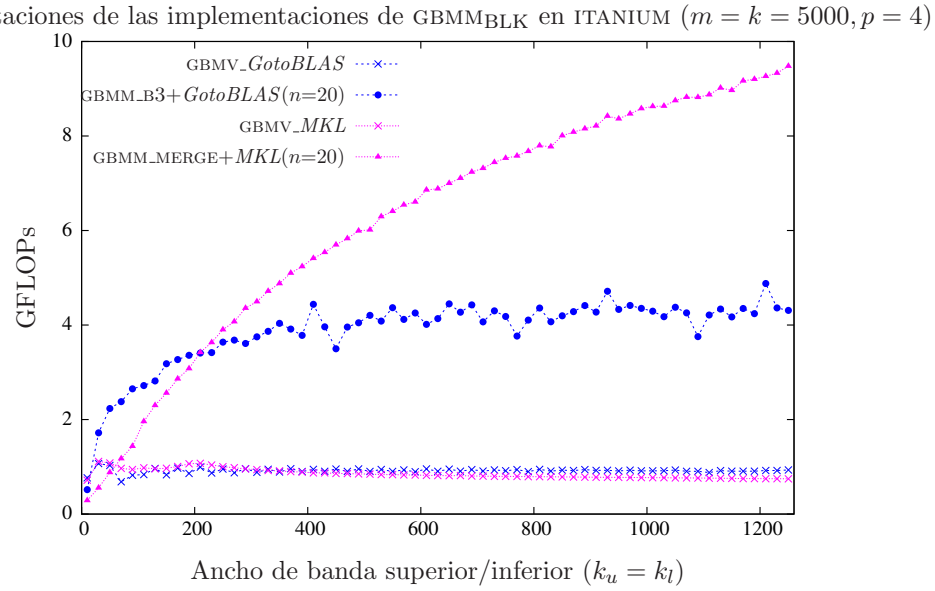


Figura 3.21: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

Arquitectura XEON

Sobre la arquitectura XEON se han experimentado las nuevas rutinas basadas en núcleos de *BLAS*-3, comparándolas con la invocación repetida de las rutinas GBMV incluidas en las bibliotecas *BLAS* en estudio, *MKL* y *GotoBLAS*.

En la figura 3.22 se muestran los resultados de las nuevas rutinas *BLAS*-3 cuando éstas invocan rutinas de *MKL* y *GotoBLAS*. En ambos casos, la eficiencia depende del ancho de banda de la matriz A, de forma que crece rápidamente conforme aumenta éste y se estabiliza para valores de $k_u (= k_l)$ superiores a 200. El otro factor que determina la eficiencia de estas rutinas es el valor de n; así, cuando $n = 20$ las prestaciones son hasta un 65 % mayores para *MKL* y hasta un 100 %

superiores en *GotoBLAS*. Ambas rutinas *BLAS*-3 obtienen prestaciones similares, siendo GBMM_B3 ligeramente superior para *GotoBLAS* y GBMM_MERGE para *MKL*.

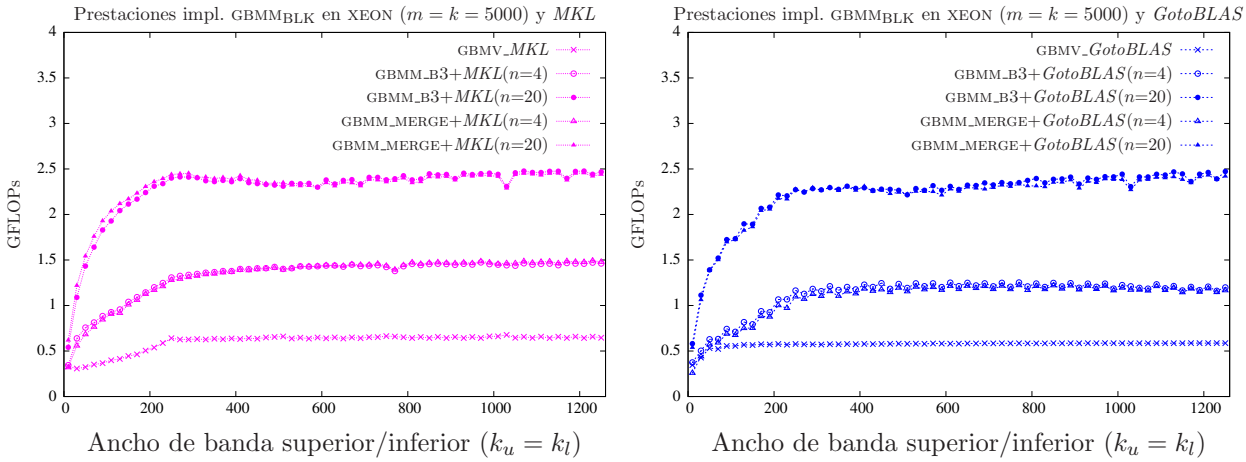


Figura 3.22: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

BLAS secuencial Para facilitar el análisis, la figura 3.23 muestra únicamente los resultados de las mejores implementaciones, GBMM_B3+*GotoBLAS* y GBMM_MERGE+*MKL*, para $n = 4$, junto con las derivadas del uso repetitivo de las rutinas GBMV_GOTO y GBMV_MKL. Como se puede observar, incluso para valores pequeños de n, las nuevas implementaciones son más rápidas que la invocación repetida de las rutinas GBMV. La implementación GBMM_B3+*GotoBLAS* se presenta como la mejor opción cuando $k_u = k_l < 30$, mientras que para anchos de banda mayores GBMM_MERGE es la rutina con mayores prestaciones.

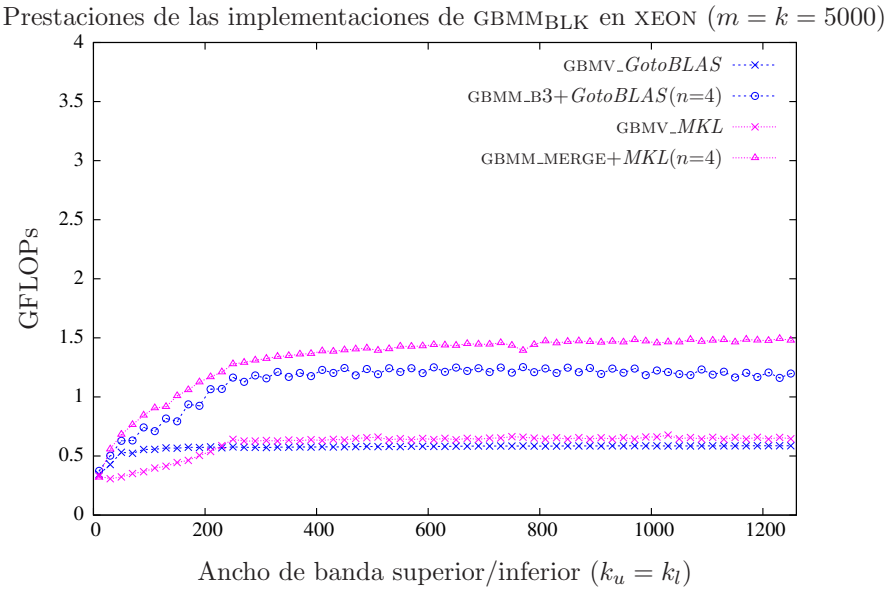


Figura 3.23: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

BLAS paralelo La figura 3.24 recoge las prestaciones de las nuevas rutinas cuando invocan a implementaciones paralelas de *BLAS*.

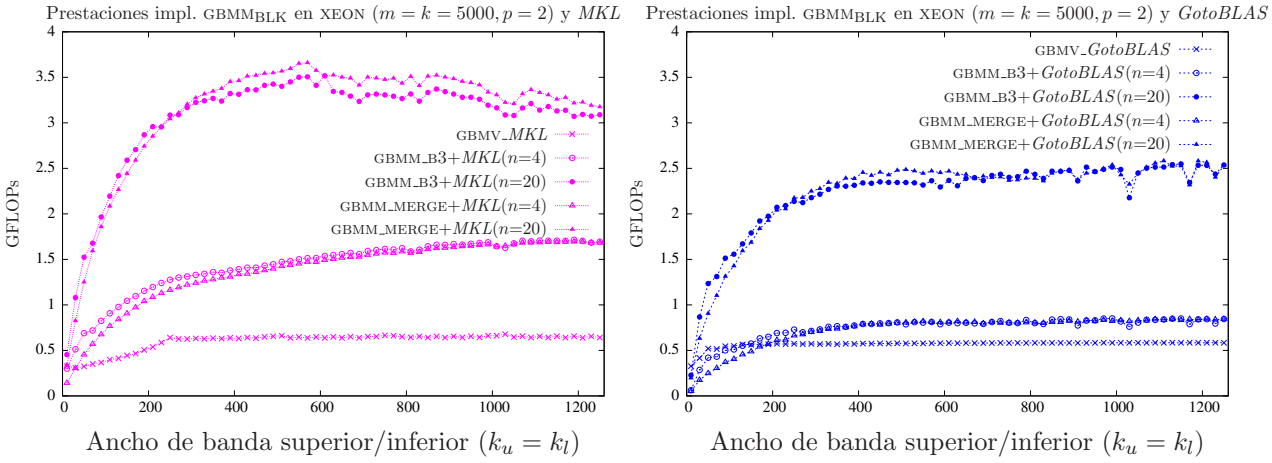


Figura 3.24: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

Como figura en la gráfica de la izquierda, correspondiente a los resultados para *MKL*, tanto GBMM_B3 como GBMM_MERGE obtienen resultados muy superiores a GBMV_MKL. GBMM_B3 es ligeramente más eficiente que GBMM_MERGE cuando A es una matriz de banda estrecha, pero cuando A es una matriz de banda media o ancha es GBMM_MERGE la rutina con mejores prestaciones. Como se ha comentado con anterioridad, GBMM_MERGE aumenta el nivel de paralelismo, lo que posiblita una mejor utilización de los recursos en un entorno paralelo como el planteado. No obstante, para ello precisa un mínimo coste computacional que no se alcanza cuando A es una matriz de banda estrecha y n toma valores tan pequeños como 4 ó 20.

Una lectura similar se puede hacer de la gráfica de la derecha, donde se muestran los resultados obtenidos al invocar rutinas de *GotoBLAS*.

Como queda patente en la figura 3.24, las implementaciones basadas en *MKL* obtienen mayores prestaciones que las basadas en *GotoBLAS*.

La figura 3.25 muestra los resultados de las mejores implementaciones de esta operación. Se ofrecen únicamente los resultados cuando $n = 20$, puesto que la carga computacional y el nivel de paralelismo cuando $n = 4$ es menor. Tanto para *GotoBLAS* como para *MKL* la mejor implementación basada en *BLAS*-3 es GBMM_MERGE, aunque como se ha visto en la figura anterior, cuando A es una matriz de banda estrecha GBMM_B3 es ligeramente más rápida. En el caso de *GotoBLAS*, GBMM_MERGE es más de cuatro veces más eficiente que GBMV_GOTO, mientras que en el caso de *MKL* la diferencia es todavía mayor, siendo hasta seis veces más rápida la nueva rutina GBMM_MERGE que GBMV_MKL.

3.2.5. Conclusiones

En el presente estudio se han evaluado dos implementaciones *BLAS*-3 para la operación (3.23), comparando sus resultados con la única implementación posible a partir de las rutinas incluidas en la especificación *BLAS*, consistente en la invocación repetida de la rutina GBMV. Los resultados indican que el uso de rutinas basadas en *BLAS*-3 para la operación en estudio mejora enormemente las prestaciones, especialmente cuando la biblioteca empleada es una implementación paralela de *BLAS*. No obstante, mejoras ostensibles también son alcanzadas con versiones secuenciales de *BLAS* incluso cuando A es una matriz de banda media y el número de columnas de C es bastante pequeño.

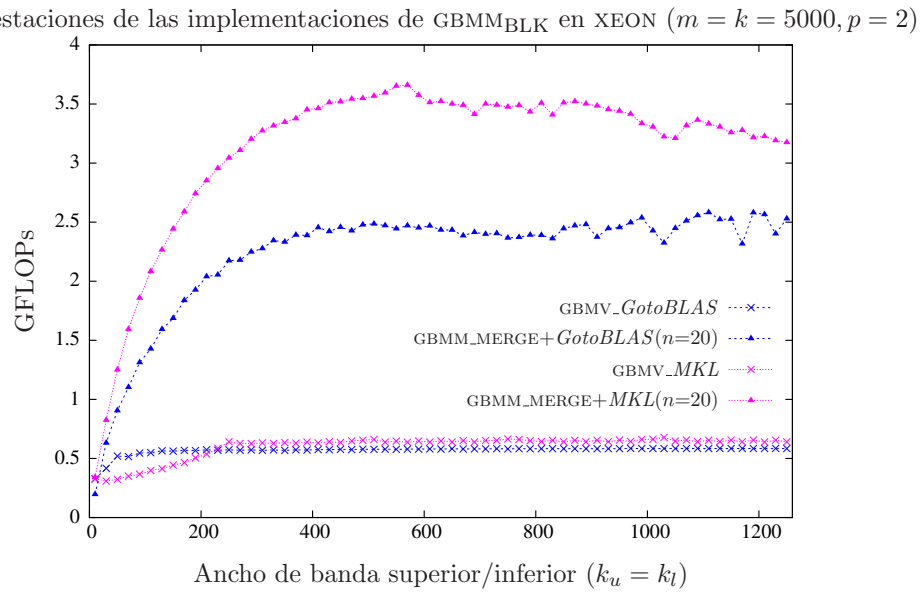


Figura 3.25: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

El uso de *MKL* es recomendable para esta operación tanto en ITANIUM como en XEON, con la salvedad del caso en que A es una matriz de banda estrecha en ITANIUM. La superioridad de GBMM_B3 o GBMM_MERGE depende básicamente de la biblioteca *BLAS* utilizada y de la arquitectura, aunque en todos los casos las diferencias entre las prestaciones obtenidas por ambas implementaciones son reducidas. En ITANIUM, la nueva implementación GBMM_B3 es la mejor opción cuando se enlaza con la biblioteca *GotoBLAS*, mientras que la nueva GBMM_MERGE es la implementación más eficiente para *MKL*. En el caso de XEON, en general GBMM_MERGE obtiene mejores prestaciones y GBMM_B3 únicamente se presenta como la mejor opción en el caso secuencial cuando se emplea *GotoBLAS* y A es una matriz de banda estrecha.

3.3. Producto de una matriz triangular banda por una matriz

En esta sección se aborda el producto

$$B := \alpha \cdot op(A) \cdot B \quad \text{o} \tag{3.45}$$

$$B := \alpha \cdot B \cdot op(A), \tag{3.46}$$

donde $op(X) = X$ o X^T, $B \in \mathbb{R}^{m \times n}$, y $op(A) \in \mathbb{R}^{m \times m}$ en (3.45) u $op(A) \in \mathbb{R}^{n \times n}$ en (3.46) es una matriz triangular banda.

Siguiendo el estándar *BLAS*, la nomenclatura que se propone para esta rutina es TBMM y su especificación puede encontrarse en la figura 3.26. El valor del argumento `SIDE` especifica cuál de los productos en (3.45)–(3.46) se calcula. Además, los valores de `UPLO` y `TRANS` determinan, respectivamente, si se opera con una matriz triangular superior o inferior, y si ésta aparece en el producto como transpuesta o no. El cuarto argumento, `DIAG`, permite especificar que todos los elementos de la diagonal de la matriz A toman el valor 1. El significado de los restantes argumentos es el habitual en las rutinas de *BLAS*.

Por simplicidad, durante el resto de esta sección consideraremos únicamente la formulación de la operación en (3.45), con la matriz A triangular inferior banda sin transponer, y con la diago-

```
      SUBROUTINE DTBMM( SIDE, UPLO, TRANS, DIAG, M, N, K, ALPHA,
                        A, LDA, B, LDB )
*     .. Scalar Arguments ..
      DOUBLE PRECISION   ALPHA
      INTEGER            K, LDA, LDB, M, N
      CHARACTER          DIAG, SIDE, TRANS, UPLO
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, *), B( LDB, *)
*
*  Purpose
*  =======
*
*  DTBMM performs one of the matrix-matrix operations
*
*  (1)     B := alpha*op(A)*B,
*
*  or
*
*  (2)     B := alpha*B*op(A),
*
*  where  op( X ) is one of
*
*     op( X ) = X   or   op( X ) = X',
*
*  A is a lower triangular band matrix with k sub-diagonals
*  or an upper triangular band matrix with k super-diagonals,
*  and B an m by n matrix.
*
*  In (1) op( A ) is an m by m matrix
*  In (2) op( A ) is an n by n matrix
*
```

Figura 3.26: Especificación propuesta para la rutina TBMM.

nal conteniendo valores que puede ser distintos a 1. De este modo, la operación simplificada que obtenemos responde a la expresión

$$B := A \cdot B. \tag{3.47}$$

A pesar de estas simplificaciones, el estudio mostrado a continuación puede ser fácilmente extendido al resto de casos.

3.3.1. Implementaciones basadas en TBMV

La operación en (3.47) puede ser descompuesta en n productos de una matriz triangular banda por vector, siendo n el número de columnas de la matriz B. Así, cada producto matriz-vector se puede ejecutar invocando la rutina TBMV. Aplicando este algoritmo, es posible obtener una implementación de la rutina TBMM por cada implementación de TBMV (ver sección 2.3).

3.3.2. Algoritmo TBMM_{BLK}

El algoritmo especificado en las figuras 3.27 y 3.28, compuesto por dos bucles, permite la utilización de rutinas *BLAS*-3 para la ejecución de la operación en estudio.

Algorithm: $[B] := \text{TBMM}_{\text{BLK}}(B, A, k_l)$

Partition $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array}\right)$
where B_L has 0 column;

while $n(B_L) < n(B)$ **do**
Determine block size c
Repartition

$$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$
where B_1 has c columns

$B_1 := \text{TBMM}_{\text{INNER\_LOOP}}(B_1, A, k_l)$

Continue with

$$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$
endwhile

Figura 3.27: Bucle externo del algoritmo por bloques TBMM_{BLK} para la operación $B := A \cdot B$.

Algorithm: $[C] := \text{TBMM}_{\text{INNER\_LOOP}}(A, C, k_l)$

Partition $x \rightarrow \left(\begin{array}{c} C_T \\ \hline C_M \\ \hline C_B \end{array}\right)$, $A \rightarrow \left(\begin{array}{c|c|c} A_{TL} & & \\ \hline A_{ML} & A_{MM} & \\ \hline & A_{BM} & A_{BR} \end{array}\right)$
where C_B and C_M are 0×0; A_{BR} and A_{MM} are 0×0
while $m(C_T) > 0$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c} C_T \\ \hline C_M \\ \hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \\ \hline C_3 \\ \hline C_4 \end{array}\right), \left(\begin{array}{c|c|c} A_{TL} & & \\ \hline A_{ML} & A_{MM} & \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & & & & \\ \hline A_{10} & A_{11} & & & \\ \hline A_{20} & A_{21} & A_{22} & & \\ \hline & A_{31} & A_{32} & A_{33} & \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right)$$

where C_1 has b_T rows; A_{11} is $b_T \times b_T$;
C_2 has b_M rows; A_{22} is $b_M \times b_M$,
with $b_T := \text{mín}(b, m(C_T))$ and $b_M := \text{mín}(k_l - b, m(C_M))$

$C_2 := A_{21} \cdot C_1 + C_2$
$C_3 := A_{31} \cdot C_1 + C_3$
$C_1 := A_{11} \cdot C_1$

Continue with

$$\left(\begin{array}{c} C_T \\ \hline C_M \\ \hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \\ \hline C_3 \\ \hline C_4 \end{array}\right), \left(\begin{array}{c|c|c} A_{TL} & & \\ \hline A_{ML} & A_{MM} & \\ \hline & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & & & & \\ \hline A_{10} & A_{11} & & & \\ \hline A_{20} & A_{21} & A_{22} & & \\ \hline & A_{31} & A_{32} & A_{33} & \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right)$$

endwhile

Figura 3.28: Bucle interno del algoritmo por bloques TBMM_{BLK} para la operación $B := A \cdot B$.

El bucle externo (figura 3.27) recorre la matriz B de izquierda a derecha, tomando en cada iteración c columnas de B. El bucle interno (figura 3.28) opera con los elementos de estas columnas y

con la matriz A al completo, recorriendo completamente esta última matriz a lo largo de su diagonal, de derecha a izquierda (comenzando por el extremo inferior derecho de la matriz y terminando en el superior izquierdo). En cada iteración del bucle, se opera con b columnas de A y se calculan los elementos de b filas de C (según la notación de la figura 3.28, B_1 en la figura 3.27), al tiempo que se actualizan los elementos de las k_l filas anteriores.

La matriz A se recorre en sentido inverso al habitual para eliminar las dependencias de datos presentes entre las operaciones. Con este mismo propósito, la última operación que se realiza en cada iteración es aquella que modifica los elementos de C_1, ya que el resto de operaciones precisan los valores originales contenidos en este bloque.

Los tres cálculos ejecutados en cada iteración son productos matriz-matriz, operaciones todas del nivel 3 de *BLAS*. Dos de los productos, aquellos que involucran a los bloques A_{11} y A_{31}, son de tipo matriz triangular-matriz, mientras que el restante es un producto entre dos matrices generales. Las rutinas *BLAS* que implementan estas operaciones son TRMM y GEMM respectivamente.

3.3.3. Implementaciones basadas en rutinas de BLAS-3 denso

El uso de la rutina TRMM precisa de un trabajo adicional, ya que esta rutina almacena el resultado en la matriz operando, con lo que para poder utilizarla será necesario obtener una copia del bloque C_1 (mediante la rutina LACPY), operar con la copia, y después actualizar C_3 con el resultado de TRMM (mediante dos bucles anidados). La rutina TBMM_B3 implementa este algoritmo realizando las operaciones tal y como se muestra seguidamente:

$$
\begin{aligned}
&(\text{GEMM}) && C_2 && := A_{21} \cdot\ C_1 + C_2, && (3.48)\\
& && C_3 && := A_{31} \cdot\ C_1 + C_3, && (3.49)\\
&(\text{LACPY}) && && W := C_1, && (3.50)\\
&(\text{TRMM}) && && W := A_{31} \cdot\ W, && (3.51)\\
& && && C_3 := C_3 + W, && (3.52)\\
&(\text{TRMM}) && C_1 && := A_{11} \cdot\ C_1. && (3.53)
\end{aligned}
$$

De esta forma, la mayor parte de las operaciones aritméticas son ejecutadas por rutinas de *BLAS*-3 denso, asegurando la obtención de altas prestaciones. No obstante, TBMM_B3 puede mejorarse puesto que:

- Invoca a tres rutinas *BLAS* por iteración, propiciando operaciones con bloques pequeños y reduciendo con ello el grado de paralelismo.
- Realiza las operaciones (3.50) y (3.52) que, en principio, no eran necesarias. Esto justifica el desarrollo de rutinas alternativas.

Implementación Merge (TBMM_B3_MERGE)

La rutina TBMM_B3_MERGE reduce el número de invocaciones a rutinas *BLAS* por iteración, de forma que se opere con bloques de mayor tamaño. Esto se consigue uniendo A_{21} y A_{31} en un sólo bloque y actuando de igual forma con los bloques C_2 y C_3. Al proceder de este modo, las operaciones (3.48) y (3.49) se pueden ejecutar mediante una única llamada a la rutina *BLAS* GEMM. La rutina TBMM_B3_MERGE simula que la existencia de un bloque rectangular completo formado por $[A_{21};\ A_{31}]$, actuando sobre los elementos donde la parte triangular inferior estricta

de A_{31} se encuentran almacenados. La secuencia de operaciones ejecutadas por esta rutina es la siguiente:

$$\begin{bmatrix} C_2 \\ C_3 \end{bmatrix} := \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot C_1 + \begin{bmatrix} C_2 \\ C_3 \end{bmatrix}, \tag{3.54}$$

$$W := \text{STRIL}(A_{31}), \tag{3.55}$$

$$\text{STRIL}(A_{31}) := 0, \tag{3.56}$$

$$\text{(GEMM)} \quad \begin{bmatrix} C_2 \\ C_3 \end{bmatrix} := \begin{bmatrix} C_2 \\ C_3 \end{bmatrix} - \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot C_1, \tag{3.57}$$

$$\text{STRIL}(A_{31}) := W \tag{3.58}$$

$$\text{(TRMM)} \quad C_1 := A_{11} \cdot C_1. \tag{3.59}$$

Como ventaja respecto a TBMM_B3, en esta nueva rutina se invoca únicamente a dos rutinas *BLAS* en cada iteración y además se realizan operaciones con bloques de mayor tamaño. Las operaciones (3.55) y (3.56) se ejecutan simultáneamente mediante dos bucles anidados y de igual forma se implementa la operación (3.58). Al igual que sucede con TBMM_B3, el sobrecoste debido a las copias adicionales es reducido, ya que el tamaño de los bloques a copiar depende de b, que habitualmente toma valores reducidos.

3.3.4. Resultados experimentales

Arquitectura ITANIUM

BLAS secuencial Hasta el momento se han mostrado dos posibilidades para la implementación de la rutina TBMM: la primera de ellas consiste en invocar repetidamente a la rutina TBMV y la segunda es la codificción del algoritmo TBMM_{BLK} obteniendo una rutina *BLAS*-3. Se han descrito dos posibles codificaciones para este último caso, TBMM_B3 y TBMM_MERGE.

En este estudio se incluyen las nuevas implementaciones *BLAS*-3 para la operación (3.47), comparando sus prestaciones con las obtenidas al invocar repetidamente a la rutina TBMV incluida en las implementaciones *BLAS MKL* (rutina TBMV_MKL) y *GotoBLAS* (TBMV_GOTO). Las prestaciones de las rutinas TBMV_MKL y TBMV_GOTO no varían con el valor de n, pero sí las prestaciones de las rutinas *BLAS*-3; por este motivo, se han evaluado dos valores para n, 4 y 20. Estos valores son especialmente pequeños, hecho que no beneficia a las rutinas *BLAS*-3, pero que permiten constatar que, incluso en estas condiciones, las nuevas rutinas pueden competir con el uso repetido de la rutina TBMV.

La figura 3.29 recoge los resultados de las diferentes implementaciones de TBMM_{BLK} propuestas en la sección cuando los núcleos computacionales invocados pertenecen a *MKL* o *GotoBLAS*. En ambos casos queda patente la relevancia que el valor de n tiene en las prestaciones obtenidas, ya que si bien con $n = 4$ las nuevas rutinas duplican las prestaciones de la invocación repetida de la rutina TBMV de *BLAS*, con $n = 20$ esta diferencia se incrementa mucho más, llegando las nuevas rutinas a ser hasta seis veces más rápidas. Los incrementos en el valor de n posibilitan a las rutinas basadas en *BLAS*-3 aumentar el número de operaciones ejecutadas por cada acceso a memoria, de forma que las prestaciones crecen considerablemente.

Las implementaciones basadas en *MKL* alcanzan grandes prestaciones con matrices de banda media o ancha, mientras que las basadas en *GotoBLAS* son preferibles cuando A es una matriz de banda estrecha, especialmente cuando $n = 20$.

Por último, la figura 3.30 compila los resultados de las mejores versiones de TBMM tanto para *GotoBLAS* como para *MKL*. Para mayor simplicidad se han incluido únicamente los resultados para

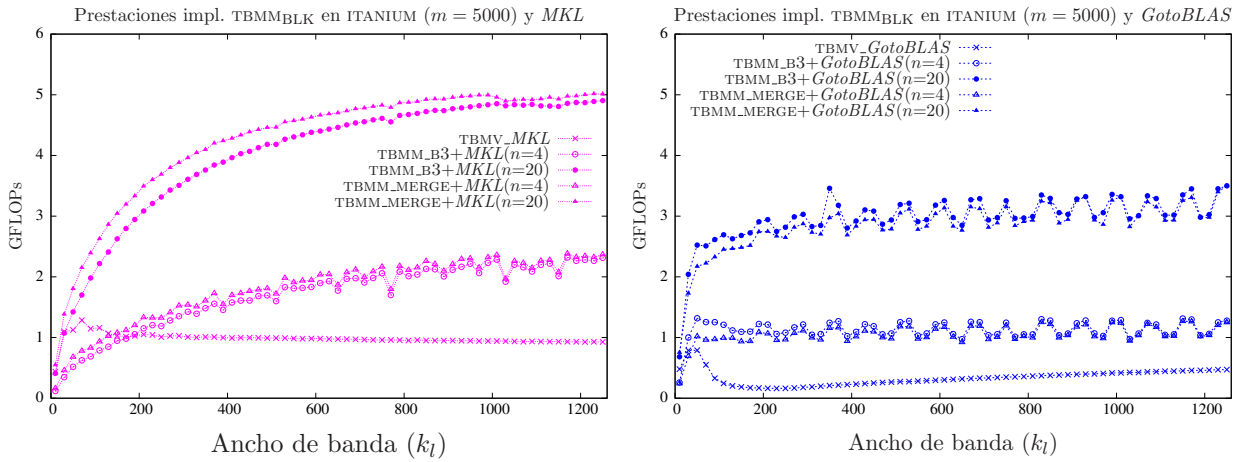


Figura 3.29: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

$n = 4$. La gráfica muestra la eficiencia de TBMM_B3, ya que es esta rutina, utilizando la biblioteca *GotoBLAS*, la que obtiene los mejores resultados cuando A es una matriz de banda estrecha, mientras que al operar con matrices de banda media o ancha es de nuevo TBMM_MERGE+*MKL* la opción que presenta las mejores prestaciones.

BLAS paralelo A continuación se evalúan las diferentes implementaciones de TBMM$_{\text{BLK}}$ cuando éstas invocan a rutinas de una versión paralela de *BLAS*.

En la figura 3.31 se muestran los resultados generados por las nuevas rutinas *BLAS*-3 utilizando versiones paralelas de *BLAS*. Al igual que sucedía con la utilización de versiones secuenciales de *BLAS*, las implementaciones basadas en rutinas de *GotoBLAS* obtienen mejores prestaciones cuando A es una matriz de banda estrecha; sin embargo, cuando la matriz A es una matriz de banda media o ancha, el uso de rutinas de *MKL* posibilita la obtención de prestaciones muy superiores. De nuevo, la eficiencia de las nuevas rutinas varía notablemente con el valor de n; cuando n toma un valor mayor, las rutinas *BLAS*-3 realizan un acceso más óptimo a los elementos de la matriz A y además hay un mayor nivel de paralelismo. En consecuencia, las prestaciones cuando $n = 20$ son hasta tres veces superiores a las obtenidas con $n = 4$. Estas mismas razones explican la aceleración superlineal obtenida cuando A es una matriz de banda ancha y $n = 20$.

La figura 3.32 recoge las prestaciones de las mejores implementaciones de TBMM$_{\text{BLK}}$ cuando $n = 20$ y las compara con el uso iterativo de las rutinas TBMV incluidas en las bibliotecas *MKL* y *GotoBLAS*. Como puede verse, las nuevas rutinas son más rápidas, especialmente TBMM_B3. Esta nueva rutina, usada conjuntamente con *GotoBLAS*, es la opción más eficiente cuando el ancho de banda de la matriz A inferior es menor que 400, mientras que para anchos de banda superiores, su uso junto con *MKL* alcanza las mayores prestaciones. Como se puede observar, las prestaciones de las implementaciones basadas en las rutinas TBMV son muy inferiores.

Arquitectura XEON

BLAS secuencial Seguidamente se analizan las prestaciones de las nuevas rutinas *BLAS*-3, comparándolas con las rutinas derivadas del uso iterativo de las implementaciones de TBMV incluidas en las bibliotecas *MKL* y *GotoBLAS*.

Los resultados de esta evaluación se resumen en la figura 3.33. La gráfica de la izquierda muestra los resultados al emplear rutinas de *MKL*, mientras que la gráfica de la derecha muestra los

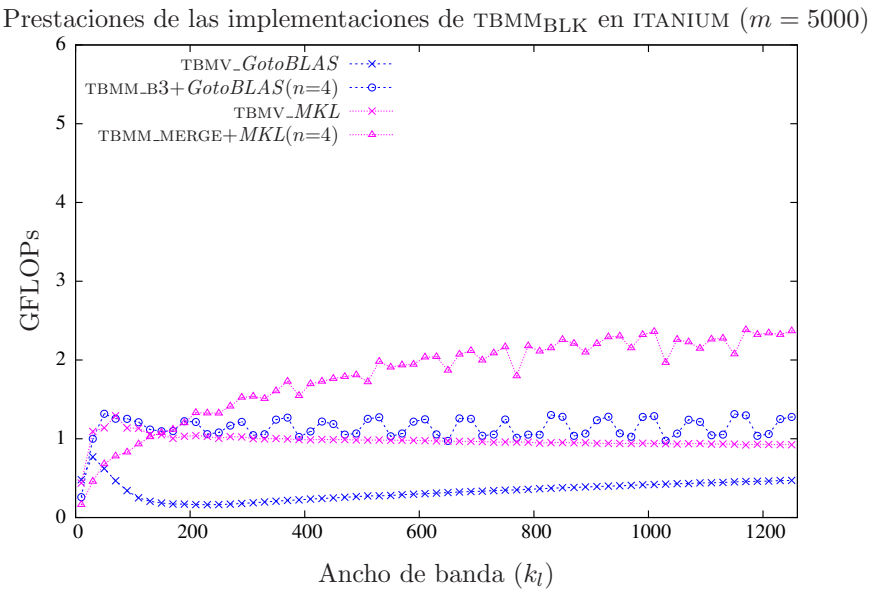


Figura 3.30: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

resultados al invocar a rutinas de *GotoBLAS*.

En ambos casos las nuevas rutinas mejoran notablemente las prestaciones de las rutinas basadas en TBMV, incluso cuando n toma un valor tan reducido como 4 y A es una matriz de banda estrecha, circunstancias éstas poco favorables para las rutinas *BLAS*-3. La implementación TBMM_MERGE es ligeramente más eficiente que TBMM_B3 al emplear rutinas de *MKL*. Por contra, al emplear núcleos de *GotoBLAS*, ambas implementaciones obtienen similares prestaciones.

La figura 3.34 muestra las mejores implementaciones vistas en la figura 3.33 cuando $n = 4$. La figura revela que la nueva rutina TBMM_MERGE es la más eficiente independientemente del ancho de banda de la matriz A, llegando a ser prácticamente tres veces más rápida que la aproximación tradicional.

BLAS paralelo En este estudio también se evalúan las nuevas rutinas utilizando implementaciones paralelas de *BLAS*. La figura 3.35 refleja sus resultados junto a los de la utilización repetida de las rutinas TBMV incluidas en *MKL* y *GotoBLAS*. La gráfica de la izquierda incluye los resultados al utilizar rutinas de *MKL*. TBMM_B3+*MKL* es la implementación más efectiva de las evaluadas con la biblioteca *MKL* para cualquier ancho de banda de la matriz A. Las mejoras alcanzadas son de hasta un 200 % con $n = 4$ y de hasta un 600 % con $n = 20$. Respecto a la utilización de rutinas de la biblioteca *GotoBLAS*, las nuevas rutinas *BLAS*-3 mejoran a la rutina TBMV_GOTO, basada en el uso iterativo de la implementación de la rutina TBMV incluida en *GotoBLAS*. Cuando n toma valores pequeños, $n = 4$, las nuevas rutinas obtienen prestaciones hasta un 25 % superiores a TBMV_GOTO, mientras que si $n = 20$ las nuevas rutinas son hasta un 400 % más rápidas utilizando únicamente dos procesadores, es decir, aparece una aceleración superlineal. Valores mayores de n permiten el incremento en el número de operaciones realizadas por cada acceso a memoria que resulta la clave para poder alcanzar esta aceleración.

Finalmente, la figura 3.36 muestra los resultados de las mejores implementaciones vistas para ambas implementaciones *BLAS*. Como se ilustra en esta figura, TBMM_B3+*MKL* obtiene las mejores prestaciones con independencia del ancho de banda de la matriz A. Las implementación TBMM_B3 obtiene grandes prestaciones tanto con *GotoBLAS* como con *MKL*, mejorando un 400 % y un 600 %

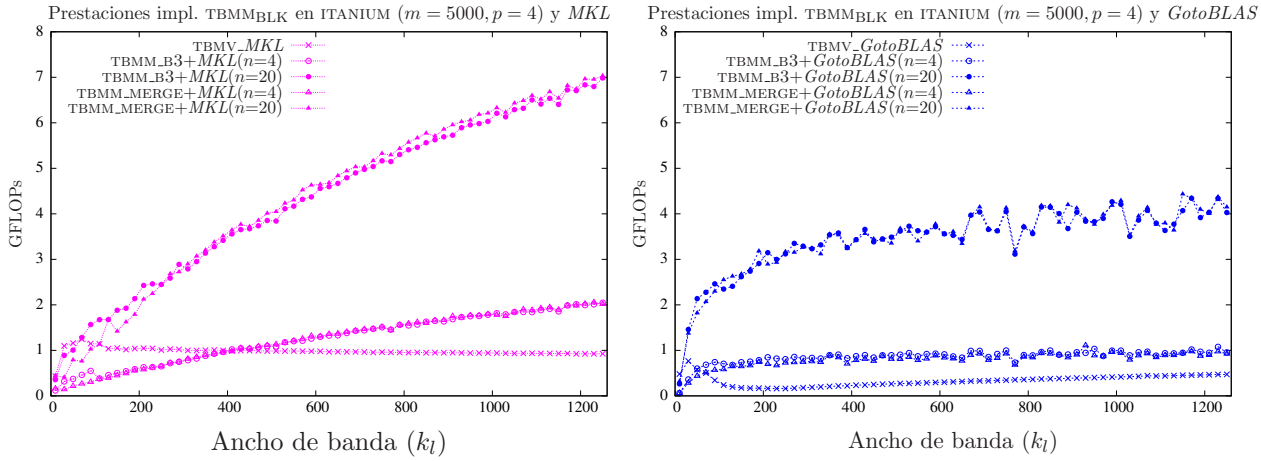


Figura 3.31: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

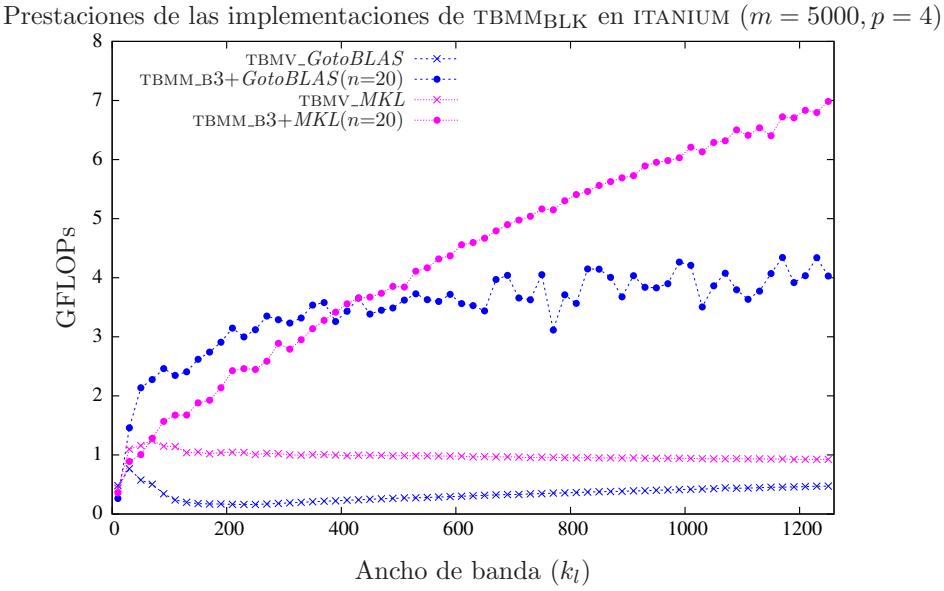


Figura 3.32: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

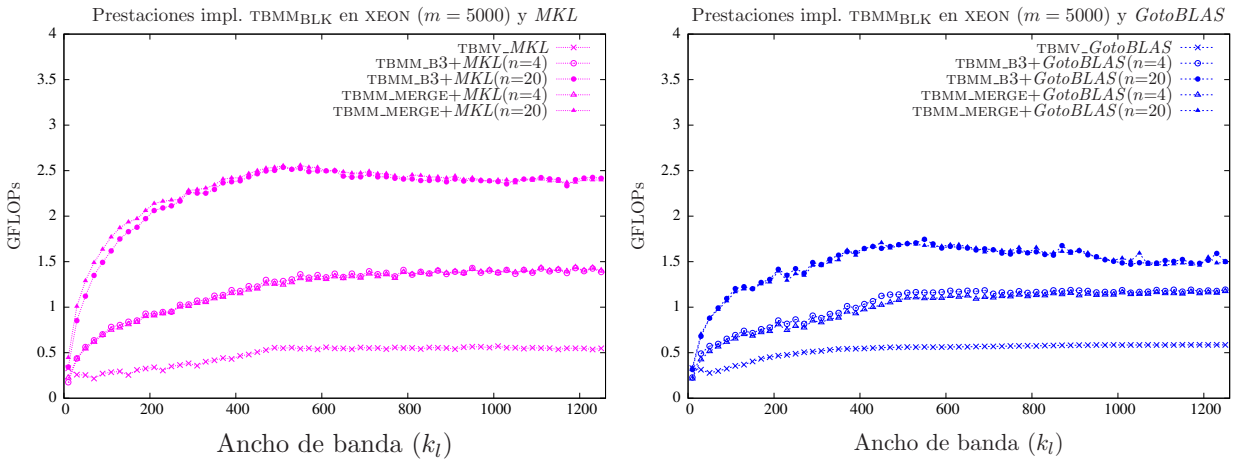


Figura 3.33: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

respectivamente el uso iterativo de la rutina TBMV.

3.3.5. Conclusiones

Las nuevas rutinas *BLAS*-3 mejoran claramente las prestaciones alcanzadas por el método tradicional: el uso iterativo de la rutina TBMV. Como era de esperar, estas rutinas mejoran sus prestaciones cuando la matriz A es una matriz de banda ancha y cuando n toma valores más elevados, pero en ambas arquitecturas la eficiencia de las nuevas rutinas con valores reducidos k_l y de n queda patente, incluso en estas condiciones mejorando en ocasiones al uso de rutinas de niveles inferiores de *BLAS*.

Los experimentos han demostrado la importancia que el valor de n tiene en las prestaciones de las rutinas *BLAS*-3. Cuanto más grande es n mayor es el nivel de paralelismo y, sobre todo, mayor es el ratio entre operaciones y el número de accesos a memoria realizados por las rutinas del tercer nivel de *BLAS*. Estas dos características hacen que la mejora en la eficiencia de las nuevas implementaciones sea tan elevada al aumentar el valor de n.

Respecto a la comparativa entre las dos implementaciones *BLAS*-3, en general al emplear rutinas de *MKL* TBMM_MERGE alcanza mejores prestaciones, mientras que con *GotoBLAS* es TBMM_B3 la implementación más rápida.

3.4. Solución de múltiples sistemas lineales triangulares banda

En esta sección se estudia la resolución de múltiples sistemas de ecuaciones lineales que comparten una misma matriz de coeficientes con estructura triangular banda. Esta operación es la definida por las expresiones:

$$op(A) \cdot X = \alpha \cdot B \quad \text{o} \tag{3.60}$$

$$X \cdot op(A) = \alpha \cdot B, \tag{3.61}$$

donde $\alpha \in \mathbb{R}$, las matrices de incógnitas y de términos independientes $X, B \in \mathbb{R}^{m \times n}$, y $A \in \mathbb{R}^{m \times m}$ en (3.60) o $A \in \mathbb{R}^{n \times n}$ en (3.61) es una matriz triangular con estructura banda. En la práctica X y B se almacenan sobre una misma matriz, que inicialmente contiene los elementos de B, y que a lo

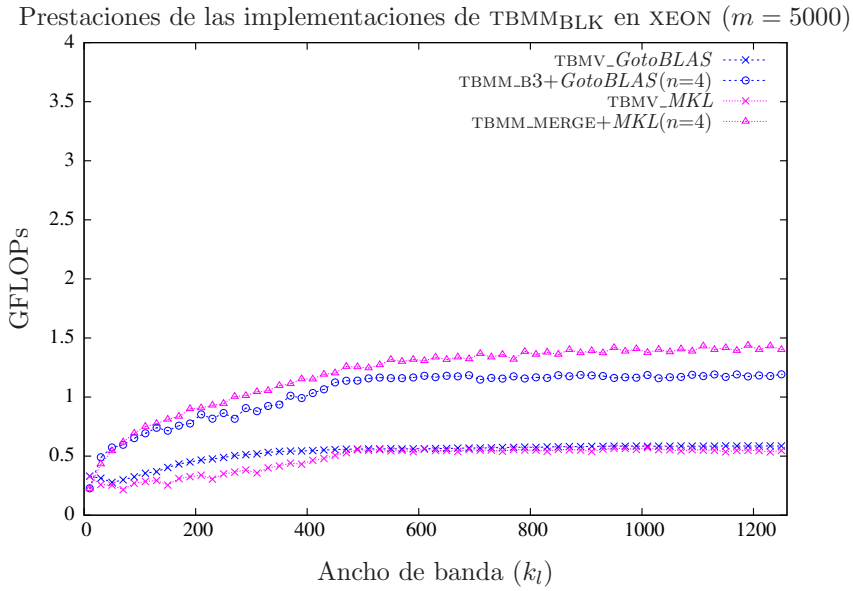


Figura 3.34: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

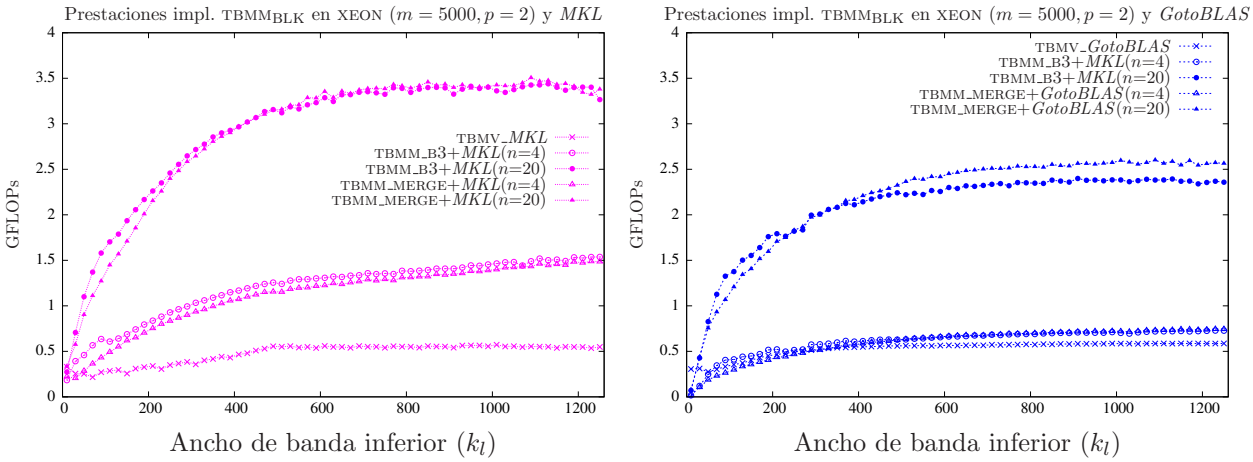


Figura 3.35: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

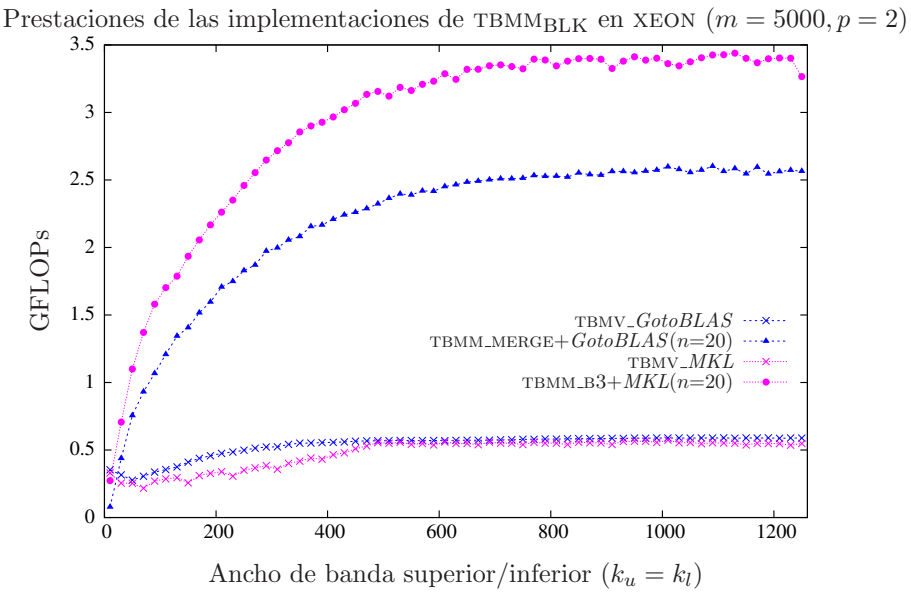


Figura 3.36: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

largo del algoritmo es sobreescrita con los elementos de la solución X. En adelante, nos referiremos a esta matriz como B.

Según la convención utilizada en *BLAS*, el nombre de la rutina que desarrollara la funcionalidad contemplada en esta sección sería TBSM, notación que usamos a partir de este momento. En cuanto a su especificación, se propone la ofrecida en la figura 3.37. Los significados de los argumentos se mantienen con respecto a la rutina TBMM: `SIDE` especifica cuál de los productos en (3.60)–(3.61) se calcula; `UPLO` determina si se opera con una matriz triangular superior o inferior; `TRANS` indica si la matriz banda aparece en el producto como transpuesta o no; y `DIAG` permite especificar que la matriz A únicamente en su diagonal valores iguales a 1. Los restantes argumentos mantienen el significado habitual de *BLAS*.

Por simplicidad, consideraremos el caso en que A es una matriz triangular inferior, con ancho de banda k_l, que aparece en el producto a la izquierda de X (sistema (3.60)) sin transponer y contiene valores distintos de uno en la diagonal, es decir,

$$A \cdot X = B. \tag{3.62}$$

A pesar de esta simplificación, el estudio mostrado a continuación es fácilmente generalizable al resto de casos.

3.4.1. Implementaciones basadas en TBSV

La operación en estudio puede ser descompuesta en n soluciones de sistemas triangulares banda, uno para cada columna de X y B. Este enfoque permite resolver la operación tratada realizando n invocaciones a la rutina de *BLAS*-2 TBSV, operación ésta estudiada en la sección 2.4.

3.4.2. Algoritmo TBSM_BLK

El algoritmo TBSM_BLK, descompuesto en los dos bucles de las figuras 3.38 y 3.39, permite su implementación mediante invocaciones a rutinas del tercer nivel de *BLAS* denso, ya que en cada

```
      SUBROUTINE DTBSM( SIDE, UPLO, TRANS, DIAG, M, N, K, ALPHA,
                        A, LDA, B, LDB )
*     .. Scalar Arguments ..
      DOUBLE PRECISION   ALPHA
      INTEGER            K, LDA, LDB, M, N
      CHARACTER          DIAG, SIDE, TRANS, UPLO
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, *), B( LDB, *)
*
*  Purpose
*  =======
*
*  DTBSM solves one of the systems of equations
*
*  (1)     op(A) X = alpha*B,
*
*  or
*
*  (2)     X op(A) = alpha*B,
*
*  where  op( X ) is one of
*
*     op( X ) = X   or   op( X ) = X',
*
*  A is a lower triangular band matrix with k sub-diagonals
*  or an upper triangular band matrix with k super-diagonals,
*  and B an m by n matrix.
*
*  In (1) op( A ) is an m by m matrix
*  In (2) op( A ) is an n by n matrix
*
```

Figura 3.37: Especificación propuesta para la rutina TBSM.

iteración del algoritmo únicamente se ejecutan operaciones (aritméticas) matriz-matriz. Mientras el bucle exterior recorre la matriz B (y, por tanto, X) por columnas, el bucle interno recorre por filas estos bloques de columnas calculando sus elementos.

Las operaciones aritméticas ejecutadas durante una iteración del bucle interno del algoritmo son:

$$C_1 := A_{11}^{-1} \cdot C_1, \tag{3.63}$$

$$C_2 := C_2 - A_{21} \cdot C_1, \tag{3.64}$$

$$X_3 := C_3 - A_{31} \cdot C_1. \tag{3.65}$$

El cálculo del bloque C_1, operación (3.63), es el resultado de la resolución de múltiples sistemas triangulares. Por otro lado, las actualizaciones de los bloques C_2 y C_3 son el resultado de sendos productos matriz-matriz, si bien el último de ellos presenta la peculiaridad de que una de las matrices, A_{31}, presenta una estructura triangular superior. Las rutinas de *BLAS*-3 denso TRSM, GEMM y TRMM implementan las operaciones (3.63), (3.64) y (3.65), respectivamente.

La rutina TBSM_B3 codifica el algoritmo TBSM_{BLK} invocando a las rutinas TRSM, GEMM y TRMM. Por lo tanto, esta rutina realiza la totalidad de las operaciones aritméticas mediante llamadas a rutinas del nivel 3 de *BLAS* denso. Sin embargo, a fin de ajustarse a la funcionalidad ofrecida por la

Algorithm: $[B] := \text{TBSM}_{\text{BLK}}(B, A, k_l)$

Partition $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array} \right)$
where B_L has 0 columns;

while $n(B_L) < n(B)$ **do**
Determine block size c
Repartition

$$\left(\begin{array}{c|c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right)$$
where B_1 has c columns

$B_1 := \text{TBSM}_{\text{INNER\_LOOP}}(B_1, A, k_l)$

Continue with

$$\left(\begin{array}{c|c} B_L & B_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right)$$
endwhile

Figura 3.38: Bucle externo del algoritmo por bloques TBSM_{BLK} para la resolución de del sistema triangular banda $B := A^{-1} \cdot B$.

rutina TRMM de *BLAS*, la secuencia concreta de operaciones realizada por TBSM_B3 es la siguiente:

$$\begin{aligned}
(\text{TRSM}) \quad & C_1 := A_{11}^{-1} \cdot C_1, && (3.66)\\
(\text{GEMM}) \quad & C_2 := C_2 - A_{21} \cdot C_1, && (3.67)\\
& C_3 := C_3 - A_{31} \cdot C_1, && (3.68)\\
(\text{LACPY}) \quad & W := C_1, && (3.69)\\
(\text{TRMM}) \quad & W := A_{31} \cdot W, && (3.70)\\
& C_3 := C_3 - W. && (3.71)
\end{aligned}$$

En consecuencia, aparecen dos problemas en la implementación de la rutina TBSM_B3:

- El número elevado de invocaciones a rutinas (4 por iteración) conlleva un sobrecoste.
- La operación (3.71) no puede ser ejecutada por ninguna rutina optimizada de *BLAS*.

A cambio, se produce un acceso favorable a los elementos de A y B, y se utilizan rutinas *BLAS*-3 para el cómputo de todas las operaciones aritméticas.

3.4.3. Implementaciones basadas en rutinas de BLAS-3 denso

Implementación Merge (TBSM_B3_MERGE)

La rutina TBSM_B3_MERGE tiene como objetivo paliar los problemas encontrados en TBSM_B3. En particular, en esta nueva rutina se implementa el algoritmo TBSM_{BLK}, pero las operaciones (3.67) y (3.68) se unen de forma que se realicen con una única invocación a GEMM. Para ello es preciso simular que la matriz formada por los bloques A_{21} y A_{31} está almacenada de forma contigua, según

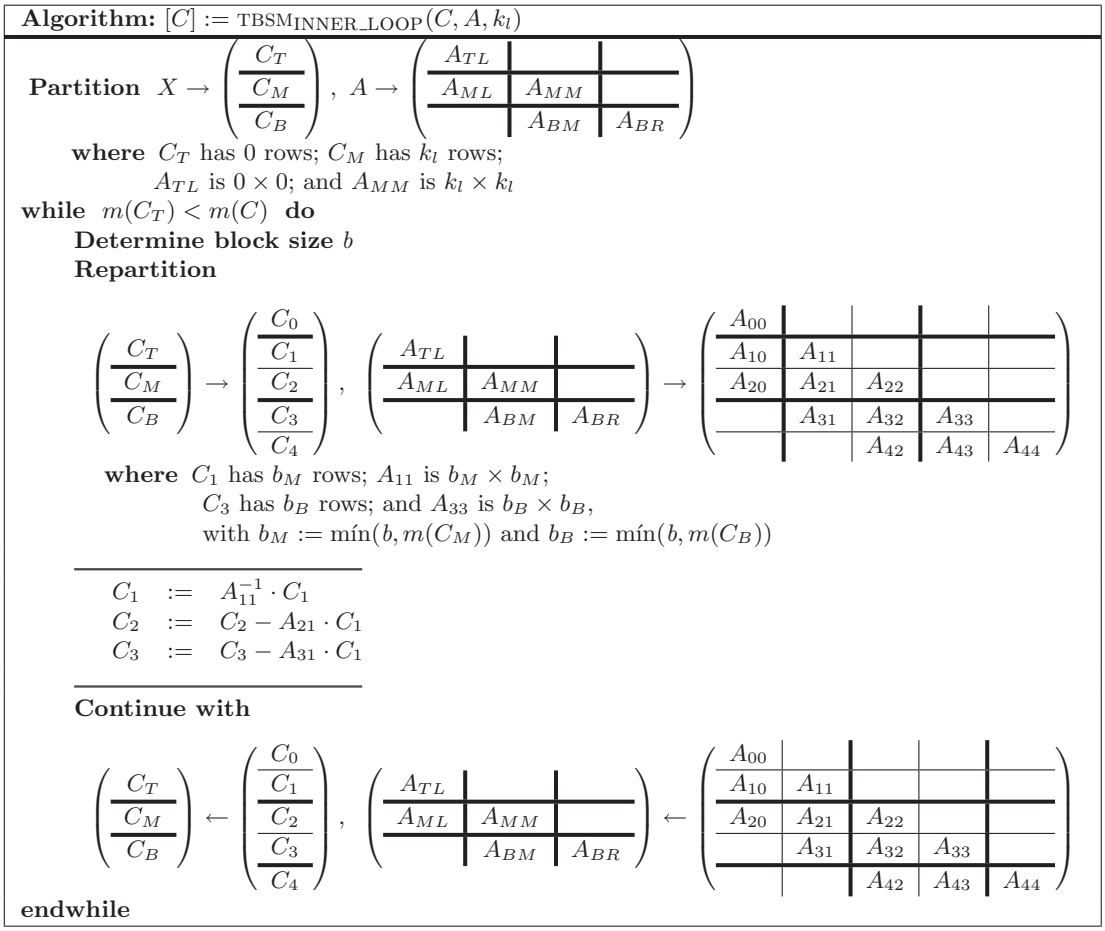


Figura 3.39: Bucle interno del algoritmo por bloques TBSM_{BLK} para la resolución del sistema triangular banda $B := A^{-1} \cdot B$.

el esquema de almacenamiento para matrices densas. Esto se consigue con el siguiente proceso:

$$
\begin{array}{lll}
(\text{TRSM}) & C_1 & := A_{11}^{-1} \cdot\ C_1, \qquad (3.72) \\
 & \left[\begin{array}{c} C_2 \\ C_3 \end{array}\right] & := \left[\begin{array}{c} X_2 \\ X_3 \end{array}\right] - \left[\begin{array}{c} A_{21} \\ A_{31} \end{array}\right] \cdot\ X_1, \qquad (3.73) \\
 & W & := \text{STRIL}(A_{31}), \qquad (3.74) \\
 & \text{STRIL}(A_{31}) & := 0, \qquad (3.75) \\
(\text{GEMM}) & \left[\begin{array}{c} X_2 \\ X_3 \end{array}\right] & := \left[\begin{array}{c} X_2 \\ X_3 \end{array}\right] - \left[\begin{array}{c} A_{21} \\ A_{31} \end{array}\right] \cdot\ X_1, \qquad (3.76) \\
 & \text{STRIL}(A_{31}) & := W \qquad (3.77)
\end{array}
$$

Las operaciones (3.74) y (3.75), respectivamente, realizan una copia de la región de memoria en la que se almacena la parte triangular inferior de A_{31} (según el esquema de almacenamiento para matrices densas) y rellenan esa misma área con ceros. Ambas operaciones se implementan con dos bucles anidados. Es interesante recordar que el número de elementos de la parte triangular inferior de A_{31} es inferior a $\frac{b}{2}$, es decir, no es una operación con un coste elevado.

Tras haber realizado la primera copia, se invoca a GEMM para realizar el producto. Finalmente,

la operación (3.77) devuelve a la región de memoria que almacena la parte triangular inferior de A_{31} sus valores originales almacenados temporalmente en W.

3.4.4. Resultados experimentales

Arquitectura ITANIUM

BLAS secuencial En la figura 3.40 se comparan las prestaciones de las rutinas TBSM basadas en las implementaciones de TBSV incluidas en las bibliotecas *MKL* y *GotoBLAS* con las nuevas implementaciones *BLAS*-3. Para estas últimas se han obtenido resultados con $n = 4$ y $n = 20$ (recordar que las prestaciones de las implementaciones basadas en la rutina TBSV no varían con el valor de n). Como se puede observar, incluso con $n = 4$ las prestaciones obtenidas por las nuevas rutinas *BLAS*-3 mejoran las alcanzadas con TBSV, mientras que con $n = 20$ las variantes *BLAS*-3 logran con facilidad prestaciones muy superiores, siendo hasta 3 veces más eficiente para *GotoBLAS* y 5 veces para *MKL*. La eficiencia de las nuevas rutinas depende en gran medida del valor de n,

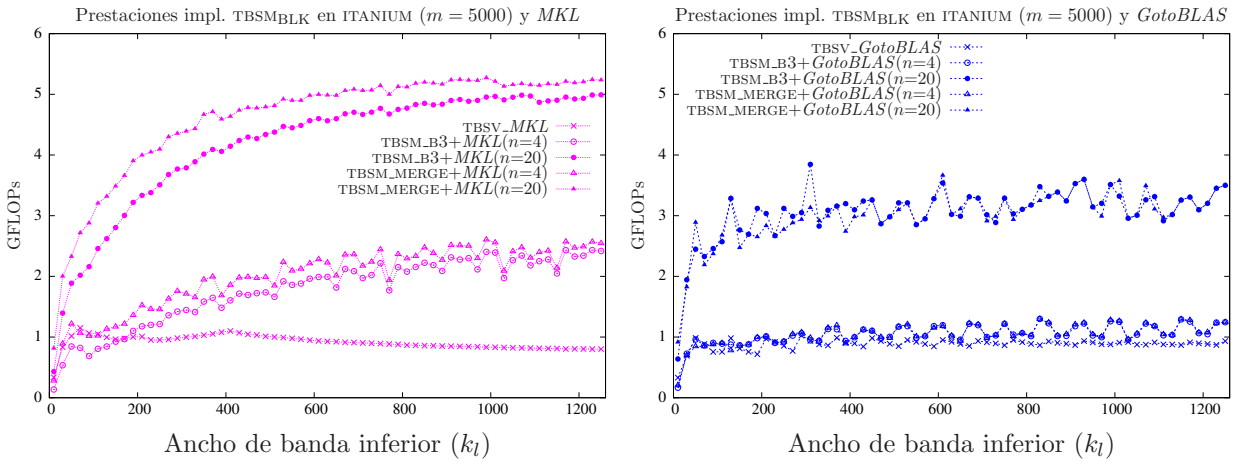


Figura 3.40: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

de forma que cuanto más grande es éste mayores prestaciones alcanzan. En el caso de *MKL*, las prestaciones varían igualmente con el ancho de banda de la matriz, viéndose favorecidas por valores elevados de k_l. Por el contrario, en el caso de *GotoBLAS* las prestaciones crecen rápidamente al incrementar el ancho de banda hasta $k_l = 50$, mientras que para valores superiores las prestaciones se mantienen estables. A consecuencia de estos comportamientos dispares, la utilización de rutinas de *GotoBLAS* genera mejores prestaciones cuando A es una matriz de banda estrecha, mientras que si A es una matriz de banda media o ancha es preferible emplear rutinas de *MKL*.

Respecto a la comparativa entre las dos implementaciones *BLAS*-3 propuestas, si se emplea la biblioteca *GotoBLAS* ambas son igualmente eficientes. En el caso de invocar rutinas pertenecientes a *MKL*, TBSM_MERGE es ligeramente más rápida.

Por último, la figura 3.41 recoge los resultados obtenidos con las mejores versiones de ambas bibliotecas. Por simplicidad, en esta gráfica solamente se muestran los resultados con $n = 4$. De esta figura podemos destacar la eficiencia de la rutina TBSM_MERGE+*MKL*, que se muestra como la mejor opción para matrices con cualquier ancho de banda, superando claramente los resultados del resto de las implementaciones.

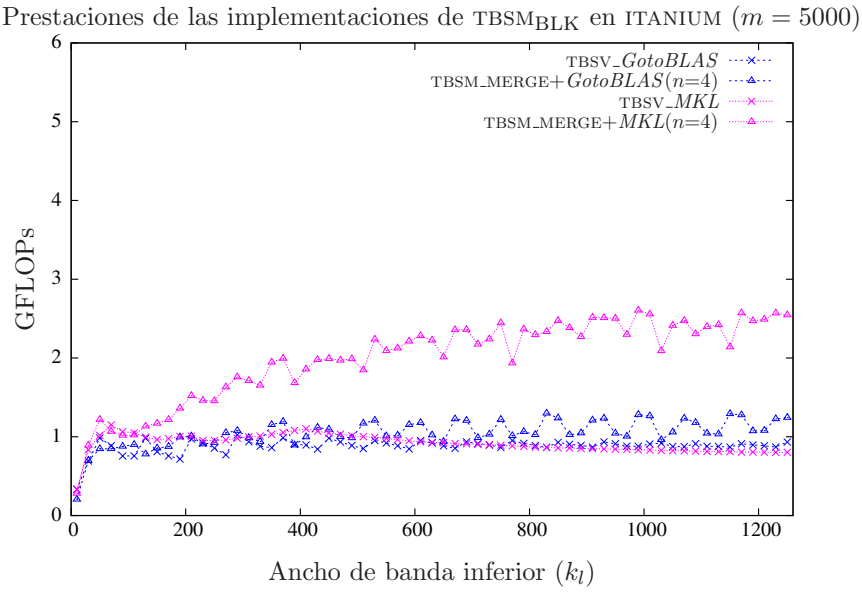


Figura 3.41: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

BLAS paralelo El mismo estudio realizado con implementaciones paralelas de *BLAS* revela como las nuevas implementaciones *BLAS*-3 son capaces de explotar eficientemente la existencia de múltiples procesadores.

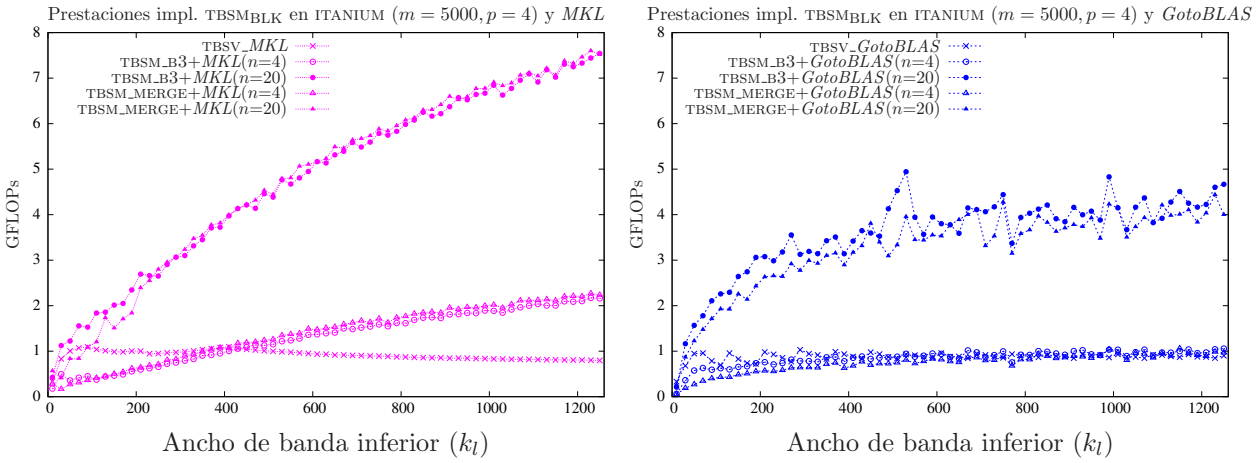


Figura 3.42: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

La figura 3.42 muestra los resultados de las nuevas implementaciones junto con los del uso repetido de las rutinas TBSV de las bibliotecas *MKL* y *GotoBLAS*. Al invocar a rutinas paralelas, el crecimiento de las prestaciones con matrices de banda estrecha es más lento, pero al operar con matrices de banda ancha las prestaciones obtenidas son muy superiores. El empleo de versiones paralelas de *BLAS* precisa una mínima carga computacional para ser efectivo. Con *MKL* y $n = 4$, TBSM+*MKL* es la mejor rutina cuando el ancho de banda de la matriz A es inferior a 400, mientras que para anchos de banda superiores la rutina *BLAS*-3 TBSM_MERGE+*MKL* es la más eficiente. Con $n = 20$, TBSM_B3 es la rutina más rápida para cualquier ancho de banda de la matriz A. En el caso

de *GotoBLAS* se obtienen resultados similares, TBSV_*GotoBLAS* alcanza las mejores prestaciones con anchos de banda menores que 600 y $n = 4$, mientras que TBSM_B3 es más eficiente para $n = 4$ y anchos de banda superiores. Con $n = 20$ las rutinas *BLAS*-3 son notablemente más eficientes, especialmente la rutina TBSM_B3.

Las prestaciones de las mejores implementaciones vistas cuando $n = 20$ se muestran en la figura 3.43. La rutina TBSM_B3+*GotoBLAS* es la rutina más eficiente cuando el ancho de banda de la matriz es inferior a 400, mientras que para anchos de banda superiores es TBSM_B3+*MKL* la implementación que genera las mejores prestaciones. La superioridad de las rutinas *BLAS*-3 es evidente, llegando a ser hasta diez veces más eficientes que el uso repetido de las rutinas TBSV.

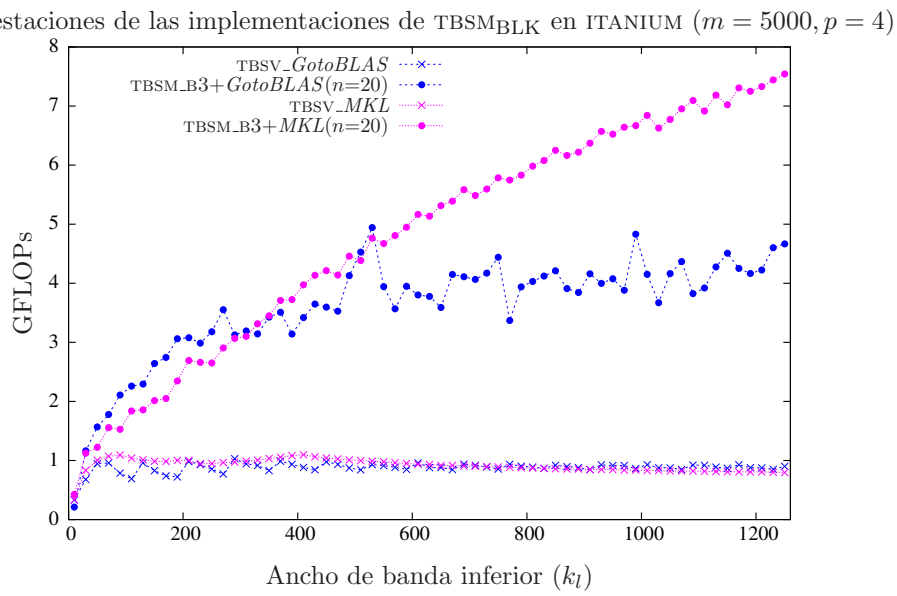


Figura 3.43: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

Arquitectura XEON

BLAS secuencial Los experimentos realizados sobre la arquitectura XEON comparan la eficiencia de las implementaciones *BLAS*-3 propuestas para la rutina TBSM con el uso repetitivo de las rutinas TBSV incluidas en *MKL* y *GotoBLAS*. La figura 3.44 muestra los resultados obtenidos con todas estas implementaciones, tanto para *MKL* (gráfica de la izquierda) como para *GotoBLAS* (gráfica de la derecha).

Si analizamos los resultados obtenidos con las diferentes implementaciones basadas en rutinas *MKL*, podemos comprobar como las nuevas rutinas mejoran notablemente el uso repetido de TBSM, llegando a ser hasta cuatro veces más rápidas. Como era de esperar, las rutinas *BLAS*-3 se ven beneficiadas por valores más altos de n y de k_l, pero incluso con valores reducidos de estas dos variables su eficiencia es mayor que la de TBSV_*MKL*. La implementación TBMM_MERGE es ligeramente más eficiente que TBSM_B3, especialmente cuando se incrementa el valor de n.

Si las rutinas invocadas pertenecen a *GotoBLAS* los resultados obtenidos son similares, al menos en comportamiento. De nuevo el valor de n es clave para alcanzar altas prestaciones, aunque ya con $n = 4$ se consiguen mejorar notablemente los resultados por la rutina TBSV de *GotoBLAS* (llegando ambas implementciones *BLAS*-3 a ser hasta dos veces más rápidas que TBSV_*GotoBLAS*). Tanto TBMM_B3 como TBMM_MERGE alcanzan prestaciones parecidas para cualquier valor de n y k_l.

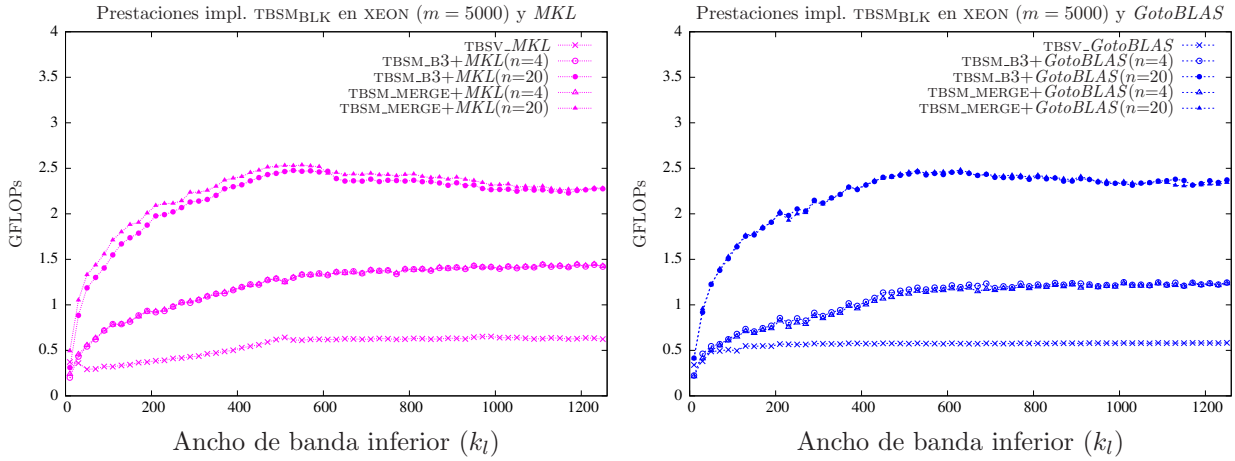


Figura 3.44: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

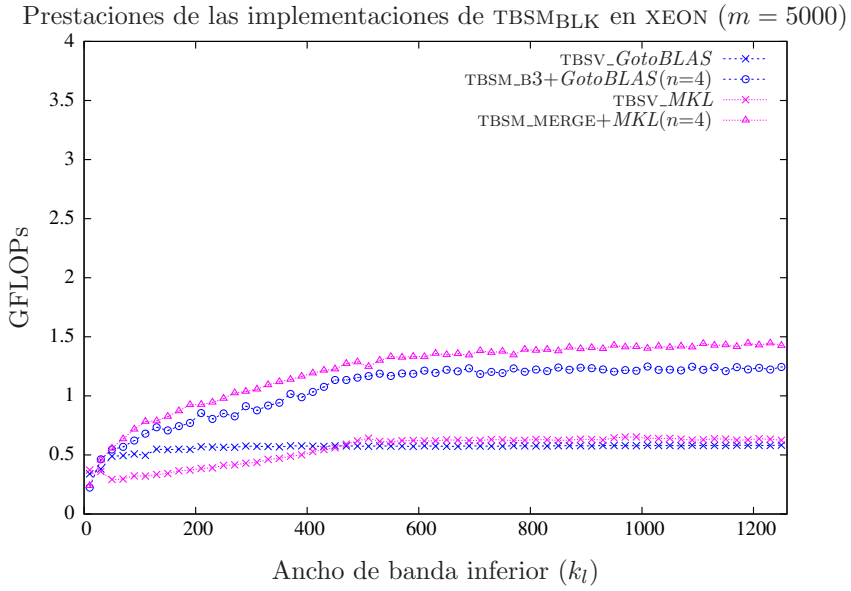


Figura 3.45: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

La figura 3.45 muestra las prestaciones de las mejores implementaciones en la figura anterior cuando $n = 4$. Como se puede apreciar, la rutina TBMM_MERGE es la más eficiente sea cual sea el ancho de banda de la matriz A. Cabe destacar las buenas prestaciones ofrecidas por las implementaciones *BLAS*-3 propuestas en esta sección, ya que éstas son capaces de duplicar las alcanzadas por TBSV, incluso cuando n toma un valor tan reducido como cuatro.

BLAS paralelo Es de suponer que las implementaciones basadas en rutinas *BLAS*-3 se beneficiarán del uso de computación paralela especialmente cuando los valores de n, k_u y k_l aumenten, ya que en estas circunstancias el nivel de paralelismo será mayor. Así se constata en la gráfica 3.46. Como se puede observar, cuando $n = 4$, al emplear rutinas paralelas de *MKL* se alcanzan prestaciones similares a las obtenidas al emplear una versión secuencial de *MKL*; por contra, los resultados cuando $n = 20$ obtenidos con la versión paralela de *MKL* son manifiestamente superiores a los mostrados por el *BLAS* secuencial.

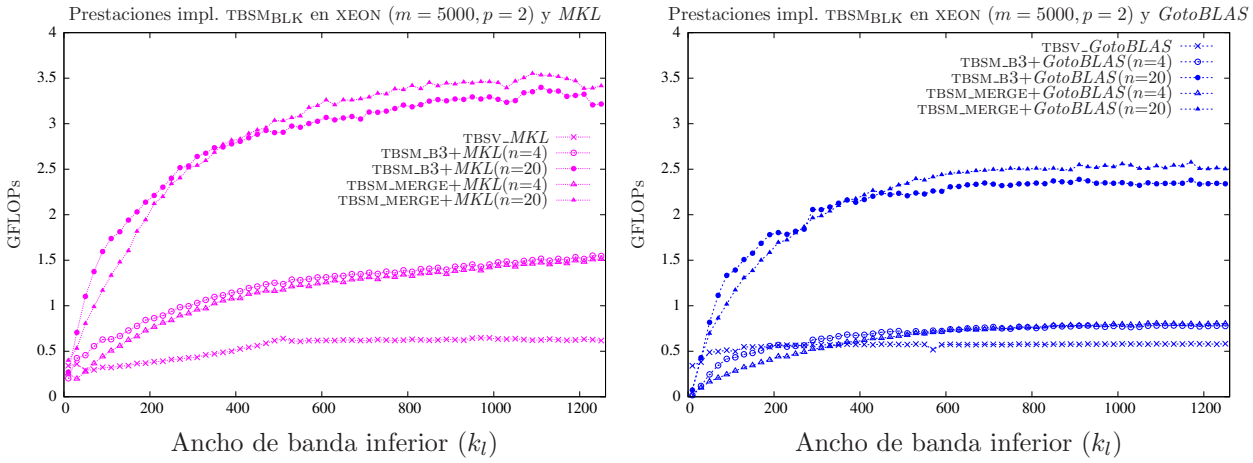


Figura 3.46: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

Si se utiliza *GotoBLAS*, los resultados son todavía más llamativos, ya que las prestaciones con la versión paralela resultan inferiores si $n = 4$ y similares si $n = 20$.

Una circunstancia común a ambas bibliotecas *BLAS* es que la rutina TBSM_B3 mejora las prestaciones obtenidas por TBSM_MERGE cuando A es una matriz de banda estrecha, es decir, obtiene mejores resultados en situaciones con baja carga computacional. Por contra, TBSM_MERGE es la rutina más rápida cuando A es una matriz de banda media o ancha.

Como queda reflejado en la figura 3.46, las implementaciones basadas en *MKL* alcanzan mayores prestaciones que las basadas en rutinas de *GotoBLAS*.

Por último, la figura 3.47 recopila las prestaciones de las mejores implementaciones vistas en esta sección cuando $n = 20$ y se invocan a rutinas paralels de *BLAS*-3. La figura muestra la gran eficiencia de la rutina TBSM_MERGE, especialmente cuando invoca a rutinas de *MKL*. Estas rutinas son hasta seis veces más rápidas que la rutina TBSV, utilizando únicamente dos hebras de ejecución. Esto se debe al mejor aprovechamiento de que las rutinas *BLAS*-3 hacen de los recursos *hardware* presentes.

3.4.5. Conclusiones

Se han comparado los resultados de las nuevas implementaciones *BLAS*-3 para la rutina TBSM con el uso repetido de TBSV. Las nuevas rutinas obtienen las mejores prestaciones en todos los

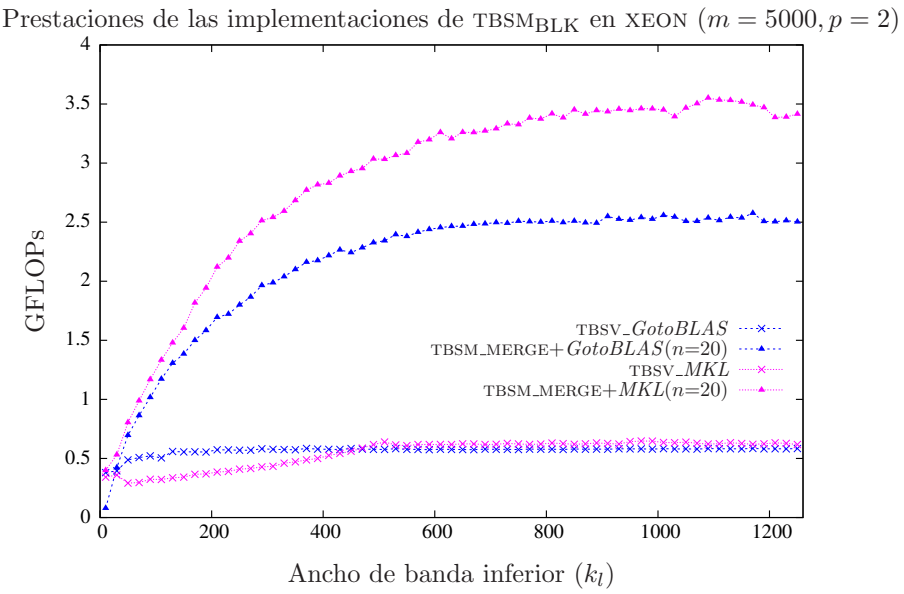


Figura 3.47: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

experimentos realizados, tanto en un entorno secuencial como paralelo. El empleo de rutinas *BLAS*-3 resulta más favorable incluso cuando n toma valores pequeños y A es una matriz de banda estrecha; es decir, incluso cuando el coste computacional de la operación es reducido. En cuanto a la comparativa entre las dos implementaciones de *BLAS*, las rutinas basadas en *MKL* alcanzan mejores prestaciones que las que emplean *GotoBLAS*.

3.5. Producto de dos matrices generales banda

La operación contemplada a continuación es el producto matricial

$$C \quad := \quad \alpha \cdot op(A) \cdot op(B) + \beta \cdot C, \tag{3.78}$$

donde $\alpha, \beta \in \mathbb{R}$, $C \in \mathbb{R}^{m \times n}$, y $op(X)$ es un operador que puede transponer o no la matriz sobre la que está aplicado, $op(A) \in \mathbb{R}^{m \times k}$ y $op(B) \in \mathbb{R}^{k \times n}$ en (3.78). Además, las matrices A/B presentan ambas una estructura banda con anchos de banda superior e inferior $k_u(A)/k_u(B)$ y $k_l(A)/k_u(B)$ respectivamente de modo que el resultado C comparte esta misma estructura con anchos de banda superior e inferior $k_u(A) + k_u(B)$ y $k_l(A) + k_u(B)$, respectivamente.

Utilizaremos el nombre GBGBMM para referirnos a la rutina que implementa esta operación. La especificación Fortran-77 que se propone (para la implementación en doble precisión) es la ilustrada en la figura 3.48. Los argumentos `TRANSA` y `TRANSB` determinan, respectivamente, si se opera con la transpuesta de las matrices A y B. El propósito de los restantes argumentos es el habitual en *BLAS*.

3.5.1. Implementaciones basadas en BLAS-2 y BLAS-3

Se han desarrollado dos implementaciones para el cálculo del producto GBGBMM. Debido a la complejidad de estas implementaciones, derivada de la casuística existente en función de la relación entre los anchos de banda de A y B, en este caso no se presenta el pseudo-código del algoritmo

```
      SUBROUTINE GBGBMM( TRANSA, TRANSB, M, N, K, KLA, KUA,
                         KLB, KUB, KLC, KUC, ALPHA, A, LDA,
                         B, LDB, BETA, C, LDC)
*     .. Scalar Arguments ..
      DOUBLE PRECISION ALPHA, BETA
      INTEGER          K, KLA, KLB, KLC, KUA, KUB, KUC,
                       LDA, LDB, LDC, M, N
      CHARACTER        TRANSA, TRANSB
*     .. Array Arguments ..
      DOUBLE PRECISION A( LDA, *), B( LDB, *), C( LDC, *)
*
*  Purpose
*  =======
*
*  DGBMM performs the matrix-matrix operations
*
*  (1)     C := alpha*op(A)*op(B) + beta*C,
*
*  where  op( X ) is one of
*
*     op( X ) = X   or   op( X ) = X',
*
*  alpha and beta are scalars, A is a band matrix with
*  kla sub-diagonals and kua super-diagonals,
*  B is a band matrix with
*  klb sub-diagonals and kub super-diagonals,
*  and C an m by n matrix with
*  klc sub-diagonals and kuc super-diagonals.
*
*  In (1) op( A ) is an m by k matrix and op( B ) is a k by n matrix
*
```

Figura 3.48: Especificación propuesta para la rutina GBGBMM.

ni de las rutinas. Tan sólo mencionar aquí que estas dos implementaciones difieren en el tipo de operaciones de *BLAS* denso que utilizan para realizar la mayor parte de los cálculos, *BLAS*-2 o *BLAS*-3, y nos referiremos a ellas en el estudio experimental que sigue como GBGBMM_B2 y GBGBMM_B3, respectivamente.

3.5.2. Resultados experimentales

Arquitecturas ITANIUM y XEON

En la figura 3.49 se muestran los resultados de la evaluación de las rutinas usando las implementaciones secuenciales de las bibliotecas *GotoBLAS* y *MKL*. En todos los experimentos de la figura las tres dimensiones del producto coinciden ($m = n = k$) y los anchos de banda inferior y superior de las matrices A y B también lo hacen. En ambas arquitecturas los resultados muestran la mayor eficiencia de la versión *BLAS*-3. En el caso de la implementación *BLAS*-3 en la arquitectura ITANIUM, la invocación de rutinas de *BLAS* en *MKL* reporta claramente mayor eficiencia.

No se han evaluado las prestaciones del producto GBGBMM utilizando implementaciones paralelas de *BLAS* debido al reducido coste computacional del mismo. Frente al producto de una matriz banda por otra densa, que presenta un coste cuadrático en las dimensiones de la matriz densa, el

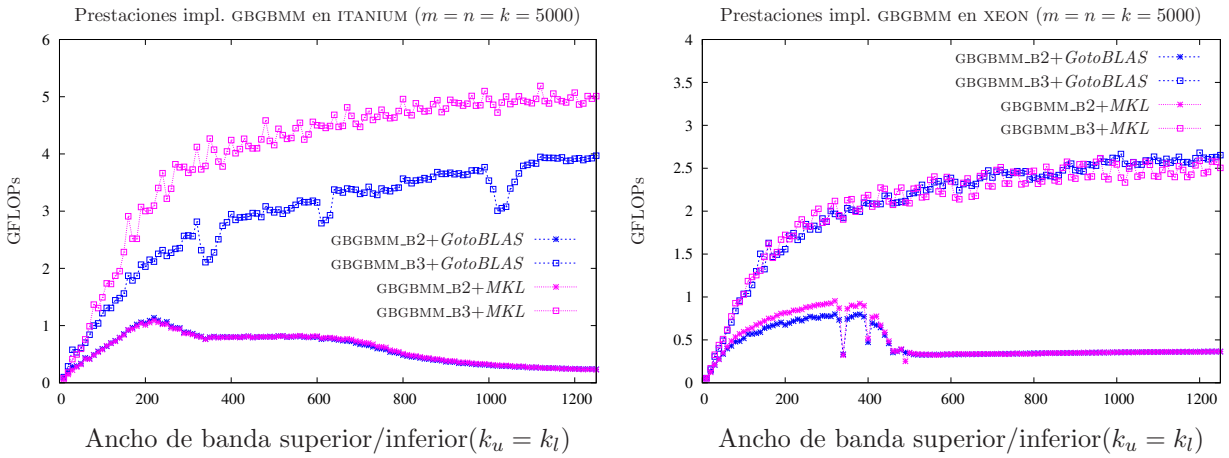


Figura 3.49: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

coste del producto de matrices banda responde a una expresión que es lineal en la dimensión común de las matrices banda. Así pues, como en el caso del *BLAS*-2 banda, la utilización de un *BLAS* multihebra para la ejecución de la operación GBGBMM está poco justificada.

3.5.3. Conclusiones

El producto entre dos matrices banda es una operación no soportada por *BLAS* actualmente. Aun así estimamos conveniente su estudio por su utilidad en problemas de ingeniería y por el gran ahorro computacional que se puede obtener al explotar la estructura de ambas matrices. Se han evaluado dos implementaciones para este producto, una de ellas basada en operaciones *BLAS*-2 y otra basada en operaciones *BLAS*-3. La complejidad de estas implementaciones es grande debida a la casuística que han de contemplar.

Se han evaluado las dos nuevas implementaciones al invocar a rutinas de *GotoBLAS* y *MKL* en las arquitecturas ITANIUM y XEON. Debido al bajo coste computacional de la operación, únicamente se ha experimentado con versiones secuenciales de *BLAS*. Los experimentos han demostrado la gran eficiencia de la implementación basada en *BLAS*-3, que mejora en todos los casos los resultados obtenidos por la implementación *BLAS*-2.

Capítulo 4

LAPACK banda

Los métodos directos de resolución de sistemas de ecuaciones lineales

$$Ax = b,$$

utilizados habitualmente cuando la matriz de coeficientes del sistema A presenta una estructura densa o banda, requieren en primer lugar del cálculo de algún tipo de descomposición (o factorización) de la matriz [40]. En este capítulo se consideran las rutinas disponibles en la biblioteca *LAPACK* para el cálculo de la factorización de Cholesky y la factorización LU con pivotamiento parcial de filas cuando la matriz A presenta una estructura banda. La primera de estas factorizaciones se aplica cuando A es además simétrica definida positiva, reduciendo el coste computacional y de almacenamiento del problema, mientras que la segunda requiere únicamente que la matriz sea invertible.

LAPACK incluye rutinas escalares y por bloques para el cálculo de estas dos factorizaciones banda que extraen su eficiencia del uso de una implementación optimizada de *BLAS*. Es más, la combinación de *LAPACK* con una implementación de *BLAS* multihebra es la forma habitual de extraer paralelismo en este tipo de problemas sobre arquitecturas multiprocesador con memoria compartida. En este capítulo se evalúan las prestaciones de las rutinas escalares y por bloques para el cálculo de las factorizaciones de Cholesky y LU con pivotamiento utilizando una única hebra de ejecución (ejecución secuencial) y múltiples hebras (ejecución paralela), en combinación con las implementaciones optimizadas de *BLAS* en *GotoBLAS* y *MKL*. La evaluación de los algoritmos de *LAPACK* nos conduce al desarrollo de nuevos algoritmos para el cálculo de las factorizaciones, que resultan más eficientes en una ejecución paralela, y que se presentan también en este capítulo. Además, siguiendo técnicas de formulación de algoritmos por bloques y planificación dinámicas planteadas recientemente para el caso denso [25], se presenta también una adaptación de estas estrategias para el caso banda, que ofrece una notable ganancia en una ejecución paralela del cálculo de la factorización de Cholesky banda. Estas técnicas pueden ser empleadas en la factorización LU con pivotamiento parcial o en factorizaciones como las que veremos en el siguiente capítulo.

El capítulo está estructurado como sigue: en la sección 2 se evalúan extensamente las prestaciones de las rutinas de *LAPACK* para el cálculo de la factorización de Cholesky, y se proponen nuevas rutinas a partir de los resultados obtenidos. A continuación, en la sección 3, se presenta un estudio menos detallado de la factorización LU con pivotamiento parcial de filas (por simplicidad, debido a la similitud de resultados con la factorización de Cholesky, se reduce el contenido de esta sección) y asimismo se proponen nuevos algoritmos para el cálculo de la factorización LU. Finalmente, en la sección 4 se presentan los nuevos algoritmos por bloques que, combinados con una planificación dinámica de tareas, resultan claramente superiores en el caso de una ejecución paralela con un elevado número de procesadores.

Las plataformas de evaluación utilizadas en las 2 primeras secciones son Intel Itanium2 e Intel Xeon, con 1 o más procesadores, dependiendo de la utilización de un *BLAS* secuencial o multihebra. Para la última sección, las plataformas utilizadas han sido Intel Itanium2(numa) y AMD Opteron(smp) que gozan de un número de procesadores/núcleos mucho más elevado (16 procesadores en la primera y 8×2 núcleos en la segunda)

4.1. Factorización de Cholesky

La factorización de Cholesky descompone una matriz $A \in \mathbb{R}^{n \times n}$ simétrica definida positiva (en adelante spd) en el producto de una matriz triangular y su transpuesta, de la forma

$$A = L \cdot L^T \tag{4.1}$$

o bien

$$A = U^T \cdot U \tag{4.2}$$

en función de si se desa que la matriz triangular resultante sea una matriz inferior (4.1) o superior (4.2). Se cumple que la descomposición de *Cholesky* para una matriz es única.

La factorización de Cholesky se emplea, entre otros, en la resolución de sistemas de ecuaciones lineales y en problemas de mínimos cuadrados. Si bien la descomposición de *Cholesky* únicamente se puede aplicar a matrices spd, presenta unos costes computacional y de almacenamiento reducidos comparado con los de otras factorizaciones (por ejemplo la factorización LU o la factorización QR), lo que la convierte en una alternativa muy interesante.

En esta sección estudiaremos el caso definido en (4.1), siendo $A \in \mathbb{R}^{n \times n}$ una matriz spd banda con ancho de banda k_d, que se descompone en el producto de una matriz triangular inferior banda, $L \in \mathbb{R}^{n \times n}$, con ancho de banda k_d, y su traspuesta. El estudio efectuado en esta sección puede ser fácilmente adaptado al cálculo de (4.2).

La biblioteca de álgebra lineal *LAPACK* incluye dos rutinas para obtener la factorización de Cholesky de una matriz banda, pbtf2 y pbtrf, que aplican un algoritmo escalar y otro por bloques respectivamente.

Por razones de economía de almacenamiento, únicamente se almacena la parte inferior de A, según el esquema de almacenamiento compacto empleado por *BLAS* y *LAPACK*. Siguiendo este esquema los elementos de la matriz resultado L sobreescriben a los de la matriz A.

4.1.1. Algoritmo $\text{PBTRF}_{\text{UNB}}$

El algoritmo $\text{PBTRF}_{\text{UNB}}$, mostrado en la figura 4.1, calcula la factorización de Cholesky de la matriz A. Se trata de un algoritmo iterativo que recorre la matriz A por columnas y que, en cada iteración, realiza tres operaciones. En este algoritmo sólo la parte triangular inferior de A es accedida, de modo que aquellas partes de la matriz denotadas mediante el símbolo "$\star$" referencian elementos, trozos de vectores o bloques no accedidos.

Las dos primeras operaciones computan los elementos de una columna de L; la primera calcula el elemento de la diagonal mediante una raíz cuadrada y la segunda los k_d elementos restantes. La última operación actualiza la parte triangular inferior de un bloque de $l \times l$ elementos de A, donde $l = \text{mín}(k_d, m(A) - m(A_{TL} - 1))$.

Este algoritmo presenta, como mayor ventaja, el patrón de acceso a los elementos de A/L, ya que éste se hace por columnas (tal y como se muestra en la figura 4.2). Dado el esquema de almacenamiento empleado por Fortran, ésta constituye la forma más eficaz de recorrer la matriz.

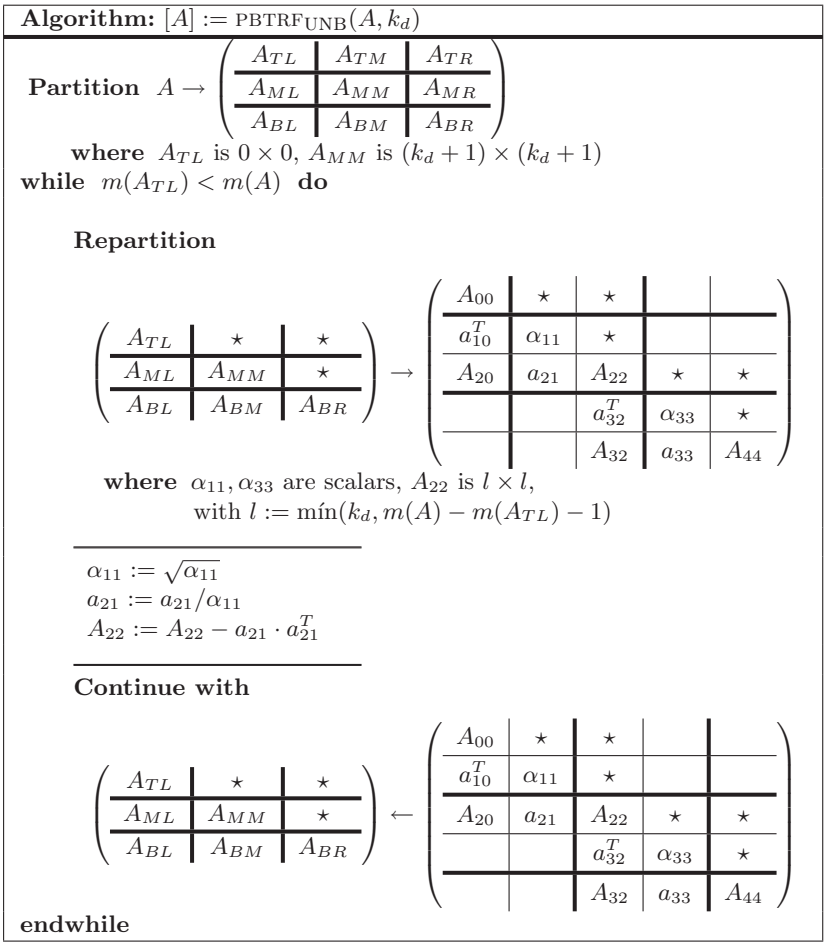


Figura 4.1: Algoritmo $\text{PBTRF}_{\text{UNB}}$ para la factorización $A = L \cdot L^T$.

No obstante, el acceso a memoria presenta cierto margen de mejora, ya que cada elemento de A es accedido hasta en k_d ocasiones.

4.1.2. Implementación LAPACK PBTF2

La implementación del algoritmo $\text{PBTRF}_{\text{UNB}}$ correspondiente a la rutina PBTF2 de *LAPACK* hace uso de dos rutinas del *BLAS*-denso, SCAL y SYR. La primera de ellas calcula a_{21} y representa una operación de escalado de un vector, mientras que la rutina SYR implementa una actualización de rango 1, operación requerida sobre A_{22}. El cálculo de α_{11} se realiza mediante una raíz cuadrada.

Así pues, en esta implementación, n operaciones aritméticas comprenden a raíces cuadradas, un número reducido de operaciones (aproximadamente $k_d \times n$ operaciones) son ejecutadas por la rutina SCAL del nivel 1 de *BLAS*, y la mayoría de los cálculos son realizados por una rutina perteneciente al nivel 2 de *BLAS*, SYR.

La implementación PBTF2 presenta dos características que favorecen la eficiencia: el acceso por columnas a los elementos y la invocación de núcleos eficientes del *BLAS* para la ejecución de la mayoría de las operaciones aritméticas.

Por otra parte, como se ha comentado en el algoritmo $\text{PBTRF}_{\text{UNB}}$, el número de accesos a memoria que realiza esta implementación puede reducirse ya que cada elemento es accedido hasta

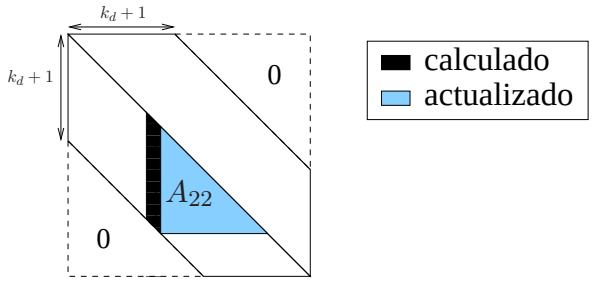


Figura 4.2: Acceso a los elementos durante una iteración (del algoritmo PBTRF$_{\text{UNB}}$ y) de la rutina PBTF2.

en k_d ocasiones. Además, en el caso de que A sea una matriz de banda muy estrecha, el número de operaciones realizadas por SCAL y SYR en cada una de sus invocaciones será muy reducido, pudiendo darse la circunstancia de que el coste de invocación de estas rutinas sea mayor que el beneficio de emplearlas.

4.1.3. Implementaciones basadas en rutinas de BLAS-1 y BLAS-2 denso

Implementación PBTF2_SCAL_SYR

La rutina PBTF2_INLINE implementa el algoritmo PBTRF$_{\text{UNB}}$, al igual que la rutina PBTF2, pero a diferencia de esta última, PBTF2_INLINE trata de eliminar las invocaciones a rutinas con un bajo coste computacional. El motivo es que, en ocasiones, el número de operaciones cubierto por la rutina invocada es tan reducido que resulta imposible compensar el propio coste de la invocación, a pesar de que se trate de una rutina altamente optimizada. Esta circunstancia se presenta principalmente para la rutina SCAL, ya que realiza menos operaciones aritméticas por invocación que SYR. Para evitar esta situación, se han implementado las siguientes versiones:

- PBTF2_SCAL_SYR: incluye el código embebido requerido por las funciones SCAL y SYR.
- PBTF2_SCAL: incluye el código embebido requerido por la función SCAL.

Estas rutinas presentan la ventaja de dedicar menos tiempo a la invocación de subrutinas; por contra no hacen uso de núcleos optimizados de *BLAS*. Como resultado, cabe esperar que sean más rápidas que PBTF2 cuando el tamaño de la banda (k_d) sea reducido, mientras que para valores altos o medios de esta variable, PBTF2 será más eficiente gracias a la utilización de rutinas optimizadas de *BLAS*. En todo caso, al igual que cualquier rutina que implemente el algoritmo PBTRF$_{\text{UNB}}$, estas rutinas realizan repetidos accesos a cada elemento de A.

4.1.4. Algoritmo PBTRF$_{\text{BLK}}$

El algoritmo PBTRF$_{\text{UNB}}$ sólo permite su implementación mediante rutinas de los niveles 1 y 2 de *BLAS*, pero la factorización de Cholesky, tanto por el número de operaciones realizadas como por el número de datos implicados, es una operación del *BLAS*-3. En respuesta, el algoritmo por bloques PBTRF$_{\text{BLK}}$ (figura 4.3) plantea la factorización de Cholesky como una secuencia de operaciones entre matrices, permitiendo el uso de rutinas del tercer nivel de *BLAS*.

PBTRF$_{\text{BLK}}$ recorre la matriz A de izquierda a derecha, computando en cada iteración los elementos de b columnas de L y actualizando elementos de las siguientes k_d columnas. Realiza un

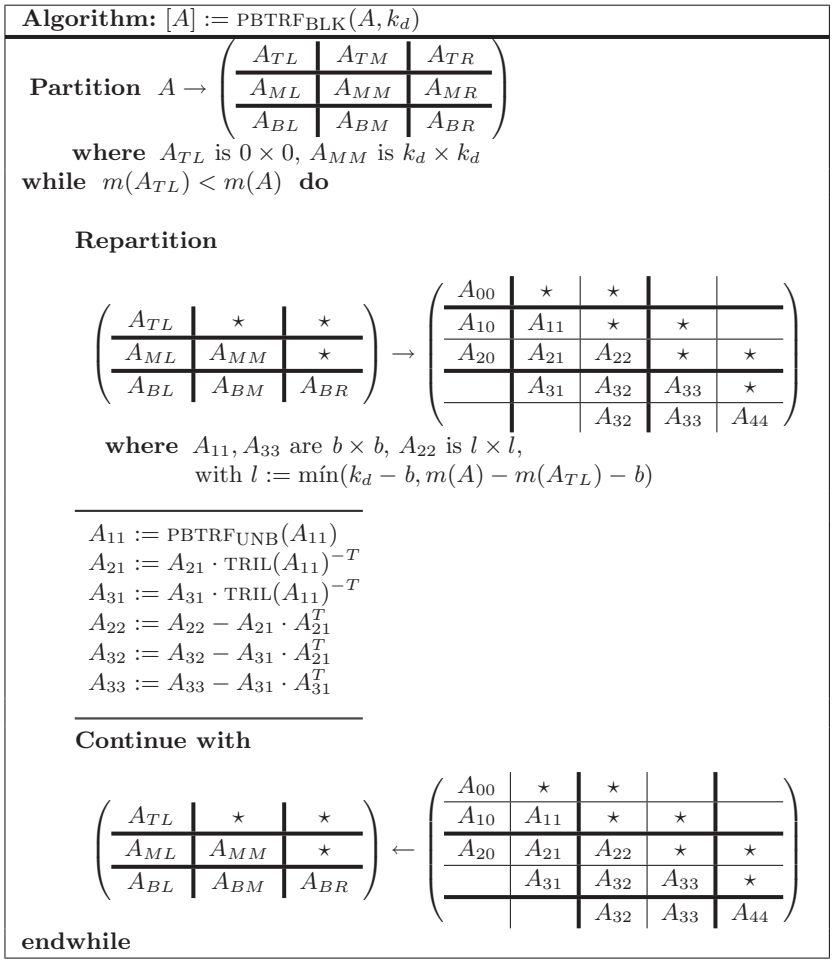


Figura 4.3: Algoritmo por bloques $\text{PBTRF}_{\text{BLK}}$ para la factorización $A = L \cdot L^T$.

total de 6 operaciones en cada iteración en las que se calculan otros tantos bloques (o submatrices, ver figura 4.4): Inicialmente es necesario calcular la factorización de Cholesky del bloque A_{11}. Una vez se ha obtenido la descomposición de A_{11}, es posible actualizar los bloques A_{21} y A_{31}, y a continuación se procede a la actualización del resto.

Al finalizar el algoritmo, los elementos de A han sido reemplazados por los elementos de L.

4.1.5. Implementaciones basadas en rutinas de BLAS-3 denso

Implementación LAPACK PBTRF

La rutina PBTRF de *LAPACK* implementa el algoritmo $\text{PBTRF}_{\text{BLK}}$ realizando las siguientes operaciones en cada una de las iteraciones

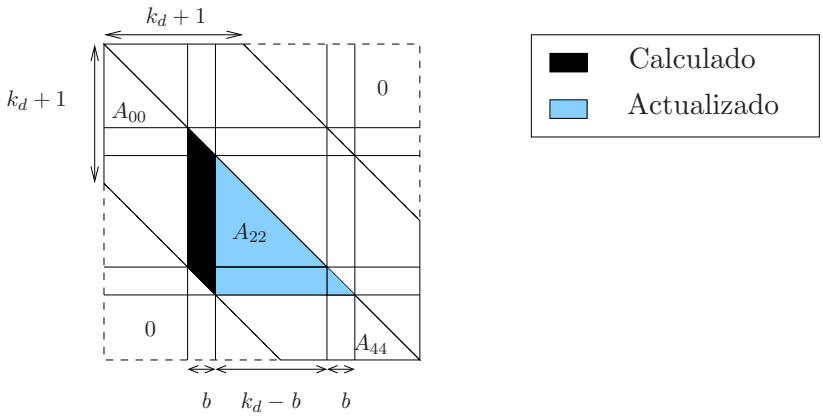


Figura 4.4: Acceso a los elementos durante una iteración (del algoritmo PBTRF$_{\text{BLK}}$ y) de la rutina PBTRF.

$$
\begin{aligned}
(\text{POTF2}) \quad & A_{11} && := \text{CHOL}_{UNB}(A_{11}), && (4.3)\\
(\text{TRSM}) \quad & A_{21} && := A_{21} \cdot \ \text{TRIL}(A_{11})^{-T}, && (4.4)\\
& A_{31} && := A_{31} \cdot \ \text{TRIL}(A_{11})^{-T}, && (4.5)\\
& W && := A_{31}, && (4.6)\\
(\text{TRSM}) \quad & W && := W \cdot \ \text{TRIL}(A_{11})^{-T}, && (4.7)\\
(\text{SYRK}) \quad & A_{22} && := A_{22} - A_{21} \cdot \ A_{21}^T, && (4.8)\\
(\text{GEMM}) \quad & A_{32} && := A_{32} - W \cdot \ A_{21}^T, && (4.9)\\
(\text{SYRK}) \quad & A_{33} && := A_{33} - W \cdot \ W^T, && (4.10)\\
& A_{31} && := W, && (4.11)
\end{aligned}
$$

La rutina PBTRF emplea POTF2 para la obtención de la factorización de *Cholesky* del bloque denso A_{11} (operación (4.3)). Esta rutina está basada en operaciones de *BLAS*-2, si bien, dado que se trata de un bloque de dimensión reducida, obtendrá buenas prestaciones. Una vez computado A_{11}, se calculan los bloques $A_{21}(L_{21})$ y $A_{31}(L_{31})$. El cálculo de L_{21} es ejecutado por la rutina *BLAS*-3 TRSM, rutina que se empleará igualmente para el bloque L_{31} en (4.7), pero dado que este último es un bloque triangular, antes es necesario copiar su contenido a un bloque rectangular (el espacio de trabajo W). Además, al realizar esta copia, es posible invocar a la rutina TRSM en (4.7), GEMM en (4.9) y SYRK en (4.10).

Una vez calculados los elementos de los bloques L_{11}, L_{21} y L_{31}, se actualizan los elementos de A_{22} y A_{32} con sendas llamadas a las rutinas SYRK y GEMM. A continuación se actualiza el bloque A_{33}, para lo que se invoca a la rutina SYRK, y se devuelve el contenido de W a A_{31}.

La mayor parte de los cálculos realizados en la rutina PBTRF son ejecutados por rutinas del tercer nivel de *BLAS*. Esta propiedad incrementa su eficiencia, ya que asegura una buena utilización del sistema de memoria jerarquizada y reduce el número de accesos a memoria.

Sin embargo, a pesar del uso de rutinas *BLAS*-3, la implementación presentada por PBTRF muestra ciertos problemas:

- Alta dependencia de la implementación *BLAS* utilizada.

- Ejecución de dos copias de matrices triangulares de tamaño $b \times b$.
- Uso de un espacio de trabajo de tamaño $b \times b$.
- Invocación de 6 rutinas en cada iteración.

Estas deficiencias son, precisamente, las que motivan las siguientes mejoras.

Implementación PBTRF_POTF2

El cálculo del bloque A_{11} se realiza por la rutina POTF2, que internamente invoca a núcleos de los niveles 1 y 2 del *BLAS*-denso. En esta versión se sustituye la llamada a la rutina POTF2 por una llamada a una nueva rutina, POTF2_INLINE. Esta nueva rutina es similar a POTF2, pero en ella no se invoca a DOT, GEMV y SCAL, sino que su código está embebido en la propia POTF2_INLINE.

Esta versión dedica menos tiempo a la invocación de rutinas, pero a cambio no utiliza ciertos núcleos optimizados de *BLAS*. Cabe esperar que para valores de b pequeños el tiempo ahorrado al evitar las invocaciones de rutinas sea superior a las ganacias que estas rutinas nos pueden ofrecer.

Implementación PBTRF_AM

Esta nueva versión persigue dos objetivos: reducir el sobrecoste de las invocaciones y trabajar con bloques de mayor tamaño. Las ganancias que puedan obtenerse con esta última mejora serán más visibles al utilizar versiones paralelas de *BLAS*, ya que al trabajar con bloques con un mayor número de datos, el grado de paralelismo también es mayor.

Las modificaciones introducidas por PBTRF_AM afectan también al esquema de almacenamiento. Esta rutina requiere que la matriz A esté almacenada en un espacio de memoria de $(k_d + b) \times n$ elementos, en los que las primeras $k_d + 1$ filas contienen la matriz A almacenada según el formato compacto para matrices simétricas banda empleado por *LAPACK* y *BLAS*, mientras que las últimas (b-1) filas contienen inicialmente elementos nulos. La figura 4.5 muestra este esquema de almacenamiento.

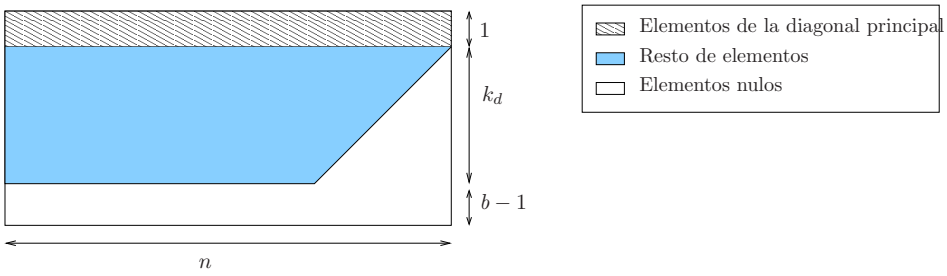


Figura 4.5: Esquema de almacenamiento de los elementos de A utilizado por la rutina PBTRF_AM. Las últimas $b-1$ filas inicialmente almacenan valores nulos.

Gracias a la modificación en el esquema de almacenamiento, los bloques A_{21} y A_{31} pueden unirse en un único bloque, $\bar{A}_{21}$, y con una sóla invocación a TRSM computar todo el bloque. De igual manera, los bloques A_{22}, A_{32} y A_{33} se pueden unir, formando el bloque $\bar{A}_{22}$, que se computa con una única invocación a SYRK.

De esta forma, las operaciones a ejecutar en cada iteración serán las siguientes:

$$(\text{POTF2}) \quad A_{11} := \text{CHOL}_{UNB}(A_{11}), \tag{4.12}$$

$$(\text{TRSM}) \quad \bar{A}_{21} := \bar{A}_{21} \cdot \text{TRIL}(A_{11})^{-T}, \tag{4.13}$$

$$(\text{SYRK}) \quad \bar{A}_{22} := \bar{A}_{22} - \bar{A}_{21} \cdot \bar{A}_{21}^{T} \tag{4.14}$$

La figura 4.6 muestra el patrón de acceso a los elementos de A durante una iteración de la rutina PBTRF_AM. De especial importancia es la forma en que se encuentran almacenados los elementos de la región formada por los bloques con los que se operará durante la presente iteración, denominada región activa. En esta rutina la región activa está formada por los bloques $\bar{A}_{11}$, $\bar{A}_{21}$ y $\bar{A}_{22}$. El bloque $\bar{A}_{11}$ se corresponde con el bloque A_{11} del algoritmo $\text{PBTRF}_{\text{BLK}}$, $\bar{A}_{21}$ con los bloques A_{21} y A_{31}, y $\bar{A}_{22}$ con los bloques A_{22}, A_{32} y A_{33}. La parte triangular inferior de $\bar{A}_{21}$ está almacenada en las $b-1$ filas inferiores, añadidas al almacenamiento para cumplir con este cometido, ya que de esta forma $\bar{A}_{21}$ es una matriz rectangular y se pueden realizar las operaciones (4.13) y (4.14). Igualmente se muestra en la figura que el bloque a la izquierda de la región activa almacena elementos de la matriz L ya calculados, mientras que los que están a la derecha de la misma son elementos de la matriz A que todavía no han sido modificados.

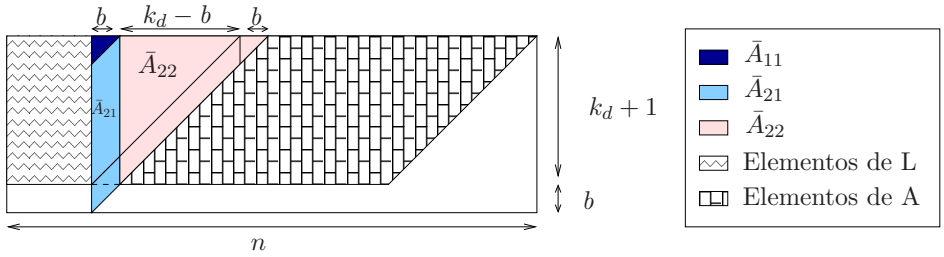


Figura 4.6: Acceso a los elementos de A realizado durante una iteración de la rutina PBTRF_AM.

PBTRF_AM reduce el número de invocaciones a rutinas *BLAS* a tan sólo tres por iteración, incrementando el tamaño de los bloques con los que trabaja cada rutina y aumentando, previsiblemente, las ventajas al utilizar *BLAS* paralelo.

Para conseguir este objetivo, ha sido necesario hacer crecer los requerimientos de almacenamiento en $n \times (b-1)$, circunstancia no demasiado costosa si tenemos en cuenta que, habitualmente, b toma valores pequeños.

Implementación PBTRF_AM_POTF2_INLINE

PBTRF_AM_POTF2 incluye en una rutina el enfoque con almacenamiento modificado visto en PBTRF_AM y el enfoque embebido de la rutina PBTRF_POTF2. Esta nueva rutina, basada en PBTRF_AM, invoca a POTF2_INLINE en lugar de a la rutina POTF2 para el cálculo de $\bar{A}_{11}$. A diferencia de POTF2, POTF2_INLINE incluye el código embebido de las rutinas DOT, SCAL y GEMV, para reducir con ello el tiempo dedicado a la invocación de rutinas.

Implementación PBTRF_MERGE

El hecho de ejecutar con una sola invocación varias de las operaciones propuestas por el algoritmo $\text{PBTRF}_{\text{BLK}}$ indudablemente presenta ciertas ventajas: simplifica el código, aumenta las prestaciones al reducir el número de llamadas a subrutinas, e incrementa el paralelismo y con ello las prestaciones al utilizar versiones paralelas de *BLAS*.

No obstante, la rutina PBTRF_AM precisa de una modificación en el esquema de almacenamiento, requiriendo mayor cantidad de memoria, y lo que probablemente es más problemático, utiliza un esquema de almacenamiento no incluido en la especificación *BLAS* y dependiente de b. Para solventar este problema se plantea la rutina PBTRF_MERGE que, a cambio del pequeño sobrecoste de realizar dos copias de la parte triangular inferior estricta de un bloque $b \times b$, permite calcular la factorización de Cholesky de forma similar a la empleada en la rutina PBTRF_AM.

La secuencia de operaciones realizadas en una iteración de PBTRF_MERGE es la siguiente:

$$
\begin{aligned}
(\text{POTF2}) \quad A_{11} &:= \text{CHOL}_{\text{UNB}}(A_{11}), && (4.15)\\
W &:= \text{TRIL}(A_{11}), && (4.16)\\
\text{STRIL}(A_{31}) &:= 0, && (4.17)\\
(\text{TRSM}) \quad \bar{A}_{21} &:= \bar{A}_{21} \cdot \ \text{TRIL}(W)^{-T}, && (4.18)\\
(\text{SYRK}) \quad \bar{A}_{22} &:= \bar{A}_{22} - \bar{A}_{21} \cdot \ \bar{A}_{21}^{-T}, && (4.19)\\
\text{TRIL}(A_{11}) &:= W && (4.20)
\end{aligned}
$$

Inicialmente, se computa el bloque A_{11} mediante una invocación a la rutina POTF2. A continuación, para simular que el bloque $\bar{A}_{21}$ está almacenado completamente (incluyendo aquellos elementos que se encuentran fuera de la banda), se realiza una copia de A_{11} en W (4.16), y se rellena con ceros la parte estrictamente inferior de A_{31} (4.17). Este proceso es el ilustrado en la figura 4.7.

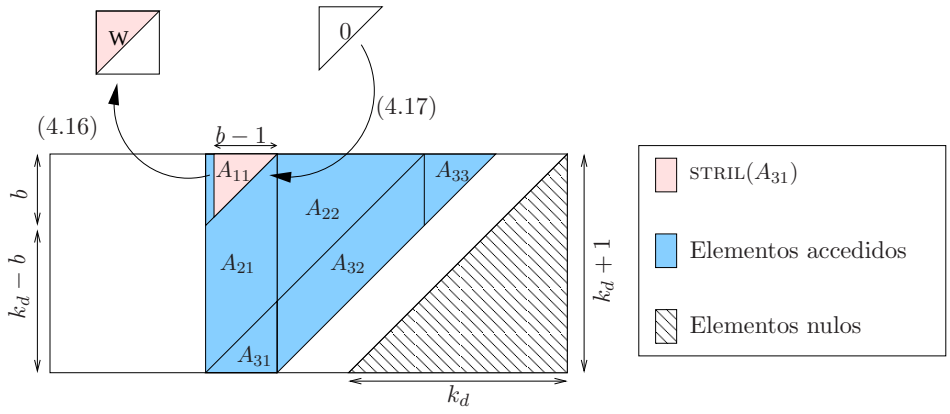


Figura 4.7: Copias de elementos de A realizadas durante una iteración de la rutina PBTRF_MERGE.

Las operaciones (4.16) y (4.17) se pueden ejecutar de forma simultánea mediante dos bucles anidados. Tras realizar las copias, se pueden ejecutar las operaciones de cálculo de $\bar{A}_{21}$ y de actualización de $\bar{A}_{22}$ mediante sendas llamadas a TRSM y SYRK. Finalmente, devolvemos los elementos copiados en W a su posición original, tal y como se muestra en la figura 4.8.

Implementación PBTRF_MERGE_POTF2_INLINE

De nuevo, se plantea la variante embebida en este caso aplicada a la rutina PBTRF_MERGE. Esta versión, a diferencia de PBTRF_MERGE, invoca a la rutina POTF2_INLINE, rutina que incluye el código embebido de las rutinas *BLAS* DOT, SCAL y GEMV, con el objetivo de reducir el tiempo requerido por la invocación de rutinas.

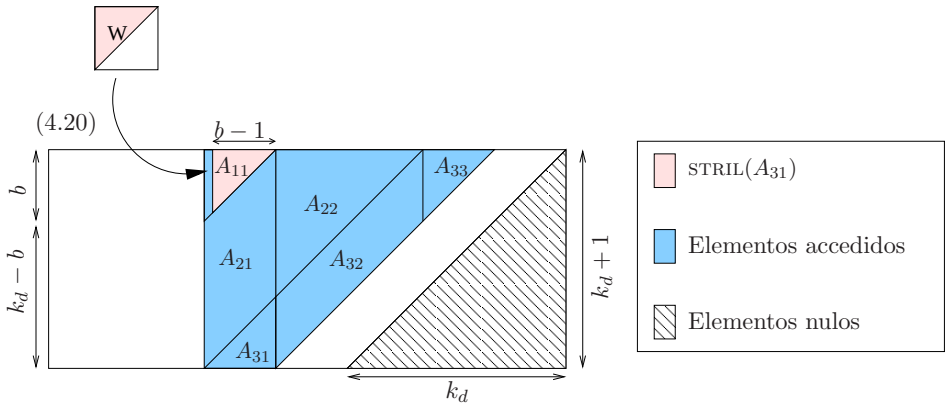


Figura 4.8: Copias de elementos de A realizadas durante una iteración de la rutina PBTRF_MERGE.

4.1.6. Resultados experimentales

Arquitectura ITANIUM

BLAS secuencial El estudio realizado en primer lugar evalúa la eficiencia en la actualización de cada bloque medida en términos del ratio entre el porcentaje de flops (operaciones en aritmética de coma flotante) y de tiempo dedicado. El objetivo es identificar aquellos bloques cuyo cálculo es más ineficiente. Los resultados mostrados en esta primeras gráficas corresponden a matrices con ancho de banda menor o igual a 300.

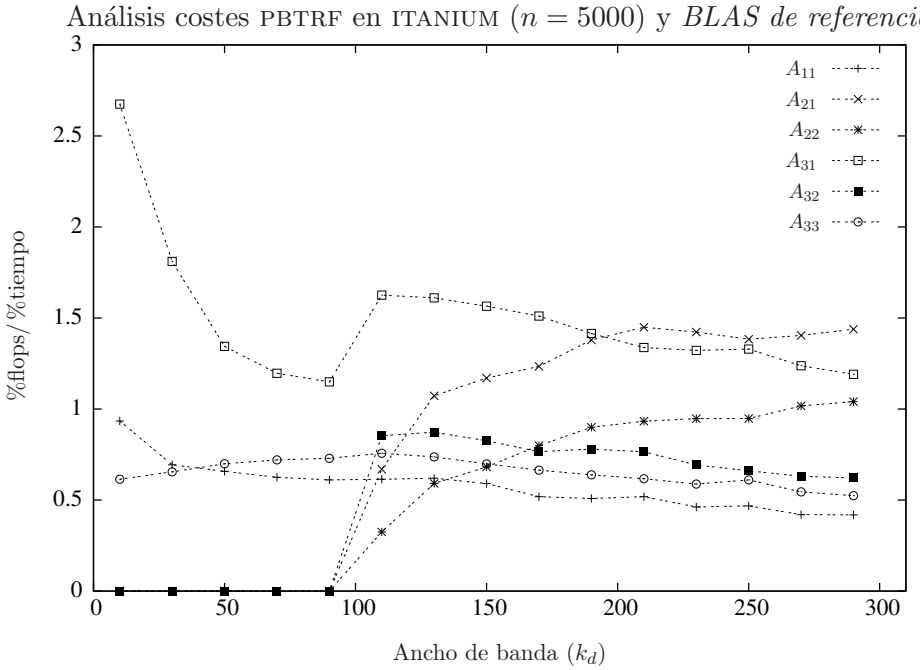


Figura 4.9: Eficiencia en el cálculo de cada uno de los bloques en PBTRF y *BLAS de referencia*.

La figura 4.9 muestra el ratio entre los porcentajes de flops y de tiempo de ejecución dedicado a cada uno de los bloques de la región activa al invocar a las rutinas de la biblioteca *BLAS de*

referencia. Como se puede observar, los cálculos de los bloques A_{11}, A_{33} y A_{32} son los más ineficientes. Por contra, las actualizaciones de los bloques A_{21} y A_{31} (ambas ejecutadas por la rutina TRSM) son muy eficientes. Los bloques A_{22} y A_{33} son actualizados por la rutina SYRK, presentando bajas prestaciones cuando se realiza con bloques pequeños (A_{33}) y buenas prestaciones al operar con bloques mayores (A_{22}), característica ésta que beneficiará especialmente a las implementaciones PBTRF_AM y PBTRF_MERGE. Respecto a la elección del tamaño de bloque óptimo, cuando se opera con una matriz de banda estrecha el tamaño de bloque se selecciona de modo que la región activa está formada únicamente por los bloques A_{11}, A_{31} y A_{33}. Esto se traduce en que el ratio $\%flops/\%tiempo$ del resto de bloques es 0.

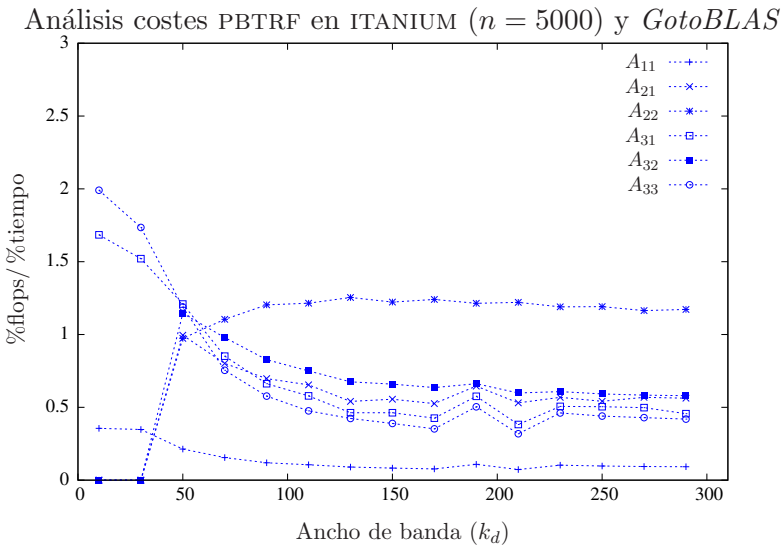


Figura 4.10: Eficiencia en el cálculo de cada uno de los bloques en PBTRF y *GotoBLAS*.

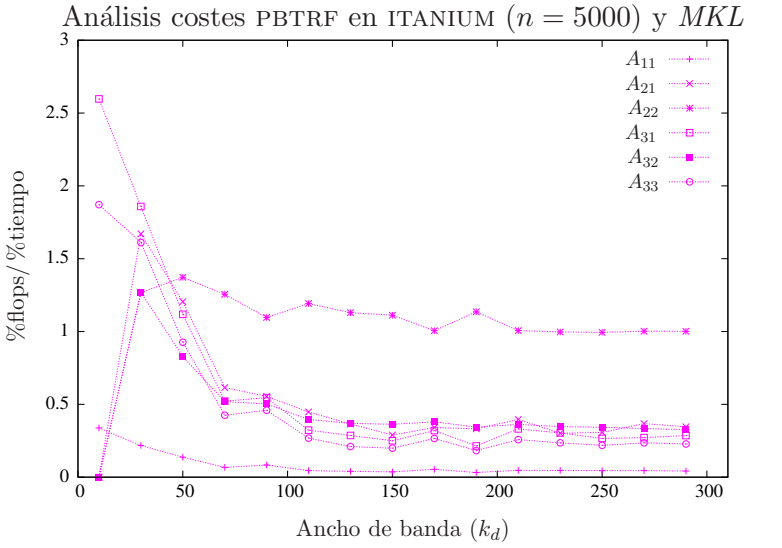


Figura 4.11: Eficiencia en el cálculo de cada uno de los bloques en PBTRF y *MKL*.

La figura 4.10 muestra los resultados del estudio realizado con *GotoBLAS*. Los datos del blo-

que A_{21} muestran que la rutina TRSM de *GotoBLAS* presenta mejores prestaciones al operar con matrices rectangulares que al hacerlo con matrices cuadradas de talla pequeña (bloque A_{31}). La actualización más ineficiente es la del bloque A_{11}, realizada por la rutina POTF2. Por contra, la más eficiente es la actualización de A_{22}.

Finalmente, en la figura 4.11 se muestran los resultados para la rutina PBTRF y *MKL*. De nuevo destaca la eficiencia de la actualización del bloque A_{22}, y con ello la eficiencia de la rutina SYRK. Por contra, la actualización de A_{11} es más ineficiente. Por último comentar la diferencia entre la eficiencia mostrada por el bloque A_{22} y el bloque A_{33}, pese a que ambos bloques son actualizados por SYRK. Esto demuestra que esta rutina al operar con bloques de pequeño tamaño no alcanza grandes prestaciones.

La figura 4.12 resume las prestaciones obtenidas por la rutina PBTRF al emplear las implementaciones secuenciales de *BLAS*: *BLAS de referencia*, *GotoBLAS* y *MKL*. Estas dos últimas implementaciones obtienen resultados similares y muy superiores a los de *BLAS de referencia*. Aunque muy ligeramente, *MKL* alcanza mayores prestaciones con matrices de banda estrecha, mientras que *GotoBLAS* mejora a *MKL* con matrices de banda ancha.

Las prestaciones de las nuevas implementaciones se muestran en la gráfica 4.13. Tanto en el caso de *GotoBLAS* como en el de *MKL*, las nuevas implementaciones mejoran sólo ligeramente las prestaciones de la rutina PBTRF incluida en *LAPACK*. Las implementaciones PBTRF_AM y PBTRF_MERGE generan idénticas prestaciones, por lo que es recomendable el uso de PBTRF_MERGE ya que no requiere modificaciones en el esquema de almacenamiento de A.

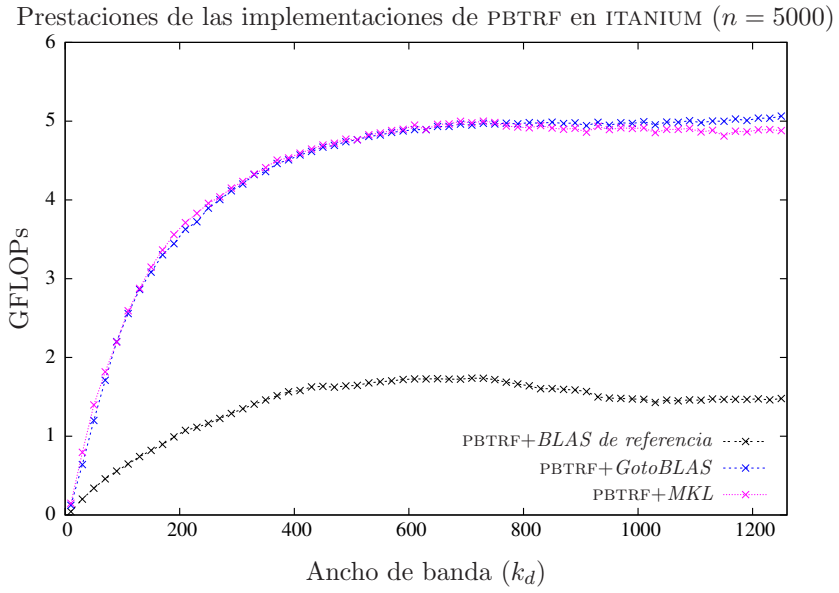


Figura 4.12: Comparativa de PBTRF empleando diferentes implementaciones de *BLAS* en ITANIUM.

La figura 4.14 muestra las prestaciones de las mejores implementaciones, y en ella se puede observar como la nueva rutina PBTRF_MERGE mejora muy ligeramente a la rutina PBTRF.

BLAS paralelo A continuación se repite el estudio relativo a la eficiencia en la actualización de cada uno de los bloques de la región activa de la rutina PBTRF, en este caso para las implementaciones paralelas de *GotoBLAS* y *MKL*. No existe una implementación paralela del *BLAS de referencia*, con lo que esta biblioteca no se incluye en el siguiente estudio.

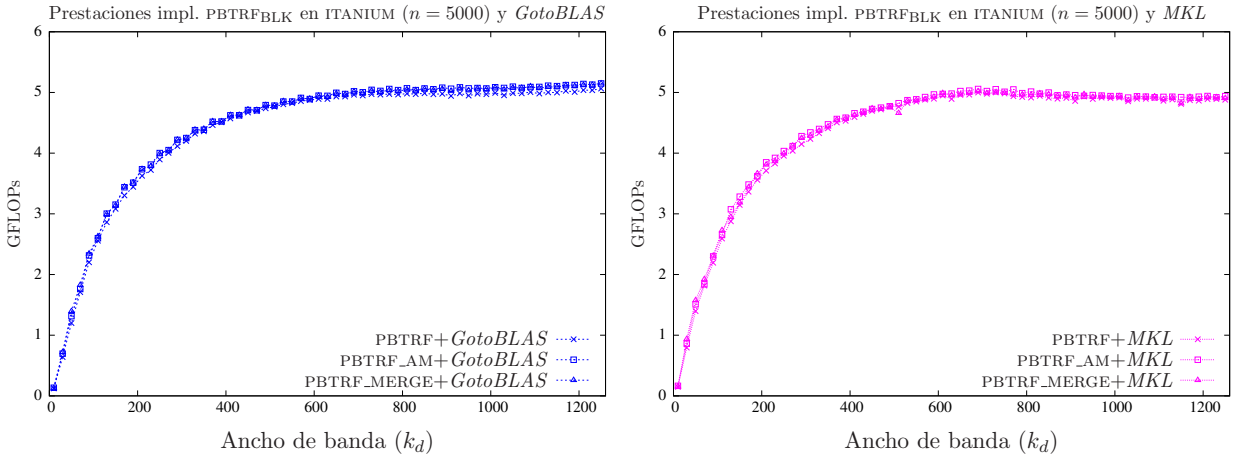


Figura 4.13: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

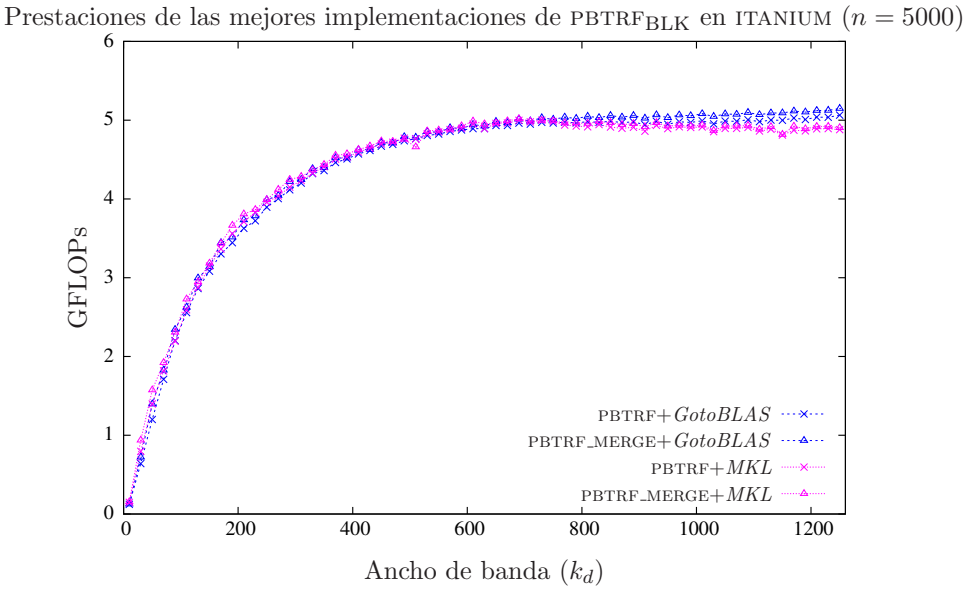


Figura 4.14: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

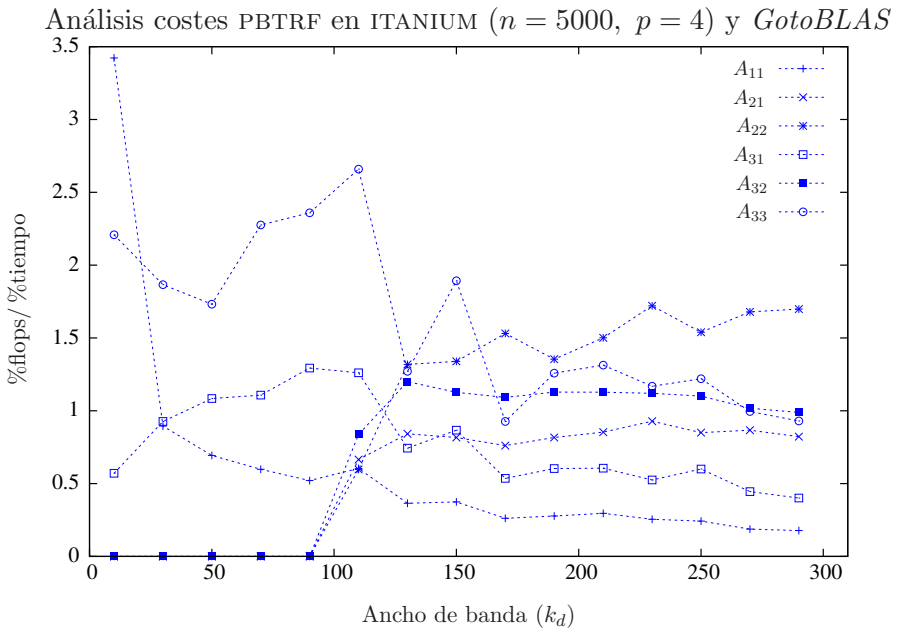


Figura 4.15: Eficiencia en el cálculo de cada uno de los bloques en PBTRF y *GotoBLAS*.

La figura 4.15 recoge esta distribución para la rutina PBTRF y *GotoBLAS*. Un punto interesante de la gráfica es que para $k_d \leq 100$ el valor óptimo de b supone que la región activa se componga únicamente de los bloques A_{11}, A_{31} y A_{33}. De esta forma se reduce el número de invocaciones a rutinas *BLAS* y se explotan las buenas prestaciones de la rutina SYRK (que actualiza A_{33}) al operar con matrices de tamaño medio. Por contra, al aumentar el valor de b, aumenta el tamaño del bloque A_{11} y la rutina POTF2 empeora sus prestaciones. Para anchos de banda superiores a 100, el tamaño de bloque decrece drásticamente, volcando la mayor parte del trabajo sobre el bloque A_{22}, de forma que se aprovecha de nuevo las buenas prestaciones de la rutina SYRK al operar con matrices de talla media o alta (en este caso con el bloque A_{22}). Podemos concluir que, al emplear *GotoBLAS*, la elección del tamaño de bloque óptimo está muy ligada a las prestaciones de las rutinas POTF2 y SYRK, especialmente de esta última.

En el caso de *MKL* (figura 4.16), existe una diferencia clara en las prestaciones obtenidas con los bloque de mayor tamaño y los de menor tamaño. Así, los cálculos de A_{11}, A_{31} y A_{33} resultan los más ineficientes. Esta circunstancia beneficiará a las nuevas implementaciones PBTRF_AMy PBTRF_MERGE.

La figura 4.17 contiene las prestaciones obtenidas con las implementaciones presentadas en esta sección, a excepción de aquellas que incluyen código embebido, ya que no aportan mejoras. La gráfica de la izquierda realiza la comparativa de las diferentes rutinas empleando *GotoBLAS*, mientras que la de la derecha es su análoga con *MKL*. En ambos casos, la rutina con el almacenamiento modificado, PBTRF_AM, es la más eficiente. No obstante, la rutina PBTRF_MERGE presenta unos resultados similares y no requiere modificaciones en el esquema de almacenamiento de la matriz. Por este motivo parece recomendable el uso de esta rutina.

La figura 4.18 ilustra los resultados de la rutina de *LAPACK* PBTRF y de la nueva implementación PBTRF_MERGE, tanto para *GotoBLAS* como para *MKL*. La ganancia obtenida por PBTRF_MERGE sobre PBTRF queda patente con ambas bibliotecas *BLAS*.

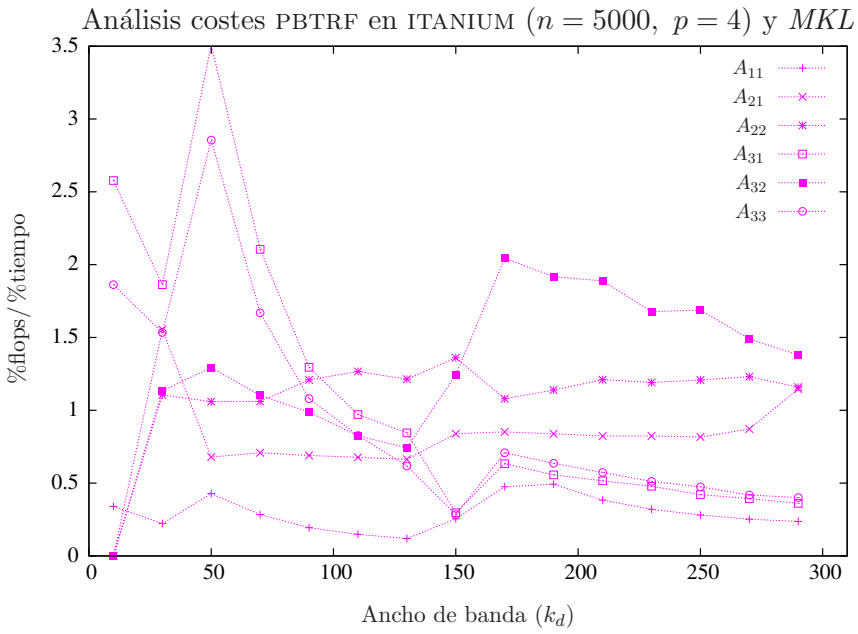


Figura 4.16: Eficiencia en el cálculo de cada uno de los bloques en PBTRF y *MKL*.

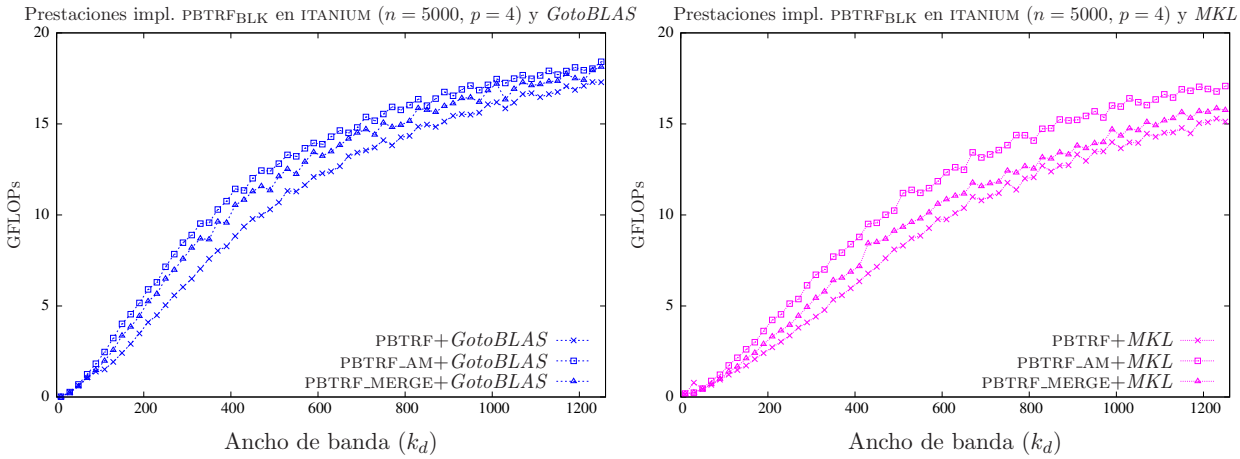


Figura 4.17: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

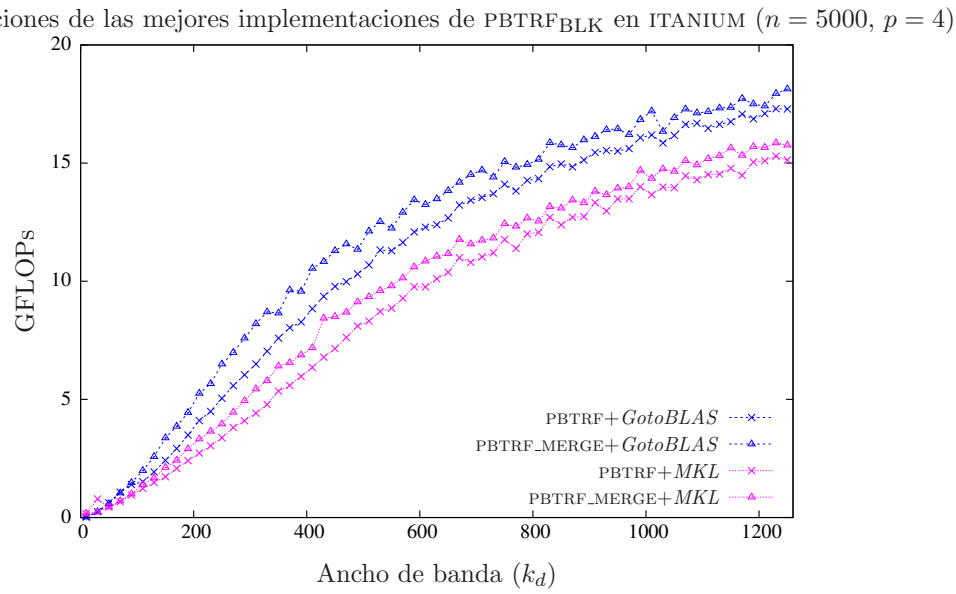


Figura 4.18: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

Arquitectura XEON

BLAS secuencial A continuación se muestran los resultados del estudio de la eficiencia de la actualización de cada bloque en la rutina PBTRF y las diferentes implementaciones de *BLAS* estudiadas.

Los resultados con *BLAS de referencia* (figura 4.19) indican que la mayor eficiencia se obtiene con el bloque A_{22}. Sorprende en este caso la eficiencia del bloque A_{11}.

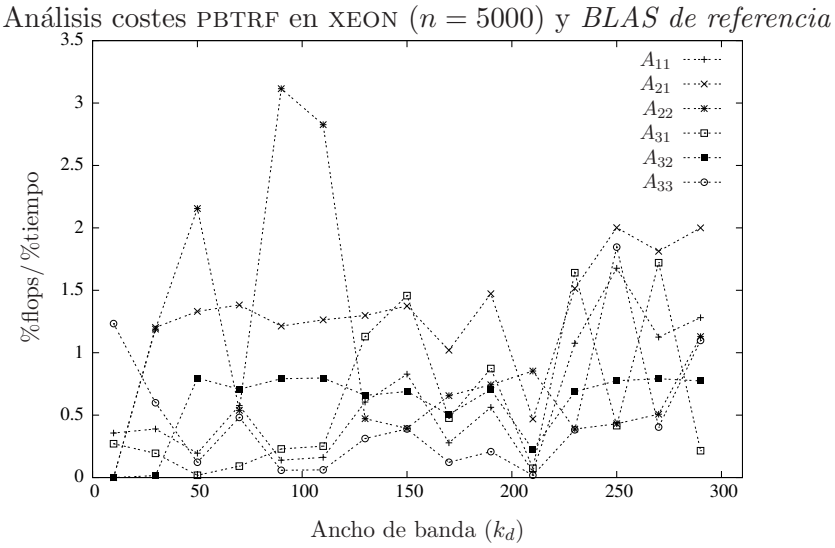


Figura 4.19: Eficiencia en el cálculo de cada uno de los bloques en PBTRF y *BLAS de referencia*.

La figura 4.20 muestra el ratio entre el porcentaje de flops y de tiempo dedicados para la actualización de cada uno de los bloques de la región activa cuando la rutina PBTRF invoca a

núcleos computacionales de la librería *GotoBLAS*. De nuevo, al operar con una matriz de banda estrecha, el tamaño de bloque óptimo se selecciona de forma que la región activa esté formada únicamente por los bloques A_{11}, A_{31} y A_{33}. De esta forma se reduce el número de invocaciones a rutinas. Cuando el ancho de banda de la matriz aumenta, el bloque más eficiente es A_{22}. Una vez más las prestaciones de la rutina SYRK difieren al operar con bloques de pequeña talla (A_{33}) y de talla mayor (A_{22}).

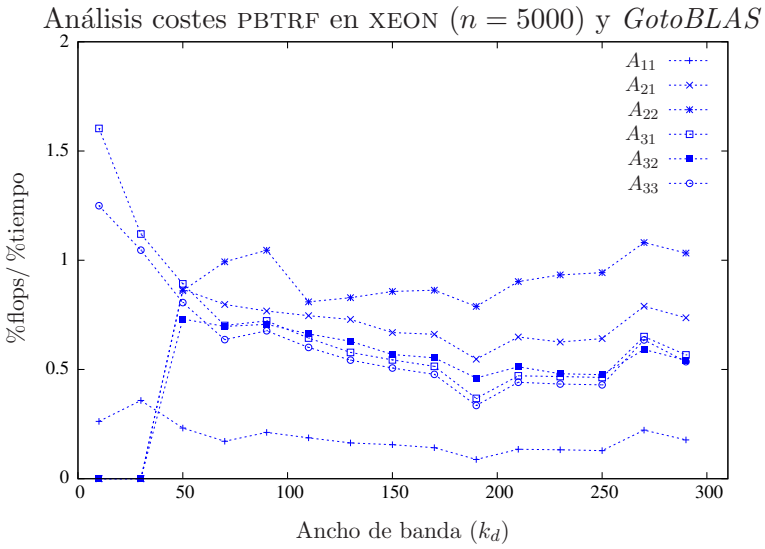


Figura 4.20: Eficiencia en el cálculo de cada uno de los bloques en PBTRF y *GotoBLAS*.

Las resultados obtenidos con la biblioteca *MKL* se ilustran en la figura 4.21. Al igual que sucede con *GotoBLAS*, las mayores prestaciones se obtienen con el bloque A_{22}, mientras que A_{11} es el que obtiene las peores prestaciones (recordemos que es el único bloque que se actualiza mediante una rutina *BLAS*-2, mientras que el resto son actualizados por rutinas *BLAS*-3).

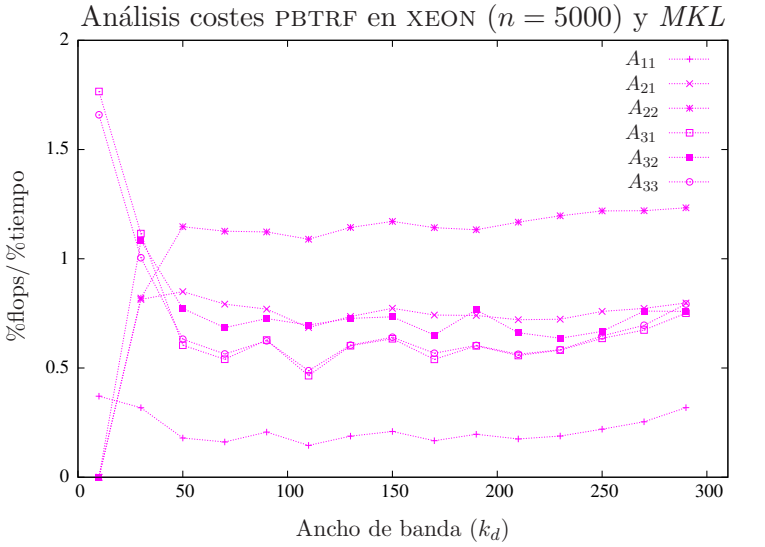


Figura 4.21: Eficiencia en el cálculo de cada uno de los bloques en PBTRF y *MKL*.

A continuación se evalúan las diferentes implementaciones sobre la arquitectura XEON y versiones secuenciales de *BLAS*. Aquí también las versiones con código embebido se han omitido de las gráficas puesto que no ofrecen mejoras en las prestaciones.

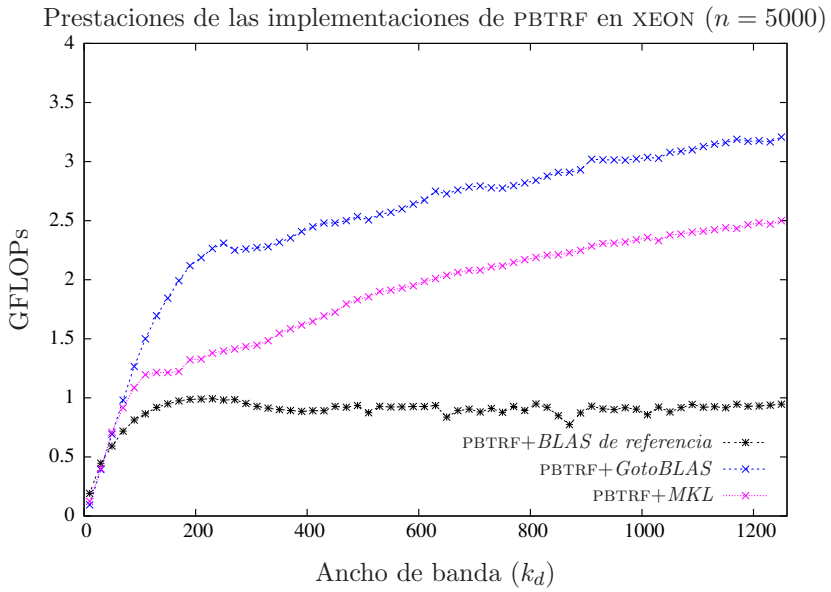


Figura 4.22: Comparativa de PBTRF empleando diferentes implementaciones de *BLAS* en XEON.

Las prestaciones de la rutina PBTRF y las diferentes bibliotecas *BLAS* estudiadas se presentan en la figura 4.22. El *BLAS de referencia* obtiene las peores prestaciones, mientras que las implementaciones de *BLAS* específicas para la arquitectura, *GotoBLAS* y *MKL*, ofrecen un rendimiento muy superior. Como se observa en la gráfica, para esta operación y arquitectura, *GotoBLAS* se presenta como la mejor opción.

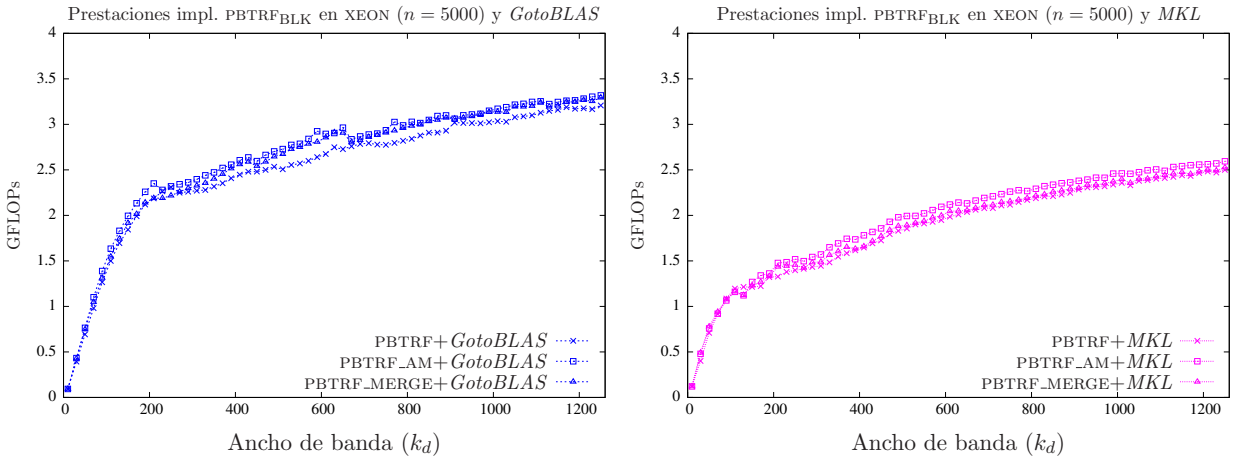


Figura 4.23: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

Los resultados de las implementaciones PBTRF_AM y PBTRF_MERGE se comparan con los de la rutina *LAPACK* (PBTRF) en la figura 4.23. La nueva rutina PBTRF_AM es la que ofrece mejores prestaciones, tanto con *GotoBLAS* como con *MKL*. La rutina PBTRF_MERGE iguala las prestaciones

de PBTRF_AM con *GotoBLAS* al operar com matrices de banda media o ancha, mientas que si la biblioteca empleada es *MKL*, sus prestaciones son inferiores a las de PBTRF_AM, pero superiores a las de PBTRF. En consecuencia, las dos nuevas implementaciones propuestas presentan resultados ligeramente superiores que la rutina *LAPACK*. Pese a que la rutina PBTRF_AM es más eficiente que PBTRF_MERGE, la diferencia de prestaciones no parece ser suficiente como para justificar el cambio en el esquema de almacenamiento que precisa.

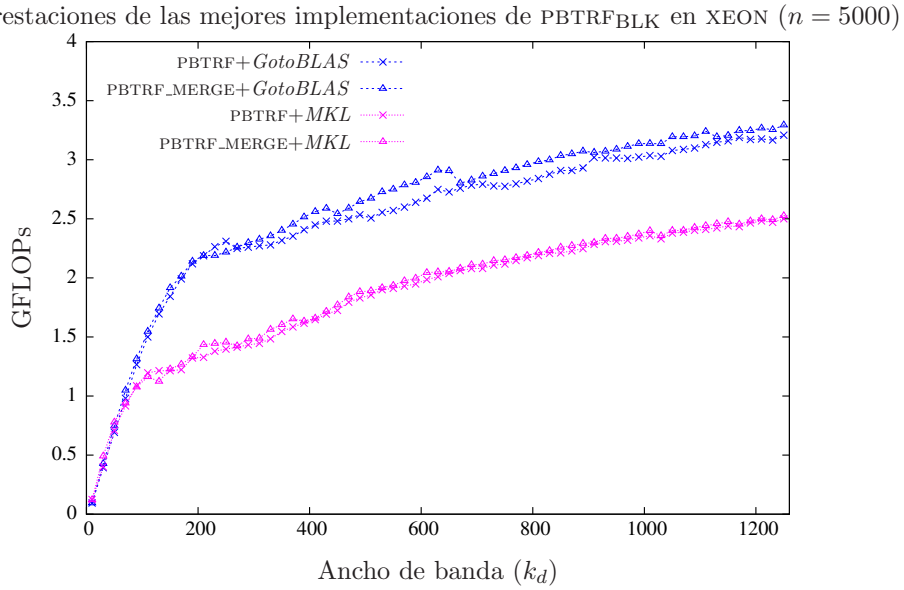


Figura 4.24: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

Finalmente la figura 4.24 presenta las prestaciones de las rutinas PBTRF y PBTRF_MERGE, para *GotoBLAS* y *MKL*. Las mejores prestaciones con matrices de banda media o ancha se obtienen invocando a rutinas de *GotoBLAS*, mientras que si se opera con matrices de banda estrecha ambas implementaciones *BLAS* son igualmente eficientes. La nueva rutina PBTRF_MERGE es ligeramente más rápida que la rutina *LAPACK* PBTRF.

BLAS paralelo Seguidamente se evalúa el rendimiento por bloques de la rutina PBTRF basada en las bibliotecas de *BLAS* paralelo *GotoBLAS* (figura 4.25) y *MKL* (figura 4.26).

Con ambas implementaciones de *BLAS*, cuando la matriz presenta una banda estrecha el tamaño de bloque óptimo provoca que la región activa esté formada únicamente por los bloques A_{11}, A_{31} y A_{33}. Cuando el ancho de banda aumenta, la actualización más eficiente es la del bloque A_{22} (rutina SYRK) y la menos eficiente la de A_{11} (rutina POTF2).

La figura 4.27 recoge los resultados obtenidos con las nuevas rutinas PBTRF_AM y PBTRF_MERGE así como con la rutina de *LAPACK* PBTRF. Las dos nuevas rutinas son más eficientes que la incluida en *LAPACK*, especialmente PBTRF_AM que es hasta un 20 % más rápida. No obstante la aplicación de PBTRF_AM conlleva un mayor coste espacial y la modificación del esquema de almacenamiento de la matriz, hecho que desaconseja su uso en favor de PBTRF_MERGE. En el caso de *GotoBLAS*, las prestaciones de PBTRF_AM y PBTRF_MERGE son muy similares para cualquier ancho de banda de la matriz. En el caso de *MKL*, ambas rutinas tienen similares prestaciones cuando la matriz presenta un ancho de banda $k_d > 600$, mientras que para anchos de banda inferiores las prestaciones de PBTRF y PBTRF_MERGE son semejantes.

La figura 4.28 muestra las prestaciones de la mejor implementación de entre las nuevas propues-

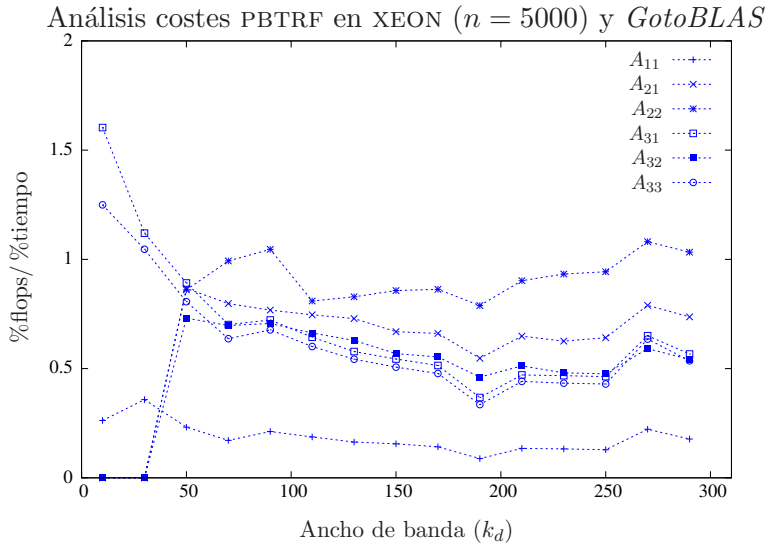


Figura 4.25: Eficiencia en el cálculo de cada uno de los bloques en PBTRF y *GotoBLAS*.

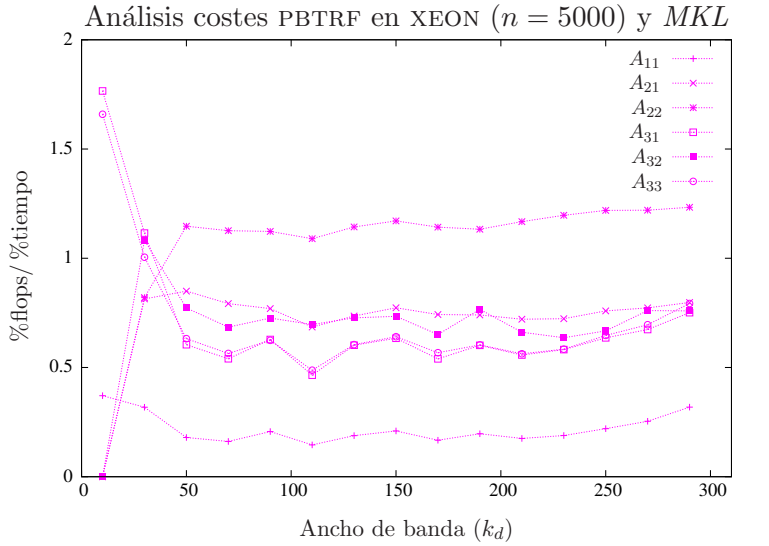


Figura 4.26: Eficiencia en el cálculo de cada uno de los bloques en PBTRF y *MKL*.

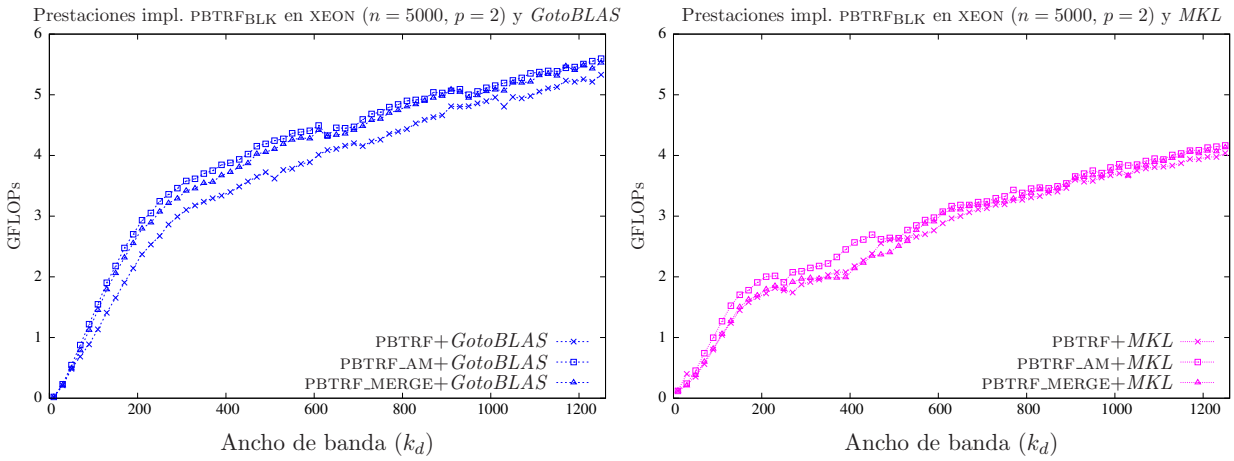


Figura 4.27: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

tas, PBTRF_MERGE, y de la rutina PBTRF. Si la biblioteca invocada es *MKL*, la mejora introducida por PBTRF_MERGE es reducida, mientras que si se emplea *GotoBLAS* la nueva rutina presenta prestaciones claramente superiores.

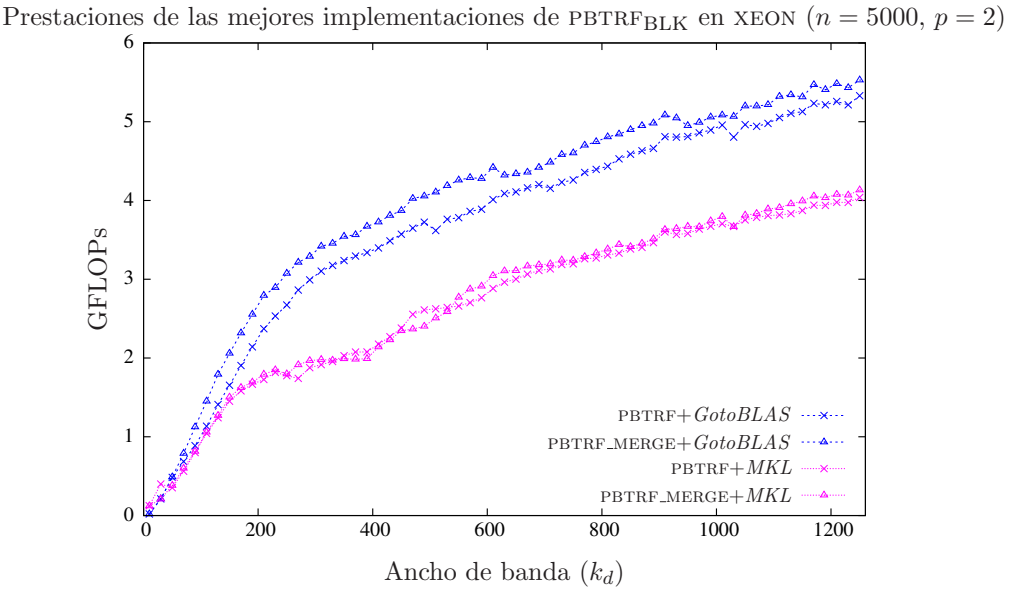


Figura 4.28: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

4.1.7. Conclusiones

Se ha realizado un estudio para evaluar la eficiencia de la actualización de cada uno de los bloques de la región activa. Este estudio se basa en la comparación de los porcentajes de tiempo y operaciones aritméticas empleados en la actualización de cada uno de los bloques. De este estudio se extrae que la actualización de A_{22} realizada mediante la rutina SYRK es muy eficiente, y por ello el tamaño de bloque se selecciona de forma que la mayor parte de las operaciones se dediquen a la actualización de este bloque. Por contra, el cálculo del bloque A_{11} por parte de la rutina POTF2 es

muy ineficiente. En el caso particular de que la matriz presente una banda estrecha, el tamaño de bloque óptimo es aquel que provoca que la región activa esté formada únicamente por los bloques A_{11}, A_{31} y A_{33}, de forma que el número de invocaciones a rutinas *BLAS* se reduce y se evitan operaciones con bloques de dimensión reducida.

También se han evaluado las prestaciones de las diferentes implementaciones para la factorización de Cholesky. En esta evaluación se han omitido los resultados de las implementaciones con código embebido por no aportar mejoras en las prestaciones. Las nuevas rutinas PBTRF_AM y PBTRF_MERGE se han comparado con la rutina *LAPACK* PBTRF, demostrándose su mayor eficiencia, en especial de PBTRF_AM. El principal problema planteado por la utilización de PBTRF_AM es que requiere modificar el esquema de almacenamiento de la matriz es almacenada. Las ganancias obtenidas por esta rutina no son suficientes como para justificar el cambio en el almacenamiento de la matriz. Por contra, la rutina PBTRF_MERGE no precisa cambio alguno en el esquema de almacenamiento y sí que alcanza mejoras de rendimiento respecto a PBTRF.

Como era previsible, las nuevas rutinas son más beneficiadas por el uso de versiones paralelas de *BLAS*, ya que operan con bloques de mayor tamaño y reducen el número de operaciones con bloques de tamaño reducido. Esta mejora se puede observar con ambas arquitecturas y versiones de *BLAS* y es previsible que las ganancias aumenten conforme se incremente el número de procesadores empleados.

4.2. Factorización LU

La operación tratada en esta sección es

$$L_{n-2}^{-1} \cdot P_{n-2} \cdots L_1^{-1} \cdot P_1 \cdot L_0^{-1} \cdot P_0 \cdot A = U, \tag{4.21}$$

donde $A \in \mathbb{R}^{n \times n}$ es una matriz banda con ancho de banda superior e inferior k_u y k_l respectivamente; $P_0,\ P_1, \ldots, P_{n-2} \in \mathbb{R}^{n \times n}$ son matrices de permutación; $L_0,\ L_1, \ldots, L_{n-2} \in \mathbb{R}^{n \times n}$ son transformaciones de Gauss y $U \in \mathbb{R}^{n \times n}$ es una matriz triangular superior con ancho de banda $k_u + k_l$. Si bien es posible reordenar las matrices de permutación y de Gauss en (4.21), obteniendo una expresión para la factorización de la forma

$$P \cdot A = L \cdot U, \tag{4.22}$$

la matriz triangular inferior L obtenida de este modo, en general , no presenta una estructura banda, requiriendo un coste de almacenamiento muy superior. En consecuencia, estas permutaciones no se realizan en la factorización LU de matrices banda. En la práctica esto significa que, cuando se factoriza una columna de la matriz A, las permutaciones sólo se aplican sobre las columnas de la matriz a la derecha de la región factorizada y no a la izquierda (sobre el factor L) tal y como es habitual en la factorización LU con pivotamiento parcial. En lo que resta de la sección, por simplicidad, nos referiremos a la factorización en (4.22), si bien la factorización realmente calculada es la que aparece en (4.21).

La factorización LU se emplea principalmente, al igual que la factorización de Cholesky, en la resolución de sistemas de ecuaciones lineales. A diferencia de esta última, la factorización LU puede aplicarse a cualquier matriz con independencia de cúal sea su estructura. El coste computacional ligado a la factorización de Cholesky es menor que el de la factorización LU, motivo por el cual la factorización LU se emplea únicamente cuando la matriz no es SPD.

En esta sección se diseñan diferentes implementaciones para (4.21) cuando la matriz A es una matriz banda. Esta operación está incluida dentro de la biblioteca *LAPACK* y recibe el nombre de GBTRF en su versión por bloques y GBTF2 en su variante escalar.

Algorithm: $[A, p] := \text{GBTRF}_{\text{UNB}}(A, k_u, k_l, p)$

Partition $A \rightarrow \left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right), p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$

where A_{TL} is 0×0; A_{MM} is $k_l \times k_u + k_l$; and p_T has 0 elements

while $m(A_{TL}) < m(A)$ **do**

Repartition

$$\left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & a_{01} & A_{02} & & \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T & & \\ \hline A_{20} & a_{21} & A_{22} & a_{23} & A_{24} \\ \hline & & a_{32}^T & \alpha_{33} & a_{34}^T \\ \hline & & A_{42} & a_{43} & A_{44} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

where α_{11} is scalar; α_{33} is scalar if $m(A_{BR}) > 0$; and π_1 is a scalar

---

$$\begin{array}{rcl} \pi_1 & := & \text{COMPUTAR\_PIVOTE}([\alpha_{11};\ a_{21}], \pi_1) \\ A_{MM} & := & p(\pi_1)(A_{MM}, \pi_1) \\ a_{21} & := & a_{21}/\alpha_{11} \cdot a_{21} \\ A_{22} & := & A_{22} - a_{21} \cdot a_{12}^T \end{array}$$

Continue with

$$\left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & a_{01} & A_{02} & & \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T & & \\ \hline A_{20} & a_{21} & A_{22} & a_{23} & A_{24} \\ \hline & & a_{32}^T & \alpha_{33} & a_{34}^T \\ \hline & & A_{42} & a_{43} & A_{44} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Figura 4.29: Algoritmo $\text{GBTRF}_{\text{UNB}}$ para la factorización $P \cdot A = L \cdot U$.

4.2.1. Algoritmo $\text{GBTRF}_{\text{UNB}}$

El algoritmo $\text{GBTRF}_{\text{UNB}}$ en la figura 4.29 calcula la factorización LU con pivotamiento parcial de una matriz general banda. El algoritmo recorre la matriz de izquierda a derecha a lo largo de su diagonal principal y durante la iteración j realiza las siguientes operaciones:

- Inicialmente se elige como pivote el mayor elemento en valor absoluto del vector columna $[\alpha_{11};\ a_{21}]$, y se almacena en π_1 la fila a la que pertenece el pivote.

- Seguidamente se permutan las filas j y π_1 del bloque A_{MM}, de forma que α_{11} tome el valor del pivote. En el algoritmo denotamos la permutación que produce este intercambio de filas como $P(\pi_1)$. La secuencia de pivotamientos aplicada queda registrada en el vector p, de n elementos.

- En tercer lugar, se calculan los elementos del vector a_{21} dividiéndolos por el pivote.

- Finalmente se actualizan los elementos de A_{22} con el producto entre $a_{21} \cdot a_{12}$.

4.2.2. Implementación LAPACK GBTF2

El algoritmo GBTRF$_{\text{UNB}}$ es el implementado por la rutina *LAPACK* GBTF2. Esta rutina, por motivos de economía de almacenamiento, guarda los factores L y U obtenidos durante la factorización (4.22) sobreescribiendo a la matriz A. Los elementos de $L_0, L_1, .., L_{n-2}$ (también conocidos como multiplicadores) se almacenan sobre la parte triangular inferior estricta de A, puesto que no es preciso almacenar los elementos de la diagonal principal de L al tomar todos ellos el valor 1. Por su parte, debido al pivotamiento, el ancho de banda superior del factor U toma un valor entre k_u y $k_u + k_l$; por este motivo la rutina *LAPACK* precisa que la matriz A se encuentre inicialmente almacenada usando $k_l + k_u + 1 + k_l$ filas, de la cuales las primeras k_l filas, inicialmente a ceros, son necesarias para almacenar los elementos de U. La figura 4.30 muestra la forma en que los elementos de A son almacenados y como son reemplazados por los de los factores L y U.

$$\begin{array}{|cccccc|}\hline \alpha_{00} & \alpha_{01} & * & * & * & * \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & * & * & * \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} & * & * \\ * & \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} & * \\ * & * & \alpha_{42} & \alpha_{43} & \alpha_{44} & \alpha_{45} \\ * & * & * & \alpha_{53} & \alpha_{54} & \alpha_{55} \\ \hline \end{array} \qquad \begin{array}{|cccccc|}\hline * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & \alpha_{01} & \alpha_{12} & \alpha_{23} & \alpha_{34} & \alpha_{45} \\ \alpha_{00} & \alpha_{11} & \alpha_{22} & \alpha_{33} & \alpha_{44} & \alpha_{55} \\ \alpha_{10} & \alpha_{21} & \alpha_{32} & \alpha_{43} & \alpha_{54} & * \\ \alpha_{20} & \alpha_{31} & \alpha_{42} & \alpha_{53} & * & * \\ \hline \end{array} \qquad \begin{array}{|cccccc|}\hline * & * & * & \mu_{03} & \mu_{14} & \mu_{25} \\ * & * & \mu_{02} & \mu_{13} & \mu_{24} & \mu_{35} \\ * & \mu_{01} & \mu_{12} & \mu_{23} & \mu_{34} & \mu_{25} \\ \mu_{00} & \mu_{11} & \mu_{22} & \mu_{33} & \mu_{44} & \mu_{55} \\ \lambda_{10} & \lambda_{21} & \lambda_{32} & \lambda_{43} & \lambda_{54} & * \\ \lambda_{20} & \lambda_{31} & \lambda_{42} & \lambda_{53} & * & * \\ \hline \end{array}$$

Figura 4.30: Almacenamiento de una matriz 6×6 con anchos de banda superior e inferior $k_u = 1$ y $k_l = 2$ respectivamente (izquierda); almacenamiento según el esquema utilizado por la rutina GBTF2 de *LAPACK* (centro); almacenamiento de la matriz resultado de la factorización LU con pivotamiento, donde $\mu_{i,j}$ representa el elemento (i, j) del factor U y $\lambda_{i,j}$ representa el elemento i-ésimo de la transformación L_j.

La rutina *LAPACK* GBTF2 implementa el algoritmo GBTRF$_{\text{UNB}}$ invocando a rutinas de *BLAS*. La secuencia de operaciones y rutinas que invoca esta rutina durante la iteración j es la siguiente:

1. Establecer como pivote, el mayor elemento de α_{11} y a_{21}:

$$\pi_1 := \text{IDAMAX} \begin{pmatrix} \alpha_{11} \\ a_{21} \end{pmatrix}. \tag{4.23}$$

2. Permutar las filas j e π_1 en el bloque A_{MM}:

$$(\text{LASWP}) \quad \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & a_{22} \end{pmatrix} := P(\pi_1) \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & a_{22} \end{pmatrix}. \tag{4.24}$$

3. Actualizar los bloques a_{21} y A_{22}:

$$(\text{SCAL}) \quad a_{21} := a_{21}/\alpha_{11}, \tag{4.25}$$

$$(\text{GER}) \quad A_{22} := A_{22} - a_{21}a_{12}^T. \tag{4.26}$$

Como se puede observar, GBTF2 invoca a cuatro rutinas *BLAS*. La mayor carga de trabajo recae sobre las actualizaciones (4.25) y (4.26), que son ejecutadas por las rutinas SCAL y GER de los niveles 1 y 2 de *BLAS* respectivamente.

Los requerimientos en cuanto a cantidad de memoria son reducidos. Por contra, cada elemento de A es accedido hasta en k_u+k_l ocasiones. Por otro lado, las operaciones aritméticas son ejecutadas por rutinas de los niveles 1 y 2 de *BLAS*, mientras que el uso de rutinas *BLAS*-3 puede aprovechar mejor la localidad de referencia.

4.2.3. Algoritmo GBTRF$_{\text{BLK}}$

El algoritmo GBTRF$_{\text{BLK}}$ descrito en la figura 4.31 calcula la factorización LU de una matriz banda principalmente mediante operaciones matriz-matriz. En consecuencia, este algoritmo permite su implementación mediante invocaciones a rutinas de *BLAS*-3.

Al igual que GBTRF$_{\text{UNB}}$, GBTRF$_{\text{BLK}}$ recorre la matriz de izquierda a derecha a lo largo de su diagonal principal. En cada iteración la matriz es dividida según un particionado 5×5 en los que hasta nueve bloques conforman la región activa (ver figura 4.32) con la que se operará. La forma en la que la matriz es particionada y los bloques de la región activa tal y como se encuentran físicamente almacenados se muestran en la figura 4.33.

Las operaciones ejecutadas durante una iteración son las siguientes:

1. Calcular la factorización LU con pivotamiento parcial del bloque de columnas $[A_{11}^T, A_{21}^T, A_{31}^T]^T$. Esta operación puede realizarse mediante una rutina que implemente el algoritmo GBTRF$_{\text{UNB}}$. Parte del trabajo realizado en esta operación es el establecimiento de los pivote s y la actualización del vector p.

2. Aplicar las permutaciones a los bloques de columnas $[A_{12}^T,\ A_{22}^T,\ A_{32}^T]^T$ y $[A_{13}^T,\ A_{23}^T,\ A_{33}^T]^T$.

3. Actualizar los bloques $[A_{12}^T,\ A_{22}^T,\ A_{32}^T]^T$. Todas las operaciones necesarias en este paso pertenecen al nivel 3 de *BLAS*.

4. Actualizar los bloques A_{13}, A_{23} y A_{33}. De nuevo todas las operaciones necesarias en este paso pertenecen al nivel 3 de *BLAS*.

La utilización de rutinas *BLAS*-3 acarreará un mejor acceso a memoria, reduciendo el número de accesos totales y aumentando las prestaciones obtenidas.

4.2.4. Implementaciones basadas en rutinas de BLAS-3 denso

Implementación LAPACK GBTRF

La rutina GBTRF de *LAPACK* implementa el algoritmo GBTRF$_{\text{BLK}}$ con algunas particularidades.

1. Para el cálculo de la factorización LU de $[A_{11}^T, A_{21}^T, A_{31}^T]^T$, en lugar de invocar la rutina GBTF2, la rutina GBTRF incluye el código necesario. La misma recorre por columnas el bloque y para cada una ejecuta las siguientes operaciones: calcular el pivote, aplicar la permutación a las restantes columnas a la derecha del bloque e invocar a las rutinas SCAL y GER, tal y como se explicó en el algoritmo GBTRF$_{\text{UNB}}$. Al mismo tiempo, guarda una copia de A_{31} en el espacio de trabajo W_{31}.

2. Aplicar las permutaciones p_1 a $[A_{12}^T,\ A_{22}^T,\ A_{32}^T]^T$. La ru tina *LAPACK* LASWP será la encargada de realizar esta tarea.

3. Aplicar las permutaciones a $[A_{13}^T,\ A_{23}^T,\ A_{33}^T]^T$. Debido a la naturaleza triangular inferior del bloque A_{13} y al esquema de almacenamiento empleado (en el que los elementos de la parte triangular superior estricta no se almacenan) no es posible emplear la rutina LASWP. En su lugar, las permutaciones son realizadas por dos bucles anidados.

Algorithm: $[A, p] = \text{GBTRF}_{\text{BLK}}(A, k_u, k_l, p)$

Partition $A \rightarrow \left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right), p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$

where A_{TL} is 0×0; A_{MM} is $k_l \times (k_u + k_l)$; and p_T has 0 elements

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & A_{01} & A_{02} & & \\ \hline A_{10} & A_{11} & A_{12} & A_{13} & \\ \hline A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ \hline & A_{31} & A_{32} & A_{33} & A_{34} \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

where A_{11}, A_{33} are $b \times b$;
A_{22} is $(k_l - b) \times (k_u + k_l - b)$; and p_1 has b elements

$$\left(\begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix}, p_1\right) := \text{GBTRF}_{\text{UNB}}\left(\begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix}, p_1\right)$$

$$\begin{bmatrix} A_{12} & A_{13} \\ A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} := P(p_1)\left(\begin{bmatrix} A_{12} & A_{13} \\ A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}\right)$$

$$\begin{aligned} A_{12} &:= \text{TRILU}(A_{11})^{-1} \cdot A_{12} \\ A_{22} &:= A_{22} - A_{21} \cdot A_{12} \\ A_{32} &:= A_{32} - A_{31} \cdot A_{12} \\ A_{13} &:= \text{TRILU}(A_{11})^{-1} \cdot A_{13} \\ A_{23} &:= A_{23} - A_{21} \cdot A_{13} \\ A_{33} &:= A_{33} - A_{31} \cdot A_{13} \end{aligned}$$

Continue with

$$\left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & A_{01} & A_{02} & & \\ \hline A_{10} & A_{11} & A_{12} & A_{13} & \\ \hline A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ \hline & A_{31} & A_{32} & A_{33} & A_{34} \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Figura 4.31: Algoritmo por bloques $\text{GBTRF}_{\text{BLK}}$ para la factorización $P \cdot A := L \cdot U$. La notación $\text{TRILU}(A)$ describe la sección triangular inferior unitaria de la matriz A

4. Actualizar el resto de bloques de la región activa:

$$\begin{aligned}
(\text{TRSM}) \quad A_{12} &:= \text{TRILU}(A_{11})^{-1} \cdot A_{12}, && (4.27)\\
(\text{GEMM}) \quad A_{22} &:= A_{22} - A_{21} \cdot A_{12}, && (4.28)\\
(\text{GEMM}) \quad A_{32} &:= A_{32} - W_{31} \cdot A_{12}, && (4.29)\\
A_{13} &:= A_{13} \cdot A_{11}^{-1}, && (4.30)\\
W_{13} &:= A_{13}, && (4.31)\\
(\text{TRSM}) \quad W_{13} &:= \text{TRILU}(A_{11})^{-1} \cdot W_{13}, && (4.32)\\
A_{13} &:= W_{13}, && (4.33)\\
(\text{GEMM}) \quad A_{23} &:= A_{23} - A_{21} \cdot W_{13}, && (4.34)\\
(\text{GEMM}) \quad A_{33} &:= A_{33} - W_{31} \cdot W_{13}, && (4.35)
\end{aligned}$$

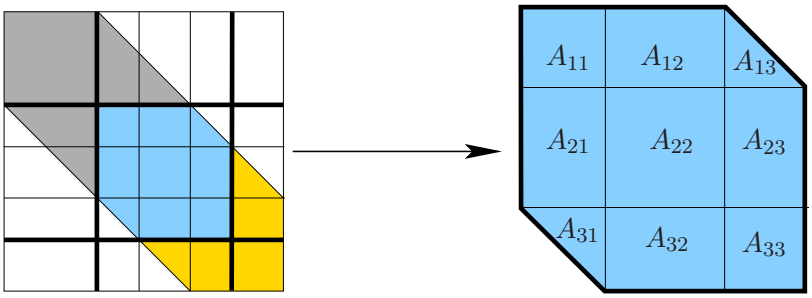


Figura 4.32: Particionado 5×5 aplicado a la matriz (izquierda) y detalle de la región activa (derecha).

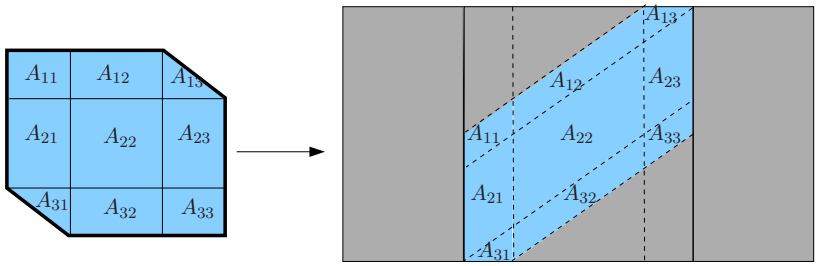


Figura 4.33: Forma en la que la región activa se almacena según el esquema para matrices banda empleado por *LAPACK*. Como se puede observar, sólo las partes triangular superior e inferior de los bloques A_{31} y A_{13}, respectivamente, son almacenadas.

5. Deshacer las permutaciones realizadas en $[A_{11}^T,\ A_{21}^T,\ A_{31}^T]^T$ con el fin de que éstos almacenen los multiplicadores (elementos de la transformada de Gauss) utilizados en la factorización y se mantenga la estructura triangular superior del bloque A_{31}.

6. Copiar el contenido de W_{31} sobre A_{31}.

La implementación practicada en la rutina GBTRF presenta como principal ventaja la utilización de rutinas *BLAS*-3 para la ejecución de la mayor parte de las operaciones. No obstante la adaptación de las necesidades del algoritmo a la funcionalidad de las rutinas incluidas en *BLAS* provoca ciertos problemas:

- Al emplear directamente la rutina TRSM sobre el bloque A_{13} en (4.30) el resultado se almacenaría sobre el propio bloque A_{13}, perdiendo por tanto el valor de los elementos de almacenados físicamente en la parte estricta triangular superior de este bloque. Para evitarlo, es necesario operar con el espacio de trabajo W_{13}. Una vez realizada la copia del bloque A_{13} sobre el espacio de trabajo, W_{13} se utiliza en (4.32), (4.34) y (4.35). En estas dos últimas operaciones se utiliza W_{13} como si el bloque A_{13} fuera una matriz general, es decir, operando con los elementos nulos de la parte triangular superior estricta de A_{13}. De esta forma se ejecutan algunas operaciones aritméticas 'no útiles'.

- Existe una diferencia entre la funcionalidad de TRMM y las operaciones de actualización de A_{32} y A_{33}, pues si bien esta rutina calcula el producto entre una matriz triangular y una general, al igual que ocurre con TRSM, almacena el resultado sobre una de las matrices operando. La

utilización del espacio de trabajo W_{31} permite operar con este bloque como si de una matriz general se tratase. Como se muestra, la funcionalidad de la rutina para el producto de matrices generales (GEMM) sí que es capaz de realizar la actualización de los bloques A_{32} y A_{33}. El problema introducido por GEMM es que opera con los elementos nulos de la parte triangular superior estricta de A_{31}, realizando de esta forma más operaciones de las que realmente son necesarias.

Implementación GBTRF_INLINE

La rutina GBTRF_INLINE es una modificación de GBTRF. El objetivo de esta nueva rutina es reducir el número de llamadas a *BLAS*, y en especial a rutinas de los niveles inferiores de *BLAS*. La mayoría de estas rutinas ejecutan un número reducido de operaciones y es probable que su eficiencia no sea capaz de contrarestar el tiempo empleado en su invocación y en el control de parámetros que internamente realizan. La rutina GBTRF_INLINE sustituye las invocaciones a las rutinas SWAP, LASWP, SCAL y GER por su código interno.

Implementación GBTRF_AM

La rutina GBTRF_AM requiere la modificación del esquema de almacenamiento de la matriz A. Al esquema ilustrado en la figura 4.30 se le añaden b filas de elementos nulos en la parte inferior. Una vez modificado de esta forma el almacenamiento, los bloques A_{31} y A_{13} se encuentran almacenados completamente, como se muestra en la figura 4.34, con lo que pueden ser tratados como matrices generales sin necesidad de utilizar los espacios de trabajo W_{13} y W_{31}.

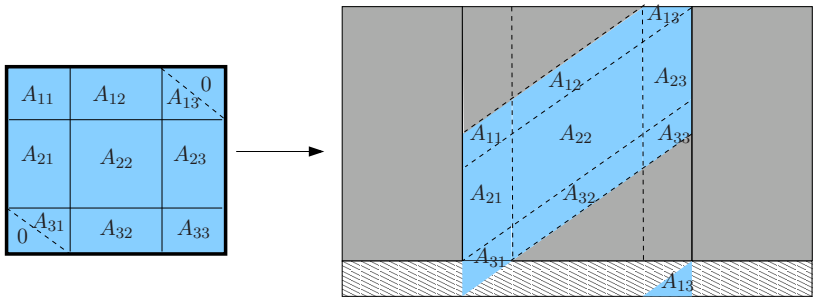


Figura 4.34: Región activa en la rutina GBTRF_AM (izquierda) y esquema de almacenamiento que emplea (derecha).

En esta nueva rutina, las operaciones (4.27) y (4.30) se pueden ejecutar con una simple invocación a TRSM. De igual forma, las operaciones (4.28), (4.29), (4.34) y (4.35) se pueden unir en una sóla operación que la rutina GEMM puede computar.

A estas mejoras hay que unir otras menores, como por ejemplo que las permutaciones de todos los bloques se pueden realizar de forma simultánea con una única invocación a LASWP, o que las copias de los bloques A_{13} y A_{31} a los espacios de trabajo W_{13} y W_{31}, respectivamente, ya no son necesarias.

La secuencia de operaciones a ejecutar en cada iteración de esta rutina es la siguiente:

1. Cálculo de la factorización LU con pivotamiento parcial de $[A_{11}^T,\ A_{21}^T,\ A_{31}^T]^T$ al igual que se hace en GBTRF, pero salvo q ue la copia a W_{31} que ya no es necesaria.

2. Aplicar las permutaciones a los bloques $[A_{12}^T, A_{22}^T, A_{32}^T]^T$ y $[A_{13}^T, A_{23}^T, A_{33}^T]^T$ mediante una invocación a LASWP.

3. Actualizar los bloques $\bar{A}_{12}\left(= \begin{bmatrix} A_{12} & A_{13} \end{bmatrix}\right)$ y $\bar{A}_{22}\left(= \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}\right)$:

$$(\text{TRSM}) \quad \bar{A}_{12} := \text{TRILU}(A_{11})^{-1} \cdot \bar{A}_{12} \tag{4.36}$$

$$(\text{GEMM}) \quad \bar{A}_{22} := \bar{A}_{22} - \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot \bar{A}_{12} \tag{4.37}$$

4. Deshacer las permutaciones realizadas en $[A_{11}^T, A_{21}^T, A_{31}^T]$, con el fin de que estos bloques almacenen los multiplicadores utilizados en la factorización y mantener la estructura triangular superior del bloque A_{31}.

En esta nueva implementación, el número de invocaciones a rutinas de *BLAS* es notablemente inferior reduciendose así el sobrecoste debido a las invocaciones. Además se consigue otra importante mejora, ya que las rutinas *BLAS*-3 invocadas por GBTRF_AM operan con bloques mayores, lo que permitirá un mayor nivel de paralelismo y mejores prestaciones cuando se invoquen a rutinas paralelas de *BLAS*.

El coste que hay que pagar para alcanzar todas estas mejoras es modificar el esquema de almacenamiento, ampliándolo en $b \times n$ elementos.

Implementación GBTRF_AM_INLINE

Esta rutina combina las modificaciones comentadas en GBTRF_AM y GBTRF_INLINE.

Implementación GBTRF_MERGE

La rutina GBTRF_AM presenta considerables ventajas sobre la rutina *LAPACK* GBTRF, pero precisa un cambio en el esquema de almacenamiento y un incremento en los requerimientos de memoria. La implementación GBTRF_MERGE, al igual que GBTRF_AM, trata de reducir el número de invocaciones a rutinas *BLAS* agrupando las operaciones ejecutadas en cada iteración, pero evitando modificar el esquema de almacenamiento.

La implementación GBTRF_MERGE simula que los bloques triangulares A_{13} y A_{31} se encuentran almacenados completamente. Para lograrlo, guarda una copia del contenido de las regiones de memoria en las cuales residirían la parte triangular superior estricta de A_{13} y la inferior estricta de A_{31} en los espacios de trabajo W_{13} y W_{31} y los rellena con ceros antes de operar con estos bloques. Este proceso es el mostrado en la figura 4.35.

La secuencia de operaciones ejecutadas en cada iteración es pues la siguiente:

1. Copiar la región de memoria que almacena $\text{STRIL}(A_{31})$ y rellenar esta región con ceros.

2. Calcular la factorización LU con pivotamiento parcial de los bloques A_{11}, A_{21} y A_{31}.

3. Copiar la región de memoria que almacena $\text{STRIU}(A_{13})$ y rellenar esta región con ceros.

4. Aplicar las permutaciones a $[A_{12}^T, A_{22}^T, A_{32}^T]^T$, $[A_{13}^T, A_{23}^T, A_{33}]^T$ mediante una invocación a LASWP.

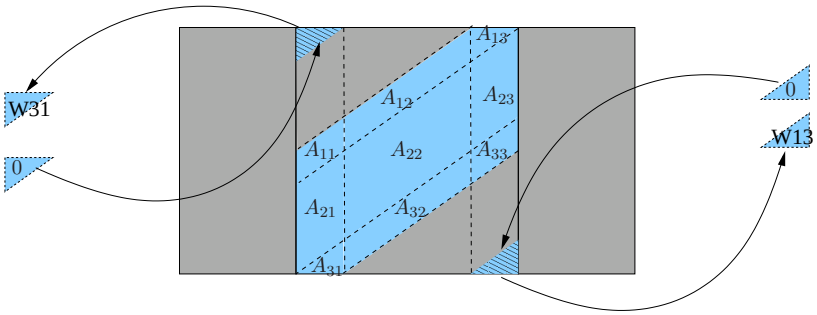


Figura 4.35: Copia de STRIL(A_{31}) a W_{31}, de STRIU(A_{13}) a W_{13} y relleno con ceros de las regiones de memoria que los almacenan.

5. Actualizar los bloques $\bar{A}_{12}$ $(= \begin{bmatrix} A_{12} & A_{13} \end{bmatrix})$ y $\bar{A}_{22}$ $\left(= \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}\right)$:

$$(\text{TRSM}) \quad \bar{A}_{12} := \text{TRILU}(A_{11})^{-1} \cdot \bar{A}_{12}, \tag{4.38}$$

$$(\text{GEMM}) \quad \bar{A}_{22} := \bar{A}_{22} - \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot \bar{A}_{12}. \tag{4.39}$$

6. Deshacer las permutaciones realizadas en $[A_{11}^T, A_{21}^T, A_{31}^T]^T$, con el fin de que estos bloques almacenen los multiplicadores utilizados en la factorización y mantener la estructura triangular superior del bloque A_{31}.

7. Copiar los elementos almacenados en W_{13} y W_{31} a sus posiciones originales.

La rutina GBTRF_MERGE reduce el número de operaciones por iteración, opera con bloques de mayor tamaño y trabaja con el mismo esquema de almacenamiento que la rutina *LAPACK*.

Implementación GBTRF_MERGE_INLINE

Esta nueva implementación es similar a GBTRF_MERGE, pero en este caso las invocaciones a las rutinas *BLAS* SWAP, LASWP, SCAL y GER se sustituyen por código embebido.

4.2.5. Resultados experimentales

Arquitectura ITANIUM

BLAS secuencial La figura 4.36 muestra los resultados de la rutina GBTRF al utilizar núcleos computacionales de las implementaciones secuenciales de *BLAS*: *BLAS de referencia*, *GotoBLAS* y *MKL*. Las bibliotecas *GotoBLAS* y *MKL*, que son específicas para la arquitectura ITANIUM, son más eficientes que la biblioteca *BLAS de referencia*.

La figura 4.37 compara las prestaciones de las nuevas rutinas con las de la rutina *LAPACK* GBTRF. La gráfica de la izquierda, que presenta los resultados obtenidos al emplear las rutinas de la biblioteca *GotoBLAS*, muestra como las nuevas rutinas mejoran, aunque sólo levemente, los resultados de GBTRF al operar con matrices de banda media o ancha, mientras que con matrices de banda estrecha alcanzan las mismas prestaciones que la rutina *LAPACK*. La gráfica de la derecha reúne las prestaciones obtenidas con la biblioteca *MKL* y, de nuevo, las tres rutinas obtienen prestaciones similares para matrices de banda estrecha, mientras que con matrices de banda media

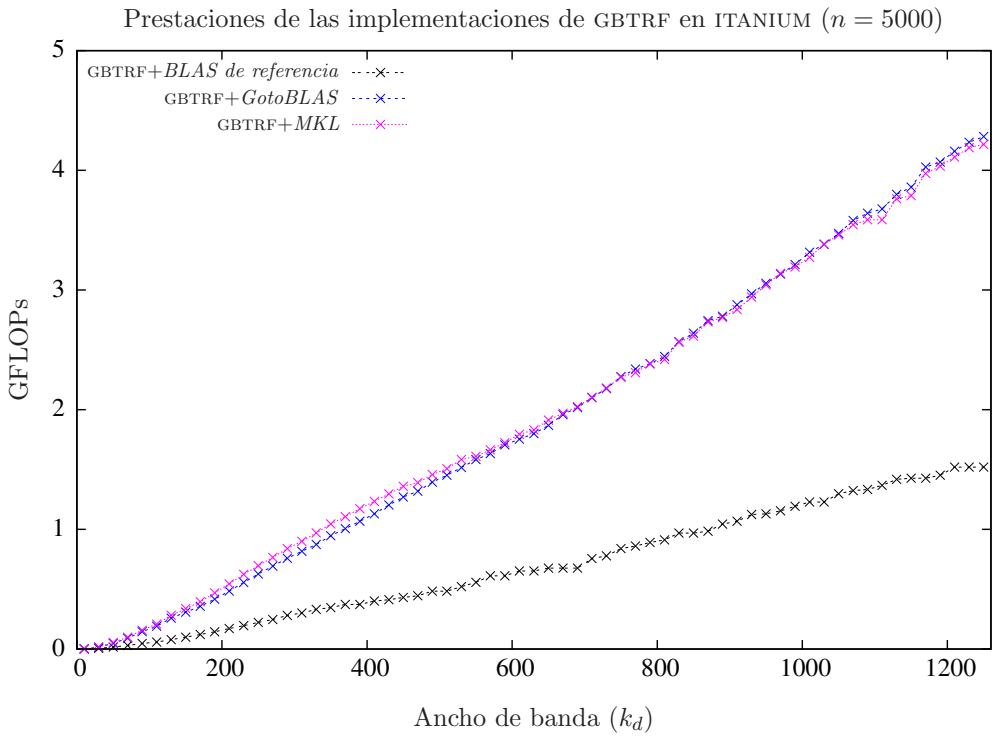


Figura 4.36: Comparativa de GBTRF empleando diferentes implementaciones de bibliotecas *BLAS*.

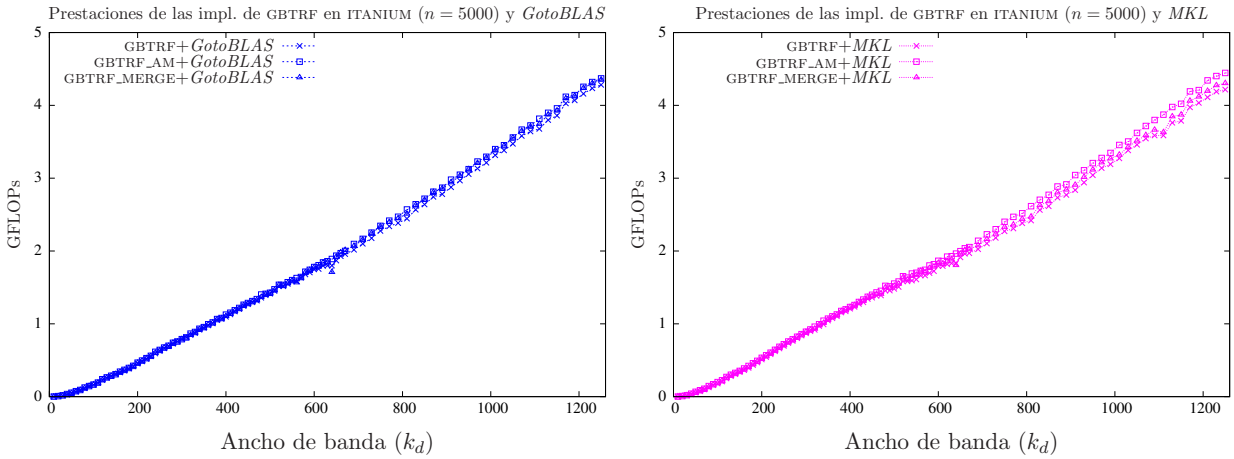


Figura 4.37: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

o ancha la rutina GBTRF_AM es la más eficiente. No obstante, la rutina GBTRF_MERGE es también ligeramente más rápida que GBTRF.

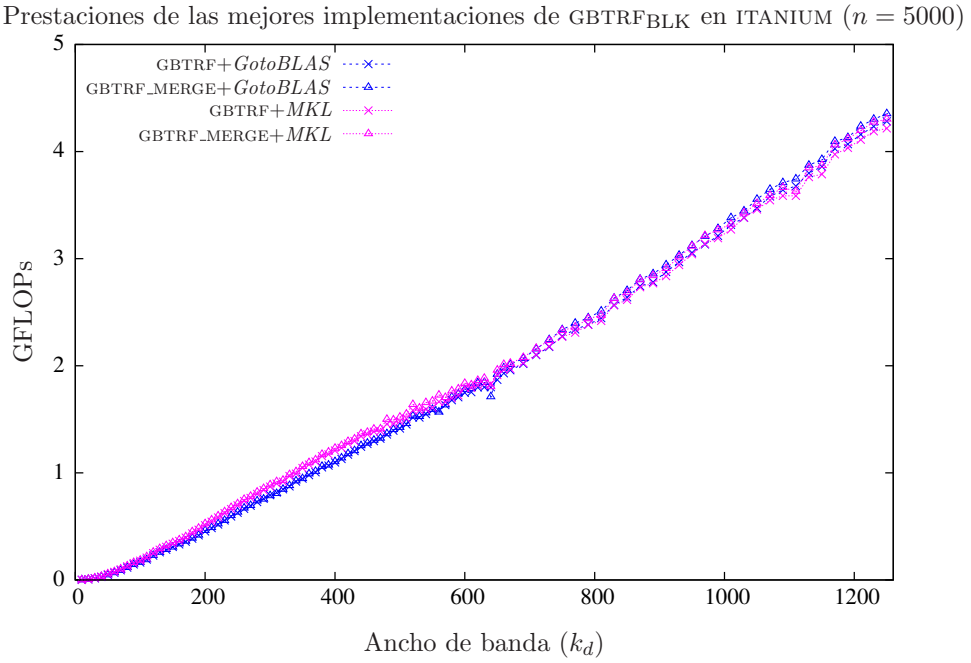


Figura 4.38: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

La figura 4.38 recoge los resultados de las mejores implementaciones para ambas bibliotecas. Pese a que la rutina GBTRF_AM alcanza mayores prestaciones que GBTRF_MERGE, la diferencia de prestaciones es demasiado reducida como para justificar la modificación en el esquema de almacenamiento requerida por GBTRF_MERGE. Es por este motivo que en la figura se muestran las prestaciones de GBTRF_MERGE. Cuando la matriz presenta una banda estrecha, la rutina GBTRF_MERGE+*MKL* es la más rápida, mientras que para matrices de banda mayor es GBTRF_MERGE+*GotoBLAS* la implementación más eficiente.

BLAS paralelo El mismo análisis se ha realizado invocando a las versiones paralelas de las bibliotecas *GotoBLAS* y *MKL*. La figura 4.39 recoge las prestaciones de las implementaciones propuestas así como las de la rutina GBTRF. La gráfica de la izquierda muestra los resultados conseguidos con la implementación paralela de *GotoBLAS* y cuatro hebras de ejecución. Las dos nuevas implementaciones mejoran las prestaciones ofrecidas por GBTRF, especialmente cuando la matriz presenta una banda ancha. En caso de que *MKL* (gráfica de la derecha) sea la implementación *BLAS* empleada, los resultados son ligeramente diferentes. GBTRF_AM continúa siendo la rutina más eficiente, y en este caso GBTRF_MERGE iguala las prestaciones de GBTRF pero no consigue mejorarlas.

Al igual que sucede en el caso secuencial, los resultados mostrados por GBTRF_AM son superiores a los alcanzados por GBTRF_MERGE, pero la diferencia de prestaciones no justifica el mayor requerimiento espacial y la modificación del esquema de almacenamiento precisado por GBTRF_AM. En consecuencia, en la figura 4.40 que recopila los resultados de las mejores implementaciones, se omiten los generados por GBTRF_AM en favor de los de GBTRF_MERGE.

La figura 4.40 muestra la superioridad de la biblioteca *MKL*. No obstante, la diferencia entre ambas bibliotecas es muy reducida, especialmente al emplear la nueva rutina GBTRF_MERGE. Esta

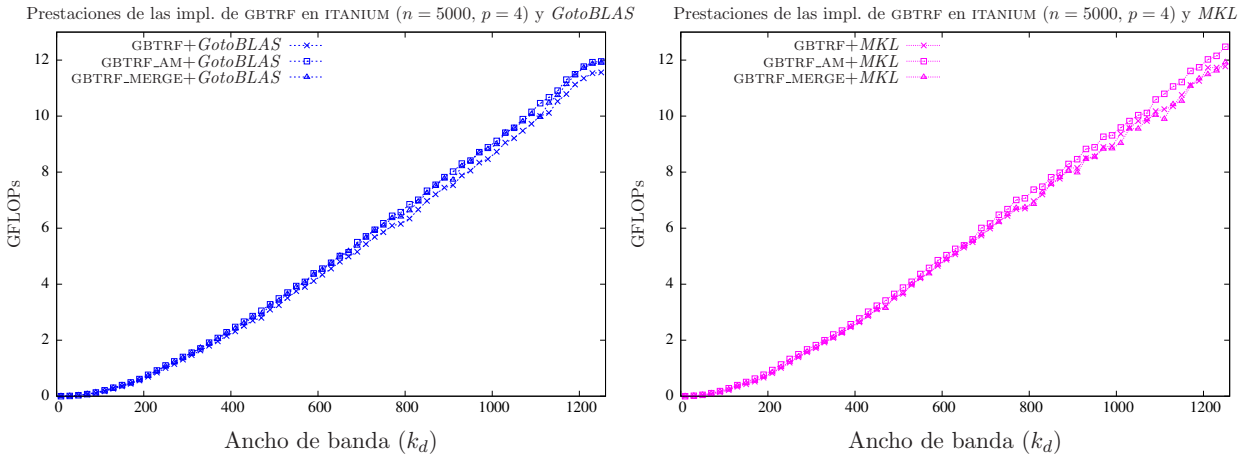


Figura 4.39: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

nueva rutina iguala los resultados de GBTRF con *MKL* y los mejora ligeramente con *GotoBLAS*.

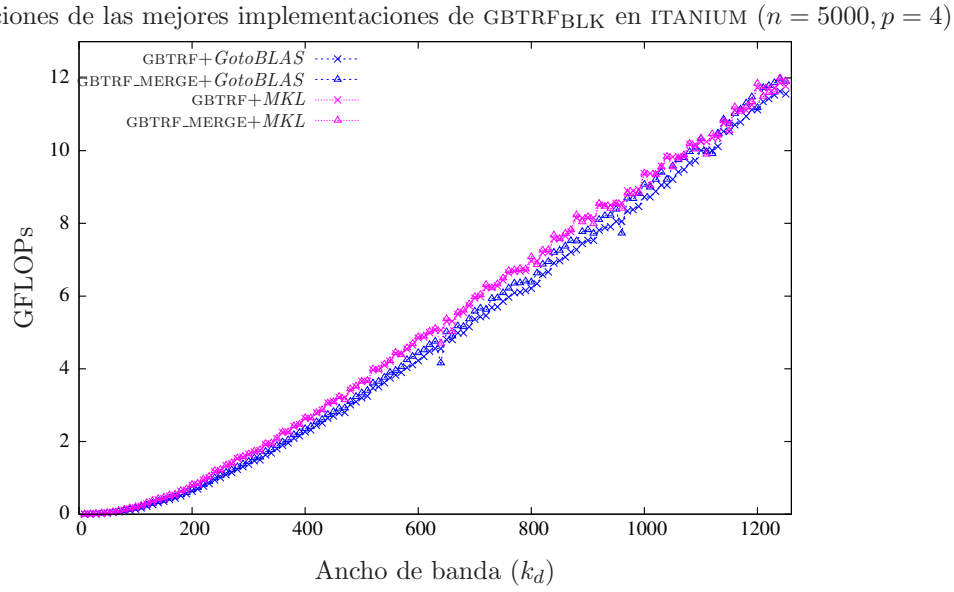


Figura 4.40: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

Arquitectura XEON

BLAS secuencial La figura 4.41 presenta las prestaciones de la rutina *LAPACK* (GBTRF) al trabajar con las versiones secuenciales de las bibliotecas *BLAS de referencia*, *GotoBLAS* y *MKL*. Los mejores resultados se obtienen con la biblioteca *GotoBLAS*, aunque *MKL* presenta prestaciones cercanas a las de *GotoBLAS*. Como era de esperar, la biblioteca *BLAS de referencia*, que no está optimizada para la arquitectura XEON, es notablemente más ineficiente.

Las prestaciones de las rutinas presentadas en este estudio se resumen en la figura 4.42. Tanto en el caso de *GotoBLAS* (gráfica de la izquierda) como en el de *MKL* (gráfica de la derecha), las nuevas

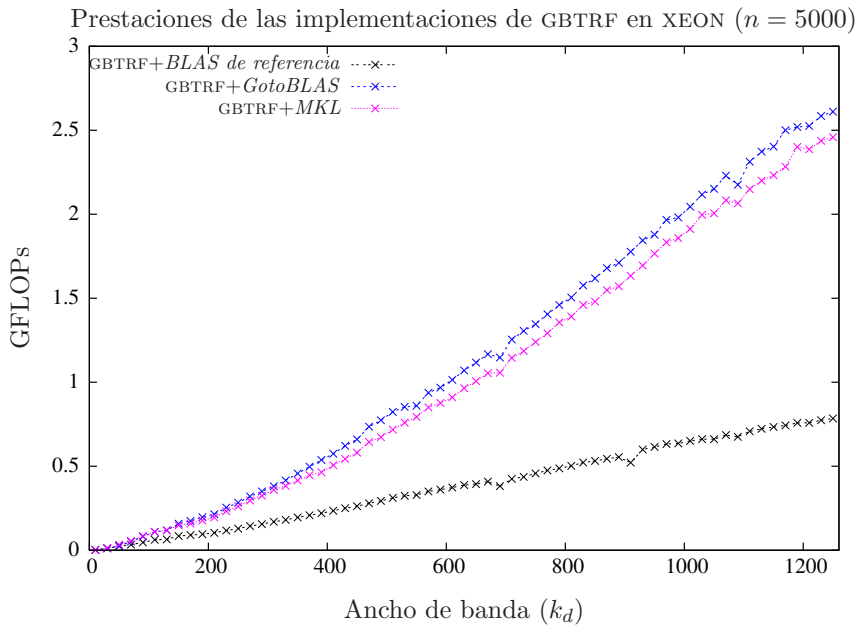


Figura 4.41: Comparativa de GBTRF empleando diferentes implementaciones de bibliotecas *BLAS*.

rutinas son ligeramente más rápidas. Ambas son igualmente eficientes, por lo que se recomienda la utilización de GBTRF_MERGE, que no precisa modificar el esquema de almacenamiento de la matriz.

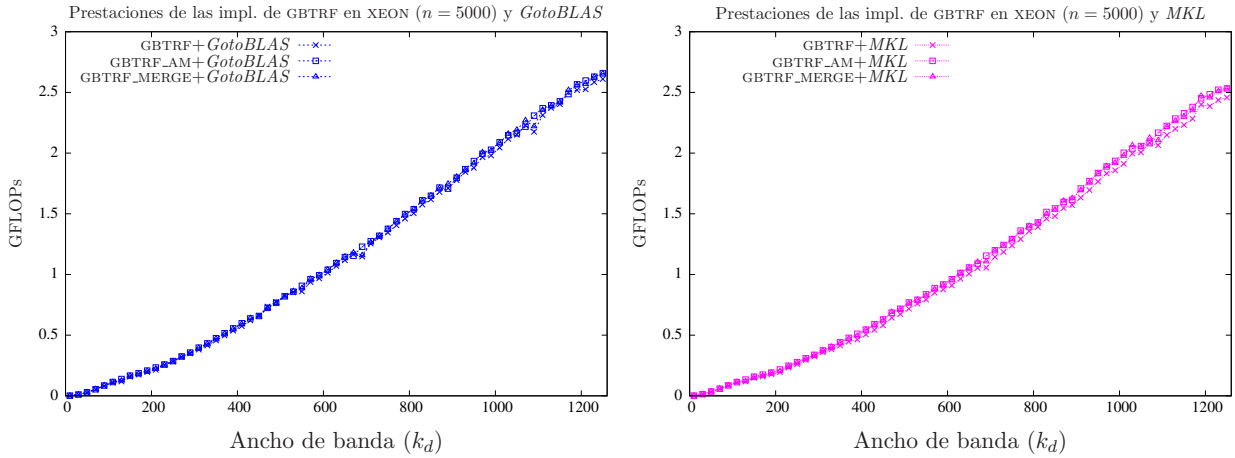


Figura 4.42: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

La figura 4.43 permite comparar los resultados de las mejores implementaciones basadas en *GotoBLAS* con los de las mejores implementaciones basadas en *MKL*. Como se puede observar, la rutina *GotoBLAS* es sensiblemente más eficiente, siendo GBTRF_MERGE+*GotoBLAS* la implementación que alcanza las mayores prestaciones.

BLAS paralelo La figura 4.44 muestra la eficiencia de las diferentes rutinas al emplear implementaciones paralelas de *BLAS* y dos hebras de ejecución. Si la bibliteca empleada es *GotoBLAS*, las dos nuevas rutinas ofrecen prestaciones similares y ligeramente superiores a las de GBTRF. Por otro lado, si la biblioteca utilizada es *MKL* la diferencia de prestaciones de las nuevas rutinas y

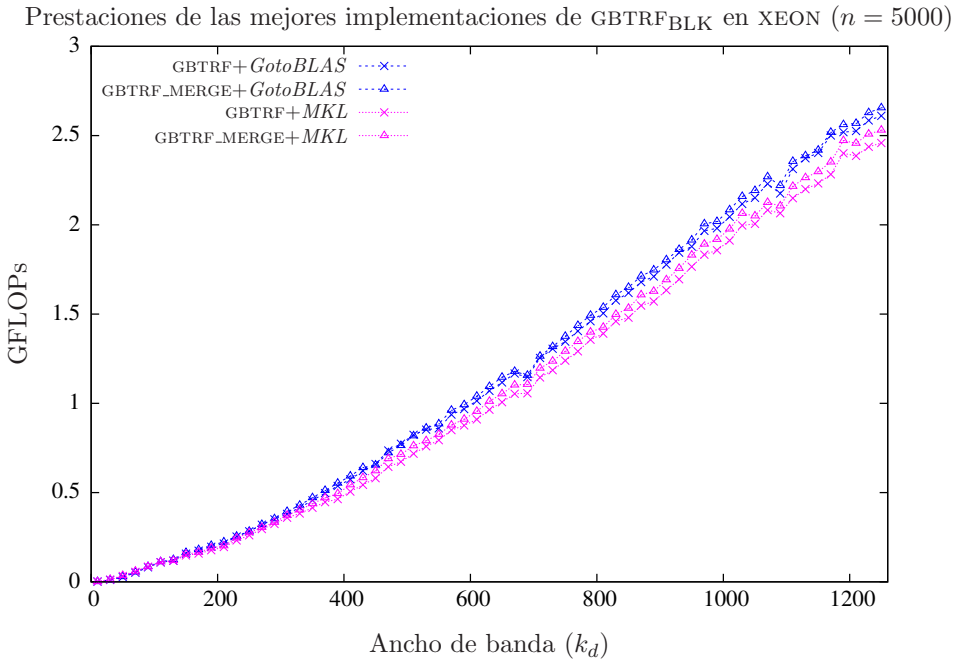


Figura 4.43: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

GBTRF aumenta, especialmente cuando la matriz A es una matriz de banda media o ancha.

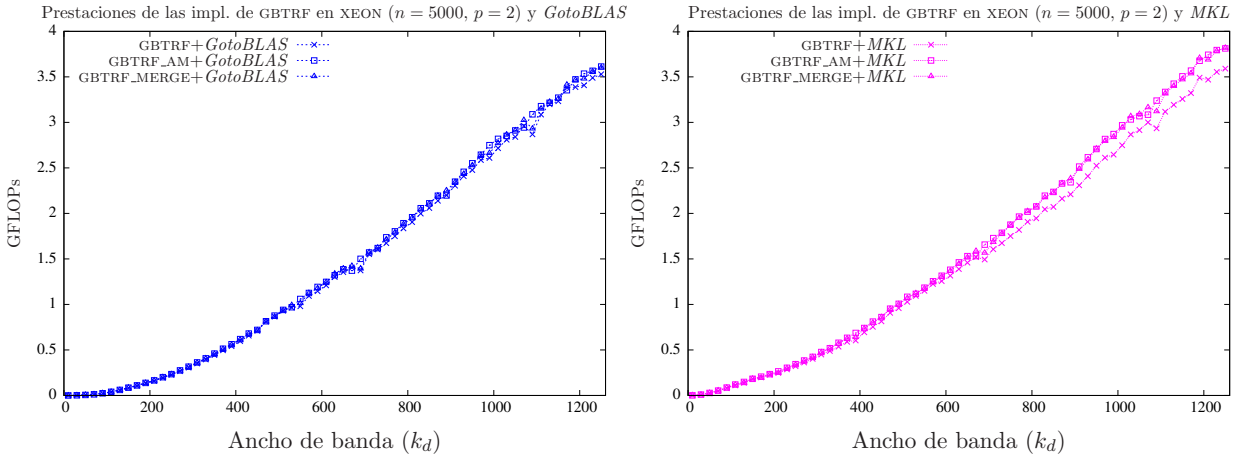


Figura 4.44: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

Dado que las diferencias entre las prestaciones de GBTRF_AM y GBTRF_MERGE son nulas, se recomienda el uso de la rutina GBTRF_MERGE, que no exige modificar el esquema de almacenamiento de la matriz. La gráfica 4.45 reune las prestaciones de las mejores implementaciones y en ella queda plasmada la superioridad de la nueva rutina GBTRF_MERGE.

4.2.6. Conclusiones

Se han evaluado varias nuevas rutinas comparándolas con la rutina *LAPACK* GBTRF. Las implementaciones basadas en la inclusión de código embebido no mejoran las prestaciones, salvo

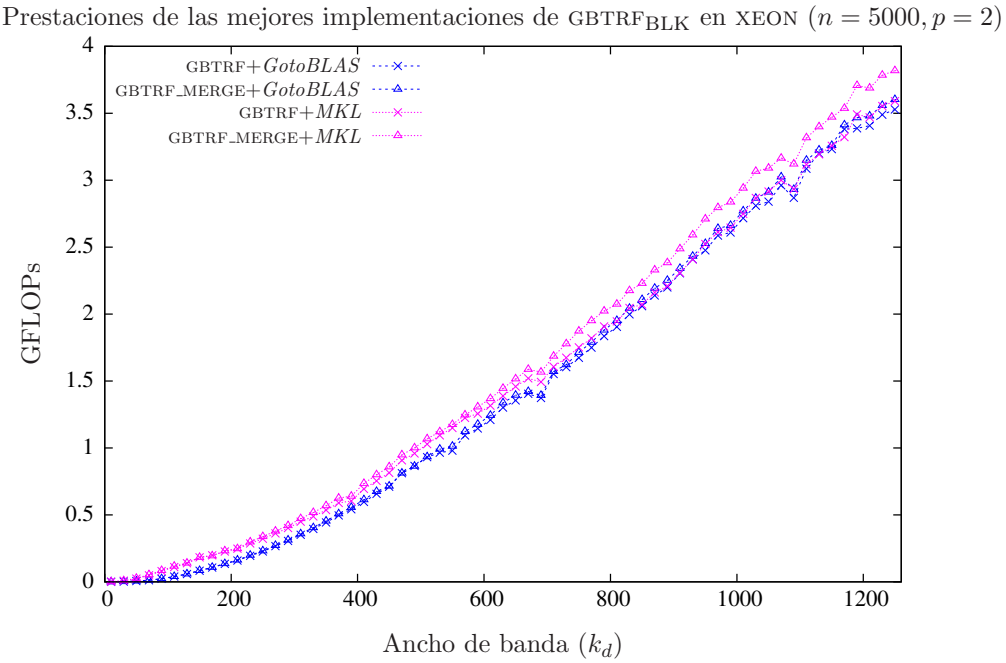


Figura 4.45: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

mínimamente para matrices de banda muy estrecha, Por simplicidad y claridad se han eliminado sus resultados de las gráficas.

Las nuevas rutinas que agrupan el número de llamadas a *BLAS* reduciéndolo mejoran en la mayoría de los casos e igualan en el resto las prestaciones de GBTRF, especialmente al invocar a implementaciones paralelas de *BLAS* y operar con matrices de banda media o ancha. La rutina más eficiente es PBTRF_AM, pero sus características y especialmente el hecho de requerir modificaciones en el esquema de almacenamiento empleado, desaconseja su uso en favor de GBTRF_MERGE, que es sólo ligeramente menos eficiente y no precisa modificaciones en el esquema de almacenamiento.

4.3. Algoritmos por bloques y planificación dinámica

La ejecución paralela de los algoritmos de factorización de matrices banda expuestos en las secciones anteriores está basada en el uso de una implementación multihebra de la biblioteca *BLAS*. En concreto, la ejecución de una iteración procede de manera secuencial, con una única hebra de ejecución, hasta el momento en que se invoca a una rutina del *BLAS*. Entran entonces en ejecución varias hebras que calculan la operación correspondiente y que se sincronizan al finalizar el cálculo, pasando el código a ejecutarse de nuevo secuencialmente hasta la siguiente invocación a una rutina de *BLAS*. Si consideramos por ejemplo las operaciones que se suceden durante una iteración del

algoritmo por bloques para la factorización de Cholesky (algoritmo en la figura 4.3):

$$
\begin{aligned}
(\text{POTF2}) \quad & A_{11}(L_{11}) && := \text{CHOL\_UNB}(A_{11}), && (4.40)\\
(\text{TRSM}) \quad & A_{21}(L_{21}) && := A_{21} \cdot \ \text{TRIL}(A_{11})^{-T}, && (4.41)\\
& A_{31}(L_{31}) && := A_{31} \cdot \ \text{TRIL}(A_{11})^{-T}, && (4.42)\\
& W && := A_{31}, && (4.43)\\
(\text{TRSM}) \quad & W && := W \cdot \ \text{TRIL}(A_{11})^{-T}, && (4.44)\\
(\text{SYRK}) \quad & A_{22} && := A_{22} - A_{21} \cdot \ A_{21}^T, && (4.45)\\
(\text{GEMM}) \quad & A_{32} && := A_{32} - W \cdot \ A_{21}^T, && (4.46)\\
(\text{SYRK}) \quad & A_{33} && := A_{33} - W \cdot \ W^T, && (4.47)\\
& A_{31}(L_{31}) && := W, && (4.48)
\end{aligned}
$$

la ejecución siguiendo esta aproximación supone que cada una de las operaciones (4.41),(4.44)–(4.48) se ejecuta en paralelo (la operación (4.40) no corresponde a una rutina de *BLAS*), con las hebras sincronizándose al término de cada una de las llamadas.

Esta *aproximación tradicional* para la paralelización de operaciones de *LAPACK* presenta varios inconvenientes, que exponemos a continuación para el algoritmo de cálculo de la factorización de Cholesky:

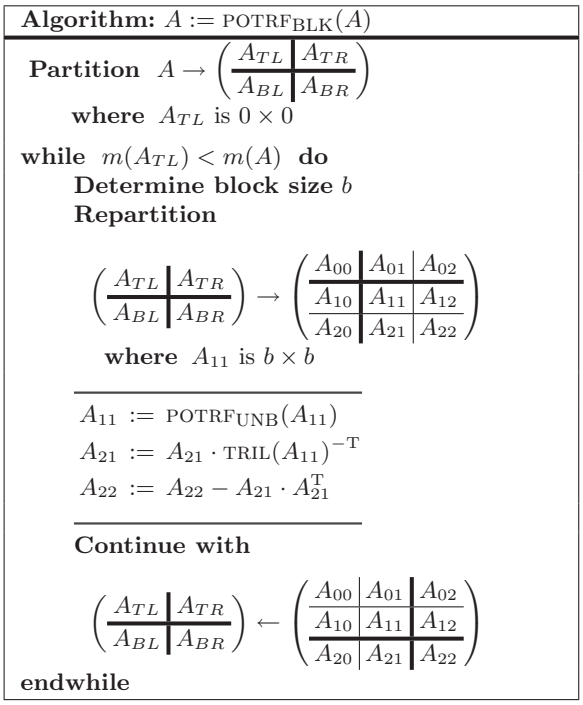
- En primer lugar, el número de sincronizaciones de las hebras es elevado, una por cada llamada a *BLAS*: esto supone hasta 5 sincronizaciones por iteración en el algoritmo de la figura 4.3.
- En segundo lugar, no se explota el paralelismo entre operaciones de una misma iteración. Por ejemplo, en la factorización de Cholesky las actualizaciones de los bloques A_{21} y A_{31} pueden proceder en paralelo, pues son operaciones independientes, y la misma situación se repite con las actualizaciones de A_{22}, A_{32} y A_{33}.
- Finalmente, se obvia el paralelismo entre operaciones de iteraciones diferentes: sea $\bar{A}_{11}$ la parte de A_{22} que se convertirá en A_{11} durante la siguiente iteración. Una vez calculada la factorización de A_{11} y actualizado $\bar{A}_{11}$, sería posible factorizar este último bloque en paralelo con la actualización del resto de A_{22}.

Estos inconvenientes limitan el grado de paralelismo del algoritmo lo que, en una situación donde el número de procesadores es elevado en comparación con el tamaño del problema, puede reducir el rendimiento.

Para resolver estos problemas, en [85] se propusieron técnicas de *look-ahead* que, de manera estática (previamente a la ejecución), adelantan cálculos correspondientes a iteraciones posteriores a la iteración actual. Esta aproximación aumenta el grado de paralelismo de la ejecución paralela. A cambio, complica considerablemente la codificación del algoritmo y resulta poco flexible pues el número de iteraciones posteriores en las que se "busca" operaciones para adelantarlas está fijado de antemano y no varía durante la ejecución del algoritmo.

Los proyectos FLAME, PLASMA y SMPSs [90, 49, 10] plantean una aproximación dinámica de extracción del paralelismo, que tiene sus raíces en las propuestas del proyecto Cilk [26]. En particular, expondremos a continuación los principios del proyecto FLAME, por la relación de este trabajo con el mismo. Los proyectos PLASMA y SMPSs siguen aproximaciones muy similares.

Los elementos principales de FLAME son una notación para expresar operaciones de álgebra lineal (que ha sido utilizada en la presentación de algoritmos de nuestro trabajo), un proceso sistemático de derivación formal de algoritmos, y las interfaces de programación de aplicaciones que




```
FLA_Error FLA_Potrf_blk( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;
  int b;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {
    b = min( FLA_Obj_length(ABR), nb_alg );
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       b, b, FLA_BR );
    /*-----------------------------------------*/
    FLA_Potrf_unb( A11 );
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11, A21 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A21, FLA_ONE,       A22 );
    /*-----------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

Figura 4.46: Algoritmo por bloques $\text{POTRF}_{\text{BLK}}$ para el cálculo de la factorización de Cholesky densa (izquierda) y el correspondiente código FLAME/C (derecha).

permiten transformar de manera sencilla algoritmos en códigos. El uso generalizado de procesadores multinúcleo (*multicore*) y la previsible aparición de sistemas con un número de núcleos por procesador muy elevado (*many-core*) han llevado al desarrollo de dos nuevos componentes de FLAME que permitan mejorar las prestaciones de los códigos de la biblioteca asociada en este tipo de plataformas, FLASH y SuperMatrix, que se describen más en detalle a continuación.

4.3.1. La interfaz de programación de algoritmos por bloques FLASH

FLASH [64] es una interfaz de alto nivel de abstracción basada en FLAME para el desarrollo de *algoritmos por bloques* que aisla el algoritmo/código del almacenamiento físico de los datos. FLASH permite la creación y gestión sencillas de matrices cuyos elementos son a su vez matrices, con varios niveles de recursión. A modo ilustrativo, en la figura 4.46 se muestra un algoritmo para el cálculo de la factorización de Cholesky de una matriz densa (simétrica definida positiva) y el correspondiente código FLAME usando la interfaz de programación FLAME/C. En la figura 4.47 se ofrece código para el algoritmo por bloques que calcula la factorización de Cholesky (izquierda) y la solución de un sistema triangular de ecuaciones (derecha) usando la interfaz de programación FLASH. En este caso, los elementos de A son bloques, pudiendo estar almacenados de manera consecutiva en

```
FLA_Error FLASH_Potrf_by_blocks( FLA_Obj A )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       1, 1, FLA_BR );
    /*-----------------------------------------------*/
    FLA_Potrf_unb( FLASH_MATRIX_AT( A11 ) );
    FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11,
                         A21 );
    FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A21,
                FLA_ONE,       A22 );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

```
void FLASH_Trsm_rltn( FLA_Obj alpha, FLA_Obj L,
                                     FLA_Obj B )
/* Caso particular con argumentos de modo
     FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                 FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                 ...                              )
   Consideraciones: L consiste de un solo bloque y
                    B consiste de una columna de bloques */
{
  FLA_Obj BT,      B0,
          BB,      B1,
                   B2;

  FLA_Part_2x1( B,    &BT,
                      &BB,    0, FLA_TOP );

  while ( FLA_Obj_length( BT ) < FLA_Obj_length( B ) ) {
    FLA_Repart_2x1_to_3x1( BT,       &B0,
                         /* ** */   /* ** */
                                     &B1,
                           BB,       &B2,    1, FLA_BOTTOM );
    /*-----------------------------------------------*/
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              alpha, FLASH_MATRIX_AT( L ),
                     FLASH_MATRIX_AT( B1 ) );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &BT,        B0,
                                          B1,
                            /* ** */   /* ** */
                              &BB,        B2,    FLA_TOP );
  }
}
```

Figura 4.47: Código FLASH para el cálculo de la factorización de Cholesky y la correspondiente resolución de sistemas triangulares.

memoria por ejemplo, redundando así en una mayor localidad de referencia a los datos y un menor tiempo de ejecución en general.

4.3.2. El entorno de ejecución SuperMatrix

SuperMatrix [25] es una herramienta de ejecución de algoritmos por bloques complementaria a FLAME que identifica tareas en el algoritmo y gestiona la planificación dinámica de las mismas.

En particular, el entorno SuperMatrix calcula la factorización de Cholesky de una matriz densa ejecutando el algoritmo por bloques en la figura 4.47 (izquierda) en dos etapas, *análisis* y *emisión*, expuestas seguidamente.

- **Análisis.** Durante esta primera etapa, el entorno ejecuta simbólicamente el código del algoritmo `FLASH_Potrf_by_blocks` de modo que, cuando encuentra una operación, en lugar de ejecutarla, encola ésta en una lista de tareas pendientes. Esto ocurre en las llamadas con `FLA_Potrf_unb`, `FLA_Trsm`, `FLA_Syrk`, y `FLA_Gemm` que se encuentran en la propia rutina `FLASH_Potrf_by_blocks` (figura 4.47 (izquierda)), así como en los códigos de `FLASH_Trsm` (figura 4.47 (derecha)) y `FLASH_Syrk` (no mostrada aquí, por simplicidad, pero con una codificación muy parecida). Esta etapa se ejecuta secuencialmente y usa el orden en que las suboperaciones aparecen en el código y los operandos que éstas leen y/o escriben (determinadas por la semántica de las llamadas a *BLAS*) para identificar las dependencias. El resultado de la ejecución de esta etapa es un grafo dirigido que contiene las dependencias entre las suboperaciones en que se ha descompuesto el problema.

- **Emsión.** En la segunda etapa, aquellas tareas que tienen todos sus operandos disponibles (las

dependencias han sido satisfechas) se extraen de la lista de tareas pendientes y se planifican para su ejecución a los núcleos de un procesador multinúcleo o los procesadores de un sistema multiprocesador. Cuando se completa una tarea, se revisa la lista de tareas pendientes actualizando las dependencias que han sido satisfechas.

Para ilustrar estas etapas, consideremos lo que ocurre durante los primeros pasos de cada etapa cuando se utiliza el código de la figura 4.47 para factorizar la matriz con 3×3 bloques:

$$A \rightarrow \left(\begin{array}{ccc} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} \end{array} \right). \tag{4.49}$$

Consideremos en primer lugar la etapa de análisis. Durante la primera iteración del bucle `while` en el código, A_{TL} es una matriz vacía de modo que $A = A_{BR}$ y

$$A \rightarrow \left(\begin{array}{c|c} \texttt{A11} & \texttt{A12} \\ \hline \texttt{A21} & \texttt{A22} \end{array} \right) = \left(\begin{array}{c|cc} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} \end{array} \right). \tag{4.50}$$

Así, cuando el entorno SuperMatrix (ejecutado por una única hebra) encuentra la llamada

```
FLA_Potrf_unb( FLASH_MATRIX_AT( A11 ) );
```

encola como tarea pendiente el cálculo de la factorización de Cholesky del bloque `A11`$= \bar{A}_{00}$. A continuación el entorno llega a la ejecución de la llamada a

```
FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
            FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
            FLA_ONE, A11, A21 );
```

pasándose a ejecutar el código en la figura 4.47 (derecha) con `L` $=$ `A11` $= \bar{A}_{00}$ y `B` $=$ `A21`$= \begin{pmatrix} \bar{A}_{10} \\ \bar{A}_{20} \end{pmatrix}$ como argumentos. Durante la ejecución de esta última rutina se encuentran dos llamadas a

```
FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
          FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
          FLA_ONE, FLASH_MATRIX_AT( L ), FLASH_MATRIX_AT( B1 ) );
```

una por bloque de `A21` ($\bar{A}_{10}$ y $\bar{A}_{20}$), encolándose en consecuencia dos nuevas tareas. Es en este momento cuando se detecta una depencencia: `A11` es una salida (resultado) de `FLA_Potrf_unb` pero una entrada (dato) para cada una de las llamadas a `FLA_Trsm` (como argumento `L`). Así pues, se registra una dependencia de tipo *lectura-tras-escritura* (RAW o *read-after-write*) como parte de la lista que implícitamente representa el grafo de dependencias. La ejecución de la etapa de análisis procede de este modo hasta que se han registrado todas las operaciones sobre los bloques. Conceptualmente, la salida es una lista con la información contenida en las dos columnas más a la izquierda de la figura 4.48.

La etapa de emisión toma la lista de tareas y dependencias (es decir, la información contenida en el grafo) como entrada. A medida que la ejecución paralela procede, las hebras ociosas consultan la lista en busca de tareas con todos los operandos disponibles (dependencias satisfechas) para descubrir que, inicialmente, la única tarea que cumple este requisito corresponde a la factorización de Cholesky de $\bar{A}_{00}$. Una única hebra ejecuta esta tarea y, al completarla, actualiza la lista marcando

	Tabla original			Después 1^a oper.			Después 3^a oper.			Después 6^a oper.		
Operación/Resultado	Dat		Dat/Res	Dat		Dat/Res	Dat		Dat/Res	Dat		Dat/Res
1. FLA_Chol_unb($\bar{A}_{00}$)			$\bar{A}_{00}\surd$									
2. $\bar{A}_{10} \cdot \text{TRIL}(\bar{A}_{00})^{-\text{T}}$	$\bar{A}_{00}$		$\bar{A}_{10}\surd$	$\bar{A}_{00}\surd$		$\bar{A}_{10}\surd$						
3. $\bar{A}_{20} \cdot \text{TRIL}(\bar{A}_{00})^{-\text{T}}$	$\bar{A}_{00}$		$\bar{A}_{20}\surd$	$\bar{A}_{00}\surd$		$\bar{A}_{20}\surd$						
4. $\bar{A}_{11}$–$\bar{A}_{10} \cdot \bar{A}_{10}^{\text{T}}$	$\bar{A}_{10}$		$\bar{A}_{11}\surd$	$\bar{A}_{10}$		$\bar{A}_{11}\surd$	$\bar{A}_{10}\surd$		$\bar{A}_{11}\surd$			
5. $\bar{A}_{21}$–$\bar{A}_{20} \cdot \bar{A}_{10}^{\text{T}}$	$\bar{A}_{20}$	$\bar{A}_{10}$	$\bar{A}_{21}\surd$	$\bar{A}_{20}$	$\bar{A}_{10}$	$\bar{A}_{21}\surd$	$\bar{A}_{20}\surd$	$\bar{A}_{10}\surd$	$\bar{A}_{21}\surd$			
6. $\bar{A}_{22}$–$\bar{A}_{20} \cdot \bar{A}_{20}^{\text{T}}$	$\bar{A}_{20}$		$\bar{A}_{22}\surd$	$\bar{A}_{20}$		$\bar{A}_{22}\surd$	$\bar{A}_{20}\surd$		$\bar{A}_{22}\surd$			
7. FLA_Potrf_unb($\bar{A}_{11}$)			$\bar{A}_{11}$			$\bar{A}_{11}$			$\bar{A}_{11}$			$\bar{A}_{11}\surd$
8. $\bar{A}_{21} \cdot \text{TRIL}(\bar{A}_{11})^{-\text{T}}$	$\bar{A}_{11}$		$\bar{A}_{21}$	$\bar{A}_{11}$		$\bar{A}_{21}$	$\bar{A}_{11}$		$\bar{A}_{21}$	$\bar{A}_{11}$		$\bar{A}_{21}\surd$
9. $\bar{A}_{22}$–$\bar{A}_{21} \cdot \bar{A}_{21}^{\text{T}}$	$\bar{A}_{21}$		$\bar{A}_{22}$	$\bar{A}_{21}$		$\bar{A}_{22}$	$\bar{A}_{21}$		$\bar{A}_{22}$	$\bar{A}_{21}$	$\bar{A}_{12}$	$\bar{A}_{22}\surd$
10. FLA_Potrf_unb($\bar{A}_{22}$)			$\bar{A}_{22}$			$\bar{A}_{22}$			$\bar{A}_{22}$			$\bar{A}_{22}$

Figura 4.48: Ilustración de la planificación de operaciones para la factorización de Cholesky densa de la matriz compuesta por 3×3 bloques en (4.49) usando el algoritmo por bloques en la figura 4.47. Las etiquetas "$\surd$" denotan a aquellos operandos que están disponibles (esto es, los operandos que no dependen de otras operaciones).

los operandos de los sistemas triangulares con $\bar{A}_{10}$ y $\bar{A}_{20}$ (operaciones 2. y 3.) como disponibles (ver la columna etiquetada como "Después 1^a oper." en la figura 4.48). Dos hebras pueden entonces desencolar estas tareas y ejecutarlas en paralelo, actualizando las correspondientes entradas de la lista una vez se completan (ver la columna etiquetada como "Después 3^a oper."). La ejecución continúa de esta manera hasta que todas las tareas se han calculado. Es interesante hacer notar que la planificación de las tareas no ocurre necesariamente en el orden descrito. En tanto las dependencias sean satisfechas, otras planificaciones son posibles. Así, por ejemplo, la operación identificada como 4. en la figura 4.48 puede calcularse antes que la etiquetada como 3. Hay otras planificaciones (ordenaciones temporales de la ejecución de las tareas) igualmente posibles.

En resumen, esta aproximación combina dos técnicas de los procesadores superescalares: planificación dinámica de instrucciones y ejecución fuera de orden, ocultando la gestión de las dependencias de datos del desarrollador de la biblioteca y de sus usuarios. Esta aproximación separa la programación de rutinas para operaciones de álgebra lineal de la ejecución específica sobre arquitecturas paralelas. Como resultado, no es necesario realizar ningún cambio sobre los códigos FLASH que implementan las operaciones de álgebra lineal para ejecutarlas usando múltiples núcleos. La planificación dinámica se adapta automáticamente para explotar eficientemente los recursos (núcleos o procesadores) del sistema explotando mejor el paralelismo existente en la operación.

4.3.3. Aplicación a la factorización de Cholesky banda

Las técnicas descritas anteriormente en esta sección son de directa aplicación al problema del cálculo en paralelo de la factorización de Cholesky de una matriz simétrica definida positiva banda (y también a otras operaciones como las factorizaciones LU o QR de matrices banda). Tan sólo es necesario tener en cuenta que, en la factorización banda, los bloques que únicamente contienen elementos que están fuera de la banda son bloques nulos y que, por tanto, no es necesario operar sobre ellos. Siguiendo esta idea, se ha realizado un estudio comparativo de las dos implementaciones siguientes:

Implementación LAPACK PBTRF

Esta implementación se describió en la sección 4.1.

Implementación FLAME PBTRF_{AB}

La implementación PBTRF_{AB} corresponde al código de un algoritmo por bloques que calcula la factorización de Cholesky de una matriz banda análogo al mostrado en la figura 4.46, con almacenamiento jerárquico de matrices usando FLASH y ejecución dinámica fuera de orden usando SuperMatrix. Las matriz banda se dispone en memoria como una matriz de bloques de dimensiones $b \times b$, donde se almacenan únicamente los bloques que tienen elementos no nulos. De este modo los elementos que están en un mismo bloques (no vacío) se encuentran en posiciones adyacentes de memoria. El uso de SuperMatrix no afecta al código, tan sólo supone que el control de la ejecución está en manos de este entorno.

4.3.4. Resultados experimentales

Los resultados que se muestran a continuación comparan las prestaciones de las rutinas PBTRF y PBTRF_{AB} sobre dos arquitecturas multiprocesador. No se muestran resultados para un único procesador pues, en tal caso, la ejecución de PBTRF_{AB} coincide en gran medida con PBTRF, encontrándose las únicas diferencias en el uso de almacenamiento jerárquico del primero y también el sobrecoste de una gestión de dependencias innecesaria cuando sólo se utiliza una hebra.

Arquitecturas ITANIUM(NUMA) y OPTERON(SMP)

La figura 4.49 contiene las prestaciones de las rutinas PBTRF y PBTRF_{AB} en las dos arquitecturas consideradas, ITANIUM(NUMA) y OPTERON(SMP), usando *MKL* (aunque no se muestran aquí, los resultados con *GotoBLAS* son similares). Los resultados revelan la clara ventaja obtenida gracias al almacenamiento por bloques y la planificación dinámica de tareas, incluso para matrices de banda estrecha. En ITANIUM los resultados de PBTRF son los correspondientes a la ejecución con únicamente 4 de los 16 procesadores del sistema. En este caso en particular se ha detectado que, debido al bajo nivel de paralelismo extraído por la aproximación tradicional empleada en esta rutina, la utilización de más de 4 procesadores incurre en un tiempo de ejecución superior.

4.3.5. Conclusiones

La implementaciones de las rutinas de factorización de matrices densas y banda en *LAPACK* extraen todo el paralelismo dentro de las llamadas a los núcleos básicos de *BLAS*. Especialmente en el caso de las matrices banda, este paralelismo resulta insuficiente para aprovechar los recursos cuando el sistema está formado por poco más de 4 procesadores/núcleos. La aproximación alternativa basada en una planificación dinámica consigue extraer un mayor grado de paralelismo de la aplicación y, combinada con el almacenamiento compacto de un algoritmo por bloques, explota la localidad de referencia de manera más eficiente. Los resultados con la factorización de Cholesky banda muestran la efectividad de esta aproximación en dos plataformas bien diferentes. Estos mismos resultados son esperables si se plantean algoritmos por bloques para las factorizaciones LU y QR.

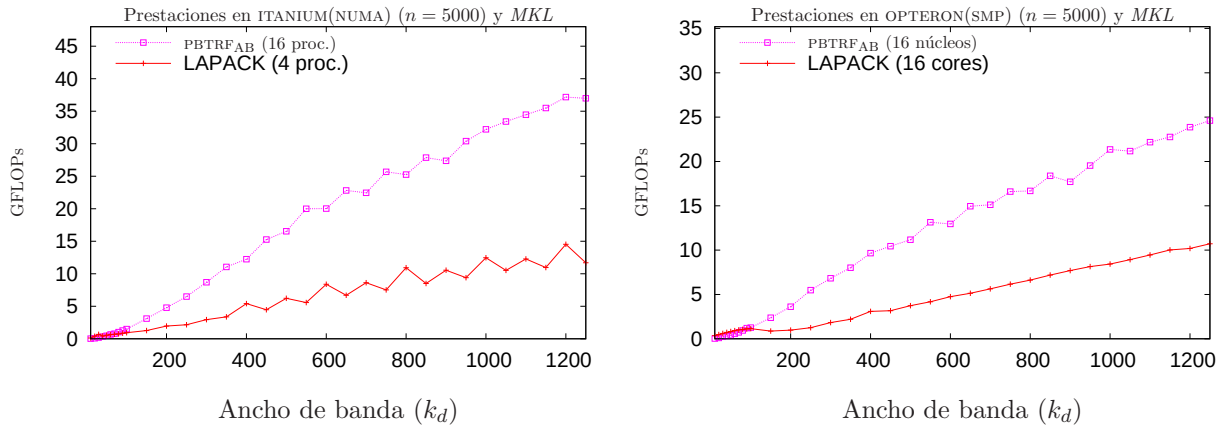


Figura 4.49: Comparativa de las diferentes implementaciones de los algoritmos para el cálculo de la factorización de Cholesky de una matriz banda.

Capítulo 5

Ampliaciones a LAPACK banda

En algunas aplicaciones que dan lugar a sistemas de ecuaciones lineales se conoce *a priori* que el uso de pivotamiento durante el cálculo de la factorización LU es innecesario. Cuando la matriz de coeficientes del sistema es densa, el uso de pivotamiento en la implementación tiene un efecto menor sobre las prestaciones. Sin embargo, cuando la matriz de coeficientes presenta una estructura banda, la situación es completamente diferente: el uso del pivotamiento incrementa el ancho de banda de la matriz triangular resultante de forma no desdeñable, incrementando en consecuencia los costes computacional y de almacenamiento del problema. Así mismo, la elaboración de una rutina para el cálculo de la factorización LU *sin pivotamiento* de una matriz banda no es trivial a partir de la correspondiente rutina densa de *LAPACK*. Así pues, una primera contribución de este capítulo es la presentación de nuevas rutinas escalares y por bloques para el cálculo de la factorización LU.

Por otro lado, la resolución de problemas lineales de mínimos cuadrados

$$\min_x \|Ax - b\|,$$

siendo $A \in \mathbb{R}^{m \times n}$ y $b \in \mathbb{R}^m$, $m \geq n$, se aborda habitualmente mediante el cálculo de la factorización QR de A [40]. Cuando esta matriz es banda, explotar su estructura puede reducir considerablemente el coste espacial y temporal de obtener la solución del problema. Sin embargo, hasta la fecha *LAPACK* no incluye rutinas para el cálculo de esta factorización, debido probablemente a la dificultad de su implementación y al menor número de aplicaciones en las que aparece este tipo de problemas, si se comparan con otros más habituales como la resolución de sistemas de ecuaciones lineales o los propios problemas de mínimos cuadrados cuando la matriz que interviene es densa. En este capítulo se aportan dos nuevas rutinas, una escalar y otra por bloques, para el cálculo de la factorización QR que permiten una resolución eficiente del problema cuando A es una matriz banda, y que son la llave para la resolución eficaz del problema lineal de mínimos cuadrados.

Todas las rutinas presentadas se evalúan sobre procesadores actuales y plataformas paralelas de memoria compartida utilizando las implementaciones optimizadas de los núcleos computacionales básicos de *GotoBLAS* y *MKL*. Los resultados se estructuran en dos secciones, con la primera de éstas dedicada a la factorización LU y la segunda a la factorización QR.

5.1. Factorización LU sin pivotamiento

En esta sección se analizan diferentes implementaciones para la factorización LU (figura 5.1). En esta operación se descompone la matriz $A \in \mathbb{R}^{n \times n}$ en el producto de una matriz triangular inferior unitaria, $L \in \mathbb{R}^{n \times n}$, y una matriz triangular superior, $U \in \mathbb{R}^{n \times n}$, de la forma

$$A = L \cdot U. \tag{5.1}$$

```
      SUBROUTINE DNBTRF( M, N, KL, KU, A, LDA, INFO)
*     .. Scalar Arguments ..
      INTEGER            M, N, KL, KU, LDA, INFO
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, *)
*
*  Purpose
*  =======
*
*  DNBTRF computes an LU factorization of a real m-by-n band matrix A
*  with ku super-diagonals and kl sub-diagonals
*
```

Figura 5.1: Especificación propuesta para la rutina NBTRF.

Aunque para garantizar una buena calidad numérica (en términos de precisión) de los factores triangulares en la práctica se utiliza el pivotamiento para el cálculo de la factorización LU [40], tal y como se mostró en el capítulo 4.2, existen matrices que no precisan de esta técnica para obtener su factorización LU con plena precisión, como por ejemplo es el caso de las matrices diagonalmente dominantes.

A continuación se muestran diferentes implementaciones para el cálculo de la descomposición (5.1) cuando la matriz A presenta anchos de banda superior e inferior k_u y k_l respectivamente. En estas circunstancias L es una matriz triangular con ancho de banda inferior k_l, mientras que U es una matriz triangular con ancho de banda superior k_u. La factorización LU sin pivotamiento (sea o no A una matriz banda) no es una operación soportada por la biblioteca *LAPACK*, que únicamente incluye rutinas para el cálculo de la factorización LU aplicando pivotamiento (operación (4.21)).

5.1.1. Algoritmo $\text{NBTRF}_{\text{UNB}}$

El algoritmo $\text{NBTRF}_{\text{UNB}}$, figura 5.2, calcula la factorización LU de una matriz general banda. Para ello, recorre la matriz a lo largo de su diagonal principal de izquierda a derecha, computando en cada iteración los elementos de una columna de A. Los elementos de A ya computados son reemplazados por los de los factores L y U.

Cada iteración del algoritmo requiere dos operaciones algebraicas: dividir los elementos de a_{21} por α_{11} y actualizar A_{22} restándole el resultado del producto entre los vectores a_{21} y a_{12}^T.

Al finalizar el algoritmo, los elementos de L sobreescriben la parte triangular inferior estricta de A, mientras que los elementos de U hacen lo propio con los elementos de la parte triangular superior de A.

5.1.2. Algoritmo $\text{NBTRF}_{\text{BLK}}$

El algoritmo mostrado en la figura 5.3 también obtiene la factorización LU de una matriz banda, en este caso trabajando por bloques. Para el cómputo de los elementos de L y U se emplean operaciones del tercer nivel de *BLAS*. El algoritmo recorre la matriz A de izquierda a derecha, computando b columnas de L y U en cada iteración. Los elementos de L y U calculados reemplazan a los de la matriz A, de forma que al finalizar el algoritmo las matrices L y U se encuentran almacenadas en el espacio que inicialmente ocupaban $\text{STRIL}(A)$ y $\text{TRIU}(A)$ respectivamente. Los

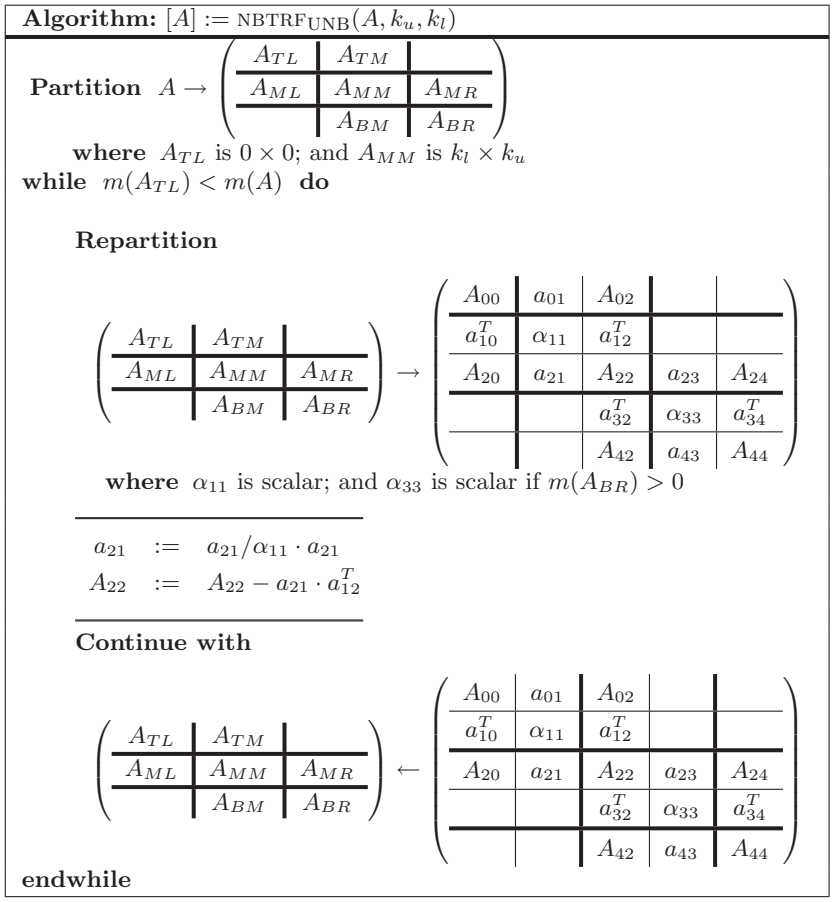


Figura 5.2: Algoritmo $\text{NBTRF}_{\text{UNB}}$ para la factorización $A = L \cdot U$.

elementos de la diagonal de L no son almacenados, ya que todos toman el valor 1.

Durante cada iteración se opera con los elementos de los bloques que conforman la región activa (mostrada en la figura 5.4). Inicialmente se factoriza el bloque $[A_{11}^T, A_{21}^T, A_{31}]^T$. Una vez calculada esta operación, se procede a la actualización del resto, para lo que se realizan diversos productos de matrices.

La cualidad más interesante de este algoritmo es que la mayoría de las operaciones aritméticas son realizadas por operaciones del nivel 3 de *BLAS*. Por contra, el número de operaciones realizadas en cada iteración es muy elevado, aumentando con ello el sobrecoste debido a la invocación de rutinas; además, se opera con bloques de dimensión reducida, reduciendo el grado de paralelismo del proceso.

Algorithm: $A := \text{NBTRF}_{\text{BLK}}(A, k_u, k_l)$

Partition $A \rightarrow \left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0; and A_{MM} is $k_l \times k_u$

while $n(A_{TL}) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & A_{01} & A_{02} & & \\ \hline A_{10} & A_{11} & A_{12} & A_{13} & \\ \hline A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ \hline & A_{31} & A_{32} & A_{33} & A_{34} \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right)$$

where A_{11}, A_{33} are $b \times b$;
A_{22} is $(k_l - b) \times (k_u - b)$

$$\begin{array}{lcl} A_{11} & := & \text{LU}(A_{11}) \\ A_{21} & := & A_{21} \cdot \text{TRIU}(A_{11})^{-1} \\ A_{31} & := & A_{31} \cdot \text{TRIU}(A_{11})^{-1} \\ A_{12} & := & \text{TRILU}(A_{11})^{-1} \cdot A_{12} \\ A_{22} & := & A_{22} - A_{21} \cdot A_{12} \\ A_{32} & := & A_{32} - A_{31} \cdot A_{12} \\ A_{13} & := & \text{TRILU}(A_{11})^{-1} \cdot A_{13} \\ A_{23} & := & A_{23} - A_{21} \cdot A_{13} \\ A_{33} & := & A_{33} - A_{31} \cdot A_{13} \end{array}$$

Continue with

$$\left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & A_{01} & A_{02} & & \\ \hline A_{10} & A_{11} & A_{12} & A_{13} & \\ \hline A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ \hline & A_{31} & A_{32} & A_{33} & A_{34} \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right)$$

endwhile

Figura 5.3: Algoritmo por bloques $\text{NBTRF}_{\text{BLK}}$ para la factorización $A = L \cdot U$.

5.1.3. Implementaciones basadas en rutinas de BLAS-3 denso

Implementación NBTRF

La rutina NBTRF implementa el algoritmo $\text{NBTRF}_{\text{BLK}}$. La secuencia de operaciones ejecutadas en esta rutina es la siguiente:

$$\begin{array}{lll} & A_{11} := \text{LU}(A_{11}) & (5.2) \\ (\text{TRSM}) & A_{21} := A_{21} \cdot \text{TRIU}(A_{11})^{-1}, & (5.3) \\ & W_{31} := \text{TRIU}(A_{31}), & (5.4) \\ (\text{TRSM}) & W_{31} := W_{31} \cdot \text{TRIU}(A_{11})^{-1}, & (5.5) \\ (\text{TRSM}) & A_{12} := \text{TRILU}(A_{11})^{-1} \cdot A_{12}, & (5.6) \\ (\text{GEMM}) & A_{22} := A_{22} - A_{21} \cdot A_{12}, & (5.7) \\ (\text{GEMM}) & A_{32} := A_{32} - W_{31} \cdot A_{12}, & (5.8) \\ & W_{13} := \text{TRIL}(A_{13}), & (5.9) \end{array}$$

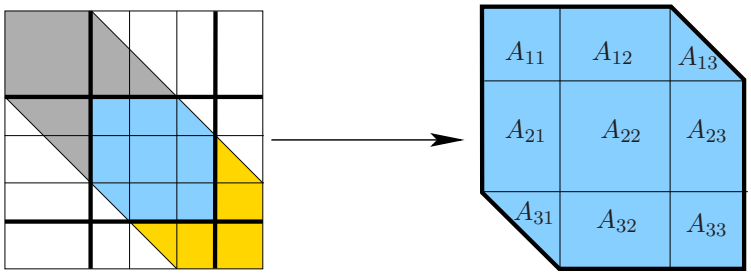


Figura 5.4: Bloques que forman la región activa en el algoritmo NBTRF$_{\text{BLK}}$.

$$
\begin{aligned}
(\text{TRSM}) \qquad W_{13} &:= \text{TRILU}(A_{11})^{-1} \cdot W_{13}, && (5.10)\\
(\text{GEMM}) \qquad A_{23} &:= A_{23} - A_{21} \cdot W_{13}, && (5.11)\\
(\text{GEMM}) \qquad A_{33} &:= A_{33} - W_{31} \cdot W_{13}. && (5.12)\\
A_{31} &:= W_{31}, && (5.13)\\
A_{13} &:= W_{13}. && (5.14)
\end{aligned}
$$

La operación (5.2) corresponde a una factorización LU sin pivotamiento de una matriz densa, y se implementa mediante tres bucles anidados siendo ésta la única no ejecutada por un núcleo optimizado de *BLAS*-3. Una vez factorizado este bloque, se actualizan los bloques A_{21}, A_{31} y A_{12} mediante invocaciones a la rutina TRSM. La resolución de estos sistemas triangulares permite actualizar los bloques A_{22} y A_{32} mediante sendas llamadas a la rutina GEMM. A continuación se actualiza el bloque A_{13} y, tras él, los bloques A_{23} y A_{33} invocando a las rutinas *BLAS* TRSM y GEMM (2 veces).

La rutina NBTRF realiza la mayoría de las operaciones aritméticas mediante núcleos computacionales optimizados de *BLAS*-3 y realiza un acceso a los elementos de A por columnas. Estas cualidades la convierten en una rutina eficiente. Sin embargo, presenta otras cualidades no tan positivas que alentan a la búsqueda de nuevas rutinas para el cálculo de la operación (5.1). Concretamente, NBTRF precisa de varias copias a/desde los espacios de trabajo W_{13} y W_{31}, e invoca a ocho rutinas *BLAS* por iteración, algunas de las cuales operan con bloques que únicamente tienen $b \times b$ elementos.

Implementación NBTRF$_{\text{AM}}$

La rutina NBTRF$_{\text{AM}}$ implementa una versión modificada del algoritmo NBTRF$_{\text{BLK}}$. El objetivo principal de esta implementación es reducir el sobrecoste introducido en NBTRF por las copias realizadas sobre/desde los espacios de trabajo W_{13} y W_{31}.

Esta variante requiere una modificación en la forma en la que la matriz A es almacenada, que consiste en aumentar el espacio de almacenamiento de A en b filas, todas ellas situadas por debajo de los elementos de A. Este espacio adicional contiene inicialmente elementos nulos y permite al algoritmo operar como si los bloques A_{31} y A_{13} se encontraran almacenados en su totalidad. Como resultado, las operaciones (5.3) y (5.5) pueden agruparse en un sóla operación y lo mismo sucede con las operaciones (5.6), (5.10) y también con (5.7), (5.8) (5.11), (5.12). Además, el nuevo esquema de almacenamiento hace innecesaria la utilización de espacios de trabajo. Así, la secuencia

de operaciones ejecutadas por la rutina NBTRF$_{\text{AM}}$ es la siguiente:

$$A_{11} := \text{LU}(A_{11}), \tag{5.15}$$

$$\text{(TRSM)} \quad \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} := \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot \text{TRIU}(A_{11})^{-1}, \tag{5.16}$$

$$\text{(TRSM)} \quad \begin{bmatrix} A_{12} \\ A_{13} \end{bmatrix} := \text{TRILU}(A_{11})^{-1} \cdot \begin{bmatrix} A_{12} \\ A_{13} \end{bmatrix}, \tag{5.17}$$

$$\text{(GEMM)} \quad \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} := \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} - \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \cdot \begin{bmatrix} A_{12} \\ A_{13} \end{bmatrix}. \tag{5.18}$$

En definitiva, esta rutina reduce notablemente el sobrecoste introducido por la rutina NBTRF, especialmente al eliminar las copias a/desde los espacios de trabajo. Además, las operaciones se realizan sobre bloques de mayor tamaño, hecho que beneficiará especialmente al uso de versiones paralelas de *BLAS*. Por contra, la rutina requiere una modificación en el esquema de almacenamiento, aumentando los requerimientos de memoria; en concreto, se requiere espacio para $b \times n$ elementos más y, aunque b habitualmente toma valores reducidos, es probable que n no lo haga. Además se opera con los elementos nulos de los bloques A_{31} y A_{13}.

Implementación NBTRF$_{\text{MERGE}}$

El principal inconveniente de la rutina NBTRF$_{\text{AM}}$ es que precisa de un cambio en el esquema de almacenamiento, que además es dependiente del tamaño de bloque empleado (el valor del tamaño de bloque se determina en función de la rutina y del *hardware* que la ejecuta). Eludir esta modificación en el esquema de almacenamiento sin renunciar a las mejoras introducidas por NBTRF$_{\text{AM}}$ es el objetivo de la implementación NBTRF$_{\text{MERGE}}$.

Esta nueva rutina simula que los bloques triangulares A_{13} y A_{31} se encuentran almacenados en su totalidad, de forma que las operaciones sobre bloques se pueden agrupar de la igual manera a como se hace en NBTRF$_{\text{AM}}$. Para simular que el bloque A_{31} se encuentra almacenado completamente, se realiza una copia sobre un espacio de trabajo de la región de memoria en la que se encontraría almacenada la parte triangular inferior estricta según el esquema de almacenamiento para matrices densas; a continuación se rellena esta región de memoria con ceros. Tras operar con A_{31}, los elementos copiados en el espacio de trabajo son devueltos a su posición original. Para operar con el bloque A_{13} se aplica la misma metodología (ver figura 5.5).

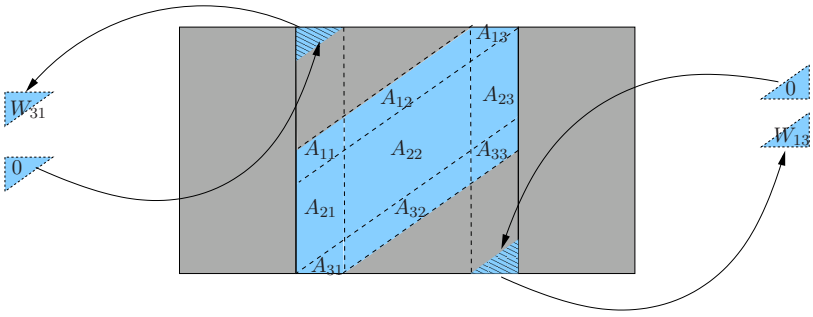


Figura 5.5: Copia de STRIL(A_{31}) a W_{31}, de STRIU(A_{13}) a W_{13} y relleno con ceros de las regiones de memoria que los almacenan.

Así pues, las operaciones ejecutadas durante una iteración de la rutina son las siguientes:

1. Copiar la región de memoria que almacena $\text{STRIL}(A_{31})$ y rellenar esta región con ceros.
2. Calcular la factorización LU de A_{11}.
3. Actualizar el bloque $\bar{A}_{21}$ $\left(= \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix}\right)$:

$$(\text{TRSM}) \quad \bar{A}_{21} := \bar{A}_{21} \cdot \text{TRIU}(A_{11})^{-1}. \tag{5.19}$$
$$\tag{5.20}$$

4. Copiar la región de memoria que almacena $\text{STRIU}(A_{13})$ y rellenar esta región con ceros.
5. Actualizar los bloques $\bar{A}_{12}$ $(= \begin{bmatrix} A_{12} & A_{13} \end{bmatrix})$ y $\bar{A}_{22}$ $\left(= \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}\right)$:

$$(\text{TRSM}) \quad \bar{A}_{12} := \text{STRIL}(A_{11})^{-1} \cdot \bar{A}_{12}, \tag{5.21}$$
$$(\text{GEMM}) \quad \bar{A}_{22} := \bar{A}_{22} - \bar{A}_{21} \cdot \bar{A}_{12}. \tag{5.22}$$

6. Copiar los elementos almacenados en W_{13} y W_{31} a sus posiciones originales.

La implementación $\text{NBTRF}_{\text{MERGE}}$ calcula la factorización LU de A_{11} mediante tres bucles anidados e invocando a rutinas de los niveles inferiores de *BLAS* para efectuar los cálculos básicos. La actualización de los bloques restantes de la región activa únicamente requiere la invocación de dos rutinas del tercer nivel de *BLAS*, TRSM y GEMM. Adicionalmente, se realizan cuatro copias de bloques de memoria de aproximadamente $\frac{b}{2} \times \frac{b}{2}$ elementos.

Respecto a NBTRF, $\text{NBTRF}_{\text{MERGE}}$ precisa de un menor número de invocaciones a rutinas *BLAS* y opera con bloques de mayor tamaño. Por contra, opera con los elementos nulos de los bloques A_{31} y A_{13}. Si se compara con NBTRF_{AM}, $\text{NBTRF}_{\text{MERGE}}$ no precisa aumentar el espacio de almacenamiento de A, mientras que aumenta ligeramente el sobrecoste debido a las copias realizadas sobre/desde los espacios de trabajo.

En consecuencia, $\text{NBTRF}_{\text{MERGE}}$ incorpora las mejores cualidades de NBTRF_{AM} al mismo tiempo que emplea el esquema de almacenamiento utilizado por el resto de rutinas *LAPACK* que operan con matrices banda.

5.1.4. Resultados experimentales

Arquitectura ITANIUM

BLAS secuencial La figura 5.6 compara las prestaciones de la rutina NBTRF con las diferentes implementaciones de *BLAS* en estudio. Las bibliotecas *GotoBLAS* y *MKL* alcanzan velocidades hasta tres veces superiores a las de *BLAS de referencia*. Esto se debe a que estas dos bibliotecas han sido optimizadas para la arquitectura ITANIUM. La biblioteca *MKL* es la más eficiente con matrices de banda media, mientras que *GotoBLAS* lo es para matrices de banda ancha.

A continuación se muestran los resultados de las diferentes rutinas propuestas en esta sección. La gráfica situada en la izquierda de la figura 5.7 muestra las prestaciones con *GotoBLAS*, mientras que la gráfica de la derecha muestra las de *MKL*. Para ambas bibliotecas, las tres rutinas propuestas presentan resultados similares, si bien NBTRF_{AM} para *GotoBLAS* y $\text{NBTRF}_{\text{MERGE}}$ para *MKL* son las opciones más eficientes. No obstante, dado que las diferencias entre las prestaciones son reducidas y que NBTRF_{AM} precisa la modificación del esquema de almacenamiento, puede considerarse que $\text{NBTRF}_{\text{MERGE}}$ es la mejor opción en ambos casos.

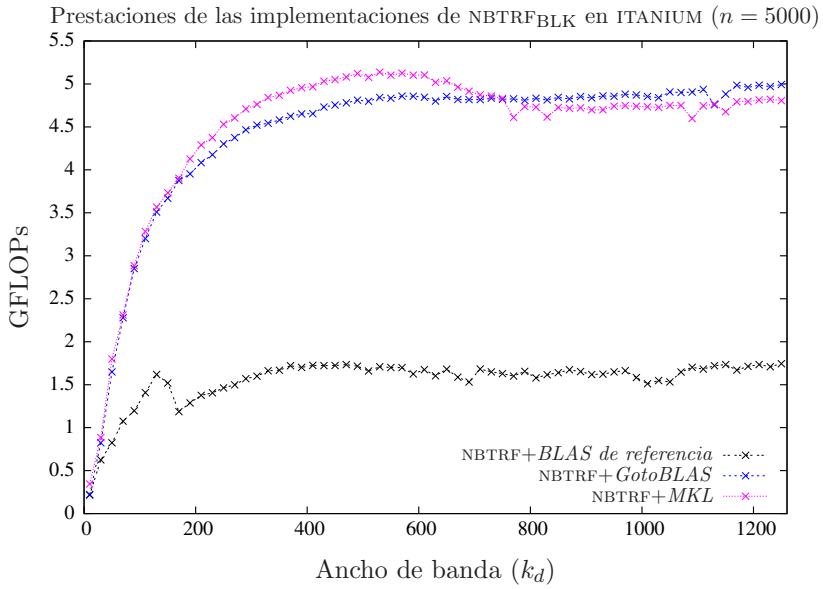


Figura 5.6: Comparativa de NBTRF empleando diferentes implementaciones de bibliotecas *BLAS* en ITANIUM.

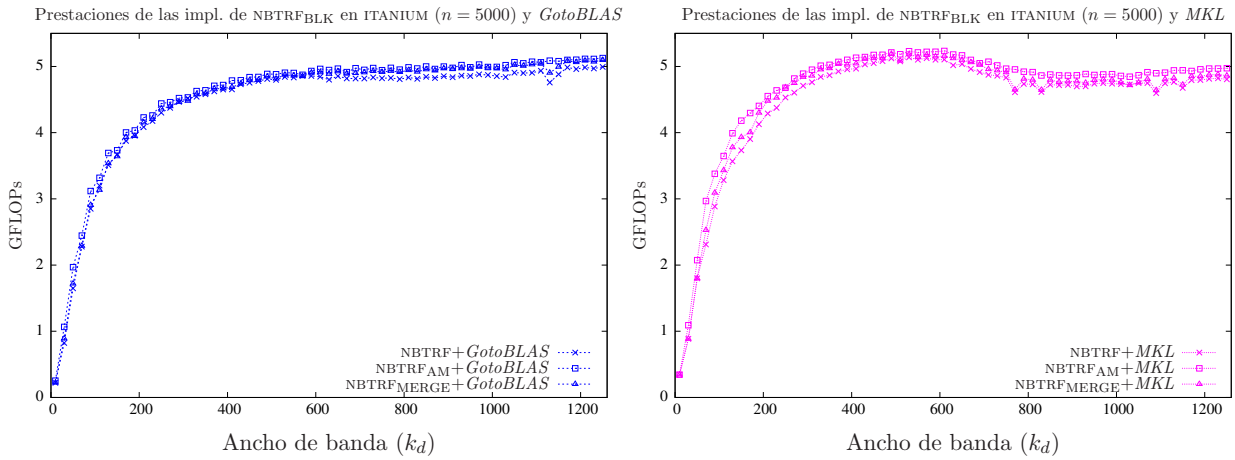


Figura 5.7: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

La figura 5.8 resume los resultados de las mejores implementaciones con cada una de las bibliotecas *BLAS* estudiadas. Como se puede observar, la rutina NBTRF$_{\text{MERGE}}$ es la más eficiente, en combinación con *MKL* cuando el ancho de banda de la matriz es inferior a 700 o junto con *GotoBLAS* para matrices con ancho de banda superior.

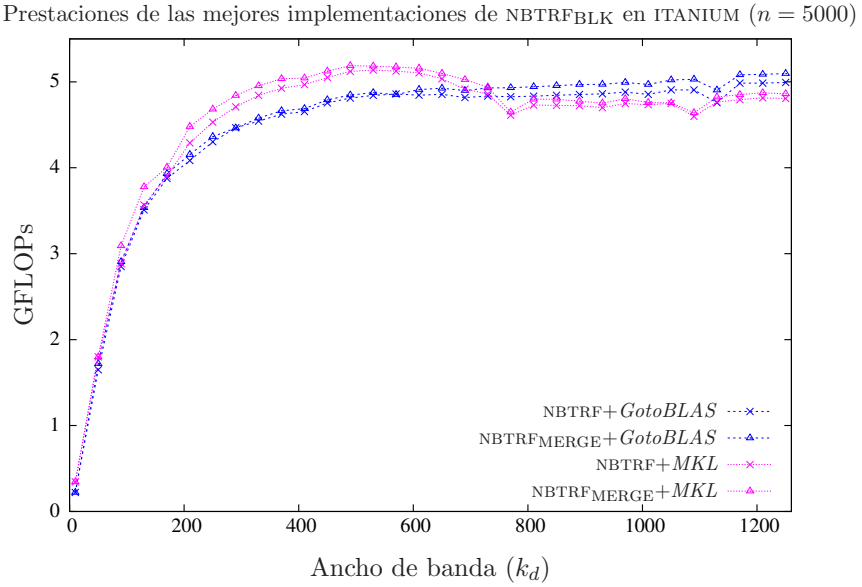


Figura 5.8: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

BLAS paralelo Las nuevas rutinas se han enlazado igualmente con versiones paralelas de las bibliotecas *BLAS*. La gráfica 5.9 presenta las prestaciones de las rutinas propuestas (NBTRF, NBTRF$_{\text{AM}}$ y NBTRF$_{\text{MERGE}}$) con las dos implementaciones de *BLAS* paralelas estudiadas, *GotoBLAS* y *MKL*. Para ambas bibliotecas, NBTRF$_{\text{AM}}$ es la implementación que mejores resultados ofrece, seguida por NBTRF$_{\text{MERGE}}$. Al igual que sucede en el caso secuencial, la diferencia de prestaciones entre estas dos rutinas no es suficientemente amplia como para justificar el mayor coste espacial y el cambio en el esquema de almacenamiento requerido por NBTRF$_{\text{AM}}$. En consecuencia, NBTRF$_{\text{MERGE}}$ se perfila como la mejor opción posible.

La figura 5.10 compara las prestaciones de las mejores implementaciones de cada una de las bibliotecas *BLAS*. NBTRF$_{\text{MERGE}}$ y *MKL* son, respectivamente, la rutina y la biblioteca más eficientes, con independencia del ancho de banda de la matriz A.

Arquitectura XEON

BLAS secuencial En la figura 5.11 se evalúa el rendimiento de la rutina NBTRF sobre la arquitectura XEON enlazada con las diferentes bibliotecas secuenciales de *BLAS* (*GotoBLAS*, *MKL* y *BLAS de referencia*). Las bibliotecas optimizadas para la arquitectura XEON son hasta 2 veces más rápidas que la biblioteca *BLAS de referencia*. De las bibliotecas evaluadas, *GotoBLAS* es la biblioteca más eficiente para esta arquitectura.

La figura 5.12 compara las tres rutinas propuestas, al mostrar la eficiencia de cada una de ellas con las implementaciones de *BLAS GotoBLAS* y *MKL*. Para ambas bibliotecas NBTRF$_{\text{AM}}$ y NBTRF$_{\text{MERGE}}$ son las rutinas más eficientes, ofreciendo ambas prestaciones similares. Conviene recordar que NBTRF$_{\text{AM}}$ exige modificar el esquema de almacenamiento de la matriz, por lo que NBTRF$_{\text{MERGE}}$ es probablemente la mejor opción.

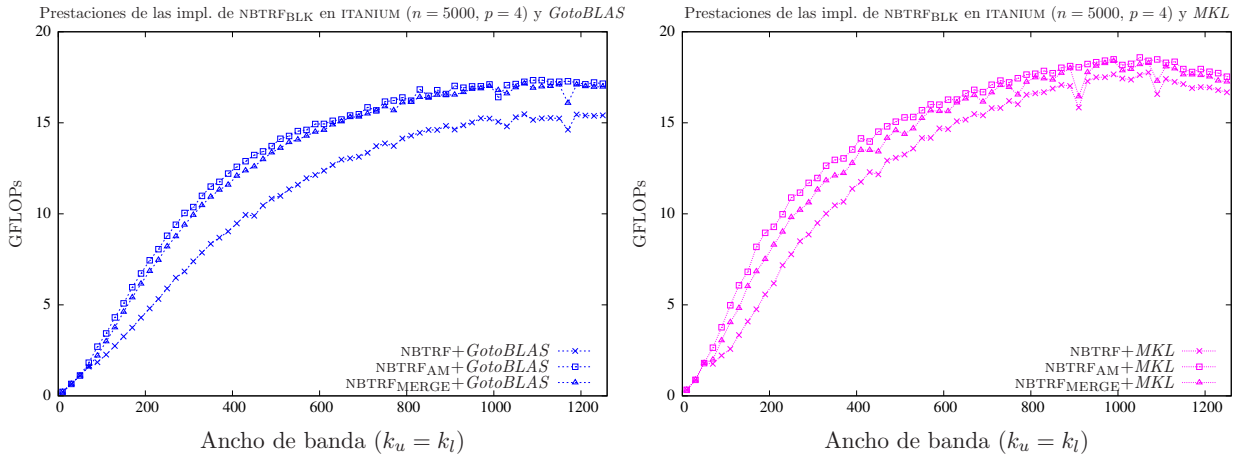


Figura 5.9: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

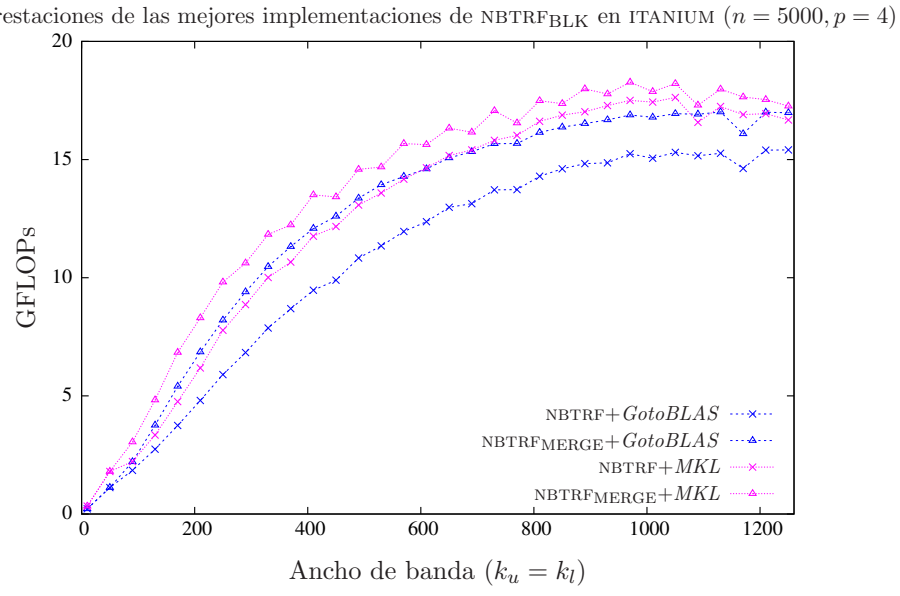


Figura 5.10: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

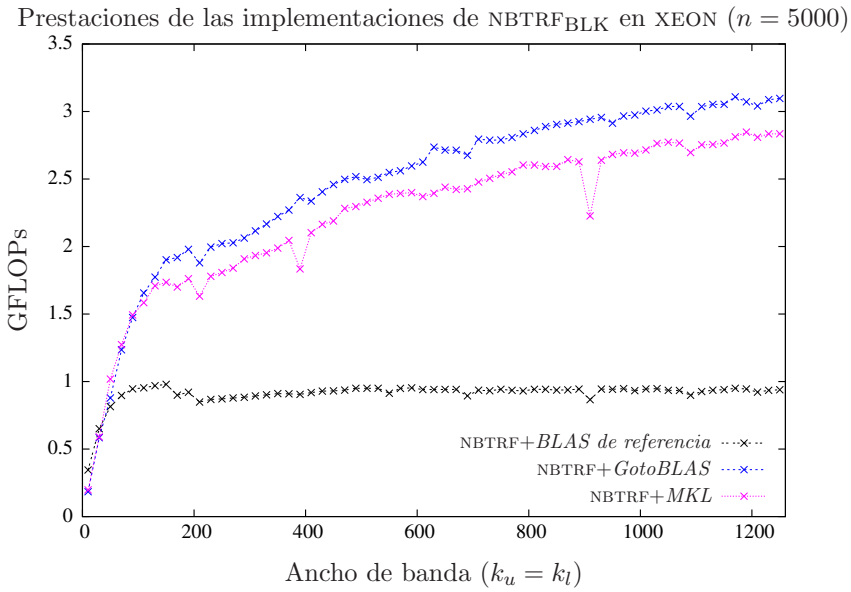


Figura 5.11: Comparativa de NBTRF empleando diferentes implementaciones de bibliotecas *BLAS* en XEON.

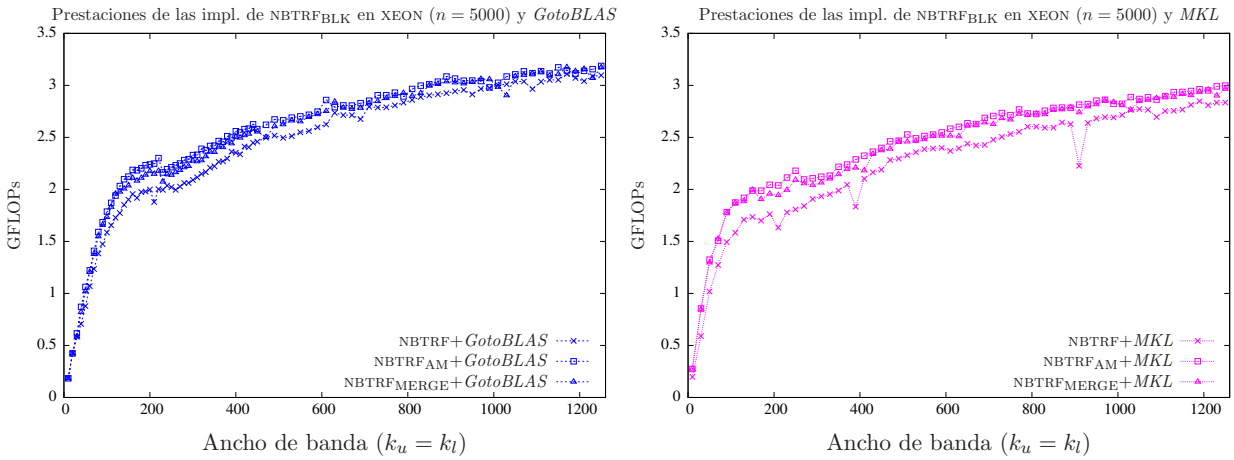


Figura 5.12: Comparativa de las diferentes implementaciones con versiones secuenciales de *BLAS*.

Finalmente se resumen en la gráfica 5.13 los resultados de las mejores rutinas con las dos implementaciones secuenciales *BLAS* más eficientes. La rutina más rápida es NBTRF$_{\text{MERGE}}$, mientras que la mejor biblioteca es *GotoBLAS*.

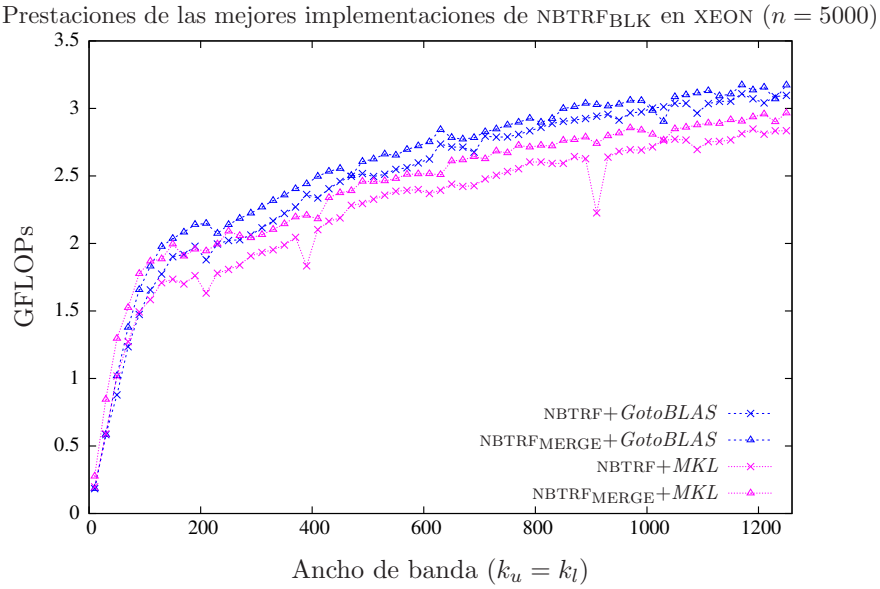


Figura 5.13: Comparativa de las mejores implementaciones con versiones secuenciales de *BLAS*.

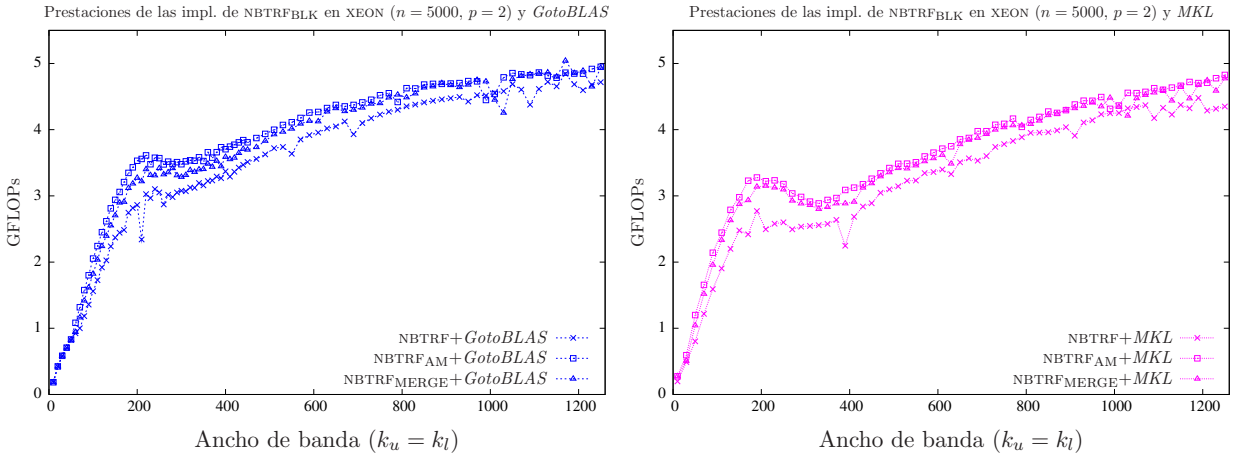


Figura 5.14: Comparativa de las diferentes implementaciones con versiones paralelas de *BLAS*.

BLAS paralelo La figura 5.14 recoge los resultados obtenidos por las tres rutinas propuestas y las dos implementaciones paralelas de *BLAS* estudiadas. Tanto para *GotoBLAS* como para *MKL*, las rutinas más eficientes son NBTRF$_{\text{AM}}$ y NBTRF$_{\text{MERGE}}$, mostrando ambas prestaciones similares.

La figura 5.15 presenta en una única gráfica las prestaciones de las mejores implementaciones para ambas bibliotecas *BLAS*. De esta gráfica se han eliminado los resultados de NBTRF$_{\text{AM}}$ puesto que, además de acarrear un mayor coste espacial, también precisa modificar el esquema de almacenamiento de la matriz. Como se refleja en la gráfica, *GotoBLAS* es más eficiente que *MKL*, y la rutina

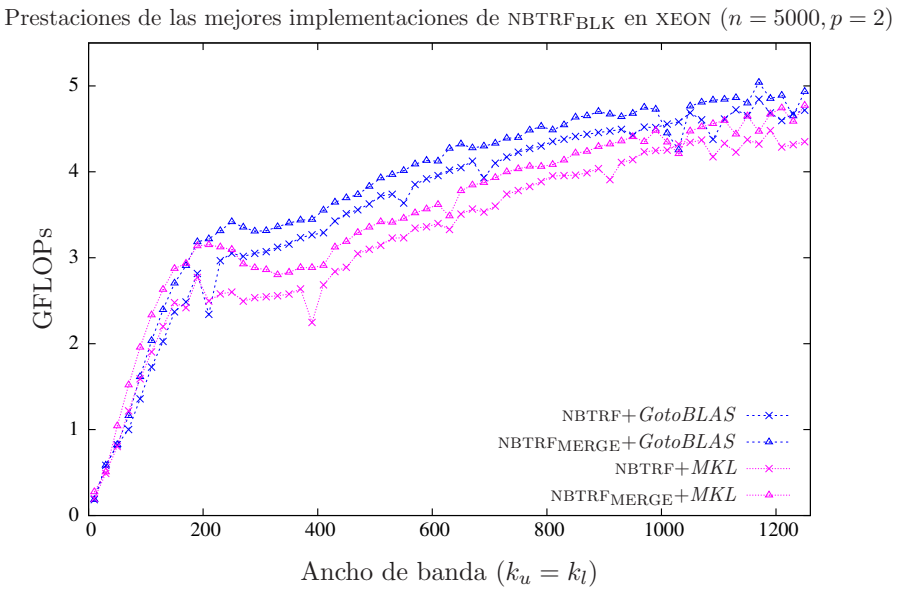


Figura 5.15: Comparativa de las mejores implementaciones con versiones paralelas de *BLAS*.

NBTRF$_{\text{MERGE}}$ más eficiente que NBTRF. Por lo tanto la combinación NBTRF$_{\text{MERGE}}$+*GotoBLAS* ofrece las mayores prestaciones.

5.1.5. Conclusiones

En esta sección se han propuesto tres implementaciones diferentes para el cálculo de la factorización LU sin pivotamiento de una matriz. Esta operación no está incluida en la biblioteca *LAPACK*, de manera que la única forma de calcularla hasta la fecha era invocando a la rutina de dicha biblioteca que calcula la factorización LU con pivotamiento. Las nuevas rutinas sin pivotamiento son considerablemente más rápidas (más de un 10 % en implementaciones secuenciales y entre un 34 % y un 50 % en implementaciones paralelas) que las correspondientes rutinas con pivotamiento evaluadas en la sección 4.2.

La rutina más eficiente planteada es NBTRF$_{\text{AM}}$, pero NBTRF$_{\text{MERGE}}$ ofrece prestaciones muy similares y, dado que la primera presenta el problema de requerir un esquema de almacenamiento diferente al empleado por las rutinas *LAPACK*, NBTRF$_{\text{MERGE}}$ se presenta como la mejor opción.

Respecto a las bibliotecas *BLAS* evaluadas, *MKL* es la más eficiente en la arquitectura ITANIUM, mientras que *GotoBLAS* lo es en la arquitectura XEON.

5.2. Factorización QR

La operación estudiada en esta sección es la factorización QR. Para una matriz $A \in \mathbb{R}^{m \times n}$ $m \geq n$, ésta se define como

$$A = Q \cdot R, \tag{5.23}$$

donde $Q \in \mathbb{R}^{m \times m}$ es una matriz ortogonal y $R \in \mathbb{R}^{m \times n}$ es una matriz triangular superior. Cuando A tiene anchos de banda superior e inferior k_u y k_l respectivamente, la matriz Q resultante presenta un ancho de banda inferior igual a k_l y R es una matriz triangular superior con ancho de banda superior $k_u + k_l$.

```
      SUBROUTINE DGBQRF( M, N, KL, KU, A, LDA, TAU, WORK, INFO)
*     .. Scalar Arguments ..
      INTEGER            M, N, KL, KU, LDA, INFO
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, *), TAU( * ), WORK( * )
*
*  Purpose
*  =======
*
*  DGBQRF computes a QR factorization of a real m-by-n band matrix A
*  with ku super-diagonals and kl sub-diagonals
*
```

Figura 5.16: Especificación propuesta para la rutina GBQRF.

Existen diversos métodos para calcular la factorización QR, basados en reflectores (o transformaciones) de Householder, rotaciones de Givens y el método de Gram-Schmidt [40]. Las implementaciones presentadas en esta sección (para la operación de la figura 5.16) utilizan transformaciones de Householder debido a que, cuando el número de elementos a anular es significativo, este tipo de técnica presenta un menor coste que las rotaciones de Givens. Además, frente al método de Gram-Schmidt, el uso de transformaciones de Householder puede formularse de manera numéricamente estable en forma de operaciones de *BLAS*-3, tal y como expondremos en esta sección [40].

5.2.1. Transformaciones de Householder

El principio básico sobre el que opera una transformación de Householder es la reflexión un vector sobre un plano. Si se elige correctamente la transformación, es posible lograr que al aplicar ésta el vector elegido quede únicamente con un elemento no nulo, tal y como se expone a continuación. Dado el vector $x = \begin{pmatrix} \chi_0 \\ x_1 \end{pmatrix}$, donde χ_0 es el primer elemento de x, el objetivo es calcular la transformación de Householder apropiada que haga ceros todos los elementos de x salvo χ_0 (que queda modificado). Para ello se calculan el vector u y el escalar τ como:

$$\mu_0 = \chi_0 + \text{signo}(\chi_0)\,\|x\|_2, \tag{5.24}$$

$$u = \begin{pmatrix} 1 \\ x_1/\mu_0 \end{pmatrix}, \quad \text{y} \tag{5.25}$$

$$\tau = \frac{2}{u^T u}. \tag{5.26}$$

Se puede comprobar entonces que la aplicación de la transformación $\left(I - \tau u u^T\right)$ sobre el vector x, elimina todos los elementos del resultado salvo el primero, que toma el valor $\eta = -\text{signo}(\chi_0)\,\|x\|_2$.

Los algoritmos para el cálculo de la factorización QR descritos a continuación calculan y aplican una transformación de Householder a cada una de las columnas de la matriz, de izquierda a derecha, para reducir ésta a la forma triangular superior. Este proceso está ilustrado gráficamente en la figura 5.17. El resultado es una matriz triangular, con ancho de banda superior $k_l + k_u$, que sobreescribe las entradas de A.

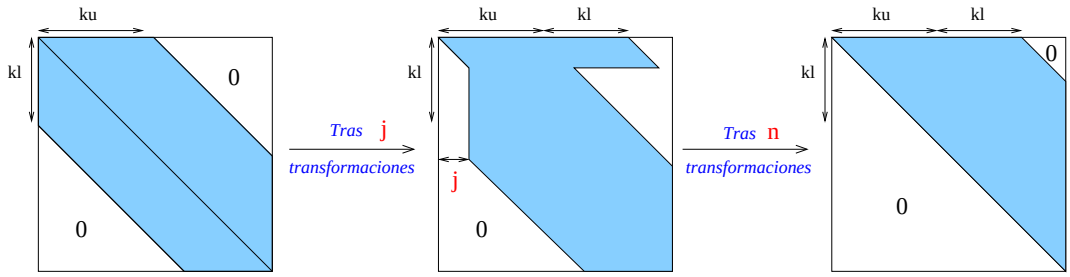


Figura 5.17: Matriz A con ancho de banda superior e inferior k_u y k_l respectivamente (izquierda); matriz A tras aplicarle j transformaciones de Householder (centro); matriz A tras aplicarle n transformaciones de Householder (derecha).

5.2.2. Algoritmo GBQRF$_{\text{UNB}}$

El algoritmo GBQRF$_{\text{UNB}}$, mostrado en la figura 5.18, calcula la factorización QR de una matriz banda, almacenando los reflectores de Householder sobre el vector $t \in \mathbb{R}^n$ y los elementos anulados de A. La matriz R resultante, por su parte, sobreescribe los correspondientes elementos en la parte triangular superior de A. El algoritmo recorre la matriz de izquierda a derecha y, en cada iteración, computa y aplica un reflector actualizando con éste un bloque de $k_l \times (k_u + k_l)$ elementos de A. En el algoritmo, la rutina HOUSEHOLDER calcula las operaciones (5.24)–(5.26), sobreescribiendo α_{11} con $\eta = -\text{signo}(\chi_0)\,\|x\|_2$ y a_{21} con los elementos de u salvo el primero; en τ_1 se devuelve el valor $2/(u^T u)$.

Las transformaciones de Householder aplicadas durante el proceso de factorización resultan en matriz triangular superior R con un banda de ancho superior $k_u + k_l$. Así pues, con el fin de disponer de espacio suficiente para almacenar las $k_u + k_l$ superdiagonales de R, al inicio del algoritmo se almacenan $k_u + k_l$ superdiagonales de A, pese a que inicialmente las k_l superdiagonales superiores contienen únicamente elementos nulos (ver figura 5.19).

La mayor virtud de GBQRF$_{\text{UNB}}$ es que los accesos a la matriz se realizan por columnas, tal y como éstos se encuentran almacenados. Por otro lado cada elemento de A es accedido hasta en $k_u + k_l$ ocasiones.

5.2.3. Implementación GBQF2

La rutina GBQF2 implementa el algoritmo GBQRF$_{\text{UNB}}$ invocando a dos núcleos de *BLAS*:

$$\text{(LARFG)} \quad \left(\begin{bmatrix} \alpha_{11} \\ a_{21} \end{bmatrix}, \tau_1 \right) := \text{HOUSEHOLDER} \left(\begin{bmatrix} \alpha_{11} \\ a_{21} \end{bmatrix} \right), \tag{5.27}$$

$$\text{(LARF)} \quad \begin{bmatrix} a_{12} \\ A_{22} \end{bmatrix} := \left(I - \tau_1 \cdot \begin{bmatrix} 1/\alpha_{11} \\ a_{21} \end{bmatrix} \cdot \begin{bmatrix} 1/\alpha_{11} \\ a_{21} \end{bmatrix}^T \right) \cdot \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix}. \tag{5.28}$$

La rutina LARFG genera el reflector para anular los elementos de a_{21}. La rutina LARF aplica el reflector a los elementos de $[a_{12}^T;\ A_{22}^T]^T$. Esta última operación en realidad se realiza en forma de un producto matriz por vector seguida de una actualización de rango 1:

$$\begin{bmatrix} a_{12} \\ A_{22} \end{bmatrix} := \begin{bmatrix} a_{12} \\ A_{22} \end{bmatrix} - \tau_1 \cdot \begin{bmatrix} 1/\alpha_{11} \\ a_{21} \end{bmatrix} \cdot \left(\begin{bmatrix} 1/\alpha_{11} \\ a_{21} \end{bmatrix}^T \cdot \begin{bmatrix} a_{12} \\ A_{22} \end{bmatrix} \right),$$

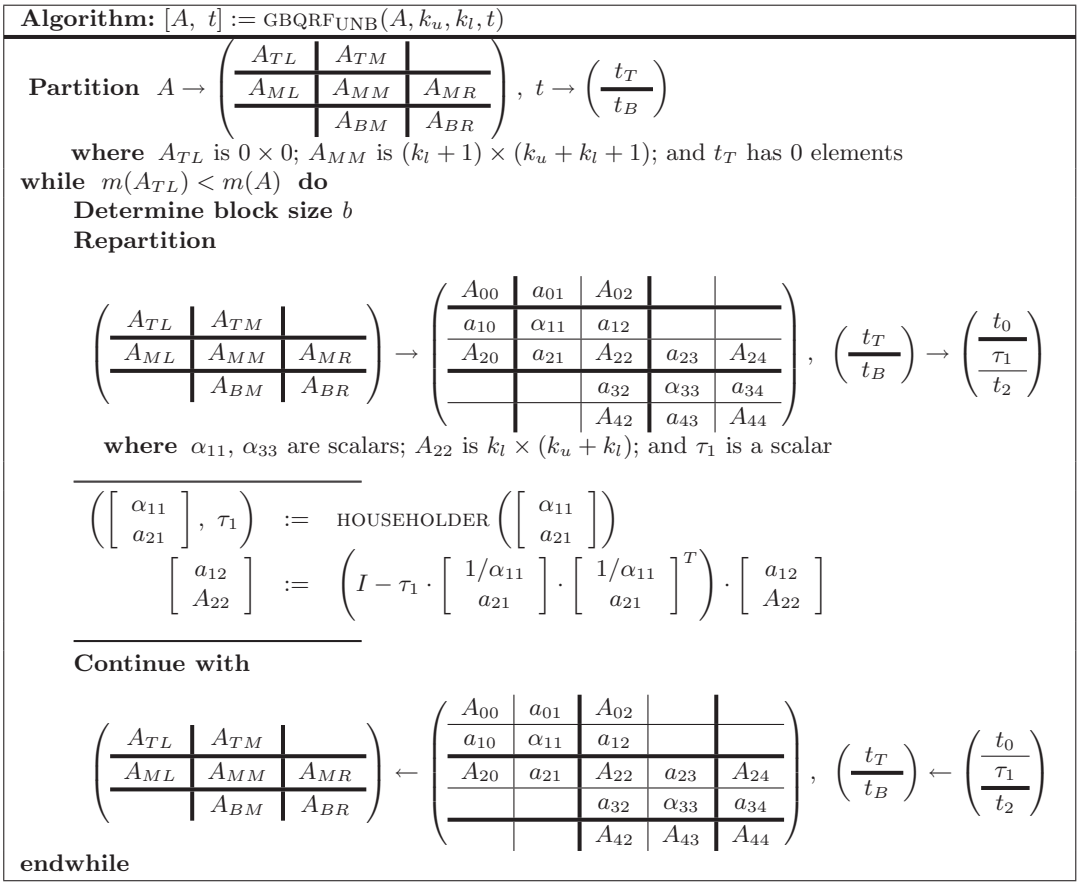


Figura 5.18: Algoritmo $\text{GBQRF}_{\text{UNB}}$ para la factorización $A = Q \cdot R$.

de modo que el coste computacional resultante es cuadrático. Con esta implementación, todas las operaciones necesarias para obtener la factorización QR son ejecutadas por rutinas *BLAS*. Aunque esta cualidad asegura cierta eficiencia, las rutinas *BLAS* empleadas pertenecen a los niveles 1 y 2 de *BLAS*, mientras que la operación estudiada pertenece al tercer nivel de *BLAS*. A continuación se expone cómo salvar este obstáculo para la factorización QR banda.

5.2.4. Algoritmo $\text{GBQRF}_{\text{BLK}}$

El algoritmo $\text{GBQRF}_{\text{BLK}}$ mostrado en la figura 5.20 calcula la factorización QR de una matriz banda almacenando los reflectores y la matriz resultado R sobre los elementos de la matriz A y el vector t. El algoritmo está basado de nuevo en el uso de transformaciones de Householder y, en cada iteración, realiza dos operaciones principales:

- Calcula y aplica las transformaciones de Householder oportunas para reducir el bloque formado por A_{11} y A_{21} a forma triangular superior.
- Aplica esas mismas transformaciones a los bloques A_{12} y A_{22}. Este cálculo se plantea mediante operaciones matriz-matriz, más concretamente mediante productos de matrices. El objetivo es que se puedan emplear rutinas *BLAS*-3 para su implementación.

Así pues, a diferencia de $\text{GBQRF}_{\text{UNB}}$ este algoritmo calcula la factorización QR en forma de operaciones entre matrices, lo que propicia una mayor eficiencia.

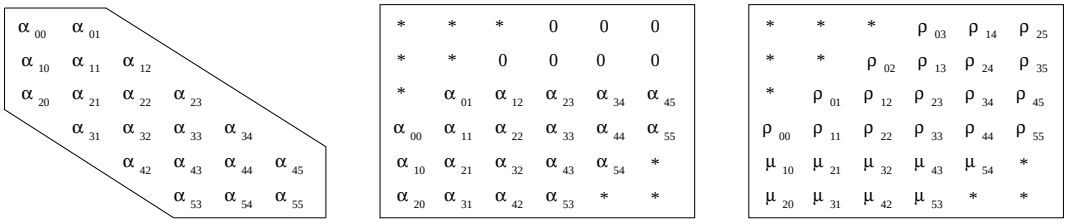


Figura 5.19: Matriz $A \in R^{6\times 6}$ con ancho de banda superior e inferior 1 y 2 respectivamente (izquierda); matriz almacenada según el esquema compacto para matrices banda con ancho de banda superior igual a $k_u + k_l$ (centro); matriz resultante de la factorización donde μ_{ij} y ρ_{ij} simbolizan, respectivamente, el elemento i del j-ésimo reflector de Householder y el elemento (i, j) de R (derecha).

Algorithm: $[A,\ t] := \text{GBQRF}_{\text{BLK}}(A, k_u, k_l, t)$

Partition $A \rightarrow \left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & \\ \hline & A_{BM} & A_{BR} \end{array}\right),\ t \rightarrow \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right)$

where A_{TL} is 0×0; A_{MM} is $(k_l + b) \times (k_u + k_l + b)$; and t_T has 0 elements

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & A_{01} & A_{02} & & \\ \hline A_{10} & A_{11} & A_{12} & & \\ \hline A_{20} & A_{21} & A_{22} & & \\ \hline & & A_{32} & A_{33} & \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right),\ \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right) \rightarrow \left(\begin{array}{c} t_0 \\ \hline t_1 \\ \hline t_2 \end{array}\right)$$

where A_{11}, A_{33} are $b \times b$; A_{22} is $k_l \times (k_u + k_l)$; and t_1 has b elements

$$\left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}, t_1\right) := geqrf_{\text{UNB}}\left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}, t_1\right)$$

$$T := \text{LARFT}\left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}, t_1\right)$$

$$\begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} := (I - U \cdot T \cdot U^T) \cdot \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} \quad \text{where} \quad U = \left[\text{STRIL}\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} + I\right]$$

Continue with

$$\left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & \\ \hline & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & A_{01} & A_{02} & & \\ \hline A_{10} & A_{11} & A_{12} & & \\ \hline A_{20} & A_{21} & A_{22} & & \\ \hline & & A_{32} & A_{33} & \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right),\ \left(\begin{array}{c} t_T \\ \hline t_B \end{array}\right) \leftarrow \left(\begin{array}{c} t_0 \\ \hline t_1 \\ \hline t_2 \end{array}\right)$$

endwhile

Figura 5.20: Algoritmo por bloques $\text{GBQRF}_{\text{BLK}}$ para la factorización $A = Q \cdot R$.

La figura 5.21 refleja el modo en que la matriz se particiona durante una iteración del algoritmo. Las transformaciones de Householder provocan el llenado de hasta k_l diagonales sobre la banda superior de la matriz. La región activa mostrada a la derecha de la figura está formada por cuatro bloques: A_{11}, A_{12}, A_{21} y A_{22}. La parte triangular inferior del bloque $b \times b$ situada en la parte

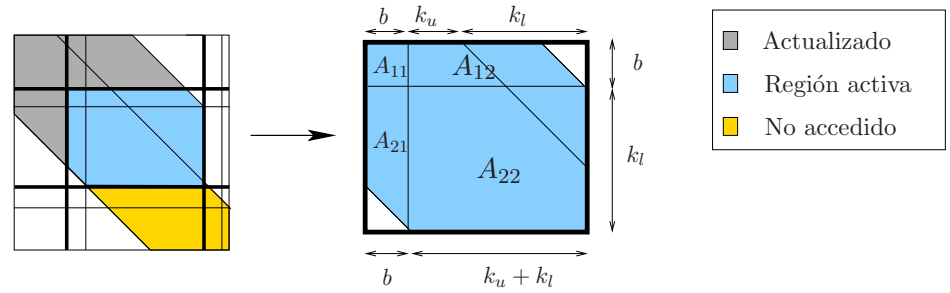


Figura 5.21: Particionado 5×5 aplicado a la matriz (izquierda) y detalle de la región activa (derecha).

inferior de A_{21} recae fuera de la banda y, por lo tanto, está formada por elementos nulos que no se encuentran físicamente almacenados debido al uso del esquema de almacenamiento compacto.

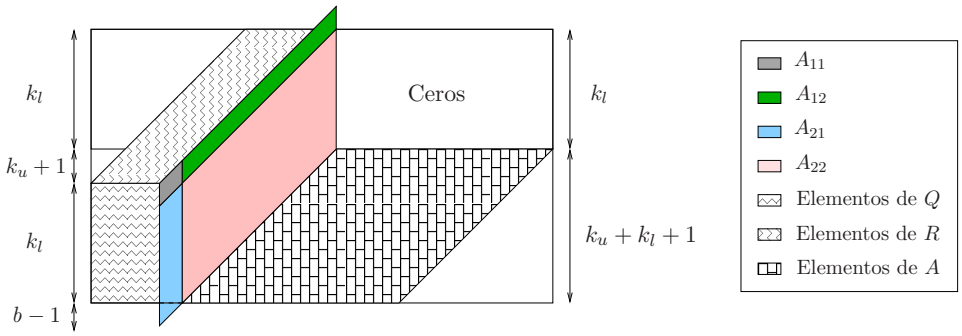


Figura 5.22: Almacenamiento físico de la matriz.

La figura 5.22 ilustra el almacenamiento físico de la matriz. Las primeras k_l filas almacenan los elementos del relleno debido a las transformaciones de Householder. Al finalizar el algoritmo, las $k_u + k_l + 1$ filas superiores almacenan los elementos de R, mientras que las k_l filas restantes almacenan los elementos de los reflectores. La figura también muestra la forma en que la región activa está almacenada. La parte inferior del bloque A_{21} no se encuentra almacenada, ya que recae fuera de la banda. Lo mismo sucede con la parte triangular superior estricta del bloque $b \times b$ situada en el estremo derecho del bloque A_{12}.

Las características que presenta GBQRF$_{\text{BLK}}$ lo convierten en un algoritmo susceptible de obtener buenas prestaciones:

- El número de accesos sobre elementos de A es reducido con respecto a los realizados por GBQRF$_{\text{UNB}}$, ya que cada elemento se accede como máximo en $\frac{k_u+k_l}{b}$ ocasiones.
- El acceso a todos los elementos se realiza por columnas.
- Las operaciones sobre los bloques A_{12} y A_{22} pueden ser ejecutadas por rutinas *BLAS*-3, lo que se traduce en un patrón de acceso con mayor localidad de referencia.

5.2.5. Implementación GBQRF

La rutina GBQRF implementa el algoritmo GBQRF$_{\text{BLK}}$ invocando a dos núcleos de *BLAS* y la rutina GEQF2 de *LAPACK*. Esta última invocación calcula la factorización QR de un bloque denso.

La secuencia de operaciones llevada a cabo en cada iteración es la siguiente:

$$W_{21} := \text{STRIL}(A_{21}), \tag{5.29}$$

$$\text{STRIL}(A_{21}) := 0, \tag{5.30}$$

$$\text{(GEQF2)} \quad \left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}, t_1 \right) := \text{GEQF2} \left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}, t_1 \right), \tag{5.31}$$

$$\text{(LARFT)} \quad T := \text{LARFT} \left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}, t_1 \right), \tag{5.32}$$

$$W_{21} := \text{STRIU}(A_{12}), \tag{5.33}$$

$$\text{STRIU}(A_{12}) := 0, \tag{5.34}$$

$$\text{(LARFB)} \quad \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} := (I - U \cdot T \cdot U^T) \cdot \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix}, \tag{5.35}$$

$$\text{STRIU}(A_{12}) := W_{12}. \tag{5.36}$$

En la primera operación de la iteración se crea una copia en el espacio de trabajo W_{21} del bloque $b \times b$ en la parte triangular inferior estricta de A_{21} para, a continuación, rellenar la región copiada con ceros. La operación (5.31) calcula la factorización QR de la matriz formada por los bloques A_{11} y A_{21}. De este proceso se obtiene un vector t_1 que, en su componente j-ésima, contiene el parámetro τ asociado al reflector de Householder que anula los elementos de la columna j-ésima de $[A_{11}^T;\ A_{21}^T]^T$. La rutina LARFT en (5.32) es la encargada de computar la matriz T a partir de los reflectores almacenados en la parte triangular inferior estricta de $[A_{11}^T;\ A_{21}^T]^T$ y los parámetros en t_1. Seguidamente, en (5.33) se realiza una copia la región de memoria que almacena la parte triangular superior estricta del bloque $b \times b$ situado más a la derecha de A_{21}, mientras que en (5.35) se rellena con ceros esa región. En (5.35) se aplican las transformaciones sobre los bloques A_{12} y A_{22} mediante la rutina LARFB de *BLAS*. La operación se realiza internamente dentro de esta rutina en forma de productos de matrices

$$\begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} := \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} - U \cdot \left(T \cdot \left(U^T \cdot \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} \right) \right),$$

y requiere un espacio de almacenamiento auxiliar para guardar el resultado del primer producto. Finalmente en (5.36) se devuelven los elementos almacenados en W_{12} a su posición original.

Hay que tener en cuenta que aunque el número de rutinas *BLAS* invocadas en esta implementación es elevado, el coste en operaciones aritméticas de la factorización QR es aproximadamente el doble que el coste de la factorización LU, por lo que en este caso esta circunstancia no es tan perjudicial.

5.2.6. Resultados experimentales

Arquitectura ITANIUM

A continuación se evalúan las prestaciones de las rutinas propuestas con diferentes implementaciones de *BLAS* sobre la arquitectura ITANIUM. La figura 5.23 incluye los resultados obtenidos con versiones secuenciales de *BLAS* (izquierda) y con versiones paralelas (derecha). En ambos casos, cuando el ancho de banda $k_u = k_l \leq 100$ las mejores prestaciones se obtienen para la rutina GBQF2 y *MKL*, mientras que la rutina *BLAS*-3 GBQRF es más eficiente con matrices de banda mayor. No hay diferencias importantes entre las prestaciones de *GotoBLAS* y *MKL* en el caso secuencial y la rutina GBQRF; sin embargo, en el caso multihebra, *GotoBLAS* es más eficiente para matrices con

ancho de banda superior e inferior ≤ 1000, mientras que *MKL* es la mejor opción con matrices de banda mayor.

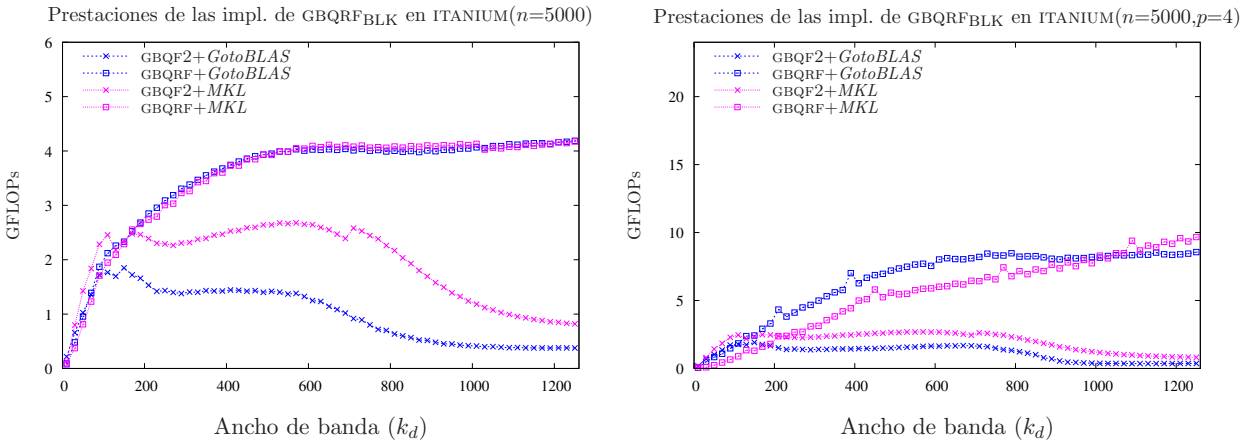


Figura 5.23: Comparativa de las diferentes implementaciones de GBQRF$_{\text{BLK}}$ en ITANIUM.

Arquitectura XEON

La figura 5.24 muestra las prestaciones de las nuevas rutinas sobre la arquitectura XEON. En el caso secuencial (a la izquierda de la figura), la rutina GBQF2 es la más eficiente al operar con matrices de banda estrecha ($k_u = k_l \leq 200$ con *GotoBLAS* y $k_u = k_l \leq 50$ con *MKL*), mientras que la rutina GBQRF, basada en *BLAS*-3, obtiene las mejores prestaciones al operar con matrices con ancho de banda mayor. Respecto a las implementaciones *BLAS* en estudio, *GotoBLAS* es más eficiente para cualquier ancho de banda. En el caso paralelo (gráfica de la derecha), las conclusiones son ligeramente diferentes, puesto que ahora ambas implementaciones de *BLAS* generan resultados muy similares. De nuevo la rutina *BLAS*-3 es mejor con matrices de banda media o ancha, mientras que GBQF2 es más rápida con matrices de banda estrecha.

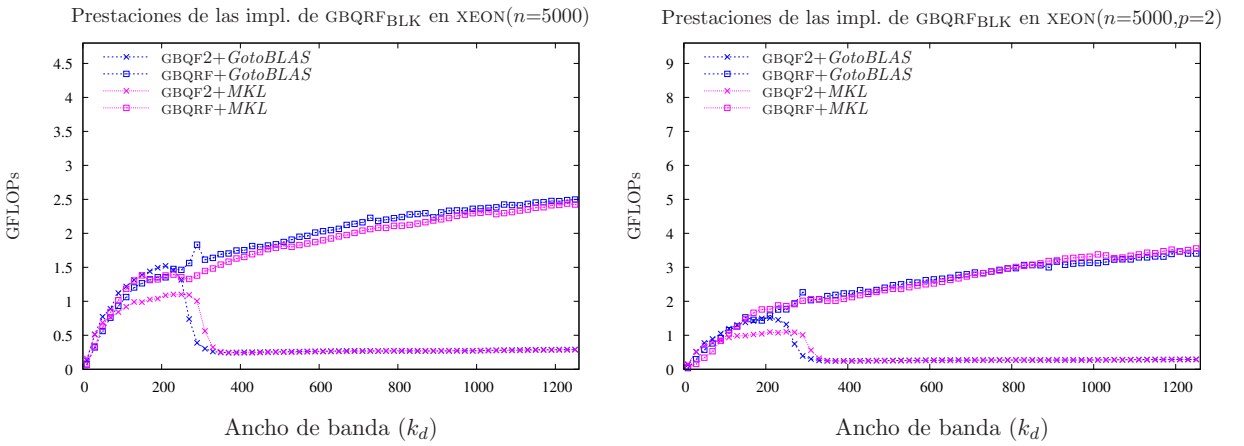


Figura 5.24: Comparativa de las diferentes implementaciones de GBQRF$_{\text{BLK}}$ en XEON.

5.2.7. Conclusiones

Se han presentado y evaluado dos nuevas rutinas para el cálculo de la factorización QR específicas para matrices banda. Aprovechar la estructura banda de la matriz reduce el coste computacional y espacial de la operación considerablemente. Las rutinas presentadas, GBQF2 y GBQRF, dejan la ejecución de la mayor parte de las operaciones aritméticas en manos de núcleos computacionales de *BLAS*. GBQF2 basa su funcionamiento en rutinas *BLAS*-2, mientras que GBQRF emplea rutinas *BLAS*-3.

En ambas arquitecturas, GBQF2 obtiene las mejores prestaciones al operar con matrices de banda estrecha, mientras que GBQRF es más rápida con matrices de banda media o ancha.

La biblioteca secuencial *MKL* es más eficiente que *GotoBLAS* en ITANIUM, mientras que en el caso paralelo la mejor implementación *BLAS* depende de la rutina a emplear y del ancho de banda de la matriz. Para la arquitectura XEON, *GotoBLAS* es más eficiente que *MKL* en el caso secuencial. Por contra, en el caso paralelo no hay diferencias relevantes entre ambas bibliotecas *BLAS*.

Capítulo 6

Reducción de modelos

En el primer capítulo de esta tesis se describió el interés que tiene disponer de rutinas optimizadas para la resolución de problemas de álgebra lineal con estructura banda: por un lado, este tipo de problemas se dan en algunas aplicaciones de forma natural, siendo un reflejo de la propia estructura del problema subyacente; por otro, una fuente importante de problemas banda aparece al reorganizar matrices con estructura dispersa mediante algoritmos de reetiquetado de grafos [37, 81].

En este nuevo capítulo se considera una de estas aplicaciones, la reducción de modelos aplicada a sistemas dinámicos lineales, con un doble propósito. En primer lugar, se utilizará una colección de problemas de esta naturaleza [48] como banco de prueba (*benchmark*) para hacer patente la necesidad de disponer de rutinas optimizadas para la resolución de sistemas lineales con estructura banda. En segundo lugar, los ejemplos de esta misma colección permitirán evaluar la utilidad práctica de las nuevas rutinas de resolución de sistemas lineales banda desarrolladas.

El capítulo está estructurado en 4 secciones. La primera introduce los conceptos básicos en la reducción de modelos, mientras que las dos siguientes describen técnicas empleadas para la reducción de modelos. Para concluir, la última sección presenta diferentes ejemplos de aplicación de la reducción de modelos extraídos de la colección Oberwolfach y los resultados experimentales obtenidos.

6.1. La técnica de reducción de modelos: conceptos básicos y aplicaciones

La formulación clásica de un *sistema dinámico lineal* (SDL) continuo e invariante en el tiempo mediante el modelo de espacio de estados [76] viene dada por un par de ecuaciones matriciales de la forma:

$$\begin{array}{rcl} E\dot{x}(t) & = & Ax(t) + Bu(t), \quad x(0) = x_0, \\ y(t) & = & Cx(t) + Du(t), \end{array} \tag{6.1}$$

siendo $x(t) \in \mathbb{R}^n$ el vector de variables de estado, con $x(0) = x_0$ el estado inicial del sistema, $u(t) \in \mathbb{R}^m$ el vector de entradas o controles, e $y(t) \in \mathbb{R}^p$ el vector de salidas. En el SDL en (6.1), $E, A \in \mathbb{R}^{n\times n}$ se conocen como el par de matrices de estados, mientras que $B \in \mathbb{R}^{n\times m}$ es la matriz de entradas y $C \in \mathbb{R}^{p\times n}$ es la matriz de salidas, y $D \in \mathbb{R}^{p\times m}$. El modelo de representación en el espacio de estados ha permitido la aplicación de numerosos resultados provenientes del álgebra lineal numérica [40] al ámbito de la teoría de control [53, 76]. Así, los nuevos métodos desarrollados para este modelo permiten abordar de modo numéricamente estable problemas de gran dimensión que serían difícilmente tratables usando otro tipo de modelos [57, 76].

Consideremos el SDL de orden n en (6.1) y la matriz de función de transferencia asociada definida por $G(s) = C(sE - A)^{-1}B + D$. Los métodos de reducción de modelos buscan obtener un segundo SDL:

$$\begin{array}{rcl} \dot{x}_r(t) & = & A_r x_r(t) + B_r u(t), \quad x_r(0) = \bar{x}_0, \\ y(t) & = & C_r x_r(t) + D_r u(t), \end{array} \tag{6.2}$$

de orden r mucho menor que n, y con función de transferencia $G_r(s) = C_r(sE_r - A_r)^{-1}B_r + D_r$, que mantenga la información fundamental sobre la dinámica del sistema original, es decir, que $G_r(s)$ presente un "comportamiento" similar a $G(s)$. A cambio de esta transformación, el sistema (6.2) resulta más manejable que (6.1), debido a su menor dimensión, y permite sustituirlo en posteriores análisis de la dinámica de la función de transferencia del sistema [6].

La mayor parte de los métodos de reducción de modelos existentes están enfocados hacia el caso denso a pesar de que, en general, las matrices de estados que aparecen ligadas a SDL de gran escala presentan una estructura dispersa. En modelos de simulaciones atmosféricas y oceánicas, reacciones químicas y de dinámica molecular, análisis de vibraciones acústicas, simulación de circuitos electrónicos, diseño de chips VLSI, etc., los sistemas dispersos pueden tener cientos de miles e incluso millones de variables de estado [6, 7, 8]. La dimensión de estos problemas hace imprescindible pues, no sólo aprovechar la estructura dispersa de la matriz de estados del SDL, sino también aplicar técnicas de computación paralela.

Los métodos de reducción de modelos más efectivos se pueden agrupar en dos grandes clases: métodos de aproximación SVD [60, 72] y métodos de aproximación por subespacios de Krylov [38, 45, 46, 52, 80]. Ambas clases permiten explotar la estructura dispersa del problema, si bien difieren en la precisión y las propiedades del modelo reducido que producen [6].

Entre los métodos de reducción de modelos basados en aproximación SVD destacan los algoritmos de truncamiento equilibrado (*balanced truncation*) [66, 82, 91, 95], los algoritmos de aproximación de perturbaciones singulares (*singular perturbation approximation*) [63] y los algoritmos de aproximación de la norma de Hankel (*Hankel-norm approximation*) [39]. Todos estos métodos presentan, como problema computacional principal, la resolución de un par de ecuaciones (matriciales) de Lyapunov que tienen por coeficiente las matrices de estados del sistema. Una de las aproximaciones más eficientes para resolver este tipo de ecuaciones, cuando las matrices coeficientes son dispersas y de gran dimensión, es el método ADI (*Alternating Direction Implicit Iteration*), reintroducido en [73]. Básicamente, se trata de un método iterativo que precisa, en cada etapa, de la resolución de una serie de sistemas de ecuaciones lineales, con matrices de coeficientes dispersas, que pueden abordarse mediante métodos directos [37] o métodos iterativos (Jacobi, SOR, multigrid, etc.) [81].

Los métodos de aproximación por subespacios de Krylov básicamente requieren el cálculo de algún tipo de factorización de la matriz de alcanzabilidad del SDL, definida por:

$$R_k(A, B) = \left[B,\ E^{-1}AE^{-1}B,\ (E^{-1}A)^2E^{-1}B,\ \ldots, (E^{-1}A)^{k-1}E^{-1}B\right].$$

Esta factorización se obtiene, habitualmente, mediante el algoritmo iterativo de Arnoldi o alguna de sus variantes [40, 81] que presentan, como operación fundamental, el producto matriz por vector, permitiendo de este modo aprovechar la estructura dispersa de las matrices del problema y preservando asimismo esta estructura.

Frente a su mayor coste computacional, los métodos de aproximación SVD presentan como ventaja principal la preservación de propiedades del sistema original como la estabilidad o la pasividad [6]. Además, este tipo de métodos también proporciona una medida del error cometido al utilizar el modelo reducido en sustitución del original.

En este capítulo nos centramos en la aplicación de los resultados elaborados en la tesis para la resolución de sistemas de ecuaciones lineales con estructura banda en la matriz de coeficientes a los métodos de aproximación SVD. Sin embargo, algunos resultados de la tesis referidos al cálculo de operaciones básicas del álgebra lineal, como el producto matriz banda por vector, son también de aplicación en los métodos de aproximación por subespacios de Krylov.

6.2. Aproximación SVD: truncamiento equilibrado

Los algoritmos de truncamiento equilibrado (TE) [66] pertenecen a la familia de métodos de error absoluto que, a su vez, están dentro de la clase de métodos de aproximación SVD [6, 70]. Los métodos de error absoluto buscan minimizar

$$\|\Delta_a\|_\infty := \|G - G_r\|_\infty,$$

donde $\|\cdot\|_\infty$ denota la norma $\mathcal{L}_\infty$ o $\mathcal{H}_\infty$ de una función matricial racional estable que, para una función de transferencia propia F, se define como

$$\|F\|_\infty := \sup_{\omega\in\mathbb{R}} \sigma_{\text{máx}}(F(\jmath\omega)),$$

con $\jmath := \sqrt{-1}$ y $\sigma_{\text{máx}}(M)$ el mayor valor singular de la matriz M.

Los algoritmos TE están fuertemente ligados a los *Gramian de controlabilidad y observabilidad* del sistema, W_c y W_o, respectivamente. Estos Gramian vienen dados por las soluciones de las ecuaciones de Lyapunov duales:

$$AW_cE^T + EW_cA^T + BB^T = 0, \tag{6.3}$$
$$A^T\widehat{W}_oE + E^T\widehat{W}_oA + C^TC = 0, \tag{6.4}$$

y $W_o = E^T\widehat{W}_oE$. Bajo ciertas condiciones habituales en problemas de reducción de modelos, W_c y W_o son matrices semidefinidas positivas y, en consecuencia, existen las factorizaciones $W_c = S^TS$ y $W_o = R^TR$. Las matrices S y R se conocen como los *factores de Cholesky* de los Gramian (aunque no son factores de Cholesky en el sentido estricto).

Considérese a continuación la descomposición en valores singulares [40] del producto

$$SR^T = U\Sigma V^T = [U_L\ U_R]\begin{bmatrix}\Sigma_L & 0\\ 0 & \Sigma_R\end{bmatrix}\begin{bmatrix}V_L^T\\ V_R^T\end{bmatrix}, \tag{6.5}$$

donde las matrices Σ, U, y V se particionan de manera coherente en una dimensión dada r tal que $\Sigma_L = \text{diag}\,(\sigma_1,\ldots,\sigma_r)$, $\Sigma_R = \text{diag}\,(\sigma_{r+1},\ldots,\sigma_n)$, y $\sigma_r > \sigma_{r+1}$. Aquí, $\sigma_1,\ldots,\sigma_n$ son los valores singulares de *Hankel* (VSH) del sistema y contienen información relevante sobre la dinámica del sistema pues miden el "peso" que tiene un estado sobre la transferencia de energía de una entrada a una salida determinadas.

El algoritmo TE *square-root* (SR) usa el producto de la SVD en (6.5) para obtener las matrices del modelo de orden reducido como

$$E_r := T_LET_R, \quad A_r := T_LAT_R, \quad B_r := T_LB, \quad C_r := CT_R, \quad D_r := D, \tag{6.6}$$

donde las matrices de truncamiento (o proyección) $T_L \in \mathbb{R}^{r\times n}$ y $T_R \in \mathbb{R}^{n\times r}$ están definidas por

$$T_L := \Sigma_L^{-1/2}V_L^TRE^{-1} \quad \text{y} \quad T_R := S^TU_L\Sigma_L^{-1/2}. \tag{6.7}$$

Con esta formulación $E_r = I_r$, la matriz identidad de orden r, y por tanto no es necesario su cálculo.

El algoritmo TE-SR proporciona una realización G_r que satisface el límite del error teórico

$$\|\Delta_a\|_\infty = \|G - G_r\|_\infty \leq 2 \sum_{j=r+1}^{n} \sigma_j, \tag{6.8}$$

permitiendo una elección adaptativa de la dimensión del modelo reducido r una vez se conocen los VSH. Más detalles sobre la aplicación de los algoritmos TE a SDL, en particular cuando E es singular, pueden consultarse en [86].

6.3. Resolución de ecuaciones de Lyapunov dispersas

En esta sección se reformulan los métodos de resolución de ecuaciones de Lyapunov introducidos en [61, 72] para el caso generalizado en (6.3)–(6.4). El paquete de *software* LYAPACK[1] [74] proporciona funciones MATLAB para resolver las ecuaciones de Lyapunov (6.3) y (6.4) usando una transformación implícita basada en una factorización de Cholesky/LU dispersa de E. En este trabajo seguimos una aproximación expuesta en [11, 59] que evita esta factorización. Otra variación de esta idea puede consultarse en [87].

Los métodos considerados a continuación se benefician de que (6.3) y (6.4) comparten las matrices A y E como coeficientes de las ecuaciones de Lyapunov. Parte de su eficiencia es debida también a que los términos constantes en estas ecuaciones, BB^T y C^TC, están formados por productos de rango k, con $k \in \{m, p\}$ en general mucho más pequeño que n. Los métodos iterativos propuestos explotan el bajo rango numérico de los términos constantes en (6.3) y (6.4) para calcular una aproximación de rango reducido de los factores de Cholesky de la solución. Estas aproximaciones pueden sustituir fiablemente a S y R en el cálculo de la descomposición (6.5) y la posterior obtención de las matrices del modelo reducido (6.7); consultar [13].

6.3.1. Gramian de controlabilidad

La relación entre el SDL (generalizado) en (6.1) y una versión estándar de éste ($E = I_n$) indica que el Gramian de controlabilidad está definido por la solución de la ecuación de Lyapunov

$$A_s W_c + W_c A_s^T + B_s B_s^T = 0,$$

con $A_s = E^{-1}A$ y $B_s = E^{-1}B$. Dado un conjunto de *parámetros de desplazamiento* $\tau = \{\tau_1, \tau_2, \ldots\}$ con parte real negativa tal que $\tau_j = \tau_{j+t_s}$, $j = 1, 2, \ldots$ (esto es, el conjunto es cíclico con periodicidad t_s), la iteración LR-ADI [61, 72] calcula un par de secuencias de matrices, $\{U_j\}_{j=1}^{\infty}$ y $\{Y_j\}_{j=1}^{\infty}$, como sigue. Inicialmente,

$$\begin{aligned} U_1 &:= \gamma_1 \left(A_s + \tau_1 I_n\right)^{-1} B_s, \\ Y_1 &:= U_1, \end{aligned}$$

donde $\gamma_1 = \sqrt{-2\,\mathrm{Re}(\tau_1)}$ y $\mathrm{Re}(\tau_k)$ representa la parte real de τ_k. A partir de entonces ($j > 1$),

$$\begin{aligned} U_j &:= \gamma_j \left(I_n - (\tau_j + \overline{\tau_{j-1}})(A_s + \tau_j I_n)^{-1}\right) U_{j-1}, \\ Y_j &:= \left[Y_{j-1},\ U_j\right]. \end{aligned}$$

[1] Disponible en `http://www.slicot.org`.

Aquí, $\overline{\tau_j}$ denota la conjugada de τ_j, y $\gamma_j = \sqrt{\frac{\mathrm{Re}(\tau_j)}{\mathrm{Re}(\tau_{j-1})}}$, $j > 1$. Tras la convergencia, después de l_c iteraciones, el procedimiento proporciona una matriz de aproximación $Y_{l_c} \in \mathbb{R}^{n\times(m\cdot l_c)}$ tal que $Y_{l_c} Y_{l_c}^T \approx S^T S = W_c$.

Reescribamos ahora la iteración LR-ADI para evitar referencias explícitas a A_s o B_s. Así, para la primera iteración,

$$\begin{aligned} U_1 &:= \gamma_1 \left(A_s + \tau_1 I_n\right)^{-1} B_s = \gamma_1 \left(E^{-1}A + \tau_1 I_n\right)^{-1} E^{-1} B \\ &= \gamma_1 \left(E^{-1}(A + \tau_1 E)\right)^{-1} E^{-1} B = \gamma_1 \left(A + \tau_1 E\right)^{-1} E E^{-1} B \\ &= \gamma_1 \left(A + \tau_1 E\right)^{-1} B, \end{aligned}$$

mientras que, para las restantes iteraciones ($j > 1$),

$$\begin{aligned} U_j &:= \gamma_j \left(I_n - (\tau_j + \overline{\tau_{j-1}})(A_s + \tau_j I_n)^{-1}\right) U_{j-1} \\ &= \gamma_j \left(I_n - (\tau_j + \overline{\tau_{j-1}})(E^{-1}A + \tau_j I_n)^{-1}\right) U_{j-1} \\ &= \gamma_j \left(I_n - (\tau_j + \overline{\tau_{j-1}})(E^{-1}(A + \tau_j E)^{-1}\right) U_{j-1} \\ &= \gamma_j \left(I_n - (\tau_j + \overline{\tau_{j-1}})(A + \tau_j E)^{-1} E\right) U_{j-1} \\ &= \gamma_j \left(U_{j-1} - (\tau_j + \overline{\tau_{j-1}})(A + \tau_j E)^{-1} E U_{j-1}\right). \end{aligned}$$

6.3.2. Gramian de observabilidad

La situación es ligeramente diferente para el Gramian de observabilidad. En lugar de trabajar con (6.4), en este caso usamos

$$A_s^T W_o + W_o A_s + C^T C = 0.$$

Emulando la idea anterior, la iteración LR-ADI puede ser reformulada para producir la secuencia $\{V_j\}_{j=1}^{\infty}$ como

$$V_1 := \gamma_1 \left(A_s^T + \overline{\tau_1} I_n\right)^{-1} C^T = \gamma_1 \, E^T (A + \overline{\tau_1} E)^{-T} C^T,$$

y, para $j > 1$,

$$\begin{aligned} V_j &:= \gamma_j \left(I_n - (\tau_j + \overline{\tau_{j-1}})(A_s^T + \overline{\tau_j} I_n)^{-1}\right) V_{j-1} \\ &= \gamma_j \left(I_n - (\tau_j + \overline{\tau_{j-1}})((E^{-1}A)^T + \overline{\tau_j} I_n)^{-1}\right) V_{j-1} \\ &= \gamma_j \left(V_{j-1} - (\tau_j + \overline{\tau_{j-1}}) E^T (A + \overline{\tau_j} E)^{-T} V_{j-1}\right). \end{aligned}$$

Así pues, durante la iteración podemos construir la secuencia $\{Z_j\}_{j=1}^{\infty}$ haciendo uso de

$$\begin{aligned} Z_1 &:= V_1, \\ Z_j &:= [Z_{j-1},\ V_j], \quad j > 1, \end{aligned}$$

de modo que, tras converger después de l_o iteraciones, se obtiene una matriz $Z_{l_o} \in \mathbb{R}^{n\times(p\cdot l_o)}$ tal que $Z_{l_o} Z_{l_o}^T \approx R^T R = W_o$. Cabe destacar aquí que no se está calculando una solución de (6.4) sino que, en su lugar, se obtiene R directamente sin pasar por la resolución de (6.4) para el factor de $\widehat{W}_o$. Operando sobre este último, y teniendo en cuenta que $\widehat{R}E = R$, en (6.5) sería necesario calcular la descomposición SVD de $SE^T\widehat{R}^T$. Por otro lado, en (6.7) sería posible utilizar $T_L := \Sigma_L^{-1/2} V_L^T \widehat{R}$ evitándose de este modo la inversión de E. Entre estas dos opciones, aquí se escoge el uso de R en lugar de $\widehat{R}$ pues con ello se retrasa tanto como es posible la aplicación de E^{-1} y, en consecuencia, la introducción de errores de redondeo debido a un mal condicionamiento de esta matriz. Además,

aunque en ambas opciones es necesario calcular una factorización de Cholesky o LU de E, si se calcula $\widehat{R}$ sería necesario utilizar esta factorización para resolver $p \cdot l_o$ sistemas de ecuaciones lineales mientras que en (6.7) sólo es necesario resolver $r \leq p \cdot l_o$. Como habitualmente $r \ll p \cdot l_o$, esto puede redundar en un ahorro significativo del tiempo de cálculo.

6.3.3. Solución de sistemas lineales en la iteración LR-ADI

Consideremos a continuación que $l_c, l_o > t_s$ (es decir, el número de iteraciones que se requieren para alcanzar la convergencia en ambas iteraciones es mayor que la cantidad de parámetros de desplazamiento, un caso habitual). Entonces, supuesto que existe la suficiente capacidad de almacenamiento, es posible lograr un gran ahorro computacional usando métodos directos para resolver los sistemas de ecuaciones lineales que aparecen en las iteraciones anteriores. Es más, únicamente es necesario factorizar las matrices $(A + \tau_j E)$ y $(A^T + \overline{\tau_j} E^T)$ una sóla vez, y los mismos factores pueden usarse en las iteraciones $j + kt_s$, $k = 0, 1, \ldots$. Si A y E son matrices simétricas, claramente sólo es necesaria una factorización. Lo mismo sucede en el caso no simétrico donde, por ejemplo, la factorización LU (con pivotamiento de filas) $(A + \tau_j E) = PLU$ ofrece también una factorización de $(A^T + \overline{\tau_j} E^T)$ como sigue:

$$A^T + \overline{\tau_j} E^T = \overline{(A + \tau_j E)^T} = \overline{U}^T \overline{L}^T P.$$

Tengamos en cuenta que los parámetros de desplazamiento se escogen siempre para que su conjunto sea cerrado bajo una transformación de conjugación. Sea $\tau_i = \overline{\tau_j}$; entonces las iteraciones j e i involucran sistemas de ecuaciones lineales con matrices de coeficientes que son las complejas conjugadas unas de otras. Así pues, únicamente es necesario calcular una de estas dos iteraciones. En resumen, si $\{\tau_1, \ldots, \tau_{t_s}\}$ está compuesto de t_s^r parámetros de desplazamiento reales y de t_s^c complejos, entonces es necesario calcular (y almacenar) t_s^r factorizaciones con aritmética real y $t_s^c/2$ con aritmética compleja.

La discusión anterior justifica la resolución de sistemas de ecuaciones lineales por métodos directos frente a los métodos iterativos [3]. Las iteraciones LR-ADI pueden utilizarse fácilmente en caso de que los pares de matrices de estado (A, E) presenten una estructura banda, utilizando en este caso los resultados de los capítulos anteriores de esta tesis.

6.4. Resolución de sistemas de ecuaciones lineales banda en reducción de modelos

La colección Oberwolfach[2] incluye un repertorio de aplicaciones en las que aparecen problemas de reducción de modelos. De un total de 15 ejemplos, se han seleccionado 7, que son utilizados en esta sección para ilustrar la importancia que tiene disponer de rutinas optimizadas para resolver sistemas de ecuaciones lineales y calcular productos de matrices con estructura banda. Los 8 ejemplos restantes de la colección se han descartado porque corresponden a modelos no lineales (y por tanto, no pueden ser resultos por los métodos descritos en las secciones anteriores), no presentan una estructura banda sencilla o requerían más memoria de la presente en las arquitecturas empleadas.

A continuación se ofrece una muy breve descripción de los ejemplos a fin de ilustrar las diferentes fuentes de aplicaciones de reducción de modelos:

B1. Micropyros thruster. Este modelo corresponde a un microimpulsor matricial que integra una pastilla sólida de combustible con un microchip [62]. El problema de diseño consiste en alcanzar la temperatura crítica para el combustible sin dañar al microchip.

[2]Disponible en `http://www.imtek.de/simulation/benchmark/`.

Example	n	m	p	nnz(A)	nnz(E)	$k_d/k_u;\ k_l$
B1.T2DAL	4,257	1	7	37,465	4,257	86
B1.T2DAH	11,445	1	7	93,781	93,781	231
B1.T3DL	20,360	1	7	265,113	20,360	1,350
B2.THERMAL	4,257	1	7	37,465	4,257	86
B3.INLET	11,730	1	2	328,323	95,396	220
B4.FLOW_METER_V0	9,669	1	5	67,391	9,669	296
B4.CHIP_COOLING_V0	20,082	1	5	281,150	20,082	1,226
B5.FILTER2D	1,668	1	5	10,750	1,668	71
B6.RAIL_20209	20,209	7	6	139,233	139,473	276
B6.RAIL_79841	79,841	7	6	553,921	554,913	550;550
B7.WINDSCREEN	22,692	1	1	1,482,390	1,481,988	758

Tabla 6.1: Ejemplos/casos de la colección Oberwolfach de reducción de modelos.

B2. Boundary condition independent thermal model. El problema que modela este ejemplo es el intercambio de calor entre las diferentes capas que pueden componer un dispositivo como un chip.

B3. Active control of a supersonic engine inlet. El objetivo de este estudio es diseñar un control que regule la entrada de flujo al motor de una aeronave.

B4. Convective thermal flow problems. Este diseño caracteriza un modelo 2-D o 3-D de un chip refrigerado por convección y se usa en la simulación del intercambio de calor entre un cuerpo sólido (el chip) y un fluido.

B5. Tunable optical filter. El objetivo de este proyecto es el diseño de un filtro óptico, en forma de una membrana, regulable mediante la temperatura.

B6. A semi-discretized heat transfer problem for optimal cooling. Este modelo surge en un método de producción de raíles de acero [14, 92]. El objetivo es diseñar un dispositivo de control que fuerce gradientes moderados de temperatura cuando el raíl se enfría. El modelo matemático corresponde a una ecuación de calor en 2-D.

B7. Structural model of a car windscreen. El objetivo de este modelo es evaluar el comportamiento de la estructura de un parabrisas de automóvil sometido a presión.

La tabla 6.1 ofrece una caracterización cualitativa de los ejemplos anteriores. Para cada ejemplo se indica el número de estados, entradas y salidas (n, m y p respectivamente) y el número de elementos no nulos de las matrices de estado, nnz(A) y nnz(E). La dimensión de los ejemplos es grande, desde 1,668 hasta 79,841 estados en el mayor de los casos, mientras que el número de entradas y salidas es mucho más reducido. La estructura dispersa de las matrices de estados también es clara; por ejemplo, para el ejemplo T2DAL, la matriz A presenta un porcentaje de dispersión del 0.207 % (tan sólo el 0.2 % de sus entradas son no nulas) mientras que para E este porcentaje se reduce al 0.023 %.

En algunos de los casos seleccionados, las matrices A y E presentan una estructura banda de forma natural mientras que en otros ha sido necesario reordenar sus entradas no nulas, aplicando el método *reverse Cuthill-McKee* [28] para obtener una forma banda. La última columna de la tabla,

Operación	Caso simétrico	Caso general
Producto de matrices	SBMM	GBMM
Resolución de sistema triangular banda	TBSM	TBSM
Factorización	PBTRF	GBTRF GBTRF_NP

Tabla 6.2: Rutinas empleadas durante la resolución de sistemas de ecuaciones lineales.

etiquetada como "k_d/k_u; k_l" indica el ancho de banda de la matriz $A + \tau E$ que interviene en las operaciones de resolución de sistemas de ecuaciones lineales y productos matriz por matriz que aparecen en la iteración LR-ADI. El resultado es que todos los ejemplos de la colección presentan características que los hacen atractivos para su resolución mediante las rutinas desarrolladas en este trabajo, pues presentan una estructura banda con un ancho que varía desde 71 hasta 1,350 en el mayor de los casos.

6.4.1. Resultados experimentales para los diferentes ejemplos extraídos de la colección oberwolfach

Para la resolución de los ejemplos de la colección Oberwolfach, se han aplicado las nuevas rutinas presentadas en capítulos anteriores, comparándolas con las incluidas en las bibliotecas *LAPACK*, *GotoBLAS* y *MKL*. En función de si se opera con una matriz simétrica o general, las rutinas utilizar varían, tal y como se muestra en la tabla 6.2. En este sentido, todas los ejemplos evaluados, excepto B6.RAIL_79841, presentan una estructura simétrica. En este caso particular, además, no ha sido posible obtener resultados en la arquitectura XEON, puesto que la cantidad de memoria de este sistema es insuficiente para almacenar las matrices involucradas en las operaciones en estudio.

Las figuras 6.1–6.7 recogen los tiempos de ejecución (en segundos) para cada operación, con las rutinas de las bibliotecas *LAPACK*, *GotoBLAS* y *MKL*, y las nuevas implementaciones basadas en el enfoque MERGE. Aunque en algunas de las operaciones MERGE no es la opción más eficiente, por simplicidad se ha seleccionado esta implementación en todos los casos. A la izquierda de todas las gráficas se muestran los tiempos secuenciales mientras que a la derecha figuran los tiempos resultantes de la ejecución paralela usando 2 y 4 procesadores en las arquitecturas XEON e ITANIUM, respectivamente.

La figura 6.1 muestra los resultados para los diferentes casos (T2DAL, T2DAH y T3DL) del ejemplo B1. Los resultados difieren entre ambas arquitecturas para el primero de los ejemplos, B1.T2DAL. Mientras que en XEON se obtienen beneficios visibles con las nuevas implementaciones sobre las tres operaciones estudiadas, en ITANIUM los tiempos sólo se reducen para las operaciones TBSM (en un 4 %) y PBTRF (en un 34 %). La carga computacional de las tres operaciones con el caso B1.T2DAL es tan reducida que el tiempo al ejecutarse en paralelo es superior al secuencial en ambas arquitecturas. Existe una fuerte semejanza entre los casos B1.T2DAL y B2.THERMAL, que hace que los resultados comentados para el primero se repitan para el segundo (ver figura 6.2).

Los resultados de B1.T2DAH y B1.T3DL muestran mejoras importantes en todas las operaciones; por ejemplo, para SBMM y TBSM, la reducción de tiempo en el caso secuencial está siempre por encima del 16 % llegando a ser de hasta un 80 %. Al igual que sucede con B1.T2DAL, tanto *GotoBLAS* como *MKL* obtienen tiempos similares al ejecutarse en secuencial y en paralelo, lo que hace suponer que en realidad ambas implementaciones supuestamente paralelas utilizan internamente una única hebra para su ejecución.

La figura 6.3 resume los tiempos para el caso B3.INLET y en ellos de nuevo se observa que los

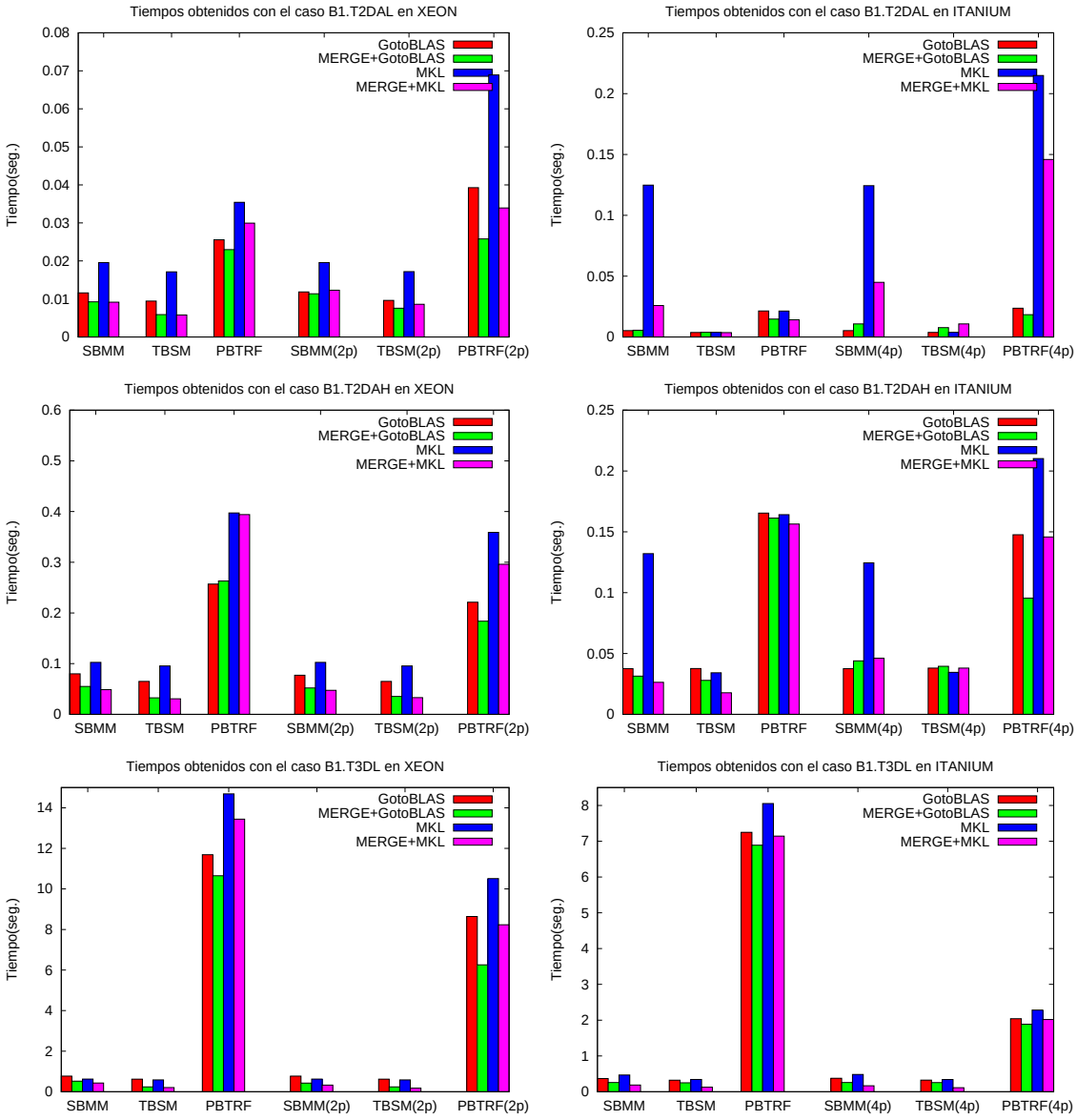


Figura 6.1: Resultados para las distintos casos del ejemplo B1.

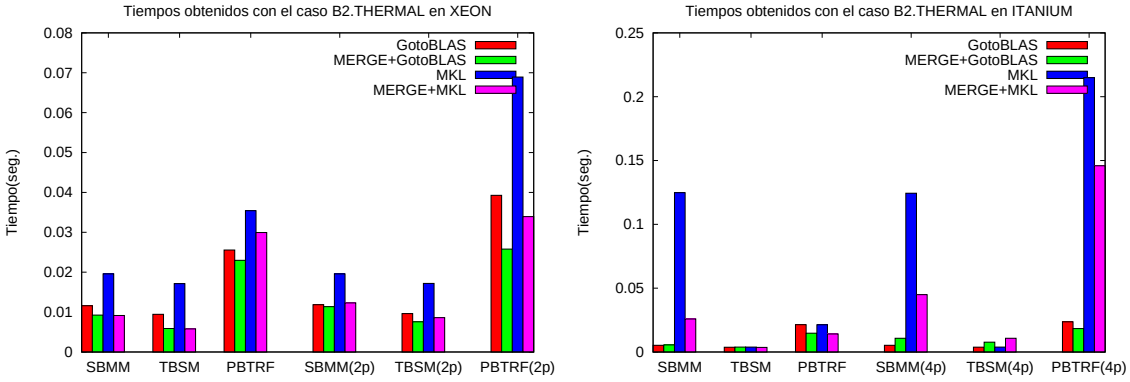


Figura 6.2: Resultados para el ejemplo B2.

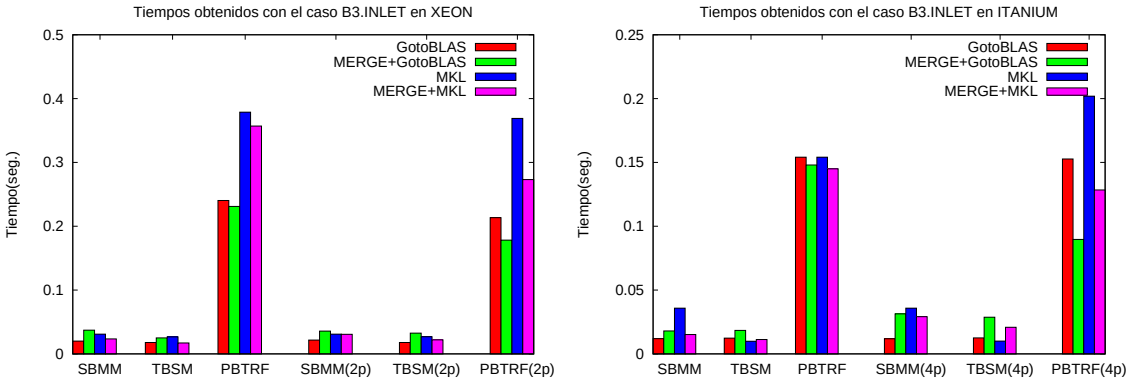


Figura 6.3: Resultados para el ejemplo B3.

resultados de las operaciones menos costosas (SBMM y TBSM), con una o varias hebras de ejecución, son similares para las rutinas de *GotoBLAS* y *MKL*, lo que lleva a pensar que estas bibliotecas usan una sóla hebra cuando la operación tiene escasa carga computacional. El hecho de que los tiempos de ejecución de la nueva implementación al utilizar varias hebras superen a los que se obtienen al usar sólo una, demuestra que ésta es una decisión acertada. En este caso, las dimensiones de la matriz y su ancho de banda hacen que la carga computacional sea reducida por lo que las mejoras obtenidas por las nuevas rutinas de resolución del sistema triangular banda (TBSM) y producto de matrices banda (SBMM) frente *MKL* resultan menores, mientras que no consiguen superar el rendimiento de las rutinas de *GotoBLAS*. Además, en este mismo caso sólo aparecen dos vectores de términos independientes en la resolución del sistema triangular banda mediante la rutina TBSM, hecho que perjudica a la rutina *BLAS*-3 propuesta y, en cambio, beneficia a las rutinas TBSV incluidas en *GotoBLAS* y *MKL*. Para la factorización de Cholesky, que conlleva un coste computacional mayor, la nueva rutina es la mejor opción, especialmente en el caso paralelo. Con ella se alcanzan mejoras sobre *MKL* del 25 % en XEONy del 36 % en la arquitectura ITANIUM.

En conclusión, en este caso la eficiencia de las rutinas de *GotoBLAS* y *MKL* es mayor para las operaciones SBMM y TBSM. No obstante, si estudiamos los tiempos de ejecución en concreto, la nueva rutina de factorización reduce el tiempo de este cálculo en la arquitectura ITANIUM en 0.063 segundos. En cambio, las rutinas más eficientes de SBMM(*GotoBLAS*) y TBSM(*MKL*) son, respectivamente, 0.003 y 0.001 segundos más rápidas que las nuevas rutinas MERGE. Algo similar sucede en la arquitectura XEON.

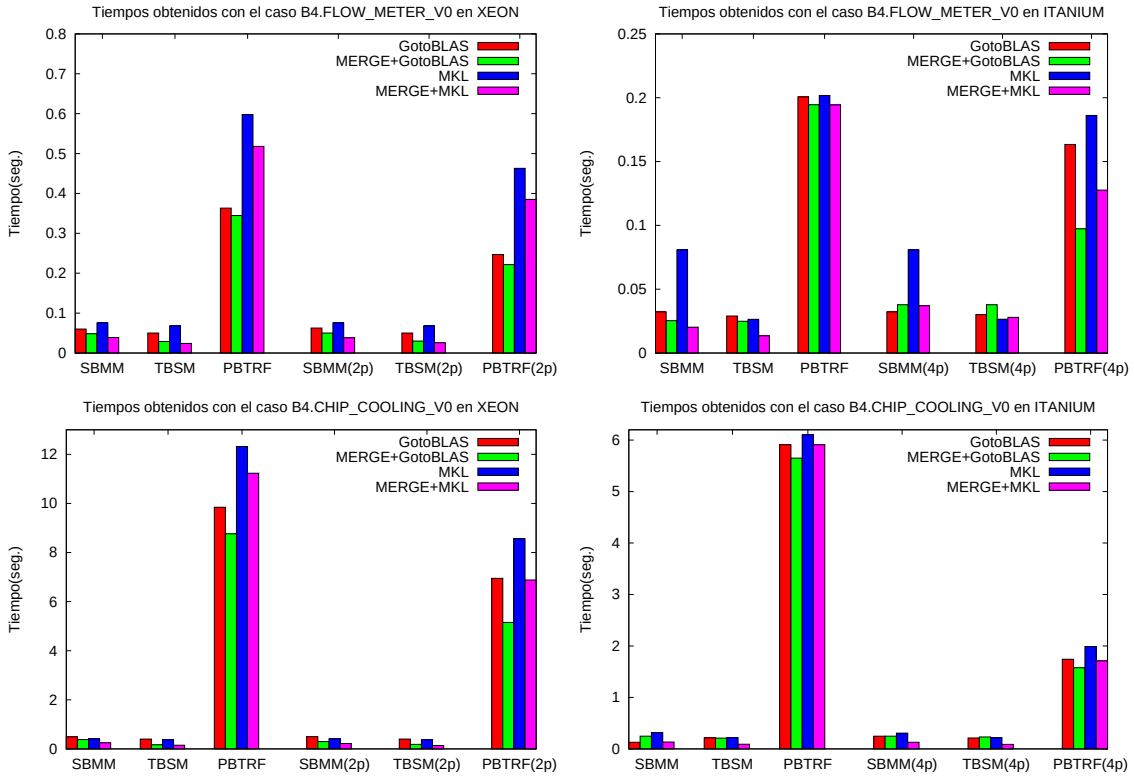


Figura 6.4: Resultados para los distintos casos del ejemplo B4.

La figura 6.4 muestra los resultados para los casos B4.FLOW_METER_V0 y B4.CHIP_COOLING_V0. Para el primero de éstos, podemos observar que los resultados en XEON son muy satisfactorios, ya

que las nuevas rutinas mejoran siempre a las de las bibliotecas *LAPACK*, *GotoBLAS* y *MKL*. Así, por ejemplo, se produce una reducción de un 20, 40 y 10 % en los tiempos de las rutinas paralelas SBMM, TBSM y PBTRF de la biblioteca *GotoBLAS*, respectivamente, mientras que los porcentajes de mejora sobre *MKL* son incluso mayores.

En la arquitectura ITANIUM, la reducción del tiempo de ejecución en todas las operaciones se sitúa en torno al 40 %. En esta arquitectura, las mayores prestaciones en el producto de matrices se alcanzan con *GotoBLAS* secuencial, mientras que con las nuevas rutinas la mejor opción es el uso de *MKL* paralelo, que es un 45 % más eficiente. Un análisis similar con TBSM y PBTRF muestra que las nuevas rutinas son un 63 % y un 25 % más rápidas respectivamente que las mejores alternativas presentadas por *GotoBLAS* y *MKL*.

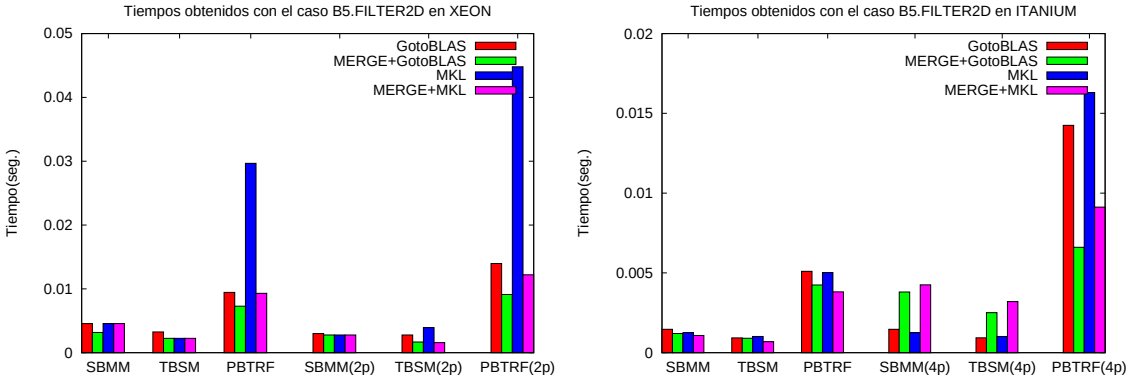


Figura 6.5: Resultados para el ejemplo B5.

El caso B5.FILTER2D presenta las matrices de estados más pequeñas de la colección y sus resultados se ilustran en la figura 6.5. Su dimensión y ancho de banda hace difícil obtener beneficios del uso de rutinas paralelas; de hecho, en la mayoría de los casos la versión secuencial es la más eficiente. Así, en ITANIUM los mejores resultados para todas las operaciones se obtienen con una única hebra de ejecución. En esta arquitectura las nuevas rutinas en su implementación secuencial mejoran en todos los casos a las rutinas de las bibliotecas *LAPACK*, *GotoBLAS* y *MKL*; por ejemplo, resultan entre un 14 y un 25 % más rápidas que las rutinas de *MKL*. En la arquitectura XEON sí es rentable el uso de versiones paralelas para el producto de matrices. También en esta arquitectura las nuevas rutinas son las más eficientes para las operaciones TBSM y PBTRF, mientras que las prestaciones quedan a la par para la operación SBMM.

Los tiempos generados con la matriz B6.RAIL_20209, en la figura 6.6, muestran de nuevo la superioridad de las nuevas rutinas, siendo entre un 12 y un 50 % más eficientes en XEON y entre un 28 y un 44 % en ITANIUM. Las mayores ganancias se obtienen en la resolución del sistema triangular banda, donde la presencia de 6 vectores de términos independientes favorece al uso de la nueva rutina *BLAS*-3.

El caso B6.RAIL_79841 presenta una estructura general (no simétrica), evaluándose pues cuatro operaciones: GBMM, TBSM y las factorizaciones LU con y sin pivotamiento. Por limitaciones de memoria no han podido obtenerse resultados para la arquitectura XEON. Los resultados para ITANIUM se recogen en la figura 6.6. Salvo en la resolución del sistema triangular banda con *GotoBLAS* y más de una hebra, en todas las combinaciones restantes las nuevas rutinas son más rápidas que las correspondientes rutinas de las bibliotecas *GotoBLAS* y *MKL*. En las operaciones GBMM y TBSM, las nuevas rutinas son un 54 % más rápidas que la más eficiente de las rutinas de las bibliotecas. Para la factorización LU, la nueva rutina únicamente reduce el tiempo de ejecución en 0.12 segundos.

No obstante, si empleamos la rutina que calcula la factorización LU sin pivotamiento, el tiempo se recorta en casi 3.8 segundos (un 54 %).

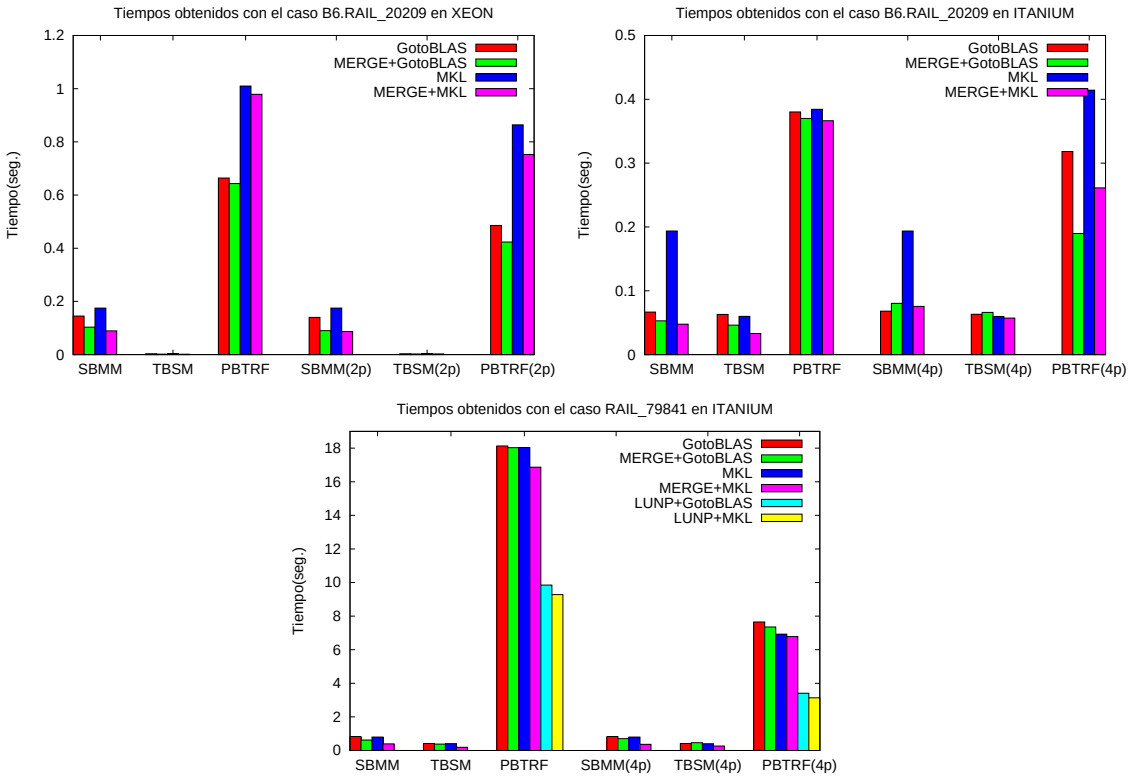


Figura 6.6: Resultados para los distintos casos del ejemplo B6.

La figura 6.7 ilustra las prestaciones de las rutinas al operar con el caso B7.WINDSCREEN. Los resultados obtenidos para las operaciones SBMM y TBSM son poco favorables, ya que las rutinas MERGE ofrecen prestaciones muy inferiores en ambas arquitecturas. Sin embargo, la nueva rutina para la factorización de Cholesky mejora ampliamente las prestaciones de la rutina *LAPACK*, por lo que en global el uso de las nuevas implementaciones puede considerarse como positivo. Así por ejemplo, en la arquitectura XEON, las rutinas MERGE para las operaciones SBMM y TBSM son 0.015 y 0.004 segundos más lentas respectivamente, mientras que la nueva rutina para la factorización de Cholesky es 0.768 segundos más rápida. Algo similar ocurre con ITANIUM.

En la figura 6.8 se resumen las diferencias entre los tiempos obtenidos por la mejor implementación incluida en las rutinas *LAPACK*, *GotoBLAS* o *MKL* para cada operación sobre la arquitectura XEON. Únicamente en dos ocasiones la rutina MERGE propuesta para SBMM es más lenta que alguna de las incluidas en las implementaciones de *BLAS*, y en una sóla ocasión la nueva rutina para TBSM se revela menos eficiente. Para la arquitectura ITANIUM, de los diez casos de estudio simétricos, en cuatro de ellos la nueva rutina para la operación SBMM es más lenta que alguna de las implementaciones de *BLAS*, y lo mismo sucede en dos ocasiones con la nueva rutina *BLAS*-3 para TBSM. No obstante, la tabla 6.3 también indica que esta pérdida de eficiencia es compensada en el resto de operaciones en todos los casos.

De los resultados expuestos podemos extraer las siguientes conclusiones:

- Las nuevas rutinas mejoran en la mayoría de los casos los resultados obtenidos por las implementaciones de *LAPACK*, *GotoBLAS* y *MKL*, especialmente en las operaciones más costosas

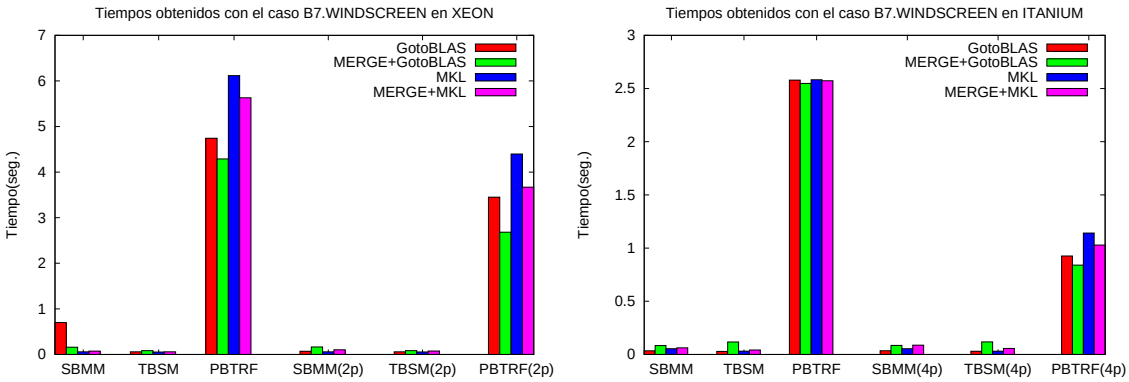


Figura 6.7: Resultados para el ejemplo B7.

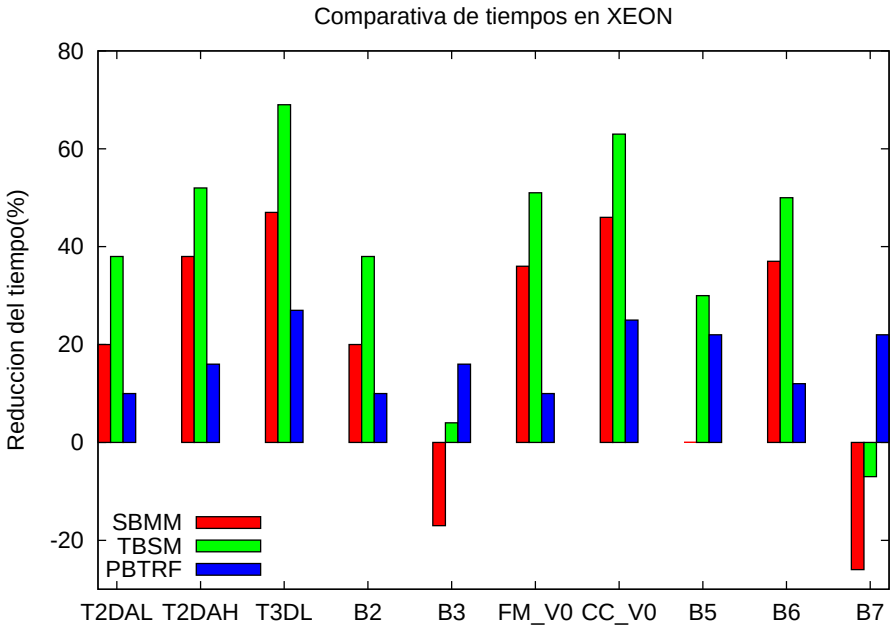


Figura 6.8: Comparativa entre las nuevas implementaciones y las de las bibliotecas *LAPACK*, *GotoBLAS* y *MKL* en XEON.

	XEON		ITANIUM			
Operación	B3.INLET	B7.WINDSCREEN	B1.T2DAL	B2.THERMAL	B3.INLET	B7.WINDSCREEN
SBMM	-4,00E-03	-1,50E-02	-2,90E-04	-2,90E-04	-3,00E-03	-2,70E-02
TBSM	1,00E-03	-4,00E-03	1,50E-04	1,50E-04	-1,00E-03	-1,30E-02
PBTRF	3,50E-02	7,68E-01	7,25E-03	7,25E-03	6,30E-02	8,60E-02

Tabla 6.3: Diferencia entre los tiempos de ejecución (en segundos) de las nuevas rutinas y las de las bibliotecas *LAPACK*, *GotoBLAS* y *MKL*.

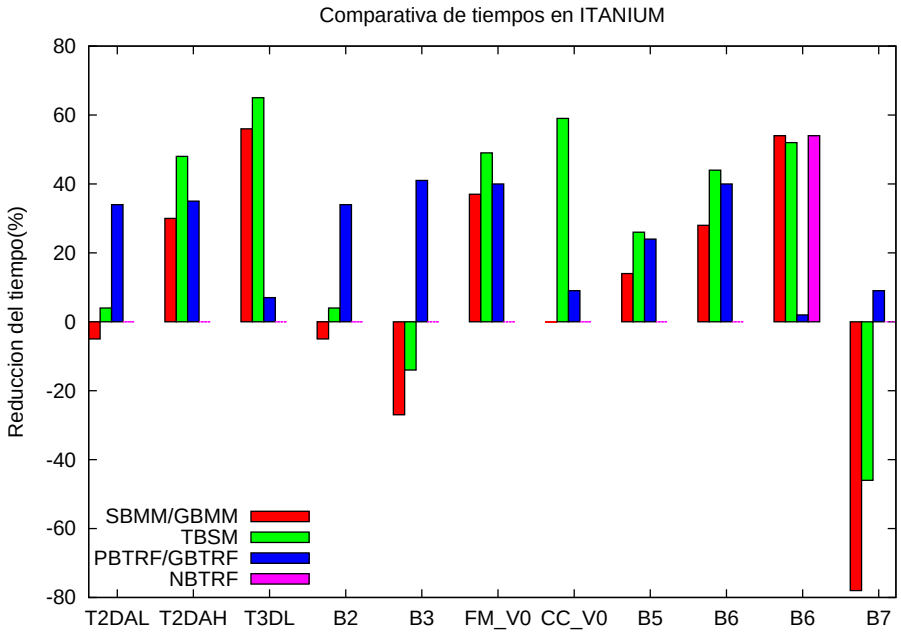


Figura 6.9: Comparativa entre las nuevas implementaciones y las de las bibliotecas *LAPACK*, *GotoBLAS* y *MKL* en ITANIUM.

y cuando se emplean varios procesadores en su ejecución.

- La eliminación del pivotamiento durante la facorización LU repercute en una importante mejora de las prestaciones para el ejemplo B6.RAIL_79841.
- El uso de las nuevas rutinas ha resultado positivo en todos los ejemplos.

Es conveniente recordar que cada una de estas operaciones se ejecuta en diversas ocasiones durante el método iterativo LR-ADI, suponiendo los cálculos más costosos que deben realizarse durante la iteración. La mejora obtenida al emplear las nuevas implementaciones en este algoritmo será verá pues acumulada en las diversas ejecuciones de cada una de las operaciones, en particular, para cada una de las iteraciones.

Capítulo 7

Conclusiones y líneas abiertas de investigación

7.1. Principales aportaciones y conclusiones

La estructura de la memoria refleja el objetivo general de la tesis, centrado en la resolución eficiente de problemas de álgebra lineal con estructura banda sobre procesadores actuales y arquitecturas paralelas con memoria compartida, agrupando estos problemas en tres capas: funcionalidad básica o *BLAS* (capítulos 2 y 3), funcionalidad avanzada o *LAPACK* (capítulos 4 y 5) y aplicaciones (capítulo 6). En cada uno de los capítulos anteriores, a partir del correspondiente estudio experimental, ya se han ofrecido conclusiones específicas para las diferentes operaciones. A continuación se ofrecen las conclusiones generales del trabajo desarrollado, destacando cuáles han sido las aportaciones originales de la tesis.

7.1.1. Funcionalidad básica: BLAS banda

BLAS-2 banda

BLAS define la especificación y funcionalidad de 5 rutinas para operaciones con matrices banda, 4 de ellas para el producto de una matriz banda por un vector (donde la matriz, además de banda, puede presentar una estructura general, simétrica, hermitiana o triangular) y la última para la resolución de un sistema de ecuaciones lineales con matriz de coeficientes triangular banda. La principal aportación en esta línea ha sido la elaboración de un estudio experimental exhaustivo (capítulo 2) de todas estas rutinas salvo el producto matriz hermitiana banda por vector. Esta última operación, que sólo tiene sentido cuando se trabaja con números complejos, presenta pocas diferencias respecto al producto matriz simétrica banda por vector.

Existe una implementación de referencia de las operaciones de *BLAS*-2 banda escrita en Fortran-77 (`http://www.netlib.org/blas`) que explota la estructura banda de la matriz para reducir el coste computacional y el espacio de almacenamiento requerido, empleando algunas optimizaciones mínimas como el recorrido de los elementos de la matriz por columnas y el uso de variables temporales para reducir el tiempo de acceso a los datos. La generación de código eficiente a partir las rutinas para operaciones con matrices banda del *BLAS* de referencia está fuertemente influenciada por el compilador utilizado, las opciones de compilación escogidas y, como es evidente, la plataforma de destino.

MKL y *GotoBLAS* son dos implementaciones optimizadas de *BLAS* para los procesadores de Intel (existen versiones de *GotoBLAS* específicas para procesadores de IBM, AMD, etc.) que

incluyen, entre sus operaciones, versiones supuestamente sintonizadas de las rutinas de *BLAS*-2 banda. Los resultados muestran que ninguna de estas dos implementaciones parece haber prestado especial atención a la optimización de las rutinas que operan sobre matrices banda.

El reducido coste computacional de las rutinas de *BLAS*-2 banda raramente justifica el uso de múltiples hebras para su ejecución paralela. Los experimentos realizados con el *BLAS* de referencia usando una herramienta de paralelización como OpenMP revelan que, debido a los problemas de compartición de datos entre las memorias caché del sistema, los beneficios que se obtienen de una ejecución paralela son insuficientes cuando se comparan con el sobrecoste de gestión de las hebras al ejecutar un código compuesto por un par de bucles anidados donde uno de éstos (el que recorre el ancho de banda de la matriz) realiza sólo unas pocas iteraciones.

En este nivel de *BLAS*, las nuevas implementaciones a menudo han mejorado las prestaciones obtenidas por las rutinas *MKL* y *GotoBLAS*, especialmente para matrices de banda media o ancha.

BLAS-3 banda

BLAS no incluye en su especificación rutinas del nivel 3 que operen con matrices banda, como el producto de una matriz banda por otra general o la resolución de múltiples sistemas de ecuaciones lineales. La necesidad de estos núcleos se ha detectado, por ejemplo, en aplicaciones de reducción de modelos y control óptimo. La única solución disponible hasta la fecha era el uso repetido de la rutina del *BLAS*-2 banda correspondiente. Sin embargo, el estudio experimental realizado demuestra que esta solución es altamente ineficiente, incluso cuando el número de repeticiones es muy pequeño. En respuesta a este problema, en esta tesis se aportan nuevas implementaciones (capítulo 3) que permiten calcular de forma eficiente operaciones como el producto de matrices, con una de las matrices general, simétrica o triangular banda, o las dos matrices con estructura general banda; y la resolución de múltiples sistemas de ecuaciones lineales con matriz de coeficientes triangular banda. Los nuevos códigos, escritos en Fortran-77 siguiendo el estilo marcado por el *BLAS* de referencia, se ajustan al formato de almacenamiento de matrices banda explotando la propia estructura de la matriz para reducir los costes. Además, los códigos hacen uso de núcleos optimizados del *BLAS* denso para ofrecer buenas prestaciones en el mayor número de plataformas posible.

El estudio experimental realizado muestra que las prestaciones de las nuevas rutinas superan claramente a la solución basada en el uso de rutinas del *BLAS*-2 banda, tanto en la ejecución secuencial como en la paralela sobre un multiprocesador con memoria compartida.

Los resultados obtenidos de este trabajo aparecen publicados en las actas de la siguiente conferencia:

1. "The implementation of *BLAS* for band matrices". A. Remón, E. S. Quintana, G. Quintana. Lecture Notes in Computer Science 4967, 7th Int. Conf. on Parallel Processing and Applied Mathematics – PPAM 2007, (Eds. R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski), pp. 668-677. Gdansk (Polonia), 2007. ISBN:978-3-540-68105-2.

7.1.2. Funcionalidad avanzada: LAPACK banda

Factorizaciones existentes

La biblioteca *LAPACK* incluye códigos secuenciales en Fortran-77 para el cálculo de las factorizaciones de Cholesky y LU con pivotamiento parcial de filas. Para cada una de estas operaciones se ofrecen una rutina *escalar* sencilla, que calcula una nueva columna/fila de la factorización en cada iteración, y una rutina por bloques que agrupa el cálculo de varias filas/columnas en una sóla iteración, de modo que hace posible el uso de núcleos eficientes del *BLAS*-3. El estudio experimental de

estos códigos, aportado en esta tesis, revela que la rutina por bloques supera a la correspondiente versión escalar a partir de un determinado umbral de tamaño de banda, que depende básicamente de las características de la plataforma *hardware* (tamaño de los diferentes niveles de memoria caché, número de unidades aritméticas, número de procesadores, velocidad de acceso a memoria, etc.).

El estudio experimental también revela una circunstancia muy significativa: algunas de las operaciones que se repiten en la iteración del algoritmo requieren un tiempo de ejecución mucho mayor del que cabría esperar conforme a su coste teórico.

Nuevas factorizaciones

Una de las aportaciones principales de la tesis viene a cubrir una carencia significativa de *LAPACK*, que no ofrece rutinas para el cálculo de la factorización LU sin pivotamiento o la factorización QR de una matriz banda. Si bien en el primero de los casos es posible utilizar la factorización LU con pivotamiento parcial, proceder de este modo es ineficiente: el pivotamiento consume un tiempo innecesario y, además, incrementa el tamaño de la banda de la matriz triangular superior resultante en un factor igual al de la banda inferior de la propia matriz, requiriendo un mayor espacio de almacenamiento y aumentando también el coste de la resolución del sistema triangular superior. Por su parte, la factorización QR resulta necesaria para la resolución de problemas lineales de mínimos cuadrados cuando la matriz involucrada presenta una estructura banda.

Incremento de la granularidad

Una aportación común de la tesis a todas estas factorizaciones ha sido el planteamiento de un esquema general que reduce el número de operaciones a realizar durante el cálculo de la factorización, agrupándolas en bloques de mayor entidad. El estudio experimental revela que en la práctica esta modificación no tiene efecto sobre la ejecución secuencial de la rutinas, si bien el beneficio en general sí es evidente cuando se utiliza un *BLAS* multihebra para el cálculo paralelo de la factorización sobre un multiprocesador con memoria compartida.

Incremento del paralelismo

La ejecución paralela tradicional de rutinas de *LAPACK*, basada en el uso de una implementación multihebra de la biblioteca *BLAS* que extrae todo el paralelismo dentro de las llamadas a las rutinas de esta última biblioteca, presenta notables problemas: por un lado, se produce un importante sobrecoste debido a la gestión de la hebras pues éstas deben sincronizarse con cada llamada a una rutina de *BLAS* lo que, en general, supone varias sincronizaciones por iteración. Por otro lado, no se aprovecha parte del paralelismo existente en la factorización, por ejemplo, entre operaciones de *BLAS* de una misma iteración o de iteraciones diferentes. Estos problemas tienen consecuencias especialmente negativas cuando se desea resolver un problema con un coste computacional reducido o con un número elevado de recursos (procesadores).

Siguiendo la aproximación planteada en la extensión SuperMatrix del proyecto FLAME, en esta tesis se aporta un algoritmo por bloques para la factorización de Cholesky de una matriz banda basado en el principio de planificación dinámica. Para un número elevado de procesadores, los resultados de este nuevo algoritmo claramente superan a los de la aproximación tradicional. La misma técnica puede aplicarse a las factorizaciones LU con y sin pivotamiento y QR.

Los resultados obtenidos en esta parte de la tesis han quedado reflejados en las siguientes publicaciones:

1. “Parallel LU factorization of band matrices on SMP systems”. A. Remón, E. S. Quintana, G. Quintana. Lecture Notes in Computer Science 4208, 2nd Int. Conf. on High Performance

Computing and Communications – HPCC 2006, (Eds. M. Gerndt, D. Kranzmüller), pp. 110-118. Munich (Alemania), 2006. ISBN: 978-3-540-39368-9.

2. "Cholesky factorization of band matrices using multithreaded *BLAS*". A. Remón, E. S. Quintana, G. Quintana. Lecture Notes in Computer Science 4699, Workshop on State-of-the-Art in Scientific and Parallel Computing – PARA 2006, (Eds. B. Kågström, E. Elmroth, J. Dongarra, J. Waśniesky,) pp. 608-616. Umea (Suecia), 2007. ISBN: 3-540-75754-6.

3. "SuperMatrix for the factorization of band matrices". G. Quintana, E. S. Quintana, A. Remón, R. van de Geijn. FLAME Working Note #27. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-07-51, 2007. 8th International Meeting on High Performance Computing for Computational Science – VECPAR 2008 (en revisión). Toulouse (Francia), 2008.

4. "*LAPACK*-Style algorithms for the QR factorization of band matrices". A. Remón, E. S. Quintana, G. Quintana. 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing – PARA 2008 (en revisión). Trondheim (Noruega), 2008.

5. "Clearer, Simpler and more Efficient *LAPACK* Routines for Symmetric Positive Definite Band Factorization". F. Gustavson, E. S. Quintana, G. Quintana, A. Remón, J. Wasniewski. 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing – PARA 2008 (en revisión). Trondheim (Noruega), 2008.
   Publicado como: IBM Research Report RC24597(W08007-034) Julio, 2008 Mathematics.

Además, en estos momentos se está procediendo a la adaptación de las rutinas de cálculo de la factorización QR con una matriz banda para su inclusión en la siguiente versión de la biblioteca *LAPACK*.

7.1.3. Aplicaciones

A fin de demostrar la utilidad de las aportaciones del trabajo desarrollado, se ha descrito con cierta profundidad una aplicación que frecuentemente da lugar a problemas de álgebra lineal con estructura banda. En particular, en reducción de modelos los métodos de resolución de ecuaciones de Lyapunov basados en la iteración LR-ADI requieren, durante cada una de las iteraciones, la resolución de un sistema de ecuaciones lineales y/o un producto de una matriz banda por una matriz general. Estas operaciones claramente se benefician de la utilización de las nuevas rutinas aportadas en esta tesis.

Existen numerosas aplicaciones, además de la reducción de modelos, que pueden también beneficiarse del trabajo realizado: en concreto, algunas aplicaciones de ciencia e ingeniería que dan lugar a problemas de álgebra lineal dispersa pueden reestructurarse para obtener una formulación banda del problema. En general, existe un mayor grado de paralelismo en el cálculo de una factorización banda que en una dispersa. Además, la propia estructura regular de la matriz banda favorece un acceso eficiente a la memoria pues, frente a la estructura irregular de las matrices dispersas, facilita la labor del técnicas de precaptación (*prefetch*) de datos.

Los estudios de aplicación de las nuevas rutinas secuenciales y paralelas para el cálculo de operaciones de álgebra lineal banda en problemas de reducción de modelos han llevado a las siguientes publicaciones:

1. "Efficient solution of large linear systems in model reduction for VLSI circuits." A. Remón, E. S. Quintana. 6th Conference on Scientific Computing in Electrical Engineering – SCEE 2006, pp. 63-64. Sinania (Rumanía), 2006. ISBN: 973-718-520-X.

2. "Solution of band linear systems in model reduction for VLSI circuits." A. Remón, E. S. Quintana, G. Quintana. Mathematics in Industry, Vol. 11, (Eds. G. Ciuprina, D. Ioan,) pp. 387-393. ISBN: 978-3-540-71979-3.

3. "Parallel implementation of LQG balanced truncation for large-scale systems." J. M. Badía, P. Benner, R. Mayo, E. S. Quintana, G. Quintana, A. Remón. Lecture Notes in Computer Science, 6th Int. Conference on Large-Scale Scientific Computations – LSSC 2007 (aceptado y pdte. de publicación). Sozopol (Bulgaria), 2007.

4. "Parallel solution of band linear systems in model reduction." A. Remón, E. S. Quintana, G. Quintana. Lecture Notes in Computer Science 4967, 7th Int. Conf. on Parallel Processing and Applied Mathematics – PPAM 2007, (Eds. R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski), pp. 678-687. Gdansk (Polonia), 2007. ISBN:978-3-540-68105-2.

5. "Toward the parallelization of GSL." J. I. Aliaga, F. Almeida, J. M. Badía, S. Barrachina, V. Blanco, M. Castillo, R. Mayo, E. S. Quintana, G. Quintana, A. Remón, C. Rodríguez, F. de Sande, A. Santos. The Journal of Supercomputing. (Eds. Springer Netherlands) ISSN:0920-8542(Print) 1573-0484(Online).

7.2. Líneas abiertas de investigación

La tesis cubre con sus objetivos la resolución eficiente de problemas de álgebra lineal con estructura banda sobre procesadores actuales y arquitecturas paralelas con memoria compartida. En esta misma línea de trabajo pueden identificarse los siguientes problemas no resueltos hasta la fecha, que constituyen líneas abiertas de investigación:

1. Un caso particular de estructura banda aparece cuando todos los elementos no nulos de la matriz están dispuestos sobre la diagonal principal y las dos diagonales adyacentes a ésta (superdiagonal y subdiagonal). Si bien los problemas que implican una *matriz tridiagonal* pueden resolverse haciendo uso de las rutinas de *BLAS* y *LAPACK* banda aportadas en esta tesis, existen algoritmos más eficientes para este caso particular, especialmente cuando se trata de explotar el paralelismo del problema al factorizar la matriz. Todos estos estudios paralelos del caso tridiagonal han sido planteados para plataformas paralelas de memoria distribuida, donde el coste de las comunicaciones juega un papel fundamental. Queda pendiente pues analizar cómo puede transladarse este trabajo a las arquitecturas paralelas con memoria compartida y en qué medida el menor coste de las comunicaciones en este tipo de multiprocesadores habilita soluciones alternativas eficientes para los problemas con estructura tridiagonal.

2. En el polo opuesto del caso tridiagonal se encuentran los problemas que presentan un ancho de banda considerable. Para estos casos, los estudios realizados en la tesis con la factorización de Cholesky demuestran que la utilización de un *algoritmo por bloques*, con planificación dinámica de las operaciones en tiempo de ejecución, consigue mejorar notablemente el rendimiento cuando el número de procesadores es elevado. Utilizando algoritmos por bloques propuestos para las factorizaciones LU y QR densas [24, 25, 79, 21, 22, 75] esta misma idea puede aplicarse para calcular estos otros tipos de factorizaciones de matrices banda.

3. El almacenamiento propuesto en *BLAS* y *LAPACK* para matrices banda, si bien eficiente en la medida en que posibilita el planteamiento de algoritmos por bloques para las operaciones habituales de álgebra lineal, presenta inconvenientes. Por un lado, al estar disociado el almacenamiento físico de los elementos de la matriz de su disposición física en memoria (el elemento

$A(i, j)$ de una matriz banda con lda filas no se almacena, como es norma en una disposición por columnas, en la posición $lda * j + i$) la codificación es más compleja y los errores son frecuentes. Por otro lado, este esquema no contempla el almacenamiento de las matrices por bloques (*block data layout*) [71, 44], es decir, un almacenamiento donde la matriz se particiona en bloques y los elementos de cada bloque ocupan posiciones consecutivas en memoria. Frente al almacenamiento por columnas habitual y los algoritmos por bloques, que permiten reducir el número de fallos en el acceso a las cachés, la utilización de este esquema alternativo de almacenamiento mejora además el número de aciertos en el acceso a las caché de TLB. Una línea abierta de investigación consiste pues en superar la complejidad de la programación de algoritmos de álgebra lineal banda y mejorar el rendimiento, introduciendo una *interfaz de codificación de alto nivel que facilite el desarrollo de algoritmos y almacenamiento por bloques.*

4. En álgebra lineal es frecuente la necesidad de resolver un problema de *actualización de una factorización*: se dispone inicialmente de una factorización (LU, Cholesky o QR) de la matriz de coeficientes del problema y se desea aprovechar esta información para obtener una segunda factorización de la matriz coeficientes a/de la que se han añadido/eliminado unas pocas filas/columnas. Existen soluciones para estos problemas aplicados en las factorizaciones densas [42, 78] que, en principio, pueden transladarse eficientemente al caso banda.

Bibliografía

[1] Advanced Micro Devices Inc., `http://www.amd.com/acml`.

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Kecler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *27th Annual Int. Symp. on Computer Architectures*, pages 248–259, 2005.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. MUMPS: a general purpose distributed memory sparse solver. In *Proc. PARA2000, 5th International Workshop on Applied Parallel Computing*, pages 122–131, 2000.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, third edition, 1999.

[5] Edward Anderson, Jack Dongarra, and Susan Ostrouchov. Installation guide for lapack. LAPACK working note 41, 1992.

[6] A. C. Antoulas. *Approximation of Large-Scale Dynamical Systems*. SIAM Publications, Philadelphia, PA, 2005.

[7] A. C. Antoulas and D. C. Sorensen. Approximation of large-scale dynamical systems: An overview. *Int. J. Appl. Math. Comp. Sci.*, 11(5):1093–1121, 2001.

[8] A. C. Antoulas, D. C. Sorensen, and S. Gugercin. A survey of model reduction methods for large-scale systems. *Contemp. Math.*, 280:193–219, 2001.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, University of California at Berkeley, Electrical Engineering and Computer Sciences, 2006.

[10] Barcelona Supercomputing Center, `http://www.bsc.es/smpsuperscalar`.

[11] P. Benner. Solving large-scale control problems. *IEEE Control Systems Magazine*, 24(1):44–59, 2004.

[12] P. Benner, V. Mehrmann, and D. C. Sorensen, editors. *Dimension Reduction of Large-Scale Systems*, volume 45 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Berlin/Heidelberg, Germany, 2005.

[13] P. Benner, E. S. Quintana-Ortí, and G. Quintana-Ortí. Balanced truncation model reduction of large-scale dense systems on parallel computers. *Math. Comput. Model. Dyn. Syst.*, 6(4):383–405, 2000.

[14] P. Benner and J. Saak. A semi-discretized heat transfer model for optimal cooling of steel profiles. Chapter 19 (pages 353–356) of [12].

[15] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Software*, 31(1):1–26, 2005.

[16] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Software*, 31(1):27–59, March 2005.

[17] Christian H. Bischof, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. Provisional contents. LAPACK Working Note 5 ANL-88-38, 1988.

[18] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[19] S. Y. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, S. S. Pawlowski, and J. R. Rattner. Platform 2015: Intel processor and platform for the next decade.

[20] Orlie Brewer, Jack Dongarra, and Danny Sorensen. Tools to aid in the analysis of memory access patterns of FORTRAN programs. *Parallel Computing*, 9(1):25–35, 1988.

[21] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical report, Innovative Computing Laboratory (University of Tennessee), September 2007.

[22] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled qr factorization for multicore architectures. Technical report, Innovative Computing Laboratory (University of Tennessee), 2007.

[23] Thuan D. Cao, John F. Hall, and Robert A. van de Geijn. Parallel Cholesky factorization of a block tridiagonal matrix. In *In Proceedings of the International Conference on Parallel Processing 2002 (ICPP-02)*, August 2002.

[24] Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *Proceedings of the ninteenth ACM symposium on Parallel Algorithms and Architectures*, pages 123–132, New York, NY, USA, 2007. ACM.

[25] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 123–132, New York, NY, USA, 2008. ACM.

[26] Cilk Arts, Inc, `http://supertech.csail.mit.edu/cilk`.

[27] Jeremy Du Croz, Peter Mayes, and Giuseppe Radicati. Factorization of band matrices using level 3 blas. In *Joint International Conference on Vector and Parallel Processing*, number 457 in Lecture Notes in Computer Science, pages 222–231. Springer-Verlag, London, UK, 1990.

[28] E. Cuthill and J. McKee. Reducing the bandwith of sparse symmetric matrices. In *Proc. of the 24th Nationanl Conference, ACM New York*, pages 157 – 172. ACM Press, 1969.

[29] James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. Prospectus for the developement of a linear algebra library for high-performance computers. Technical Report ANL/MCS-TM-07, 1987.

[30] J. J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. SIAM Philadelphia, 1979.

[31] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM Publications, Philadelphia, PA, 1979.

[32] J. J Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

[33] J. J Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14:1–17, 1988.

[34] J. J. Dongarra, J. du Croz, S. Hammarling, and I. S. Duff. Algorithm 679; a set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Software*, 16:18–28, 1990.

[35] J. J. Dongarra, J. du Croz, S. Hammarling, and R. J. Hanson. Algorithm 656: An extended set of basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Software*, 14:18–32, 1988.

[36] Jack Dongarra and David Walker. The design of linear algebra libraries for high performance computers. Technical Report UT-CS-93-188, 1993.

[37] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Oxford Science Publications, Oxford, UK, 1993.

[38] R. Freund. Model reduction methods based on Krylov subspaces. *Acta Numerica*, 12:267–319, 2003.

[39] K. Glover. All optimal Hankel-norm approximations of linear multivariable systems and their L^∞ norms. *Internat. J. Control*, 39:1115–1193, 1984.

[40] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, third edition, 1996.

[41] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Software*, 27(4):422–455, December 2001.

[42] Brian C. Gunter and Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Trans. Math. Software*, 31(1):60–78, March 2005.

[43] Anshul Gupta, Fred G. Gustavson, Mahesh Joshi, and Sivan Toledo. The design, implementation and evaluation of a symmetric banded linear solver for distributed-memory parallel computers. *ACM Trans. Math. Software*, 24(1):74–101, March 1998.

[44] F. G. Gustavson. New generalized data structures for matrices lead to a variety of high-performance algorithms. In Björn Engquist, editor, *Simulation and visualization on the grid: Parallelldatorcentrum, Kungl. Tekniska Högskolan, 7th annual conference, Stockholm, Sweden, December 1999: proceedings*, volume 13 of *Lecture Notes in Computational Science and Engineering*, pages 46–61. Springer-Verlag Inc., 2000.

[45] A. S. Hodel, B. Tenison, and Kameshwar K. R. Poolla. Numerical solution of the Lyapunov equation by approximate power iteration. *Linear Algebra Appl.*, 236:205–230, 1996.

[46] D. Hu and L. reichel. Krylov subspace methods for the Sylvester equation. *Linear Algebra Appl.*, 172:83–313, 1992.

[47] IBM, `http://www-03.ibm.com/systems/p/software/essl.html`.

[48] IMTEK - Institute of Microsystem Tech., `http://www.imtek.de/simulation/benchmark`.

[49] Innovative Computing Laboratory - University of Tennessee, `http://icl.cs.utk.edu/plasma`.

[50] Intel Corporation., `http://www.intel.com/`.

[51] Iowa State University, `http://www.cs.iastate.edu/ prabhu/Tutorial/title.html`.

[52] I. M. Jaimoukha and E. M. Kasenally. Krylov subspace methods for solving large Lyapunov equations. *SIAM J. Numer. Anal.*, 31:227–251, 1994.

[53] T. Kailath. *Systems Theory*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[54] R. Kumar, V. Zyuban, and D.M. Tullsen. Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling. In *32nd Int. Symp. on Computer Architecture, ISCA'05*, pages 408–419, 2005.

[55] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *15th international conference on Parallel architectures and compilation technique – PACT'06*, pages 23–32, 2006.

[56] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11):32–38, 2005.

[57] A. J. Laub. Numerical linear algebra aspects of control design computations. *IEEE Trans. Automat. Control*, AC-30:97–108, 1985.

[58] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5:303–323, 1979.

[59] J. R. Li and T. Penzl. Square root method via Cholesky factor ADI. In *Proc. MTNS 2000, Perpignan*, 2000. (CD Rom).

[60] J. R. Li and J. White. Reduction of large circuit models via low rank approximate gramians. *Int. J. Appl. Math. Comp. Sci.*, 11(5):1151–1171, 2001.

[61] J. R. Li and J. White. Low rank solution of Lyapunov equations. *SIAM J. Matrix Anal. Appl.*, 24(1):260–280, 2002.

[62] J. Lienemann, E. B. Rudnyi, and J. G. Korvink. MST MEMS model order reduction: Requirements and benchmarks. *Linear Algebra Appl.*, 415(2–3):469–498, 2006.

[63] Y. Liu and B. D. O. Anderson. Controller reduction via stable factorization and balancing. *Internat. J. Control*, 44:507–531, 1986.

[64] Tze Meng Low and Robert van de Geijn. An API for manipulating matrices stored by blocks. FLAME Working Note TR-2004-15 TR-2004-15, Dept. of Computer Sciences, The University of Texas, 2004.

[65] Matrix market. `http://www.matrixmarket.org`.

[66] B. C. Moore. Principal component analysis in linear systems: Controllability, observability, and model reduction. *IEEE Trans. Automat. Control*, AC-26:17–32, 1981.

[67] Netlib.org, `http://www.netlib.org/blas`.

[68] Netlib.org, `http://www.netlib.org/lapack`.

[69] Netlib.org, `http://www.netlib.org/eispack`.

[70] G. Obinata and B. D.O. Anderson. *Model Reduction for Control System Design*. Communications and Control Engineering Series. Springer-Verlag, London, UK, 2001.

[71] Neungsoo Park, Bo Hong, and Viktor K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. Parallel and Distributed Systems*, 14(7):640–654, 2003.

[72] T. Penzl. A cyclic low rank Smith method for large sparse Lyapunov equations. *SIAM J. Sci. Comput.*, 21(4):1401–1418, 2000.

[73] T. Penzl. Eigenvalue decay bounds for solutions of Lyapunov equations: the symmetric case. *Sys. Control Lett.*, 40:139–144, 2000.

[74] T. Penzl. LYAPACK Users Guide. Technical Report SFB393/00-33, Sonderforschungsbereich 393 *Numerische Simulation auf massiv parallelen Rechnern*, TU Chemnitz, 09107 Chemnitz, FRG, 2000. available from `http://www.tu-chemnitz.de/sfb393/sfb00pr.html`.

[75] Josep M. Perez, Rosa M. Badia, and Jesús Labarta. A flexible and portable programming model for smp and multi-cores. Technical report, Barcelona Supercomputing Center - Centro Nacional de Supercomputación, June 2007.

[76] P. H. Petkov, N. D. Christov, and M. M. Konstantinov. *Computational Methods for Linear Control Systems*. Prentice-Hall, Hertfordshire, UK, 1991.

[77] Eric Polizzi and Ahmed H. Sameh. A parallel hybrid banded system solver: the SPIKE algorithm. *Parallel Comput.*, 32(2):177–194, 2006.

[78] Enrique S. Quintana-Orti and Robert A. van de Geijn. Updating an lu factorization with pivoting. *ACM Trans. Math. Software*, 2008. Accepted for publication.

[79] Gregorio Quintana-Orti, Enrique S. Quintana-Orti, Alfredo Remon, and Robert van de Geijn. Supermatrix for the factorization of band matrices. FLAME Working Note 27, September 2007.

[80] Y. Saad. Numerical solution of large Lyapunov equation. In M. A. Kaashoek, J. H. van Schuppen, and A. C. M. Ran, editors, *Signal Processing, Scattering, Operator Theory and Numerical Methods*, pages 503–511. Birkhäuser, 1990.

[81] J. Saak. Effiziente numerische Lösung eines Optimalsteuerungsproblems für die Abkühlung von stahlprofilen. Diplomarbeit, Fachbereich 3/Mathematik und Informatik, Universität Bremen, D-28334 Bremen, September 2003.

[82] M. G. Safonov and R. Y. Chiang. A Schur method for balanced-truncation model reduction. *IEEE Trans. Automat. Control*, AC–34:729–733, 1989.

[83] SIAM, `http://history.siam.org/oralhistories/dongarra.htm`.

[84] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines—EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, second edition, 1976.

[85] Peter Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. *Int. J. of Parallel and Distributed Systems and Networks*, 4(1):26–35, 2001.

[86] T. Stykel. Gramian-based model reduction for descriptor systems. *Math. Control, Signals, Sys.*, 16:297–319, 2004.

[87] T. Stykel. Low rank iterative methods for projected generalized Lyapunov equations. Preprint 198, DFG Research Center MATHEON, TU Berlin, 2005. Available from `http://www.math.tu-berlin.de/~stykel/Publications/pr_04_198.pdf`.

[88] SUN microsystems, `http://developers.sun.com/sunstudio/perflib_index.html`.

[89] Texas Advanced Computing Center, `http://www.tacc.utexas.edu/~kgoto/`.

[90] Texas University, `http://www.cs.utexas.edu/users/flame`.

[91] M. S. Tombs and I. Postlethwaite. Truncated balanced realization of a stable non-minimal state-space system. *Internat. J. Control*, 46(4):1319–1330, 1987.

[92] F. Tröltzsch and A. Unger. Fast solution of optimal control problems in the selective cooling of steel. *Z. Angew. Math. Mech.*, 81:447–456, 2001.

[93] University of Tennessee, `http://math-atlas.sourceforge.net/`.

[94] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. `www.lulu.com`, 2008.

[95] A. Varga. Efficient minimal realization procedure based on balancing. In *Prepr. of the IMACS Symp. on Modelling and Control of Technological Systems*, volume 2, pages 42–47, 1991.