Universitat Jaume I de Castelló
E. S. de Tecnología y Ciencias Experimentales

Una Aproximación de Alto Nivel a la Resolución de Problemas Matriciales con Almacenamiento en Disco

Castellón, Abril de 2010

Tesis Doctoral presentada por: M. Mercedes Marqués Andrés
Dirigida por: Gregorio Quintana Ortí
Enrique S. Quintana Ortí

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Resolución de Problemas de Álgebra Lineal con Almacenamiento en Disco

1.1. Introducción

1.1.1. Motivación y objetivos

La necesidad de diseñar y simular con gran precisión complejos sistemas, que modelizan aplicaciones científicas, tecnológicas e industriales, requiere en ocasiones la resolución de problemas numéricos densos de elevada dimensión. En esta línea, trabajos recientes sobre la determinación del campo gravitatorio terrestre, la modelización de radares, la resolución de problemas de electromagnetismo y simulaciones de dinámica molecular, dan lugar a problemas numéricos bien conocidos del álgebra lineal, como sistemas de ecuaciones lineales o problemas lineales de mínimos cuadrados con matrices densas enormes, con decenas e incluso centenares de miles de filas/columnas [8, 30, 34, 56, 76, 45]. En todas estas aplicaciones, la resolución de estos problemas básicos supone la parte computacionalmente más costosa en la obtención de una respuesta, siendo su coste proporcional al cubo del tamaño de la matriz.

Para hacer frente a este alto coste computacional, las tendencias en el diseño de arquitecturas paralelas se dirigen hacia el uso de *procesadores multinúcleo* (o *multicore*), a los que a partir de este momento también nos referiremos como *arquitecturas multihebra*[1]. Así, actualmente existen procesadores comerciales de propósito general, aceleradores *hardware* y procesadores gráficos con más de un núcleo (*core*): 4 en el Intel Quad-Core y el AMD Quad-core, 6 en el SiCortex SC5832, 8 en el SUN UltraSparc "Niagara", 1+8 en la configuración heterogénea del Cell Broadband Engine, y hasta 240 núcleos simples en el procesador gráfico de la tarjeta Nvidia GTX 280. Además, las previsiones apuntan a que el número de núcleos por chip se doblará con cada nueva generación de procesadores (es decir, cada año y medio aproximadamente) [2, 7, 13].

La evolución en la capacidad y velocidad de los sistemas de memoria asociados a las arquitecturas multihebra sigue un ritmo de crecimiento menor (en parte siguiendo los dictados de la ley de Moore, que predice que el número de transistores en un procesador y en la memoria se doblan cada 18 y 24

[1] El término utilizado, arquitecturas multihebra, se refiere a que el mecanismo habitualmente empleado para explotar el paralelismo explícito en estas arquitecturas está basado en la utilización de múltiples hebras o hilos de ejecución (*threads*), bien sobre procesadores distintos, sobre núcleos distintos o sobre una combinación de éstos. De esta forma, los multiprocesadores también quedan incluidos en este término.

meses, respectivamente). La velocidad de los procesadores, medida con *benckmarks*, ha aumentado un 59 % cada año en el periodo de 1988 a 2004 [46]. Esta tendencia ha continuado después, aunque de forma más moderada. El ancho de banda de la memoria, sin embargo, ha aumentado más lentamente. Así, en el periodo de 1982 a 1994 aumentó un 38 % al año, mientras que a partir de 1995 el aumento pasó a ser de sólo un 23 % al año. Hoy en día es habitual que un procesador multinúcleo disponga de una memoria principal (RAM) de entre 4 y 8 GBytes. La adición de tantos núcleos está transformando la resolución de problemas de álgebra lineal densa de dimensión elevada, de un problema limitado por el cálculo (*processor-bounded*), en otro donde la limitación está en la capacidad de almacenamiento del sistema (*memory-bounded*).

Para salvar las restricciones de almacenamiento, en este trabajo nos planteamos el uso de almacenamiento secundario (disco) que, habitualmente, presenta una capacidad de entre uno y dos órdenes de magnitud mayor que la RAM. Aunque la gestión de este espacio adicional puede dejarse en manos del sistema operativo (memoria virtual), para obtener altas prestaciones en problemas de álgebra lineal en general resulta necesario dirigir cuidadosamente el patrón de acceso a los datos que se produce desde el programa. Esta clase de diseños, que persiguen orquestar eficazmente la entrada/salida (E/S) desde disco, se conocen como algoritmos *Out-Of-Core* (OOC), y en la práctica pueden considerarse como algoritmos por bloques que extienden la jerarquía de memoria para abarcar el nivel del almacenamiento secundario.

Además de la posibilidad de tratar problemas de mayor dimensión, una segunda razón en favor de las técnicas OOC es que son más rentables, pues abogan por reemplazar parcialmente el almacenamiento en RAM por el disco, de menor coste económico. A estos dos beneficios se suma la existencia de entornos especializados donde, por razones técnicas, la cantidad de memoria RAM es muy reducida. Un ejemplo de este tipo de plataformas muy especializado lo encontramos en los sistemas de cálculo a bordo de satélites y, más en general, en los sistemas empotrados (*embedded systems*). De nuevo es fácil identificar estas áreas como un campo fértil para la implantación de soluciones basadas en técnicas OOC, que permitan salvar la limitación que supone disponer de una cantidad muy reducida de RAM.

El objetivo general de la tesis es pues *diseñar, desarrollar y evaluar una colección de rutinas para resolver sistemas de ecuaciones lineales y problemas lineales de mínimos cuadrados, de dimensión elevada (matrices con decenas de miles de filas/columnas), sobre procesadores actuales, haciendo uso de técnicas OOC*. Este objetivo se plasma en la siguiente lista de objetivos específicos:

- Analizar la eficacia de las bibliotecas LAPACK y `libflame` [48, 68] de álgebra lineal densa para la resolución de sistemas de ecuaciones lineales y problemas lineales de mínimos cuadrados sobre procesadores de propósito general, incluidos procesadores multinúcleo.

- Extender estas bibliotecas con una capa, en la medida de lo posible transparente al usuario, que, haciendo uso de técnicas OOC, amplíe el rango (tamaño) de problemas de álgebra lineal densa tratables mediante esta solución.

- Estudiar el uso arquitecturas multihebra equipadas con procesadores de propósito específico (aceleradores *hardware*, GPU), como plataforma de resolución de problemas de álgebra lineal.

La propuesta planteada, basada en el uso de procesadores multinúcleo y técnicas OOC, puede completarse para abarcar problemas numéricos de muy elevada dimensión (matrices con cientos de miles de filas/columnas) que no pueden esperar a que la evolución de los procesadores de un solo núcleo permita su solución, y que aparecen en diversas aplicaciones. Para tomar conciencia de

qué se califica como "dimensión muy elevada" en este trabajo, cabe mencionar que la resolución de una instancia de tamaño moderado del problema lineal de mínimos cuadrados que surge en la modelización de radares requeriría el cómputo continuado sin fallo durante 30.000 días en un procesador Intel Itanium-2 (¡más de 82 años!). Y todo ello bajo la premisa irreal de que ese computador tuviese la capacidad suficiente para almacenar los datos.

Ante esta situación, queda clara la conveniencia de disponer de un *sistema paralelo* más escalable, formado por un código paralelo y una arquitectura multiprocesador distribuida, que permita obtener una solución con un tiempo de respuesta razonable: días, a lo sumo unos pocos meses, en lugar de años. En este sentido, existen actualmente bibliotecas paralelas para álgebra lineal densa como ScaLAPACK y PLAPACK [50, 69] que, haciendo uso de un potente *cluster* formado por unos 1.000 procesadores, podrían en teoría proporcionar la solución del problema ligado a la modelización de radares en aproximadamente un mes. Sin embargo, un par de factores clave juegan en contra de esta alternativa. En primer lugar, no es esperable que la eficiencia de tal sistema paralelo alcance un 100 %. Las sincronizaciones, las comunicaciones y los cálculos redundantes reducen el rendimiento en un factor no despreciable en la práctica, y habitualmente esta mengua se acentúa cuanto mayor es el número de nodos de cómputo del sistema paralelo (escalabilidad). Otro factor importante a considerar es la posibilidad de error (por ejemplo, por la caída de un nodo del sistema), que es directamente proporcional al producto del tiempo de ejecución de la aplicación por el número de nodos de cálculo.

Estos dos aspectos, escalabilidad del sistema y posibilidad de errores, favorecen una aproximación diferente, basada en una plataforma multiprocesador con un número de nodos más reducido, pero que dispongan de una mayor potencia de cálculo. Este tipo de arquitectura puede construirse, por ejemplo, a partir de un *cluster* formado por nodos equipados con aceleradores *hardware*, como el Cell Broadband Engine o el CSX600 de ClearSpeed, o procesadores de propósito específico, como los procesadores gráficos de Nvidia o AMD/ATI. A cambio, la capacidad de almacenamiento es habitualmente proporcional al número de nodos, lo que para problemas numéricos muy grandes nos devuelve a un entorno donde resulta imprescindible el uso de técnicas OOC. En este escenario, la colección de rutinas desarrollada en esta tesis dará lugar a una nueva alternativa de solución de sistemas de ecuaciones lineales y problemas lineales de mínimos cuadrados, de dimensión muy elevada (cientos de miles de filas/columnas) basada en *clusters* de computadores equipados con aceleradores *hardware* y que hagan uso de técnicas OOC.

En resumen, las rutinas desarrolladas en el marco de esta tesis permitirán resolver problemas de escala moderada en nodos con arquitectura multihebra, como por ejemplo, procesadores multinúcleo equipados (o no) con aceleradores *hardware*. Desde este punto de vista, la solución propuesta facilita una mejor difusión pues es indudable la mayor facilidad de uso que presenta una arquitectura multihebra frente a un gran *cluster* que se programa mediante paso de mensajes. Aún así, las rutinas desarrolladas en la tesis pueden combinarse con bibliotecas de paso de mensajes, como ScaLAPACK o PLAPACK, para proporcionar una solución a problemas de mayor escala, en *clusters* compuestos por nodos con arquitectura multihebra y/o con aceleradores *hardware*.

1.1.2. Estructura de la memoria

El primer capítulo de esta memoria está estructurado del modo siguiente. Tras la primera sección de introducción aparecen tres secciones donde se presentan las bibliotecas de álgebra lineal que se encuentran en la base este trabajo: BLAS en la Sección 1.2, LAPACK en la Sección 1.3 y

`libflame` en la Sección 1.4. A continuación, en la Sección 1.5, se introducen los algoritmos OOC. Para ello, se presentan las técnicas de diseño de este tipo de algoritmos, los distintos métodos que se utilizan para realizar el análisis de los mismos y las dependencias que pueden presentar respecto a la arquitectura de cada computador. La sección finaliza con una revisión de la literatura en la que se encuentran algoritmos OOC para la resolución de sistemas lineales densos. Las arquitecturas de los computadores sobre las que han sido implementados los algoritmos OOC desarrollados en esta tesis se presentan en la Sección 1.6. Por último, la Sección 1.7 expone el modo en que se va a realizar el análisis del rendimiento en este trabajo y la Sección 1.8 describe la notación utilizada en esta memoria.

Tras este primer capítulo introductorio a la resolución de problemas de álgebra lineal con almacenamiento en disco, siguen los tres capítulos principales de la tesis, en donde se tratan distintas factorizaciones de matrices. Una de las estrategias que se utilizan habitualmente para resolver sistemas de ecuaciones lineales densos del tipo $Ax = b$ comienza con la factorización de la matriz de coeficientes, de manera que ésta se descompone en el producto de dos matrices que se utilizan después para obtener la solución del sistema inicial. La factorización de Cholesky (Capítulo 2) se utiliza cuando la matriz de coeficientes del sistema lineal es simétrica y definida positiva, mientras que la factorización LU (Capítulo 3) se utiliza cuando la matriz es cuadrada pero no presenta una estructura particular. En ambas factorizaciones la matriz de coeficientes se descompone en un producto de dos matrices triangulares, por lo que la solución del sistema inicial se calcula resolviendo sendos sistemas triangulares. La factorización QR (Capítulo 4) se utiliza cuando la matriz de coeficientes es rectangular, con más filas que columnas. En este caso, el sistema lineal es sobredeterminado y la factorización se utiliza para resolver un problema lineal de mínimos cuadrados que obtiene una solución aproximada. Estos tres capítulos siguen una misma estructura. En primer lugar, se define la factorización, justificando su utilidad. A continuación, se presentan las distintas variantes algorítmicas para calcular la factorización, escalares y por bloques, que se obtienen mediante la técnica de derivación formal de algoritmos propia de FLAME. A partir de estas variantes algorítmicas, se obtienen los correspondientes algoritmos orientados a bloques. Éstos son generalizaciones de los algoritmos escalares tradicionales, en los que los cálculos sobre elementos se transforman en las correspondientes operaciones sobre bloques. Nuestro interés en la formulación de algoritmos orientados a bloques radica en que es trivial realizar implementaciones OOC a partir de ellos. Para esto, es suficiente considerar que cada bloque es un *tile* de la matriz, es decir, una submatriz que constituye la unidad elemental de transferencia de datos entre disco y memoria principal. Cuando se presentan las implementaciones OOC de los algoritmos orientados a bloques en cada uno de los capítulos, se muestran de manera progresiva las mejoras que se han ido incorporando para optimizar el rendimiento. En la mejora del rendimiento tomamos como referencia el rendimiento de la mejor implementación *in-core* disponible de cada una de las factorizaciones (denominamos *in-core* a las implementaciones que trabajan con datos que residen en la memoria RAM). Para cada factorización, se incluyen en el capítulo correspondiente los resultados experimentales obtenidos sobre distintas arquitecturas: un sistema monoprocesador, una arquitectura multihebra de propósito general y, cuando ha sido posible, una arquitectura multihebra equipada con una GPU. Cada uno de estos tres capítulos finaliza con una sección de conclusiones donde se resumen las aportaciones que se han realizado. El último capítulo de la memoria recoge las conclusiones de esta tesis y el trabajo futuro al que puede dar lugar.

1.2. BLAS

Muchos problemas científicos y de ingeniería se representan mediante modelos matématicos. En gran parte de las simulaciones y análisis que se realizan mediante estos modelos aparecen problemas fundamentales del álgebra lineal, como la resolución de sistemas de ecuaciones lineales o el cálculo de valores propios. Como ejemplos de estos problemas científicos y de ingeniería podemos citar el cálculo de estructuras, el control automático, el diseño de circuitos integrados o la simulación de reacciones químicas. Durante la resolución de estos problemas de álgebra lineal asociados a los modelos matemáticos 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*) [43, 25, 23] es la especificación de una colección de rutinas para este tipo de operaciones básicas de álgebra lineal densa, que define la funcionalidad y la interfaz que deben presentar las rutinas de la biblioteca. El desarrollo de BLAS se inició a principios de los años 70 [43] y para su elaboración se contó con la colaboración de especialistas de diferentes áreas de conocimiento, circunstancia que le da un valor especial, pues de esta forma se consiguió una especificación que cubre las necesidades requeridas en diversos campos [21].

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. Existe una implementación de BLAS genérica en un repositorio público de Netlib [47], pero lo que realmente hace útil a BLAS son las implementaciones específicas para cada arquitectura *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) [38], IBM (ESSL) [37] y SUN (Sun Performance Library) [60]. También existen otras implementaciones como GotoBLAS [61] y ATLAS [66], que han sido 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, BLAS ha tenido gran relevancia en la resolución de problemas de álgebra lineal. Su uso en la programación de aplicaciones para este tipo de problemas cuenta con múltiples ventajas:

- Eficiencia: BLAS cuenta con implementaciones específicas para cada arquitectura *hardware*.
- Robustez: las rutinas de BLAS han sido ampliamente probadas.
- Legibilidad del código: los nombres de las rutinas expresan unívocamente su funcionalidad.
- Portabilidad: 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.

Todas estas ventajas propiciaron que otras bibliotecas, como LINPACK [22] y EISPACK [58] en un principio, y más tarde LAPACK [6] o PLAPACK [71], fueran diseñadas para hacer uso internamente de las rutinas de BLAS.

Normalmente, BLAS está implementado en *C* y *Fortran* y, en ocasiones, en lenguaje 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 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 era que se desarrollaran 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 [43].

Con la aparición de nuevas arquitecturas se constató que el uso de BLAS no era la mejor manera de aumentar la eficiencia de las aplicaciones. Por ejemplo, en las máquinas vectoriales, es necesario optimizar a nivel de operaciones matriz-vector para acercarse a la eficiencia potencial de la máquina; sin embargo, el uso de BLAS inhibe esta optimización porque oculta al compilador las operaciones matriz-vector. Es por ello que en 1984 se propuso ampliar BLAS con un conjunto de operaciones matriz-vector, lo que constituyó el segundo nivel de BLAS (BLAS 2). En las rutinas de BLAS 2 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 [24, 25]. De dicha implementación, los autores destacan que los accesos a las matrices se realizan por columnas, puesto que es así como éstas se encuentran almacenadas en *Fortran*, redundando en una mayor eficiencia. Del mismo modo, se invitaba al desarrollo 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 BLAS 2, ya que la ratio entre el número de operaciones y el de datos es $O(1)$, mientras que 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) [23, 26], que implementa operaciones con un número de cálculos de orden cúbico frente a un número de datos de orden cuadrático. Es precisamente esta relación 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 (velocidad pico). Desde el punto de vista algorítmico, esto se consigue con los denominados *algoritmos por bloques*. Estos algoritmos dividen la matriz en bloques (submatrices) y al operar con éstos, 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 resultar más rápidos. En definitiva, los algoritmos por bloques extraen un mayor beneficio de la arquitectura jerarquizada del sistema de memoria. Como detalles adicionales, el tamaño óptimo de

los bloques en este tipo de algoritmos es dependiente de la arquitectura, especialmente del tamaño de la caché.

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 el hecho de 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 trasponer 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. Esta ratio es crucial en las arquitecturas con jerarquía de memoria, 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 en términos de operaciones aritméticas procesadas por unidad de tiempo. Por lo tanto, se definen los niveles en función de la 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 paralelas para arquitecturas multihebra. Con el fin de paralelizar los cálculos y emplear eficientemente los recursos disponibles, estas bibliotecas implementan un código multihebra que distribuye las operaciones entre los distintos procesadores de la arquitectura. 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 cálculo de los (múltiples) procesadores.

Como los mismos autores subrayan respecto al paralelismo, las rutinas BLAS 3 basadas en el cálculo por bloques presentan dos ventajas:

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

Desde el punto de vista de las prestaciones, se puede concluir que:

- Las prestaciones en los niveles 1 y 2 de BLAS 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 se puede realizar un mayor número de operaciones, consiguiendo prestaciones cercanas a la velocidad pico del procesador. Además, las rutinas BLAS 3 ofrecen mejores cualidades para su paralelización, pudiendo obtener por lo tanto mejoras más significativas en arquitecturas multihebra.

1.2.2. Núcleos básicos utilizados en este trabajo

En este apartado se citan las rutinas de BLAS que constituyen los núcleos básicos de los algoritmos que se presentan en este trabajo. 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 TMMOO, donde:

- T: indica el tipo de datos con los que trabaja la rutina; por ejemplo, S indica números reales de precisión simple, mientras que D indica números reales de precisión doble.
- MM: indica la estructura de las matrices con las que opera la rutina (o la estructura de la matriz más significativa); así, GE indica matrices generales, SY indica una matriz simétrica y TR indica una matriz triangular (existen otras estructuras de matrices pero no son de interés en este trabajo).
- OO: indica la operación que realiza la rutina.

Los núcleos básicos de BLAS que aparecen en esta memoria son los siguientes:

- _GEMM (BLAS 3): Producto de matrices $C := AB + C$, siendo A, B y C matrices generales (sin ninguna estructura particular).
- _TRSM (BLAS 3): Resolución de un sistema de ecuaciones del tipo $TX = B$ o $XT = B$, siendo X y B matrices generales, y T triangular.
- _SYRK (BLAS 3): Actualización de rango k de una matriz simétrica $C := AA^{\mathrm{T}} + C$, siendo C una matriz simétrica y A general.

1.3. LAPACK

LAPACK [6] (*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. Mientras que BLAS resuelve las operaciones básicas de álgebra lineal, LAPACK resuelve problemas más complejos como, por ejemplo, sistemas de ecuaciones lineales, problemas de mínimos cuadrados, y problemas de valores propios y valores 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 de las bibliotecas EISPACK [58] y LINPACK [22], y que además mejorara el rendimiento. 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 una jerarquía de memoria multinivel. 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. A consecuencia de esto, 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 [14].

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 (programación por bloques).

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, especialmente las del nivel 3, y obtener mejores prestaciones. Con este fin, los algoritmos implementados por LAPACK fueron reestructurados para trabajar con bloques [17, 27]. 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 [19].

La implementación genérica de LAPACK es de libre acceso [48] e incluye programas de comprobación y temporización. Algunos fabricantes de *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 de 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 para la factorización de matrices

LAPACK incluye una serie de rutinas para la factorización de matrices, además de rutinas que permiten resolver los sistemas obtenidos a partir de estas factorizaciones o estimar el número de condición de la matriz, entre otras. Como sucede en BLAS, el nombre de cada rutina LAPACK trata de incluir en él la información relevante para el usuario. El formato utilizado es idéntico al usado por BLAS: TMMOO, donde T indica el tipo de datos, MM indica la estructura de la matriz y OO indica la operación que realiza la rutina (puede tener hasta tres caracteres).

Las rutinas de LAPACK que realizan las factorizaciones de matrices densas son las siguientes:

- _POTRF: Algoritmo por bloques para calcular la factorización de Cholesky.
- _GETRF: Algoritmo por bloques para calcular la factorización LU con pivotamiento parcial de filas.
- _GEQRF: Algoritmo por bloques para calcular la factorización QR.

En los tres casos, la factorización de cada bloque diagonal/panel se realiza mediante la correspondiente rutina basada en BLAS 2 (_POTF2 para la factorización de Cholesky, _GETF2 para la LU y _GEQR2 para la QR), mientras que los demás cálculos se realizan, principalmente, mediante llamadas a rutinas de BLAS 3.

1.4. libflame

`libflame` es la biblioteca que se ha obtenido como resultado del proyecto FLAME (*Formal Linear Algebra Methods Environment*) [67]. El objetivo de este proyecto es el de transformar el desarrollo de bibliotecas de álgebra lineal densa, hasta ahora un arte reservado a expertos, en una ciencia que pueda ser entendida también por noveles. El proyecto, además de haber dado lugar a la biblioteca `libflame`, proporciona una nueva notación para expresar algoritmos [10, 72], una metodología para derivarlos de manera sistemática, una serie de APIs (interfaces de programación de aplicaciones) para generar código a partir de los algoritmos, y herramientas para derivar e implementar algoritmos de manera automática, así como para analizarlos.

1.4.1. Características

A continuación se enumeran algunas de las características de `libflame`:

- Los algoritmos implementados, elaborados mediante la metodología de FLAME, se han obtenido de manera sistemática, siguiendo unos principios rigurosos de derivación formal. Estos métodos se basan en teoremas fundamentales de las ciencias de la computación, garantizando así que el algoritmo obtenido es correcto. Cabe destacar que FLAME utiliza una nueva notación, más estilizada, para expresar algoritmos de álgebra lineal basados en bucles. Esta notación se presenta más adelante, en el Apartado 1.4.2.

- `libflame` está basada en objetos. Se trabaja con estructuras de datos opacas que ocultan los detalles de implementación de las matrices y se proporcionan APIs basadas en objetos para operar con estas estructuras. Esta abstracción facilita la programación sin utilizar índices en los bucles ni en los operandos (vectores y matrices).

- Al igual que LAPACK, `libflame` proporciona implementaciones de las operaciones de álgebra lineal más habituales. Las rutinas de `libflame` son funcionalmente análogas a las que se encuentran en BLAS y en LAPACK. Sin embargo, a diferencia de LAPACK, `libflame` proporciona una infraestructura para construir códigos de álgebra lineal a medida [**?**]. Este entorno es más útil, ya que permite al usuario prototipar de manera rápida una solución de álgebra lineal para una aplicación concreta.

- Las rutinas de `libflame` obtienen, a menudo, mejores prestaciones que las rutinas equivalentes de LAPACK [52]. Muchas veces esto sucede porque `libflame` dispone de todas las variantes (algoritmos) de cada operación, por lo que el usuario puede elegir aquella que es más apropiada en cada situación [11]. `libflame` depende de la presencia de un núcleo de rutinas escalares (no orientadas a bloques) altamente optimizadas, para resolver los pequeños subproblemas que aparecen en los códigos FLAME.

- Para extraer el máximo nivel de paralelismo en sistemas con memoria compartida, `libflame` cuenta con un sistema denominado *SuperMatrix* que, en tiempo de ejecución, detecta y analiza las dependencias de datos en los algoritmos por bloques de FLAME [16]. Una vez conocidas estas dependencias, el sistema planifica la ejecución simultánea de operaciones sobre bloques independientes a hebras de ejecución. A diferencia de `libflame`, LAPACK consigue el paralelismo enlazando con implementaciones de BLAS multihebra. Este paralelismo es de más bajo nivel y presenta importantes limitaciones en cuanto a eficiencia y escalabilidad, cuando se trata de problemas con matrices pequeñas o medianas. En cambio, con `libflame` el paralelismo es de mayor nivel, proporcionando mejores prestaciones que LAPACK [52].

- El almacenamiento de matrices por bloques permite obtener mejores prestaciones debido a que se incrementa la localidad espacial. `libflame` no representa las matrices como un vector, en el que la matriz se almacena físicamente por columnas/filas, sino que el esquema de almacenamiento está codificado dentro del objeto. Internamente, los elementos hacen referencia a otros objetos que representan a submatrices (bloques), posibilitando el uso de un almacenamiento jerárquico.

- `libflame` proporciona una API para programar algoritmos por bloques denominada FLASH. Los algoritmos por bloques trabajan sobre matrices que son potencialmente jerárquicas, y que son referenciadas y/o almacenadas como submatrices. FLASH contiene completamente implementado el nivel 3 de BLAS, en el que basa la mayor parte de sus operaciones, y proporciona además una serie de rutinas de servicio para la gestión de matrices (creación y destrucción, inicialización de contenidos, etc.).

- `libflame` incorpora una estructura de control para especificar de manera sencilla una ejecución algorítmica multinivel. La estructura codifica la elección de la variante y el tamaño de bloque para cada subproblema, haciendo posible ejecutar cualquier algoritmo por bloques, secuencial o paralelo, con recursión de nivel arbitrario sin cambiar el código del algoritmo.

- Cada distribución de `libflame` está integrada en un paquete con un sistema de instalación muy robusto y que sigue los estándares de GNU.

- `libflame` ofrece compatibilidad con LAPACK mediante un conjunto de rutinas que mapean las llamadas a rutinas de LAPACK con sus correspondientes implementaciones de `libflame`.

En resumen, `libflame` es actualmente una biblioteca muy profesional, completada con herramientas útiles de desarrollo y apoyo a la ejecución, por lo que ha sido escogida como base para los desarrollos de este trabajo.

1.4.2. Metodología de derivación formal de algoritmos y notación FLAME

En la programación por objetivos se comienza con una formalización de lo que se debe calcular, es decir, el objetivo, y a partir de él, se desarrolla de manera sistemática un programa que implemente dicho objetivo. Del mismo modo, en la metodología propuesta en el proyecto FLAME, se comienza especificando la operación para la que hay que implementar un algoritmo. A partir de la especificación, se van determinando predicados (asertos) de manera sistemática, y se insertan en el algoritmo antes de añadir las instrucciones que realizan los cálculos. Estas instrucciones se

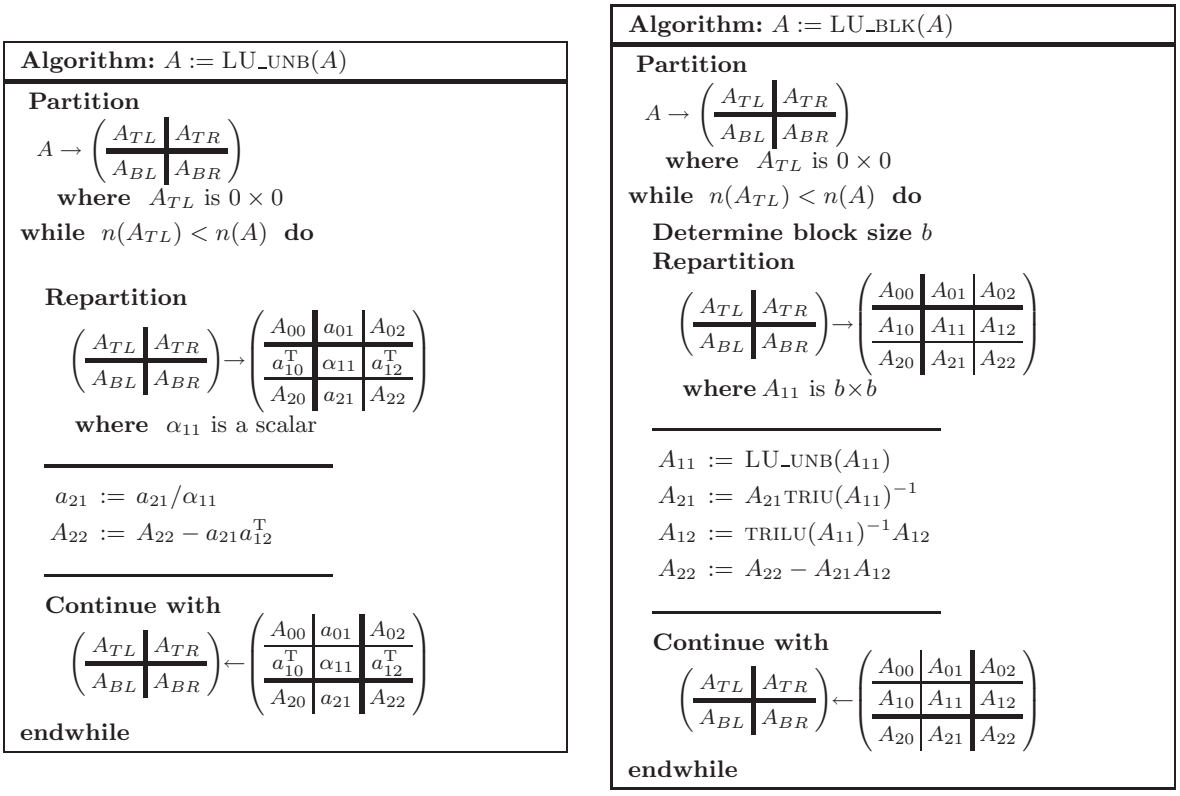


Figura 1.1: Algoritmo escalar (izquierda) y por bloques (derecha) para calcular la factorización LU en notación FLAME. TRIU(A_{11}) devuelve la matriz triangular superior que se almacena en A_{11} y TRILU(A_{11}) devuelve la matriz triangular inferior que se almacena en A_{11}, poniendo a unos la diagonal.

insertarán después, de manera que hagan verdaderos los predicados en los puntos indicados. De este modo, se desarrolla el algoritmo a la vez que se realiza la prueba de su corrección.

Los predicados expresan de manera natural el estado en el que deben estar las variables en un punto dado del algoritmo. Ya que los algoritmos de cálculo matricial conllevan el uso de bucles, para desarrollarlos se debe expresar el estado de las variables que son actualizadas en el cuerpo del bucle antes y después de cada iteración. Para ello se establece el invariante del bucle, que es un predicado que se cumple justo antes de entrar en el bucle y, en cada iteración, antes y después del cuerpo del bucle. El invariante es la clave para probar la corrección del bucle y se identifica de manera sistemática a partir de la postcondición, que es la especificación del cálculo a realizar.

Los algoritmos desarrollados con esta metodología de derivación formal se expresan mediante la notación desarrollada en el proyecto FLAME, que destaca por su concreción, regularidad y simplicidad. La Figura 1.1 (izquierda) 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 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 tienen el valor 1.

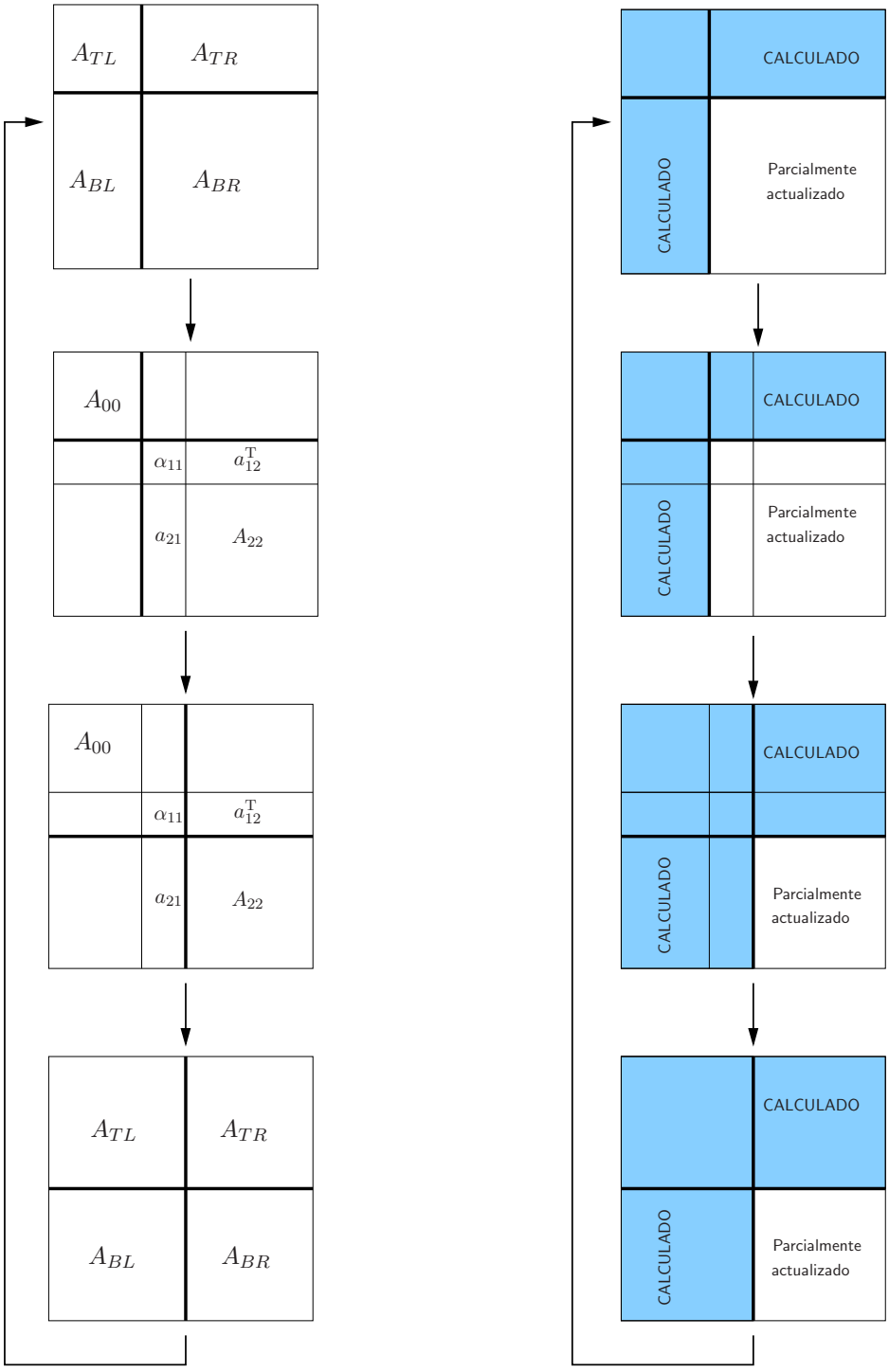


Figura 1.2: Particionado aplicado a la matriz A durante el algoritmo LU_UNB de la Figura 1.1.

En el algoritmo inicialmente se particiona A en cuatro bloques: A_{TL}, A_{TR}, A_{BL} y A_{BR} (particionado 2×2), siendo el primero de tamaño 0×0. De este modo, al inicio de la primera iteración del bucle, el bloque A_{BR} está formado por todos los elementos de A.

En cada iteración, el bloque A_{BR} se divide, tal y como se muestra en la Figura 1.2, en otros cuatro bloques: α_{11}, a_{21}, a_{12}^{T} y A_{22}, pasando de un particionado 2×2 a un particionado 3×3. Los elementos de los factores L y U correspondientes a los tres primeros de estos bloques son calculados durante la presente iteración (α_{11} y a_{12}^{T} no cambian), mientras que A_{22} es actualizado respecto a ellos. 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} es de dimensión 0×0, es decir, cuando el número de columnas del bloque A_{TL} coincide con el número de columnas de la matriz A. En este sentido, el bucle del algoritmo se repite mientras $n(A_{TL}) < n(A)$, donde $n(\cdot)$ denota el número de columnas de una matriz (o el número de elementos de un vector fila). Análogamente, $m(\cdot)$ se utilizará para denotar el número de filas de una matriz (o el número de elementos de un vector columna).

La Figura 1.1 (derecha) muestra un *algoritmo por bloques* para el cálculo de la misma operación, la factorización LU. El funcionamiento de este tipo de algoritmos, que son clave 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, se usará la letra b para denotarlo.

La notación FLAME es útil ya que permite expresar los algoritmos con un alto nivel de abstracción. Estos algoritmos deben después plasmarse en una *rutina* o *implementación*, con un mayor nivel de concreción. Así, por ejemplo, en los algoritmos de la Figura 1.1 no se indica el modo en que se realizan las operaciones que forman el cuerpo de los bucles. Una rutina que, por ejemplo, ejecutase la operación $a_{21} := a_{21}/\alpha_{11}$ mediante una llamada a la rutina _SCAL de BLAS 1 y otra rutina que calculase este escalado mediante un simple bucle que recorriese los elementos del vector a_{21}, pese a ser diferentes, responderían al mismo algoritmo. Por lo tanto, es importante distinguir entre *algoritmo* y *rutina* o *implementación*.

1.4.3. Interfaces de programación de aplicaciones

En el marco del proyecto FLAME se han desarrollado una serie de APIs para ayudar a simplificar la tarea de programar, documentar y mantener familias de algoritmos para un amplio espectro de operaciones de álgebra lineal. Algunos ejemplos de estas APIs son:

- FLAME@lab: API para el lenguaje *M-script* de *Matlab*.
- FLAME/C: API para el lenguaje C.
- PLAPACK2e: Extensión para FLAME del paquete PLAPACK (memoria distribuida).
- FLaTeX: Conjunto de órdenes para escribir algoritmos de FLAME en LaTeX.

A continuación se describe brevemente la API desarrollada para el lenguaje C, FLAME/C, que se ha utilizado en este trabajo.

- El tipo de datos de FLAME para almacenar operandos es `FLA_Obj`, ya sean escalares, vectores o matrices. Toda la información sobre el operando se encuentra encapsulada en el objeto: tipo de datos de los elementos, tamaño y dimensión principal, entre otros. Además, el objeto posee un puntero al *buffer* donde se almacenan los elementos en memoria.

- `FLA_Init()` y `FLA_Finalize()` inicializan y finalizan, respectivamente, el uso de la API.

- La rutina `FLA_Obj_create` inicializa un objeto que describe una matriz, un vector o un escalar, y crea su espacio de almacenamiento. Hay una rutina equivalente que inicializa un objeto sin espacio de almacenamiento, espacio que se puede añadir más tarde mediante otra rutina. La memoria asociada a los objetos se libera mediante la rutina `FLA_Obj_free`. También es posible crear un objeto cuyas características sean las mismas que las de otro objeto existente.

  Tanto escalares como vectores se definen como casos especiales de matrices. Así, un escalar es una matriz 1×1, un vector fila es una matriz $1 \times n$ y un vector columna es una matriz $n \times 1$.

- Hay una serie de rutinas que permiten obtener información sobre las características de un objeto `FLA_Obj`. `FLA_Obj_datatype` devuelve el tipo de datos de los elementos de un objeto, `FLA_Obj_length` devuelve el número de filas, `FLA_Obj_width` devuelve el número de columnas, `FLA_Obj_buffer` devuelve un puntero al espacio de almacenamiento donde se encuentran los datos, y `FLA_Obj_ldim` devuelve la dimensión principal (para matrices almacenadas por columnas, es el número de elementos entre dos posiciones consecutivas en la misma fila).

- El particionado y reparticionado de los datos que se lleva a cabo en los algoritmos desarrollados mediante la metodología de FLAME se realiza mediante las rutinas `FLA_Part_2x2` y `FLA_Repart_from_2x2_to_3x3`. La rutina `FLA_Cont_with_3x3_to_2x2` permite avanzar en la matriz, recuperando el particionado 2×2. Existen rutinas equivalentes a las anteriores para realizar particionados horizontales y verticales sobre matrices y también vectores.

- El conjunto de rutinas que implementan las operaciones de álgebra lineal contiene una serie de operaciones generales como las de BLAS que permiten operar con escalares, vectores o matrices. Dos sencillos ejemplos son `FLA_Copy` y `FLA_Swap`, que pueden usarse para copiar el contenido de un objeto en otro objeto e intercambiar el contenido de dos objetos, respectivamente, ya sean escalares, vectores o matrices. Además, `libflame` proporciona una serie de rutinas no incluidas en BLAS como las factorizaciones LU, Cholesky y QR, y las respectivas rutinas para la resolución de los sistemas lineales asociados.

- Los tipos de datos de los elementos son `FLA_INT`, `FLA_DOUBLE`, `FLA_FLOAT`, `FLA_DOUBLE_COMPLEX` y `FLA_COMPLEX`.

La Figura 1.3 (izquierda) muestra la implementación, mediante FLAME/C, del algoritmo de la Figura 1.1 (derecha). A su lado se muestra un programa de ejemplo que, tras crear e inicializar la matriz, llama a la rutina especificada para calcular su factorización LU.

El proyecto FLAME proporciona un generador de código denominado *Spark* [70]. A partir de un algoritmo diseñado mediante el proceso de derivación formal FLAME esta herramienta web genera el esqueleto de la función correspondiente al algoritmo, en el lenguaje de la API que se especifique: FLAME@lab, FLAME/C, PLAPACK2e o FLATEX. El esqueleto contiene todas las operaciones de particionado y reparticionado de los operandos, de modo que el programador tan solo debe

```
#include "FLAME.h"

FLA_Error LU_unb( FLA_Obj A )
{
  FLA_Obj ATL,   ATR,      A00,  a01,     A02,
          ABL,   ABR,      a10t, alpha11, a12t,
                           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,
    /* ************* */  /* ************************** */
                           &a10t, /**/ &alpha11, &a12t,
       ABL, /**/ ABR,       &A20,  /**/ &a21,     &A22,
       1, 1, FLA_BR );

    /*--------------------------------------------*/

    /* a21 := a21 / alpha11; */
    FLA_Inv_scal( alpha11, a21 );

    /* A22 := A22 - a21 * a12t; */
    FLA_Ger( FLA_MINUS_ONE, a21, a12t, A22 );

    /*--------------------------------------------*/

    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,       A00,  a01,     /**/ A02,
                              a10t, alpha11, /**/ a12t,
    /* ************** */   /* ************************ */
       &ABL, /**/ &ABR,       A20,  a21,     /**/ A22,
       FLA_TL );
  }

  return FLA_SUCCESS;
}
```

```
#include "FLAME.h"

void main()
{
  FLA_Obj A;
  int     n;

  FLA_Init( );

  printf( "Dimensión de la
           matriz (n):" );
  scanf( "%d", &n );

  FLA_Obj_create( FLA_DOUBLE,
                  n, n, &A );

  fill_matrix( A );

  LU_unb( A );

  FLA_Obj_show( "A = [", A,
                "%lf", "]" );

  FLA_Obj_free( &A );

  FLA_Finalize( );
}
```

Figura 1.3: Ejemplo de uso de la API FLAME/C para implementar el algoritmo LU_UNB de la Figura 1.1 que calcula la factorización LU de una matriz (izquierda) y un programa que utiliza la rutina implementada (derecha).

especificar las operaciones de cálculo (son, por ejemplo, las operaciones que se encuentran entre las dos líneas de comentario que contienen guiones en la función `LU_unb` de la Figura 1.3).

1.5. Algoritmos OOC

Cuando las estructuras de datos con las que debe trabajar un algoritmo son demasiado grandes para caber en la memoria, una posible solución es hacer uso del almacenamiento en disco. El acceso a disco es lento, por lo que para mantener el rendimiento se debe intentar que el acceso a los datos se circunscriba a bloques grandes y contiguos, y se reutilicen los datos cargados en memoria tanto como sea posible. Los algoritmos que se diseñan para obtener buenas prestaciones cuando sus estructuras de datos están almacenadas en disco se conocen como *algoritmos out-of-core* (OOC).

La necesidad de resolver problemas de álgebra lineal mediante algoritmos OOC queda patente por la multitud de trabajos que se han publicado al respecto en los últimos años. Toledo [63] ha realizado un estudio de los algoritmos OOC que aparecen en dichos trabajos, identificando dos técnicas de diseño que se presentan en el siguiente apartado. Basándonos en su trabajo, mostramos también un resumen de las distintas maneras en que se realiza el análisis de los algoritmos OOC, y recogemos dos de sus reflexiones; la primera en cuanto a la dependencia de la arquitectura de estos algoritmos, y la segunda respecto a la conveniencia de utilizar abstracciones a la hora de diseñar algoritmos OOC.

1.5.1. Técnicas de diseño

Toledo clasifica en dos grupos los algoritmos OOC que aparecen en la literatura, diferenciando así dos técnicas de diseño. En la primera técnica, denominadada *de particionado*, se planifican las operaciones de modo que la ejecución OOC sea eficiente. La segunda técnica, denominada *de transformación del flujo de datos*, consiste en modificar el algoritmo, de manera que cambie el flujo de datos a un modo en que se pueda conseguir una ejecución eficiente.

Un ejemplo de ambas técnicas se puede ilustrar con un problema sencillo como es el producto de matrices. Un algoritmo tradicional para el producto de matrices $C := A \cdot B$, expresado en la notación *M-script* de *Matlab*, es el que consta de tres bucles anidados:

```
function [ C ] = gemm(n, A, B, C )
for i = 1:n
  for j = 1:n
    C(i,j) = 0;
    for k = 1:n
      C(i,j) = C(i,j) + A(i,k) * B(k,j);
    end
  end
end
```

Para hacer una estimación del número de accesos a elementos en disco que realiza este algoritmo, vamos a suponer que en la memoria principal cabe menos de la mitad de una matriz. Si M es el tamaño de la memoria, estamos suponiendo pues que $M < n^2/2$. Puesto que en cada iteración del bucle exterior se accede a todos los elementos de B y, a lo sumo, sólo la mitad de ellos pueden

permanecer en memoria, al menos la otra mitad se deberá leer en cada iteración, por lo que el número de accesos a elementos en disco será, como mínimo, $n \cdot n^2/2 = n^3/2$ (cota inferior). Por otra parte, la cota superior de este número de accesos será $4n^3$ (tres lecturas y una escritura en cada iteración del bucle interior). En cualquier caso, el número de accesos a disco es del mismo orden que el número de operaciones, $O(n^3)$.

Un algoritmo OOC diseñado siguiendo la técnica del particionado consigue ahorrar accesos a disco, reduciendo este número para el producto de matrices a $O(n^3/\sqrt{M})$. Para ello, se deben considerar las matrices particionadas en bloques de tamaño $b = \sqrt{M/3}$, de modo que en cada momento sea posible almacenar en memoria un bloque de cada una de las tres matrices a la vez. Suponiendo, por simplicidad, que b es múltiplo de n, el algoritmo queda como sigue:

```
function [ C ] = gemm_blk( n, b, A, B, C )
for i = 1:b:n
  for j = 1:b:n
    C(i:i+b-1,j:j+b-1) = 0;
    for k = 1:b:n
       C(i:i+b-1,j:j+b-1) = C(i:i+b-1,j:j+b-1) + ...
                            gemm( b, A(i:i+b-1,k:k+b-1), ...
                                     B(k:k+b-1,j:j+b-1), ...
                                     C(i:i+b-1,j:j+b-1) );
    end
  end
end
```

De este modo, el cuerpo del bucle interior accede a $3b^2 = M$ elementos, que permanecen en memoria mientras se realizan $2b^3$ operaciones aritméticas con ellos, por lo que si b es suficientemente grande, puede ocultarse la latencia del disco. El número de elementos accedidos en disco con este algoritmo estará acotado por el número de transferencias realizadas en el cuerpo del bucle interior multiplicado por el número de veces que se ejecuta: $M \cdot (n/b)^3 = O(n^3/\sqrt{M})$. En resumen, en la técnica de particionado se planifican las operaciones de modo que el acceso sea más eficiente.

Entre los algoritmos de particionado que operan con matrices densas, se pueden distinguir distintos modos de realizar las particiones. Así, encontramos particiones por submatrices (bloques), particiones por paneles (bloques de columnas) y particiones recursivas. Además, un mismo tipo de partición puede dar lugar a diversos algoritmos. Por ejemplo, el primer algoritmo de particionado para calcular la factorización LU con pivotamiento parcial, que data de 1960 [9], trabaja sobre paneles y está orientado hacia la derecha (tras factorizar un nuevo bloque de columnas, se actualizan los bloques de columnas que quedan a su derecha, como sucedía en el algoritmo de la Figura 1.1). Sin embargo, el algoritmo OOC más utilizado hoy en día para la factorización LU, que fue propuesto en 1981 [18], también trabaja sobre paneles, pero está orientado hacia la izquierda (antes de calcular un bloque de columnas, éste es actualizado respecto a los bloques de columnas ya calculados a su izquierda). En consecuencia, realiza el mismo número de lecturas que el orientado hacia la derecha, pero menos escrituras.

Por otro lado, para obtener un algoritmo OOC con la técnica de transformación del flujo de datos para el producto de matrices, se debería partir de un algoritmo fundamentalmente diferente (por ejemplo, el algoritmo de Strassen) que es posible que requiera un menor número de accesos a

disco. En ocasiones, los algoritmos de transformación del flujo de datos requieren más cálculos; en otras ocasiones, son algoritmos con peores condiciones numéricas.

Ambas técnicas de diseño se llevan usando desde principios de la década de los años 50. Sin embargo, la técnica del particionado dejó de emplearse durante los años 70, en parte porque se pensaba que los algoritmos de particionado requerían que los datos se almacenaran por bloques, en tanto que los compiladores disponían los elementos de las matrices por columnas o por filas. Hay algunos trabajos de la época en los que se presentan algoritmos por columnas, y se subestiman los algoritmos por bloques argumentando que no son buenos para *Fortran*. Además, en aquella época, la disparidad de rendimiento entre la memoria RAM y el disco todavía no era tan significativa, por lo que el alto nivel de reutilización de datos de los algoritmos de particionado tampoco parecía necesario.

1.5.2. Análisis de los algoritmos

Toledo encuentra que en la literatura aparecen cuatro formas distintas analizar la E/S (entendido, en nuestro caso, como el acceso a disco) y el nivel de reutilización de datos. Una primera forma que se propone consiste en contar el número exacto de lecturas y escrituras. Sin embargo, ésta la forma más tediosa de hacerlo y difícilmente predice el impacto de la E/S en el tiempo de ejecución, ya que el tiempo que cuesta servir una petición de E/S depende de muchos factores: el tamaño de la petición, si los datos son o no son precargados (*prefetch*), si las peticiones acceden a los datos secuencialmente, etc. En definitiva, el conteo de las operaciones de E/S no parece dar una estimación precisa de su impacto sobre el tiempo de ejecución.

Un segundo modo de realizar el análisis consiste en obtener el número asintótico de operaciones de E/S, aunque puede suceder que esta medida no sea útil para comparar algoritmos. Esta afirmación parte de la suposición de que la memoria es, como mínimo, 1/1.000 el tamaño del disco. Así, suponiendo un tamaño de problema n =1.000.000, la memoria puede almacenar $O(\sqrt{n})$ datos *in-core*. Por ejemplo, en la transformada rápida de Fourier, el número asintótico de operaciones de E/S es $O(n \log n / \log M)$, que se reduce a $O(n)$ ya que $\log n / \log M \leq 2$, por lo que se concluye que esta medida no es de utilidad para comparar con un algoritmo que sólo realiza $O(n)$ operaciones de E/S.

Otro modo de realizar el análisis de los algoritmos OOC es utilizar una serie de primitivas, como el número de pasadas que se realizan sobre los datos o el número de trasposiciones de matrices que se han de realizar en disco. De este modo, es posible realizar comparaciones afirmando, por ejemplo, que un determinado algoritmo realiza dos pasadas sobre los datos y la trasposición de una matriz. En la práctica parecen más útiles este tipo de afirmaciones que el uso de expresiones asintóticas.

Una última forma de analizar y comparar algoritmos, que se utiliza con frecuencia en operaciones sobre matrices densas, es la *fracción de nivel 3 del algoritmo*: la fracción de operaciones aritméticas que se pueden realizar en términos de operaciones matriz-matriz (productos de matrices, factorizaciones y resoluciones de sistemas triangulares). Esta medida se utiliza porque las operaciones de nivel 3 se pueden planificar con un alto nivel de reutilización de datos y, por lo tanto, un algoritmo con una fracción alta de nivel 3 muy probablemente ofrecerá un buen rendimiento en un computador con cachés o como algoritmo OOC. El problema es que se mide la fracción del trabajo en que se puede disfrutar de una reutilización de datos, pero no se predice cuánta reutilización de datos se puede alcanzar realmente. En la literatura se encuentran algoritmos que tienen una fracción alta de nivel 3, pero en los que las operaciones matriz-matriz tienen una baja reutilización

de datos, por lo que se considera que estos algoritmos no serán eficientes cuando se ejecuten OOC. Por otra parte, la fracción de nivel 3 suele ser un buen indicador de las prestaciones de la caché, porque incluso un modesto nivel de reutilización de datos es suficiente para reducir la tasa de fallos de caché de manera significativa en muchos computadores (aunque esto está cambiando porque la penalización por fallo de caché ha aumentado significativamente en los últimos años).

1.5.3. Dependencia de la arquitectura

Los sistemas de memoria de los computadores actuales tienen una profunda jerarquía de memorias: registros, dos o tres niveles de caché, la memoria principal y los discos. En general se cumple que los algoritmos que han sido diseñados para explotar un nivel de la jerarquía de memoria, pueden ser también válidos para explotar los otros niveles, como el disco, dando lugar a algoritmos OOC eficientes. Sin embargo, esto no siempre es así, ya que la ratio entre el ancho de banda memoria/disco es bastante diferente de la ratio entre el ancho de banda caché/memoria, de manera que algunos algoritmos que explotan bien la caché no son eficientes como algoritmos OOC. Y, del mismo modo, algoritmos OOC eficientes dejan de serlo al trabajar *in-core*. Por ejemplo, un algoritmo obtenido mediante la técnica de transformación del flujo de datos puede realizar más trabajo pero ofrecer un mayor nivel de reutilización de datos, resultando ser un buen algoritmo OOC. En cambio, este mismo algoritmo puede resultar ineficiente *in-core*, porque lo que se gana usando la caché de manera eficiente, es un factor menor frente a la pérdida de eficiencia debido al trabajo extra realizado. Sin embargo, no se debe perder de vista el hecho de que la ratio entre el ancho de banda caché/memoria está aumentando, por lo que cada vez hay menos diferencia entre los algoritmos que explotan los primeros niveles de la jerarquía de memorias y los algoritmos OOC.

Por otra parte, los algoritmos que son eficientes en memoria distribuida no necesariamente se traducen bien en algoritmos OOC. Un ejemplo de ello son los algoritmos iterativos para resolver sistemas lineales. Muchos de estos algoritmos pueden explotar un gran número de procesadores y pueden organizarse para que la cantidad de comunicación interprocesador sea asintóticamente menor que la cantidad de trabajo. Esta alta ratio cómputos/comunicaciones garantiza que el algoritmo se ejecute eficientemente incluso cuando las comunicaciones son lentas. Aunque las comunicaciones con disco son también lentas, la gran diferencia entre estos dos entornos es que en memoria distribuida los datos no locales también están siendo actualizados (por otro procesador), mientras que en OOC no es así.

1.5.4. Abstracciones

Toledo afirma que las abstracciones resultan muy útiles cuando se trata de diseñar algoritmos OOC, ya que ayudan a descubrir, comprender e implementar planificaciones complejas, que permiten que los algoritmos se ejecuten eficientemente y en paralelo. Cuando se trata de algoritmos para matrices densas, se utilizan con frecuencia los algoritmos orientados a bloques. Cuando se trata de la transformada rápida de Fourier, los algoritmos OOC se expresan en función de múltiples transformadas sobre conjuntos de datos más pequeños y de una permutación estructurada.

Las planificaciones para la factorización de matrices dispersas se suelen expresar y generar utilizando árboles de eliminación, que son representaciones compactas de los grafos de flujo de datos de los algoritmos, que representan dependencias entre filas y/o columnas.

Hay otras abstracciones que no son tan populares, pero que también aparecen en la literatura, como la recursión. En principio esta técnica ha sido poco utilizada en el diseño de algoritmos

numéricos porque *Fortran* no la soportaba hasta hace poco tiempo y este lenguaje ha sido la base durante muchos años del desarrollo de bibliotecas numéricas. Sin embargo, Toledo muestra [62] una planificación recursiva de la factorización LU densa con pivotamiento parcial que es más eficiente que la planificación orientada a bloques. También se ha aplicado la recursión a otros problemas sobre matrices densas y para describir permutaciones en términos de operaciones matriciales sobre índices.

1.5.5. Algoritmos OOC para la resolución de sistemas lineales densos

Los métodos directos para la resolución de sistemas lineales densos del tipo $Ax = b$ (o del tipo $AX = B$, es decir, con múltiples vectores de términos independientes y soluciones) requieren un paso previo de factorización de la matriz de coeficientes A. La factorización de Cholesky es el método comúnmente utilizado para resolver sistemas lineales en los que la matriz A es simétrica y definida positiva. La factorización LU juega el mismo papel en sistemas lineales en los que la matriz A no presenta ninguna estructura particular. Por otra parte, la factorización QR se utiliza para sistemas lineales sobredeterminados.

Cuando se utiliza una implementación optimizada de LAPACK para realizar estas factorizaciones, como por ejemplo la de MKL, y la matriz del sistema no cabe en la memoria RAM, el sistema operativo hace uso de la memoria virtual, siendo ésta una primera solución para obtener algoritmos OOC. Sin embargo, el uso de la memoria virtual hace que, habitualmente, las prestaciones caigan de manera drástica, pues el gestor de la memoria virtual tiene como unidad básica de gestión la página, que en general no concuerda con el tamaño del bloque ni está alineada con éste, y además, en general, no es capaz de predecir el patrón de accesos que genera la aplicación.

En consecuencia, en estos años se han desarrollado diversas implementaciones OOC de estas factorizaciones que se revisan brevemente a continuación.

ScaLAPACK, una extensión paralela de LAPACK, proporciona una implementación OOC para arquitecturas paralelas con memoria distribuida de las tres factorizaciones [20]. Las rutinas OOC de ScaLAPACK particionan las matrices en paneles (también denominados *slabs*), por lo que inherentemente no son escalables: conforme aumenta el tamaño de la matriz a factorizar, el tamaño del panel que cabe en memoria es cada vez más estrecho, lo que afecta negativamente a las prestaciones de los núcleos computacionales que se utilizan para realizar los cálculos una vez el panel está en memoria. Es precisamente el estrechamiento del panel el que imposibilita el uso de rutinas de BLAS 3 y determina la falta de escalabilidad de esta aproximación.

SOLAR [64] es una biblioteca portable de rutinas de álgebra lineal para OOC, que se ofrece como una extensión de LAPACK y ScaLAPACK. Esta biblioteca utiliza las rutinas de ScaLAPACK para las operaciones que tienen lugar *in-core* y proporciona una capa para manejar la E/S. Esta capa permite conseguir mejores prestaciones, gracias a que utiliza un modo de almacenamiento distinto para las matrices en disco del que se utiliza en ScaLAPACK para las matrices en memoria.

Otra extensión para OOC es la que proporciona la biblioteca de paso de mensajes para operaciones de álgebra lineal PLAPACK [71], denominada POOCLAPACK [53]. Esta extensión se ha utilizado para realizar una implementación OOC para las factorizaciones de Cholesky y QR [35].

Todas las implementaciones citadas anteriormente corresponden a algoritmos cuya formulación se basa en bucles. Otro tipo de formulación para los algoritmos de álgebra lineal es la recursiva. Elmroth et al. [29] repasan los últimos avances realizados aplicando el paradigma de la recursión a la computación matricial densa. Un argumento a favor de la formulación recursiva es que mejora la

localidad de referencia, ya que si en las implementaciones *in-core* hay menos fallos de línea, entonces en las implementaciones OOC habrá menos E/S. El uso de la recursión ha llevado a utilizar nuevas estructuras de datos híbridas (recursivas, jerárquicas) que pueden encajar mejor con este tipo de algoritmos [5, 44, 29, 4]

Para el caso disperso, existen varios trabajos sobre implementaciones OOC para la factorización de Cholesky y la factorización LU [54, 31, 55, 3]. Y en cuanto a matrices complejas, Béreux [15] realiza una comparación de varias implementaciones OOC secuenciales para la factorización de Cholesky. Las implementaciones corresponden a algoritmos basados en bucles [35] y algoritmos recursivos [64], presentando estos últimos peores prestaciones que los primeros.

1.6. Computadores actuales

En esta sección se repasan brevemente las características de las arquitecturas de computadores actuales que han sido utilizadas para evaluar los algoritmos OOC diseñados e implementados en esta tesis.

1.6.1. Multiprocesadores

Los multiprocesadores con memoria compartida (o simplemente multiprocesadores) son plataformas compuestas por varios procesadores completos que disponen de una memoria común accesible mediante instrucciones *hardware* (en código máquina). 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 a cualquier posición de la memoria es uniforme.
- 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 el acceso a la memoria local de otro procesador.

Debido a la simplicidad del principio en el que están fundamentados (replicación completa de procesadores), los multiprocesadores fueron una de las primeras arquitecturas paralelas que se diseñaron. La facilidad de su programación, basada en el paradigma de variables compartidas (habitualmente, accesible mediante hebras de ejecución o herramientas de alto nivel para la 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 su escasa escalabilidad 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.

1.6.2. Procesadores multinúcleo

Los procesadores multinúcleo (*multicore processors* o *chip multiprocessors*) incluyen en un sólo chip varias unidades de proceso independientes, denominadas núcleos (*cores*). Cada núcleo es una unidad operacional completa, es decir, puede ser un procesador con técnicas sofisticadas de paralelismo y/o segmentación. Los problemas de disipación de calor y de consumo de energía de la

tecnología actual provocaron que, a partir de 2005, los principales fabricantes de procesadores comenzaran a incorporar 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 cuatro y ocho 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 [40, 41].
- Las previsiones para los procesadores multinúcleo apuntan a cientos de núcleos en un chip [13], 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 (Figura 1.4 izquierda) 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 (Figura 1.4 derecha) y tampoco en los procesadores multinúcleo cuando los datos residen en los niveles de memoria fuera del chip [42].

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 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.

Como comentamos al inicio de esta memoria, el modo de programación habitual en multiprocesadores y procesadores multinúcleo, basado en el uso de hebras, nos hace adoptar el término de arquitecturas multihebra para referirnos a ambas plataformas.

1.6.3. Procesadores gráficos y aceleradores hardware

Los procesadores gráficos (*Graphics Processing Units* o GPU) son arquitecturas que, en un principio, fueron diseñadas para el procesamiento de imágenes gráficas. En sus orígenes, estaban formados por un cauce segmentado cuyas etapas realizaban tareas fijas, explotando así el paralelismo a nivel de tareas y también el paralelismo a nivel de datos, ya que en cada etapa se trabajaba sobre varios datos a la vez. La evolución de los procesadores gráficos ha permitido que algunas de estas etapas sean programables por el usuario, por lo que actualmente no sólo son potentes motores gráficos sino que se han convertido también en procesadores programables altamente paralelos, con una capacidad de cálculo y un ancho de banda de memoria que superan ampliamente a las de cualquier CPU.

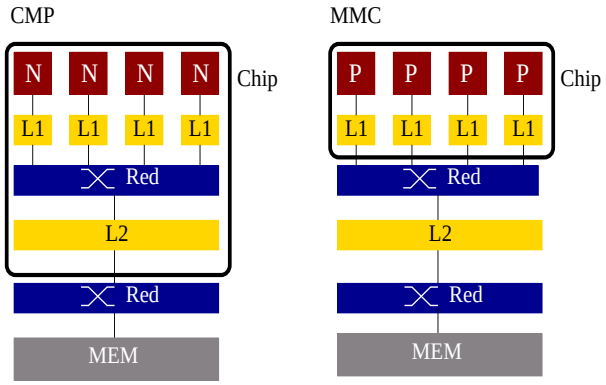


Figura 1.4: Organización habitual de un procesador multinúcleo (CMP, izquierda) y un multiprocesador (MMC, derecha). `N`=núcleo, `P`=procesador, `L1`=caché de primer nivel, `L2`=caché de segundo nivel, `MEM`=memoria principal. La línea gruesa marca los límites del chip.

El rápido avance, tanto en la programabilidad de los procesadores gráficos como en su flexibilidad, ha permitido utilizarlos para resolver un amplio rango de complejos problemas con altas necesidades computacionales. Es lo que se conoce como GPGPU (*General-Purpose Computing on the GPU*). Sin embargo, los procesadores gráficos no son la panacea, ya que están diseñados para aplicaciones con unas características muy concretas:

- Con altos requisitos computacionales.
- Con alto grado de paralelismo.
- En las que el rendimiento es más importante que la latencia.

Por otra parte, la programación ha sido, hasta hace bien poco, una tarea de expertos puesto que sólo se disponía de APIs gráficas, como OpenGL y DirectX.

En este sentido, los avances realizados en el modelo de programación y las herramientas de programación de los procesadores gráficos han resultado determinantes para su éxito. El reto de fabricantes e investigadores ha sido encontrar el equilibrio justo entre el acceso de bajo nivel al *hardware* para obtener buenas prestaciones, y los lenguajes de alto nivel y las herramientas que dan al programador flexibilidad y productividad, todo ello teniendo en cuenta los rápidos avances en el *hardware*. Actualmente existe un modelo unificado para las partes programables del cauce segmentado (*Shader Model 4.0*), y conforme el modelo evoluciona, va siendo más potente por lo que las aplicaciones para procesadores gráficos pueden ser más complejas.

Recientemente, tanto AMD como NVIDIA han puesto a disposición de los desarrolladores de GPGPU nuevos modelos de programación. Cabe destacar CUDA, de NVIDIA, que proporciona una sintaxis como la del lenguaje C, y que permite aprovechar dos niveles de paralelismo: a nivel de datos y multihebra. Además, CUDA deja visibles más recursos *hardware* que otros sistemas de programación y, aunque requiere que el programador conozca muchos detalles de bajo nivel del *hardware*, ofrece gran flexibilidad a cambio de una ganancia significativa en prestaciones.

La arquitectura de los procesadores gráficos se ha desarrollado en una dirección distinta a la arquitectura de las CPU, ya que el requisito en las GPUs es atender cálculos paralelos de gran di-

mensión y con énfasis en el rendimiento, no en la latencia. Es por ello que los procesadores gráficos están formados por un gran número de procesadores paralelos de grano fino. Por ejemplo, el procesador gráfico NVIDIA GEFORCE 9800 GX2, utilizado en la experimentación de esta tesis, contiene 256 procesadores multihebra distribuidos en dos GPUs, que cuentan con 16 multiprocesadores cada una, con 8 procesadores multihebra por multiprocesador. Cada multiprocesador contiene, además, cachés compartidas de instrucciones y de datos, lógica de control, una memoria compartida de 16KB y dos unidades funcionales especiales.

1.7. Análisis del rendimiento

En esta sección se tratan aspectos relativos al análisis del rendimiento de los algoritmos diseñados e implementados en el trabajo. En primer lugar se establece el modo en que se mide este rendimiento; a continuación, se repasan las características principales de las arquitecturas sobre las que se han realizado las experimentaciones; y, por último, se analiza el posible efecto del acceso a disco sobre las prestaciones.

1.7.1. 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 el *flop* (*floating-point arithmetic operation*) como una operación aritmética de coma flotante, 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 del tiempo de ejecución, la velocidad de procesamiento aritmético (flops/seg.) presenta una clara ventaja en la representación gráfica de resultados. Así, mientras que al crecer el tamaño del problema, el tiempo de ejecución sigue aumentando proporcionalmente (a menudo en una relación cúbica), 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 y la velocidad del bus del sistema, entre otros). 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 en FLOPS.

1.7.2. Entorno de experimentación

Las características de las arquitecturas utilizadas en la evaluación de las implementaciones realizadas en este trabajo se resumen en la Tabla 1.1. Dado que la eficiencia de las rutinas de BLAS es importante, en la Tabla 1.2 se muestran las versiones de estas rutinas que se han utilizado en cada plataforma.

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 con multiprocesadores, el

paralelismo se extrae ejecutando múltiples hebras, 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. Concretamente, explotamos el paralelismo a dos niveles. En un primer nivel utilizamos un par de hebras especializadas: una se encarga de la realizar E/S y la otra se encarga de realizar los cálculos. En el segundo nivel, la hebra encargada de los cálculos utiliza múltiples hebras de ejecución para realizarlos. En el caso de la GPU, el paralelismo se obtiene de la implementación de BLAS que proporciona NVIDIA denominada CUBLAS.

Plataforma	Arquitectura	Núcleos	Frecuencia (GHz)	Cache L2 (MBytes)	RAM (GBytes)	Discos	Sist. de ficheros
ROPE	INTEL PENTIUM 4	1	3		1	1	EXT3
TESLA	INTEL XEON	8	2	12	8	1	XFS
TESLA2	INTEL XEON	8	2,83	12	8	6	XFS
ZAPE	AMD PHENOM	4	2,2	4×512Kbytes	4	1	XFS
	NVIDIA GEFORCE 9800 GX2	2×128			1		

Tabla 1.1: Arquitecturas empleadas en la experimentación.

Plataforma	BLAS
ROPE	MKL 8.2
TESLA	MKL 10.0.1
TESLA2	MKL 10.0.1
ZAPE	MKL 10.0.1 CUBLAS 2.1

Tabla 1.2: Versiones de BLAS empleadas en la experimentación.

1.7.3. Efecto del acceso a disco

Puesto que en esta tesis se trabaja con matrices de gran dimensión que residen en disco, es importante realizar un análisis previo del efecto que tiene el tamaño de la matriz sobre la velocidad de acceso a los datos. La gráfica de la Figura 1.5 muestra este efecto cuando se realiza una operación que lee cada uno de los elementos de una matriz que reside en disco, incrementa su magnitud y los escribe de nuevo en el disco. Puesto que la matriz no cabe en memoria, las lecturas y escrituras de sus elementos se realizan en bloques de elementos de manera intercalada. Para facilitar estas operaciones se considera que la matriz original, de tamaño $n \times n$, está dividida en submatrices (los bloques) de tamaño $t \times t$, a las que denominamos *tiles*. Cada *tile* se almacena en un fichero distinto, de modo que es de esperar que sus elementos residan en posiciones cercanas del mismo. En la figura se observa que, para todas las arquitecturas sobre las que se trabaja, la velocidad de acceso decae a partir de un determinado tamaño de matriz, por lo que a partir de ese punto podemos esperar una caída en las prestaciones de los algoritmos OOC debido a que el acceso a disco pasa a ser excesivamente lento. Este comportamiento es muy similar para distintos tamaños de *tile*, obteniéndose prácticamente la misma velocidad asintótica cuando se hace crecer el tamaño

de la matriz. Un objetivo a perseguir será, por tanto, que este descenso de la velocidad de acceso al disco no se note en las prestaciones de los algoritmos implementados.

En cuanto al efecto que puede tener sobre las prestaciones el orden de acceso a los ficheros que contienen las submatrices, por filas de *tiles* o por columnas de *tiles*, en la gráfica de la Figura 1.6 se observa que este aspecto no influye en la velocidad de acceso. Dentro de cada *tile*, el acceso se organiza por columnas, en caso de una operación de BLAS 2 (sin reutilización de datos) o por bloques (si es posible reutilizar los datos).

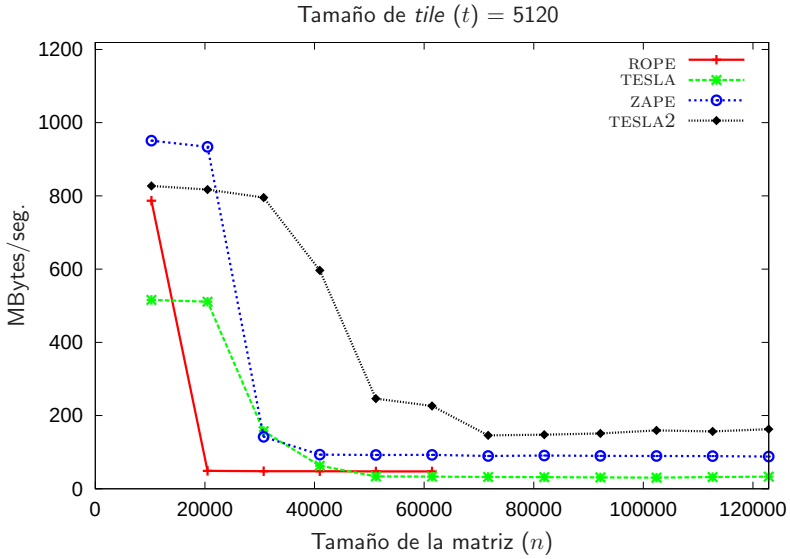


Figura 1.5: Efecto del acceso a disco sobre las prestaciones para distintos tamaños de matriz cuando se realiza una operación que incrementa el valor de todos los elementos de la matriz.

1.8. Notación

En esta sección se describe la notación que se ha utilizado en la tesis.

Un *vector* (columna) x de dimensión n es una *n-tupla* de números. Normalmente, los vectores se identifican por una letra del alfabeto latino, mientras que sus componentes, escalares todos ellos, se identifican 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 traspuesta de un vector columna:

$$x^{\mathrm{T}} = \begin{pmatrix} \chi_0, \chi_1, \ldots, \chi_{n-1} \end{pmatrix}$$

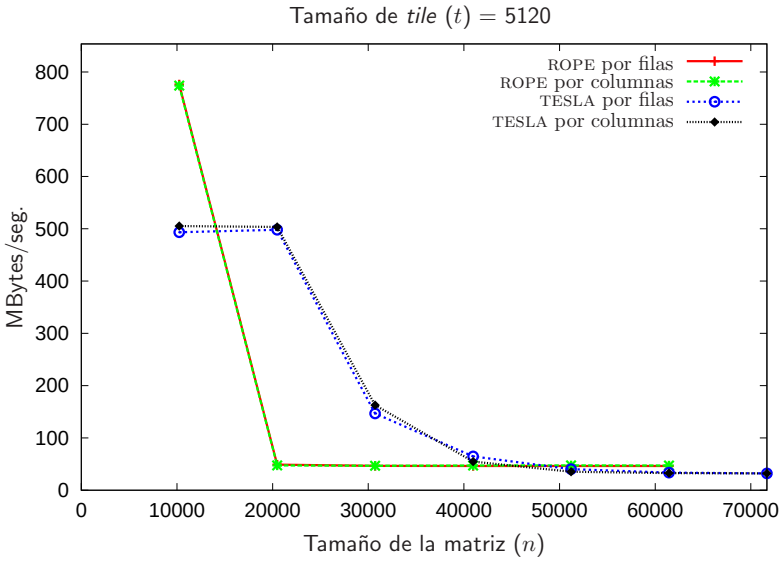


Figura 1.6: Efecto del acceso a disco sobre las prestaciones para distintos tamaños de matriz cuando se accede a los *tiles* por filas o por columnas.

Una *matriz* $A \in \mathbb{R}^{m \times n}$ está formada por n vectores columna o m vectores fila. Las matrices se identifican con una letra latina mayúscula, 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*.

La matriz I_n es la *matriz identidad* y se define como $I_n = (e_0, e_1, \ldots, e_{n-1})$, siendo $e_j \in \mathbb{R}^n$ el vector unitario que tiene el elemento j igual a 1 y el resto de elemenos nulos.

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) unidad* si $\alpha_{ij} = 0$ si $i > j$ $(j > i)$ y $\alpha_{ij} = 1$ si $i = j$.

Una matriz simétrica A es *definida positiva* si $x^{\mathrm{T}} A x > 0$ para todo vector x de dimensión conforme con las dimensiones de la matriz.

Una matriz $Q \in \mathbb{R}^{n \times n}$ es *ortogonal* si $Q^{\mathrm{T}} Q = Q Q^{\mathrm{T}} = I_n$.

$\|A\|$ denota la *norma matricial* de la matriz A, que puede instanciarse como la 1-norma, la 2-norma, la ∞-norma o la norma de Frobenius [32]. $\|x\|_2$ denota la *2-norma vectorial* de $x \in \mathbb{R}^n$ y se define como $\|x\|_2 = \sqrt{x_0^2 + x_1^2 + \cdots + x_{n-1}^2}$.

En lo que respecta a los algoritmos, los que se incluyen en el presente documento se expresan siguiendo la notación desarrollada en el proyecto FLAME, que ha sido presentada en el Apartado 1.4.2

Capítulo 2

Factorización de Cholesky en Disco

En este capítulo se aborda la factorización de Cholesky de una matriz que, por su tamaño, no es posible almacenar completamente en la memoria principal del sistema, por lo que reside inicialmente en disco. Esta operación de descomposición es clave en la resolución de sistemas de ecuaciones lineales cuando la matriz de coeficientes es simétrica y definida positiva. Nuestro trabajo analiza, desde el punto de vista teórico y práctico, la implementación *Out-of-Core* (OOC) de las tres variantes conocidas para el cálculo de esta factorización. Todos los algoritmos planteados realizan los cálculos sobre *tiles* (bloques cuadrados de dimensión regular), que constituyen la unidad de transferencia de datos entre disco y memoria principal, mejorando de este modo la eficiencia y la escalabilidad de la solución.

Tres son las arquitecturas destino de los algoritmos OOC desarrollados, reflejo de la gama de sistemas disponibles en la actualidad: una plataforma con un único procesador de propósito general, una máquina con varios procesadores y/o núcleos (*cores*) de propósito general que comparten una memoria común, y un sistema híbrido compuesto por un procesador multinúcleo de propósito general y un acelerador *hardware* de tipo GPU (*graphics processor unit* o procesador gráfico) con memorias separadas.

El capítulo está estructurado del modo siguiente. En la primera sección se define la factorización de Cholesky, se justifica su utilidad, y se ofrecen dos formulaciones recursivas, una escalar y otra por bloques, para su cálculo. En la Sección 2.2 se presentan las tres variantes algorítmicas escalares conocidas para el cálculo de la factorización de forma iterativa y las respectivas versiones por bloques. Las implementaciones de estas variantes en forma de algoritmos OOC se discuten en la Sección 2.3, que ilustra las aportaciones principales de la tesis mediante este primer ejemplo de operación matricial. Las Secciones 2.4–2.6 recogen la evaluación del impacto experimental que suponen las propuestas realizadas y, finalmente, la Sección 2.7 resume el contenido y aportaciones del capítulo.

2.1. Factorización de Cholesky

Una de las estrategias que se utilizan habitualmente para resolver sistemas de ecuaciones lineales densos comienza con la factorización de la matriz de coeficientes, de manera que ésta se descompone en el producto de dos matrices triangulares que se utilizan después para obtener la solución del sistema inicial resolviendo sendos sistemas triangulares. La factorización LU y la factorización de Cholesky son descomposiciones de este tipo.

La factorización LU, combinada con un pivotamiento parcial de filas durante el proceso de factorización, es el método comúnmente utilizado para resolver sistemas lineales generales del tipo $Ax = b$. La factorización de Cholesky juega el mismo papel en sistemas en los que la matriz A es, además, simétrica y definida positiva.

La siguiente definición establece las condiciones necesarias para que una matriz sea invertible, condición indispensable para poder calcular estas dos factorizaciones.

Definición 1 *Una matriz $A \in \mathbb{R}^{n\times n}$ es invertible si sus columnas son linealmente independientes y, por lo tanto, $Ax = 0$ si y sólo si $x = 0$, para todo $x \in \mathbb{R}^n$.*

Ya que este capítulo está dedicado a la factorización de Cholesky, es conveniente recordar la caracterización de las matrices simétricas definidas positivas.

Definición 2 *Una matriz simétrica $A \in \mathbb{R}^{n\times n}$ es definida positiva si y sólo si $x^{\mathrm{T}}Ax > 0$ para todo $x \in \mathbb{R}^n$ tal que $x \neq 0$.*

Las matrices simétricas definidas positivas tienen los elementos de mayor magnitud en la diagonal (se dice que tienen una diagonal pesada) y, en consecuencia, no necesitan pivotamiento cuando se factorizan [32]. De su propia definición se desprende que toda matriz simétrica definida positiva es una matriz invertible.

Una vez conocida la definición de matriz simétrica definida positiva, podemos enunciar el teorema que define la factorización de Cholesky.

Teorema 1 (Factorización de Cholesky) *Si una matriz $A \in \mathbb{R}^{n\times n}$ es simétrica y definida positiva, existe una única matriz triangular inferior $L \in \mathbb{R}^{n\times n}$, con elementos diagonales estrictamente positivos, tal que $A = LL^T$. A la matriz L se le denomina factor de Cholesky de A.*

La demostración de este teorema puede encontrarse en la bibliografía [32]. Existe un teorema análogo que define la factorización de Cholesky como $A = U^{\mathrm{T}}U$, con $U \in \mathbb{R}^{n\times n}$ triangular superior, aunque en esta tesis optamos por la forma triangular inferior del factor de Cholesky.

Una vez factorizada la matriz en el producto $A = LL^T$, la solución del sistema lineal $Ax = b$ puede obtenerse resolviendo el sistema triangular inferior $Ly = b$, seguido de la resolución del sistema triangular superior $L^{\mathrm{T}}x = y$. La resolución de estos sistemas triangulares cuando la matriz reside en disco no se aborda explícitamente en nuestro trabajo al presentar un coste de orden menor frente al de la propia factorización; además, la operación de resolución de sistemas triangulares aparece implícitamente como una suboperación más dentro del procedimiento de cálculo de la factorización de Cholesky.

2.1.1. Algoritmo escalar

A continuación se obtiene, mediante un proceso de derivación formal, un algoritmo para calcular la factorización de Cholesky de una matriz $A \in \mathbb{R}^{n\times n}$ simétrica definida positiva.

Consideremos la siguiente partición de la matriz de coeficientes densa A y su factor de Cholesky L con estructura triangular inferior:

$$A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right), \quad L = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right),$$

siendo α_{11} y λ_{11} escalares, a_{21} y l_{21} vectores de $n-1$ elementos, y A_{22} y L_{22} matrices de tamaño $(n-1)\times(n-1)$ (el símbolo $\star$ en la expresión anterior denota la parte simétrica de A, que no se

referenciará durante el proceso de derivación). A partir de $A = LL^{\mathrm{T}}$, se tiene entonces que:

$$\left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right)\left(\begin{array}{c|c} \lambda_{11} & l_{21}^{\mathrm{T}} \\ \hline 0 & L_{22}^{\mathrm{T}} \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11}^2 & \star \\ \hline \lambda_{11} l_{21} & l_{21} l_{21}^{\mathrm{T}} + L_{22} L_{22}^{\mathrm{T}} \end{array}\right),$$

de donde se derivan las siguientes expresiones:

$$\begin{array}{rcl} \lambda_{11} & = & \sqrt{\alpha_{11}}, \\ l_{21} & = & a_{21}/\lambda_{11}, \\ A_{22} - l_{21} l_{21}^{\mathrm{T}} & = & L_{22} L_{22}^{\mathrm{T}}. \end{array}$$

Se obtiene así una formulación recursiva de un algoritmo escalar para calcular la factorización de Cholesky, en la que se sobreescriben los elementos de la parte triangular inferior de A con los del factor de Cholesky L:

1. Particionar $A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right)$.
2. $\alpha_{11} := \lambda_{11} = \sqrt{\alpha_{11}}$.
3. $a_{21} := l_{21} = a_{21}/\lambda_{11}$.
4. $A_{22} := A_{22} - l_{21} l_{21}^{\mathrm{T}}$.
5. Continuar recursivamente con $A = A_{22}$ en el paso 1.

Este algoritmo se dice que está orientado hacia la derecha (*right–looking*) porque, una vez calculado un nuevo elemento de la diagonal del factor de Cholesky (λ_{11}), las correspondientes actualizaciones se aplican a la submatriz que queda a la derecha y por debajo del elemento calculado (es decir, a a_{21} y A_{22}).

Tal y como está formulado el algoritmo, sólo es necesario almacenar los elementos de la parte triangular inferior de la matriz simétrica A. Además, estos mismos elementos de la matriz se sobreescriben durante los cálculos con el resultado L. Por lo tanto, en el cálculo $A_{22} := A_{22} - l_{21} l_{21}^{\mathrm{T}}$ sólo es necesario actualizar la parte triangular inferior de A_{22}, operación a la que se conoce como una *actualización simétrica de rango 1*. De este modo se reduce a la mitad el coste computacional del algoritmo, que requiere un total de $n^3/3$ operaciones aritméticas. (Por simplicidad, en todas las expresiones de coste aritmético/computacional de los algoritmos, descartaremos los términos de orden menor.)

El mayor volumen de cálculo del algoritmo anterior está en la actualización simétrica de rango 1, que realiza $O(n^2)$ operaciones aritméticas sobre $O(n^2)$ datos. Esta relación entre volúmenes de cálculo y de datos no permite obtener altas prestaciones, ya que por cada tres accesos a memoria se realizan aproximadamente dos operaciones aritméticas (una resta y un producto). Para mejorar el rendimiento, se debe reformular el algoritmo en términos de productos de matrices (o actualizaciones de rango k), pues de este modo se incrementa la relación entre el número de operaciones aritméticas y el de datos (por cada acceso a memoria se hacen más operaciones aritméticas), siendo posible una mayor reutilización de los datos situados en los niveles de la caché más cercanos al procesador. Este objetivo es el que persigue precisamente el algoritmo por bloques que se presenta a continuación.

2.1.2. Algoritmo por bloques

Para obtener un algoritmo orientado a bloques, y que por tanto se base en operaciones matriz–matriz, podemos realizar el siguiente particionado de las matrices A y L:

$$A = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right), \quad L = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right),$$

donde A_{11} y L_{11} son matrices de tamaño $b \times b$ (con b mucho menor que el tamaño de A), y A_{22} y L_{22} matrices con $(n-b) \times (n-b)$ elementos. De $A = LL^{\mathrm{T}}$, con el particionado anterior tenemos que:

$$\left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} L_{11}^{\mathrm{T}} & L_{21}^{\mathrm{T}} \\ \hline 0 & L_{22}^{\mathrm{T}} \end{array}\right) = \left(\begin{array}{c|c} L_{11}L_{11}^{\mathrm{T}} & \star \\ \hline L_{21}L_{11}^{\mathrm{T}} & L_{21}L_{21}^{\mathrm{T}} + L_{22}L_{22}^{\mathrm{T}} \end{array}\right),$$

de donde se obtienen las expresiones:

$$\begin{array}{rcl} L_{11}L_{11}^{\mathrm{T}} & = & A_{11}, \\ L_{21} & = & A_{21}L_{11}^{-\mathrm{T}}, \\ A_{22} - L_{21}L_{21}^{\mathrm{T}} & = & L_{22}L_{22}^{\mathrm{T}}. \end{array}$$

El algoritmo por bloques recursivo que calcula el factor de Cholesky de A, sobreescribiendo con éste los correspondientes elementos de la parte triangular inferior de A, puede entonces formularse como sigue:

1. Particionar $A = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right)$, siendo A_{11} de tamaño $b \times b$ (o menor, si A no es suficientemente grande).
2. $A_{11} := L_{11}$ tal que $A_{11} = L_{11}L_{11}^{\mathrm{T}}$. (Es decir, obtener el factor de Cholesky de A_{11}.)
3. $A_{21} := L_{21} = A_{21}L_{11}^{-\mathrm{T}}$.
4. $A_{22} := A_{22} - L_{21}L_{21}^{\mathrm{T}}$.
5. Continuar recursivamente con $A = A_{22}$ en el paso 1.

Este algoritmo por bloques, al igual que el algoritmo escalar, está orientado hacia la derecha: tras factorizar el bloque diagonal A_{11}, se actualizan los bloques que yacen abajo y a la derecha de éste. De nuevo sólo es necesario que A contenga los datos del triángulo inferior de la matriz. De este modo, el cálculo del factor de Cholesky del bloque diagonal, $A_{11} := L_{11}$, y la actualización $A_{22} := A_{22} - L_{21}L_{21}^{\mathrm{T}}$ sólo se deben realizar sobre la parte correspondiente a dicho triángulo. El coste de esta formulación por bloques es también de $n^3/3$ operaciones aritméticas.

Este modo de plantear los algoritmos por bloques es el que se ha utilizado tradicionalmente para resolver problemas matriciales de álgebra lineal, como por ejemplo se hace en las rutinas de LAPACK [6]. En el algoritmo que nos ocupa, el cálculo del panel A_{21} y la actualización de la submatriz A_{22} se pueden realizar con rutinas de BLAS 3. La mayor parte de los cálculos se realizan, precisamente, en esta última operación, una actualización simétrica de rango b, que conlleva $O(n^2b)$ operaciones sobre $O(n^2)$ datos. Si $b \gg 1$, mediante el algoritmo por bloques se consigue, por tanto, aumentar la relación entre operaciones y datos, aumentando en consecuencia la reutilización de éstos.

2.2. Algoritmos en notación FLAME

Es conocido que existen distintas variantes algorítmicas para calcular la factorización de Cholesky, como sucede para muchas otras operaciones de álgebra lineal [32]. La Figura 2.1 muestra las tres variantes algorítmicas existentes, escalares (izquierda) y por bloques (derecha), para calcular la factorización de Cholesky de una matriz A simétrica definida positiva. Estas variantes se han obtenido utilizando la metodología de derivación formal de algoritmos FLAME. Siguiendo la práctica habitual, se sobreescribe la parte triangular inferior de A con el factor de Cholesky L, quedando intacta la parte triangular superior estricta. En las variantes por bloques, la expresión $A := \{L\backslash A\} = \text{CHOL\_UNB}(A)$ indica que el triángulo inferior del bloque A se sobreescribe con su factor de Cholesky L, no alterándose la parte estrictamente triangular superior de la matriz. Para obtener esta factorización podría utilizarse el propio algoritmo por bloques con un tamaño de bloque menor (formulación recursiva), alguna de las otras variantes por bloques, también con un tamaño de bloque menor, o cualquiera de los algoritmos escalares. Esta última opción es la escogida en la Figura 2.1. En la práctica, los algoritmos por bloques utilizan un tamaño de bloque algorítmico $\bar{b}$ definido por el usuario que, en cada iteración, se fija en b como el mínimo entre $\bar{b}$ y el tamaño de A_{BR} (el bloque que queda por factorizar y del cual se extrae A_{11} en esta iteración).

Con cualquiera de las variantes de la figura, el coste del cálculo del factor de Cholesky es de $n^3/3$ operaciones aritméticas.

A continuación revisamos la notación FLAME usando la variante 1 del algoritmo por bloques para el cálculo de la factorización de Cholesky. Una característica importante de esta notación es que los algoritmos reflejan, casi de manera gráfica, las operaciones que se realizan. En la Figura 2.2 se muestran las acciones que tienen lugar en cada uno de los pasos del algoritmo por bloques correspondiente a la variante 1 de la Figura 2.1. En la figura se ilustra también el particionado y reparticionado de la matriz.

Inicialmente se particiona la matriz A en cuatro bloques: A_{TL}, A_{TR}, A_{BL} y A_{BR}, estando los tres primeros vacíos (el tamaño de A_{TL} es 0×0 y, por tanto, A_{TR} tiene cero filas y A_{BL} cero columnas). De este modo, al comienzo de la primera iteración del bucle, el bloque A_{BR} está formado por todos los elementos de A.

En cada iteración, el bloque A_{BR} se divide a su vez en otros cuatro bloques, A_{11}, A_{21}, A_{12} y A_{22}, tal y como se muestra en la Figura 2.2. Los elementos del factor L correspondientes a los dos primeros de estos bloques, A_{11} y A_{21}, son calculados en la iteración actual, y después A_{22} es actualizado. De esta forma se sobreescriben el triángulo inferior del bloque A_{11} y todo el contenido de A_{21}, con los correspondientes elementos del factor de Cholesky L. Las operaciones que se realizan son las que forman el cuerpo del bucle:

$$\begin{array}{lrcl} & A_{11} & := & \{L\backslash A\}_{11} = \text{CHOL\_UNB}(A_{11}), \\ (\text{TRSM}) & A_{21} & := & A_{21}\,\text{TRIL}(A_{11})^{-\text{T}}, \\ (\text{SYRK}) & A_{22} & := & A_{22} - A_{21}A_{21}^{\text{T}}. \end{array} \tag{2.1}$$

Junto a las dos últimas operaciones, y entre paréntesis, se ha indicado el nombre de la rutina BLAS que realiza la operación que aparece a su derecha. La función $\text{TRIL}(\cdot)$ devuelve la parte triangular inferior de la matriz que se le pasa como parámetro. Su utilización indica, en nuestro caso, que la operación $A_{21} := A_{21}\text{TRIL}(A_{11})^{-\text{T}}$ corresponde a la resolución de un sistema triangular. Cabe recordar que solamente se actualiza la parte triangular inferior del bloque A_{22}.

Algorithm: $A := \text{CHOL\_UNB}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**

Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^{\mathrm{T}} & \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$
where α_{11} is 1×1

Variante 1:
$\alpha_{11} := \sqrt{\alpha_{11}}$
$a_{21} := a_{21}/\alpha_{11}$
$A_{22} := A_{22} - a_{21}a_{21}^{\mathrm{T}}$

Variante 2:
$a_{10}^{\mathrm{T}} := a_{10}^{\mathrm{T}}\text{TRIL}(A_{00}^{-\mathrm{T}})$
$\alpha_{11} := \alpha_{11} - a_{10}a_{10}^{\mathrm{T}}$
$\alpha_{11} := \sqrt{\alpha_{11}}$

Variante 3:
$\alpha_{11} := \alpha_{11} - a_{10}a_{10}^{\mathrm{T}}$
$\alpha_{11} := \sqrt{\alpha_{11}}$
$a_{21} := a_{21} - A_{20}a_{10}^{\mathrm{T}}$
$a_{21} := a_{21}/\alpha_{11}$

Continue with
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^{\mathrm{T}} & \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$
endwhile

Algorithm: $A := \text{CHOL\_BLK}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$
where A_{11} is $b \times b$

Variante 1:
$A_{11} := \{L\backslash A\}_{11} = \text{CHOL\_UNB}(A_{11})$
$A_{21} := A_{21}\text{TRIL}(A_{11})^{-\mathrm{T}}$
$A_{22} := A_{22} - A_{21}A_{21}^{\mathrm{T}}$

Variante 2:
$A_{10} := A_{10}\text{TRIL}(A_{00})^{-\mathrm{T}}$
$A_{11} := A_{11} - A_{10}A_{10}^{\mathrm{T}}$
$A_{11} := \{L\backslash A\}_{11} = \text{CHOL\_UNB}(A_{11})$

Variante 3:
$A_{11} := A_{11} - A_{10}A_{10}^{\mathrm{T}}$
$A_{11} := \{L\backslash A\}_{11} = \text{CHOL\_UNB}(A_{11})$
$A_{21} := A_{21} - A_{20}A_{10}^{\mathrm{T}}$
$A_{21} := A_{21}\text{TRIL}(A_{11})^{-\mathrm{T}}$

Continue with
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$
endwhile

Figura 2.1: Variantes de los algoritmos escalares (izquierda) y por bloques (derecha) para calcular la factorización de Cholesky.

Al finalizar la iteración actual, se avanza en la matriz recuperando el particionado 2×2 sobre la misma, quedando el nuevo bloque A_{BR} formado únicamente por los elementos que componían A_{22}. De esta forma, en la siguiente iteración las operaciones sobre la matriz se aplican de nuevo sobre el bloque A_{BR} que se acaba de definir.

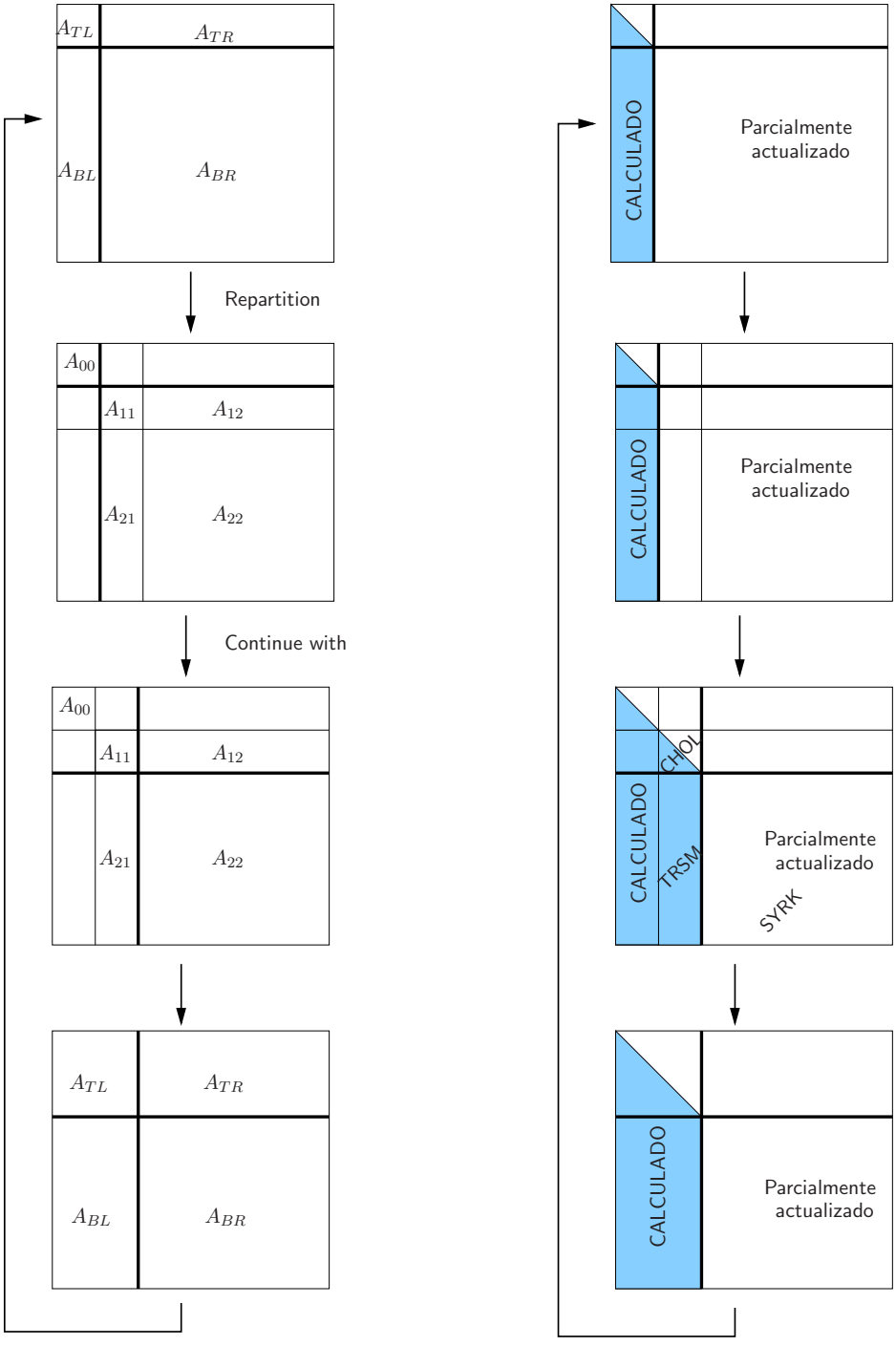


Figura 2.2: Desarrollo de la variante 1 del algoritmo por bloques de la Figura 2.1.

Algorithm: $A := \text{CHOL\_BLK\_VAR1}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$A_{11} := \{L \backslash A\}_{11} = \text{CHOL\_UNB}(A_{11})$

$A_{21} := A_{21}\text{TRIL}(A_{11})^{-\text{T}}$

$A_{22} := A_{22} - A_{21}A_{21}^{\text{T}}$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

```
FLA_Error FLA_Chol_blk_var1( 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_Chol_unb_var1( 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 2.3: Variante 1 del algoritmo por bloques para calcular la factorización de Cholesky (izquierda) e implementación FLAME/C correspondiente (derecha).

El algoritmo termina cuando el bloque A_{BR} queda vacío, es decir, cuando el número de filas del bloque A_{TL} coincide con el número de filas de la matriz A. De esta manera, el bucle del algoritmo se repite mientras $m(A_{TL}) < m(A)$, donde $m(\cdot)$ denota el número de filas de una matriz.

A partir de los algoritmos así diseñados es sencillo obtener código ejecutable en lenguaje C mediante la API FLAME/C. La Figura 2.3 muestra la variante 1 del algoritmo por bloques para la factorización de Cholesky (izquierda) y su implementación usando las rutinas de FLAME/C (derecha). En esta implementación, el particionado inicial de A para obtener A_{TL}, A_{TR}, A_{BL} y A_{BR} se realiza mediante la rutina `FLA_Part_2x2`, mientras que el reparticionado con el que comienza cada iteración se lleva a cabo mediante la rutina `FLA_Repart_2x2_to_3x3`, que obtiene los bloques A_{11}, A_{21}, A_{12} y A_{22} a partir de A_{BR}. El tamaño de bloque definido por el usuario viene dado por `nb_alg` y se utiliza en cada iteración para determinar el tamaño de bloque `b` como el mínimo entre dicho valor y el número de filas de A_{BR}, que se obtiene mediante `FLA_Obj_length`. A continuación se llevan a cabo los cálculos. En primer lugar se realiza la factorización de Cholesky del bloque A_{11} mediante la llamada a la rutina `FLA_Chol_unb_var1`. Esta rutina implementa mediante

FLAME/C el algoritmo escalar correspondiente a la variante 1 de la Figura 2.1. Seguidamente se realizan las actualizaciones de los bloques A_{21} y A_{22} mediante las rutinas `FLA_Trsm` y `FLA_Syrk`, respectivamente. Estas dos rutinas pertenecen a la biblioteca `libflame` (Sección 1.4 del Capítulo 1) e implementan las operaciones de BLAS TRSM y SYRK, con la particularidad de que los parámetros son objetos de FLAME. Para avanzar en la matriz, se recupera el particionado 2×2 al final de cada iteración del bucle mediante la rutina `FLA_Cont_with_3x3_to_2x2`.

2.2.1. Algoritmos FLAME orientados a bloques

En el cuerpo del bucle de la variante 1 del algoritmo por bloques, mostrado en la Figura 2.3, se trabaja sobre bloques, aunque no todos ellos son de las mismas dimensiones. Sin embargo, por simplicidad, consideraremos de ahora en adelante que el tamaño de la matriz n es un múltiplo exacto del tamaño de bloque b. Otro modo de plantear algoritmos por bloques es concebir el bloque como la unidad de cálculo, de manera que todas las operaciones involucren a bloques del mismo tamaño, $b \times b$. Denominaremos a éstos como *algoritmos orientados a bloques* para distinguirlos de los tradicionales algoritmos por bloques, presentados hasta el momento. Mediante este tipo de algoritmos se puede obtener un mayor grado de paralelismo y además se mejora la localidad de referencia [52]. Este tipo de planteamiento es también el más adecuado cuando se trata de trabajar con matrices en disco, ya que, como veremos más adelante, es fácil plantear una equivalencia entre el bloque y la unidad de almacenamiento en disco, conocida como *tile*, y en consecuencia entre los algoritmos orientados a bloques y los algoritmos OOC.

FLAME proporciona la API FLASH [44] que explota el encapsulamiento que FLAME/C hace de la información de la matriz en un objeto. Esto facilita trabajar con matrices de matrices, es decir, matrices cuyos elementos son, a su vez, matrices, estableciéndose una jerarquía de matrices de varios niveles.

La Figura 2.4 muestra, a la izquierda, la implementación correspondiente a la variante 1 del algoritmo orientado a bloques para la factorización de Cholesky, utilizando la API FLASH. Esta implementación es muy similar a la del algoritmo por bloques, realizada utilizando la API de FLAME/C, que se mostró en la Figura 2.3 (derecha). La diferencia fundamental entre ambas reside en la función de repartición `FLA_Repart_2x2_to_3x3`: en el caso de FLAME/C el tamaño para `A11` es $b \times b$ (contiene b^2 elementos escalares), mientras que en FLASH el bloque `A11` es 1×1, ya que es un elemento "escalar" de una matriz de matrices.

En la implementación orientada a bloques, las rutinas `FLASH_Trsm` y `FLASH_Syrk` deben definirse de manera que descompongan los cálculos en operaciones sobre bloques de dimensión $b \times b$, tal y como se muestra en las Figuras 2.4 (derecha) y 2.5.

Para ilustrar el funcionamiento de la implementación orientada a bloques se desarrolla en detalle un ejemplo para una matriz formada por 4×4 submatrices:

$$A = \begin{pmatrix} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix}.$$

```
FLA_Error FLASH_Chol_by_blocks_var1( 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_Chol_blk_var1( 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 )
/* Special case with mode parameters
     FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                 FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                 ...                              )
   Assumption: L consists of one block and
               B consists of a column of blocks     */
{
  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 2.4: Implementación FLASH de la variante 1 orientada a bloques de la factorización de Cholesky (izquierda) e implementación FLASH de la función `FLASH_Trsm` utilizada en este algoritmo (derecha).

El bucle principal de la rutina `FLASH_Chol_by_blocks_var1` realizará cuatro iteraciones y, al principio de cada iteración, el particionado contendrá los bloques que se muestran a continuación:

$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \begin{array}{cc} \text{1ª iteración} & \text{2ª iteración} \\ \left(\begin{array}{c|ccc} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right), & \left(\begin{array}{c|ccc} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right), \\ \text{3ª iteración} & \text{4ª iteración} \\ \left(\begin{array}{cc|cc} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \hline \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right), & \left(\begin{array}{ccc|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right). \end{array}$$

Utilizando la notación algorítmica de la Figura 2.3 (izquierda), las siguientes expresiones especifican las operaciones que se realizan sobre los bloques de la matriz en la primera iteración del bucle:

```
void FLASH_Syrk_ln( FLA_Obj alpha, FLA_Obj A,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                 ...                                   )
   Assumption: A is a column of blocks (column panel) */
{
  FLA_Obj AT,      A0,      CTL,   CTR,      C00, C01, C02,
          AB,      A1,      CBL,   CBR,      C10, C11, C12,
                   A2,                       C20, C21, C22;

  FLA_Part_2x1( A,    &AT,
                      &AB,    0, FLA_LEFT );
  FLA_Part_2x2( C,    &CTL, &CTR,
                      &CBL, &CBR,     0, 0, FLA_TL );

  while ( FLA_Obj_length( AL ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x1_to_3x1( AT,       &A0,
                         /* ** */   /* ** */
                                    &A1,
                           AB,       &A2,    1, FLA_BOTTOM );
    FLA_Repart_2x2_to_3x3(
       CTL, /**/ CTR,       &C00, /**/ &C01, &C02,

    /* ************* */   /* ******************** */
                           &C10, /**/ &C11, &C12,
       CBL, /**/ CBR,       &C20, /**/ &C21, &C22,
       1, 1, FLA_BR );
    /*----------------------------------------------*/
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              alpha, FLASH_MATRIX_AT( A1 ),
              beta,  FLASH_MATRIX_AT( C11 ) );
    FLASH_Gepb( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                alpha, A2,
                       A1,
                beta,  C21 );
    /*----------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,        A0,
                                          A1,
                            /* ** */   /* ** */
                              &AB,        A2,    FLA_TOP );
    FLA_Cont_with_3x3_to_2x2(
       &CTL, /**/ &CTR,       C00, C01, /**/ C02,
    /* ************** */  /* ****************** */
       &CBL, /**/ &CBR,       C20, C21, /**/ C22,
       FLA_TL );
  }
}
```

```
void FLASH_Gepb_nt( FLA_Obj alpha, FLA_Obj A,
                                   FLA_Obj B,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Gepb( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                 ...                             )
   Assumption: B is a block and
               A, C are columns of blocks (column panels) */
{
  FLA_Obj AT,      A0,     CT,      C0,
          AB,      A1,     CB,      C1,
                   A2,              C2;

  FLA_Part_2x1( A,    &AT,
                      &AB,    0, FLA_TOP );
  FLA_Part_2x1( C,    &CT,
                      &CB,    0, FLA_TOP );

  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x1_to_3x1( AT,       &A0,
                         /* ** */   /* ** */
                                    &A1,
                           AB,       &A2,    1, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( CT,       &C0,
                         /* ** */   /* ** */
                                    &C1,
                           CB,       &C2,    1, FLA_BOTTOM );
    /*----------------------------------------------*/
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
              alpha, FLASH_MATRIX_AT( A1 ),
                     FLASH_MATRIX_AT( B ),
              beta,  FLASH_MATRIX_AT( C1 ) );
    /*----------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,        A0,
                                          A1,
                            /* ** */   /* ** */
                              &AB,        A2,    FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &CT,        C0,
                                          C1,
                            /* ** */   /* ** */
                              &CB,        C2,    FLA_TOP );
  }
}
```

Figura 2.5: Implementación FLASH de la función `FLASH_Syrk` utilizada en el algoritmo orientado a bloques de la variante 1 de la factorización de Cholesky.

$$A_{11} := \{L\backslash A\}_{11} = \textsc{Chol}(A_{11}) \quad \equiv \quad \bar{A}_{00} := \{L\backslash \bar{A}\}_{00} = \textsc{Chol}(\bar{A}_{00}), \tag{2.2}$$

$$A_{21} := A_{21}\textsc{tril}(A_{11})^{-\mathrm{T}} \quad \equiv \quad \begin{pmatrix} \bar{A}_{10} \\ \bar{A}_{20} \\ \bar{A}_{30} \end{pmatrix} := \begin{pmatrix} \bar{A}_{10} \\ \bar{A}_{20} \\ \bar{A}_{30} \end{pmatrix} \textsc{tril}(\bar{A}_{00})^{-\mathrm{T}}, \tag{2.3}$$

$$\begin{aligned} A_{22} := A_{22} - A_{21}A_{21}^{\mathrm{T}} \quad \equiv \quad & \begin{pmatrix} \bar{A}_{11} & \star & \star \\ \bar{A}_{21} & \bar{A}_{22} & \star \\ \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix} := \begin{pmatrix} \bar{A}_{11} & \star & \star \\ \bar{A}_{21} & \bar{A}_{22} & \star \\ \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix} \\ & - \begin{pmatrix} \bar{A}_{10} \\ \bar{A}_{20} \\ \bar{A}_{30} \end{pmatrix} \begin{pmatrix} \bar{A}_{10} \\ \bar{A}_{20} \\ \bar{A}_{30} \end{pmatrix}^{\mathrm{T}}. \end{aligned} \tag{2.4}$$

Puesto que la operación (2.3) trabaja sobre un panel de bloques, se ha desarrollado la rutina `FLASH_Trsm` (ver la parte derecha en la Figura 2.4), de manera que las operaciones básicas se realicen sobre bloques y sea también así un algoritmo orientado a bloques. Cuando se utiliza esta rutina para resolver el sistema triangular en (2.3), que tiene la forma $B := BL^{-\mathrm{T}}$, se llevan a cabo tres iteraciones del bucle:

$$\left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right) = \overset{1^{\mathrm{a}}\text{ iteración}}{\left(\begin{array}{c} \overline{\bar{A}_{10}} \\ \hline \bar{A}_{20} \\ \bar{A}_{30} \end{array}\right)}, \quad \overset{2^{\mathrm{a}}\text{ iteración}}{\left(\begin{array}{c} \bar{A}_{10} \\ \hline \bar{A}_{20} \\ \hline \bar{A}_{30} \end{array}\right)}, \quad \overset{3^{\mathrm{a}}\text{ iteración}}{\left(\begin{array}{c} \bar{A}_{10} \\ \bar{A}_{20} \\ \hline \bar{A}_{30} \end{array}\right)}.$$

Las siguientes expresiones muestran las operaciones sobre bloques de la matriz que se llevan a cabo en cada una de estas tres iteraciones:

$$B_1 := B_1L^{-\mathrm{T}} \quad \equiv \quad \bar{A}_{10} := \bar{A}_{10}\textsc{tril}(\bar{A}_{00})^{-\mathrm{T}} \quad 1^{\mathrm{a}}\text{ iteración}, \tag{2.5}$$

$$B_1 := B_1L^{-\mathrm{T}} \quad \equiv \quad \bar{A}_{20} := \bar{A}_{20}\textsc{tril}(\bar{A}_{00})^{-\mathrm{T}} \quad 2^{\mathrm{a}}\text{ iteración}, \tag{2.6}$$

$$B_1 := B_1L^{-\mathrm{T}} \quad \equiv \quad \bar{A}_{30} := \bar{A}_{30}\textsc{tril}(\bar{A}_{00})^{-\mathrm{T}} \quad 3^{\mathrm{a}}\text{ iteración}. \tag{2.7}$$

Del mismo modo, y puesto que la actualización de rango b en (2.4) involucra a una submatriz de bloques, se han desarrollado las rutinas orientadas a bloques de la Figura 2.5.

Hemos utilizado la variante 1 del algoritmo que calcula la factorización de Cholesky para mostrar cómo pueden descomponerse las diferentes operaciones del algoritmo de modo que éstas operen sobre los componentes de una matriz de submatrices. Esta misma transformación puede desarrollarse para las variantes 2 y 3 de la factorización. De esta forma obtendríamos algoritmos orientados a bloques para todas la variantes. A continuación se establece la relación entre los algoritmos orientados a bloques y los algoritmos OOC.

2.3. Implementaciones OOC para la factorización de Cholesky

Transformar los algoritmos orientados a bloques en algoritmos OOC simplemente requiere considerar que cada bloque es un *tile* (bloque cuadrado en disco), e insertar en el código las operaciones

de entrada/salida (E/S) necesarias para leer/escribir los *tiles* de/en disco. En particular, resulta interesante almacenar las matrices en disco de modo que los elementos del mismo *tile* se encuentren en posiciones próximas entre sí en el disco. Esto puede conseguirse, por ejemplo, si cada *tile* se almacena en un fichero distinto (salvo fragmentación). Si la unidad básica de operación es el *tile*, con los algoritmos orientados a bloques resulta sencillo saber, en cada momento y sin manejo de subíndices, qué *tiles* se deben leer de disco para trabajar con ellos, escribiendo después aquellos que hayan sido actualizados.

En esta sección se presentan las distintas implementaciones desarrolladas para calcular la factorización de Cholesky OOC. Cada implementación introduce una nueva aportación que da título al apartado en que se presenta. Estas aportaciones suponen mejoras con mayor o menor impacto, como se verá en las secciones dedicadas a los experimentos, siendo siempre nuestro objetivo que las prestaciones de las implementaciones OOC igualen a las prestaciones de la implementación *in-core*. Si esto se consigue, el tamaño del problema sólo estará restringido por el tamaño del disco, sin que el rendimiento se vea afectado negativamente por los accesos a disco que requiere el trabajo OOC.

2.3.1. Variantes algorítmicas

En la Figura 2.1 se mostraban las tres variantes algorítmicas para la factorización de Cholesky. En este apartado se realiza un estudio de la implementación OOC de los correspondientes algoritmos orientados a bloques. A estas implementaciones se las denominará a partir de ahora *variantes tradicionales*, para distinguirlas de las sucesivas mejoras que les iremos incorporando. Para las implementaciones OOC tradicionales se ha desarrollado una pequeña API que permite gestionar los objetos OOC y la E/S, y que aparece descrita en el Apéndice A.

Consideremos que la matriz a factorizar A es una matriz de 7×7 submatrices, donde cada una de estas submatrices es un *tile*:

$$A = \begin{pmatrix} \bar{A}_{00} & \bar{A}_{01} & \dots & \bar{A}_{06} \\ \bar{A}_{10} & \bar{A}_{11} & \dots & \bar{A}_{16} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{60} & \bar{A}_{61} & \dots & \bar{A}_{63} \end{pmatrix}.$$

Cada *tile* se almacena en un fichero y es de tamaño $t \times t$. Se considera por simplicidad que el tamaño de la matriz n es un múltiplo exacto de t. Las Figuras 2.6, 2.7 y 2.8 muestran, mediante un ejemplo, cómo proceden los cálculos en las tres variantes implementadas, así como las lecturas y escrituras de *tiles*. Como se puede observar, éstos se van leyendo de disco conforme va siendo necesario y, cada vez que uno de ellos es actualizado, se escribe en disco inmediatamente.

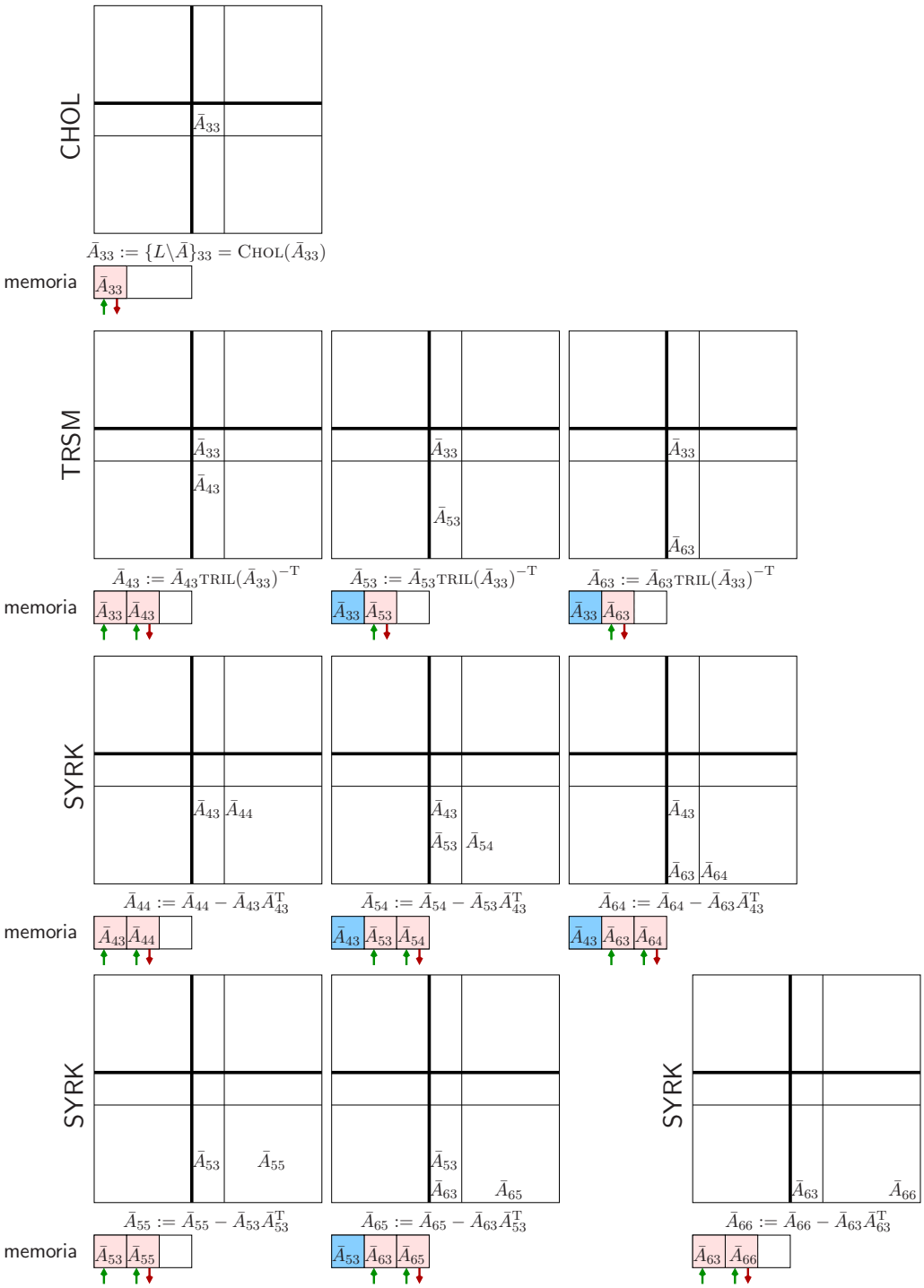


Figura 2.6: Operaciones aritméticas y de E/S que se realizan en la cuarta iteración de la variante 1 de la factorización de Cholesky OOC, cuando se factoriza una matriz de 7×7 *tiles*.

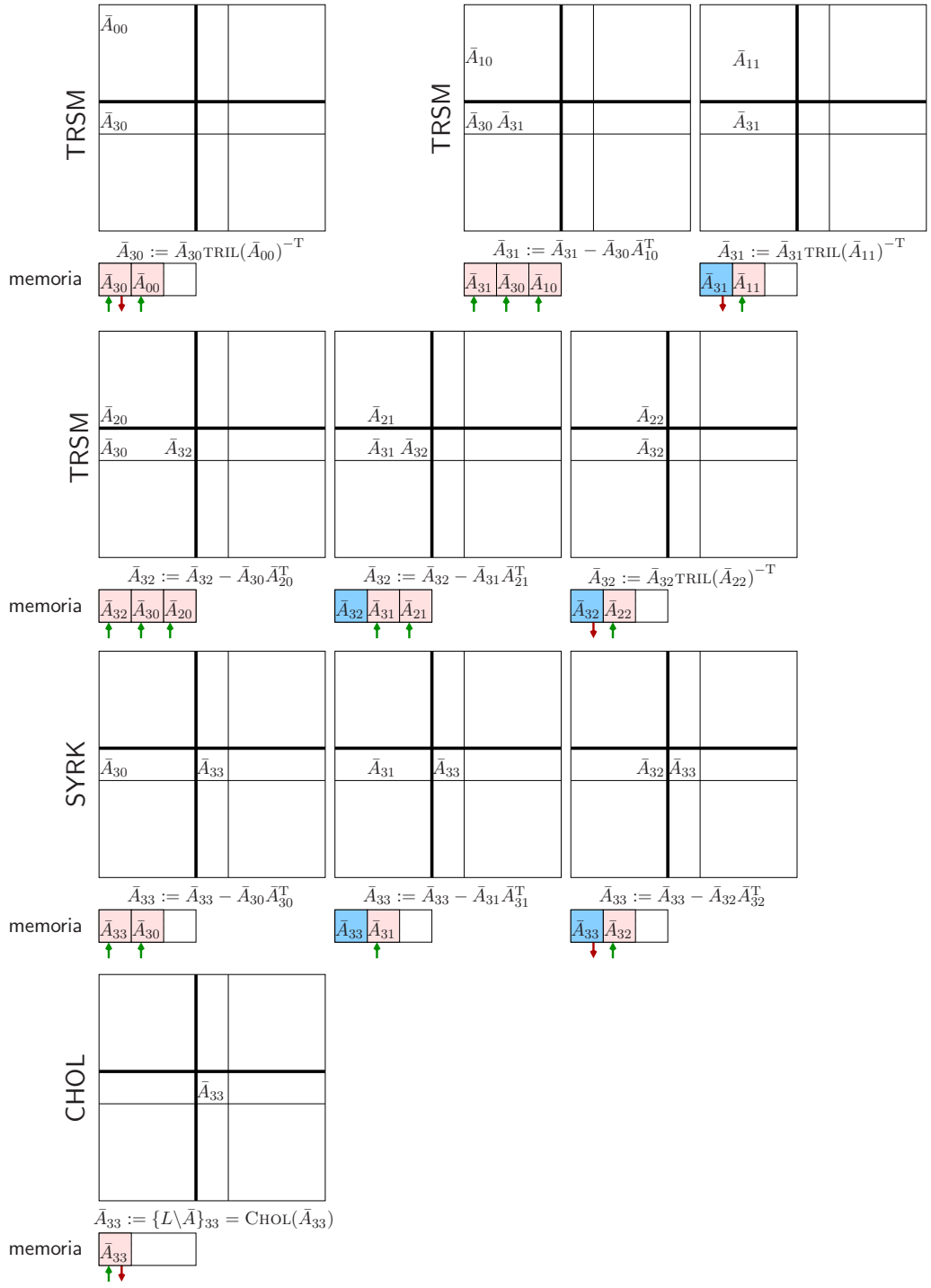


Figura 2.7: Operaciones aritméticas y de E/S que se realizan en la cuarta iteración de la variante 2 de la factorización de Cholesky OOC, cuando se factoriza una matriz de 7×7 *tiles*.

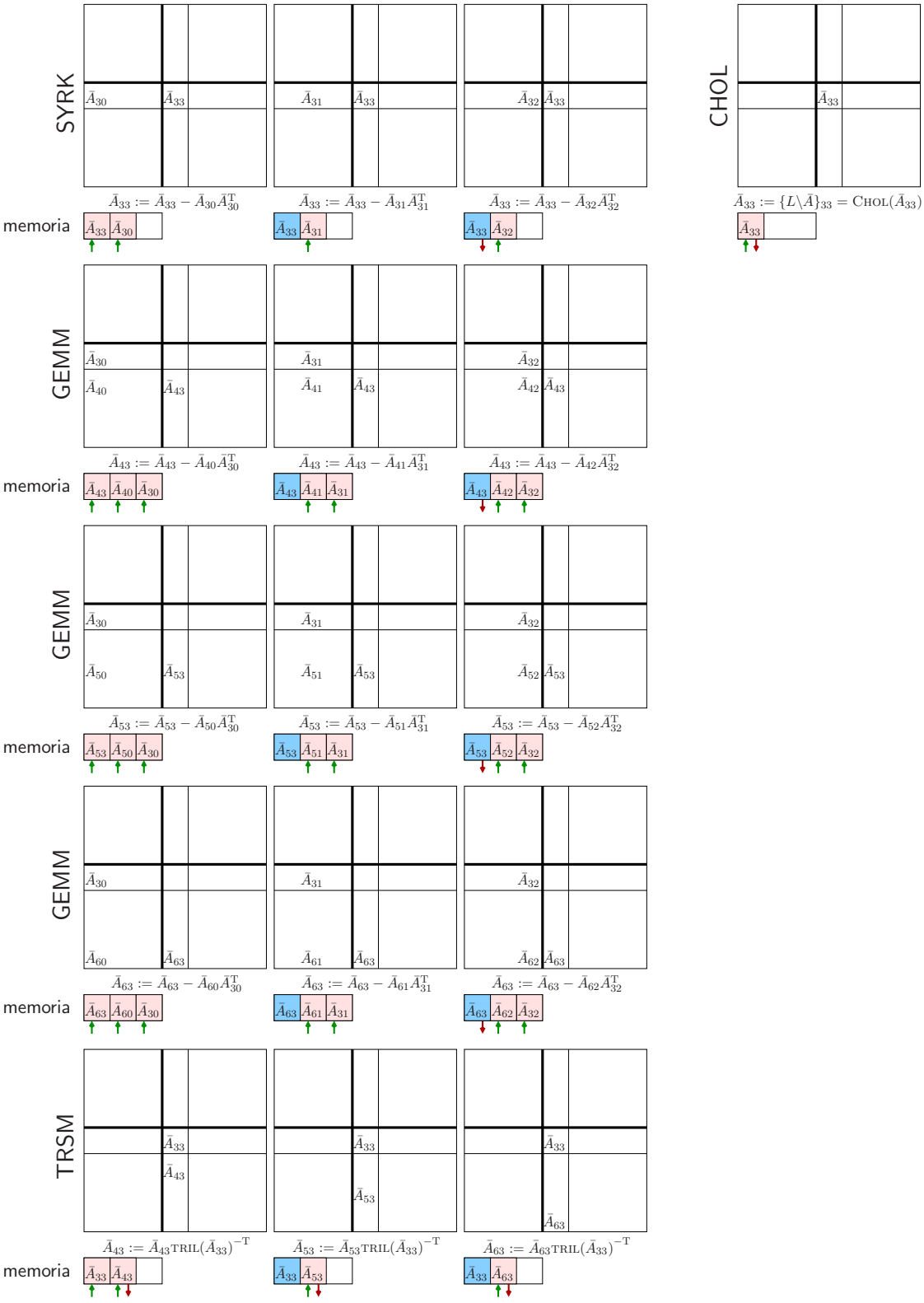


Figura 2.8: Operaciones aritméticas y de E/S que se realizan en la cuarta iteración de la variante 3 de la factorización de Cholesky OOC, cuando se factoriza una matriz de 7×7 *tiles*.

En los tres casos nos hemos situado en la cuarta iteración del bucle, en la que A_{11}, el bloque central del reparticionado 3×3 que se realiza al inicio de cada iteración, corresponde al *tile* $\bar{A}_{33}$:

$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{ccc|c|ccc} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} & \bar{A}_{04} & \bar{A}_{05} & \bar{A}_{06} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} & \bar{A}_{14} & \bar{A}_{15} & \bar{A}_{16} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} & \bar{A}_{24} & \bar{A}_{25} & \bar{A}_{26} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} & \bar{A}_{34} & \bar{A}_{35} & \bar{A}_{36} \\ \hline \bar{A}_{40} & \bar{A}_{41} & \bar{A}_{42} & \bar{A}_{43} & \bar{A}_{44} & \bar{A}_{45} & \bar{A}_{46} \\ \bar{A}_{50} & \bar{A}_{51} & \bar{A}_{52} & \bar{A}_{53} & \bar{A}_{54} & \bar{A}_{55} & \bar{A}_{56} \\ \bar{A}_{60} & \bar{A}_{61} & \bar{A}_{62} & \bar{A}_{63} & \bar{A}_{64} & \bar{A}_{65} & \bar{A}_{66} \end{array}\right).$$

En cada línea se ha indicado, a la izquierda, la operación que se realiza y sobre la matriz se señalan los *tiles* implicados. Ya que éstos han de estar residentes en la memoria principal (RAM), el contenido de esta última se ha representado bajo la matriz, mostrándose los *tiles* que almacena en cada momento, así como los *tiles* que se deben leer y escribir de/en disco (flechas hacia arriba y hacia abajo, respectivamente). Los *tiles* residentes en memoria que son reutilizados entre operaciones aparecen en color azul, mientras que en rosa aparecen los que se deben leer desde el disco. De la representación mostrada se concluye que, para poder realizar las ejecuciones, es necesario que quepan tres *tiles* completos en memoria, por lo que el tamaño máximo del *tile* estará restringido por el tamaño de la memoria principal.

Cuando se realizan implementaciones que utilizan datos que residen en disco (OOC), hay una diferencia importante entre la variante 1 (*right–looking*) y las variantes 2 y 3 (*left–looking*), pues estas últimas requieren menos escrituras. Esto se puede observar en la Figura 2.6, correspondiente a la variante 1. Las operaciones SYRK actualizan parcialmente la submatriz A_{22} que, en cada iteración, incluye un subconjunto de los *tiles* que contenía en la iteración anterior. Por ejemplo, el *tile* $\bar{A}_{66}$ formará parte de A_{22} en todas las iteraciones, excepto en la última, que es cuando se factoriza. Por lo tanto será actualizado parcialmente 6 veces, y en cada una de estas ocasiones será leído y escrito de/en disco. La variante 3 también realiza actualizaciones parciales, pero mientras que la variante 1 las realiza sobre una matriz de *tiles* (A_{22}), la variante 3 las hace sobre un vector de *tiles* (A_{21}). En consecuencia, el orden del coste de las escrituras es más reducido en esta variante. Lo mismo sucede para la variante 2. Esta última es la que presenta un menor número de escrituras, puesto que no realiza ninguna actualización parcial: cada *tile* es escrito una sola vez cuando se actualiza, excepto los *tiles* diagonales que son escritos dos veces: una primera al ser actualizados respecto a los *tiles* de su izquierda y la segunda al ser obtenida su factorización de Cholesky. La Tabla 2.1 muestra el coste de la E/S de las tres variantes de la factorización de Cholesky. Se ha considerado que la matriz a factorizar, de tamaño $n \times n$, está dividida *tiles* de tamaño $t \times t$ cada uno, por lo que el tamaño en *tiles* es $s \times s$, siendo $s = n/t$.

Cuando se trata de comparar las variantes 2 y 3, esta última es la utilizada tradicionalmente por seguir una orientación por columnas: en cada iteración se calcula (y por lo tanto se escribe) una columna de la matriz. La variante 2 es menos popular por trabajar, en cada iteración, sobre una fila de la matriz, algo que resulta ser un inconveniente cuando las matrices se almacenan por columnas, como es habitual en Fortran. En el caso de OOC no afecta negativamente el escribir por filas de *tiles* ya que, en nuestro caso, cada *tile* se almacena en un fichero distinto. Cada fichero,

Algoritmo	*Lecturas*	*Escrituras*
Variante 1	$\frac{2s^3+3s^2+7s}{6}$	$\frac{s^3+3s^2+2s}{6}$
Variante 2	$\frac{2s^3+3s^2+7s}{6}$	$\frac{s^2+3s}{2}$
Variante 3	$\frac{2s^3+3s^2+13s}{6}$	s^2+s

Tabla 2.1: Número de lecturas y escrituras de *tiles* (coste teórico de la E/S) de las distintas variantes de la factorización de Cholesky. (El número de elementos leídos/escritos puede obtenerse fácilmente multiplicando estos valores por el tamaño del *tile*: $t \times t$.)

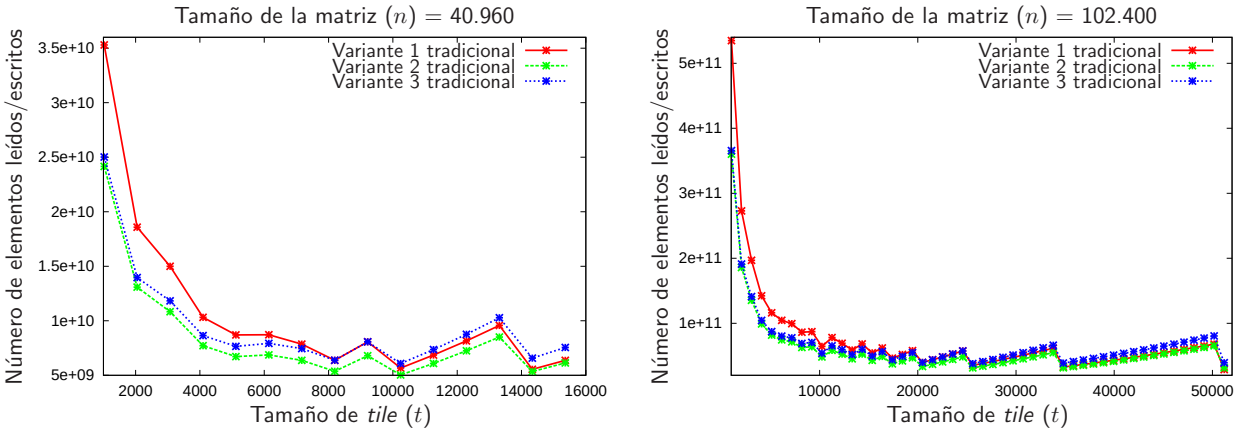


Figura 2.9: Coste teórico de la E/S de las tres variantes tradicionales de la factorización de Cholesky, en función del tamaño de *tile* para una matriz de tamaño $n = 40.960$ (izquierda) y 102.400 (derecha).

sin embargo, sigue almacenando los elementos del *tile* por columnas, por lo que sí es importante el orden en que se accede a los elementos internos de éstos.

La gráfica en la izquierda de la Figura 2.9 muestra el coste teórico de la E/S para las tres variantes de la factorización de Cholesky para una matriz de tamaño $n = 40.960$. Se observa que la variante 2 es la que menor coste presenta para todos los tamaños de *tile*, mientras que las variantes 1 y 3 intercambian sus papeles a partir de un tamaño de *tile* elevado ($t \gtrsim n/4$). Esto mismo sucede para matrices de gran tamaño (n=102.400), como muestra la gráfica en la derecha de la Figura 2.9.

Para poder llevar a cabo la ejecución de las tres variantes será preciso ajustar el tamaño de los *tiles* de modo que, al menos, quepan tres de ellos en memoria en todo momento. Además, en este primer estudio teórico se concluye que los *tiles* más grandes no aseguran necesariamente las mejores prestaciones, por lo que una tarea que deberemos realizar experimentalmente será la de determinar el tamaño de *tile* óptimo para cada variante y tamaño de problema.

2.3.2. Uso de una caché software

En las aplicaciones OOC la latencia de la E/S a menudo se convierte en un cuello de botella, por lo que nuestras propuestas de mejora estarán enfocadas a paliar su impacto. En el apartado anterior se ha comprobado que, a partir de cierto umbral, el coste de la E/S depende poco del tamaño de

tile, por lo que se intuye que no es necesario ocupar toda la memoria llenándola mediante unos pocos *tiles* de gran dimensión. Por contra, es posible contemplar la memoria disponible como si de una caché se tratara, llenándola con un mayor número de *tiles* de menor tamaño. Del mismo modo que los procesadores utilizan la memoria caché para reducir los accesos a memoria, nos planteamos utilizar la memoria RAM disponible como una memoria caché del disco. Si se saca buen partido de la caché será posible ahorrar accesos a disco, por lo que se reducirá la influencia de la E/S y podrán mejorar las prestaciones.

Para implementar una memoria de caché se debe escoger una organización que determine cómo emparejar (mapear) los *tiles* de la matriz en disco con los bloques (líneas) de la caché, gestionada por *software*, en memoria. Cada bloque de la caché será del tamaño de un *tile*. Ya que la caché es más pequeña que la matriz a factorizar, es necesario compartir los mismos bloques de caché por parte de distintos *tiles*. El modo en que se mapeen los *tiles* a los bloques de caché puede afectar a la ejecución del programa, ya que si dos *tiles* que son muy accedidos se mapean sobre el mismo bloque de caché, la tasa de fallos de caché será más alta de lo deseable. Las memorias caché se pueden organizar de varias formas: mapeo directo, completamente asociativa y asociativa por conjuntos [28]. Veamos brevemente cómo aplicar estas organizaciones a la caché que queremos implementar para el programa.

El *mapeo directo* corresponde a la organización más simple. Si en la caché caben c bloques (recordemos que cada uno alojará un *tile*), los primeros c *tiles* de la matriz se mapean directamente en los c bloques de la caché. Lo mismo sucede con los siguientes c *tiles* y con el resto de conjuntos de c *tiles* de la matriz. Así, determinar el bloque en el que se debe cargar un determinado *tile* es rápido y sencillo. El problema con este tipo de mapeo se presenta cuando se accede a una secuencia de *tiles* que recaen sobre el mismo bloque de caché. Cada acceso provoca un fallo de caché, siendo necesario vaciar el bloque que se acaba de cargar. A la sobrecarga que produce este tipo de situaciones en las memorias caché *hardware* se le denomina *trashing* y, cuando éste es muy alto, la caché se convierte en un lastre en lugar de una ventaja.

En el extremo opuesto del mapeo directo se encuentra la *caché completamente asociativa*, en la que cualquier *tile* puede mapearse sobre cualquiera de los bloque de la caché. Este tipo de memorias contienen información sobre los datos que alojan. Cuando se busca un *tile*, se debe consultar el contenido de todos los bloques de caché. Si ninguno coincide, hay un fallo de caché y se debe escoger un bloque de caché para ser reemplazado. Normalmente se escoge aquel que hace más tiempo que ha sido usado (política de reemplazo LRU). El problema del *trashing* no es habitual con las memorias caché completamente asociativas. Sin embargo, estas memorias son más lentas al requerir una gestión más compleja, que incluye la política de reemplazo y el tiempo de búsqueda del bloque en la caché.

Si se sitúan dos (cuatro, ocho,...) memorias caché de mapeo directo una junto a otra y se hace que cada *tile* corresponda a un bloque de caché de cada una de las memorias, tendremos una *caché asociativa por conjuntos*. Con esta configuración, cuando se debe traer un *tile* a la caché, se reemplaza aquel de los dos (cuatro, ocho,...) bloques que hace más tiempo que se ha usado. La búsqueda y el reemplazo se realizan como en la caché completamente asociativa, con la diferencia de que ahora sólo se debe buscar/escoger entre dos (cuatro, ocho,...) bloques y no entre todos ellos, por lo que será más rápida.

Una memoria caché asociativa por conjuntos es más inmune al problema del *trashing* que una memoria del mismo tamaño de mapeo directo y tiene una gestión más ágil que la caché completamente asociativa, por lo que ésta es la organización que se ha escogido para implementar la

```
1   void FLA_Trsm_OOCwrapper( FLA_Obj alpha, FLA_Obj L, FLA_Obj B )
2   /* OOC wrapper for FLA_Trsm                                    */
3   {
4     Task *task;
5
6     FLAOOC_Create_task( &task, FLA_Trsm, 1, 2 );
7
8     FLAOOC_Set_task_out_operand( task, B );
9     FLAOOC_Set_task_in_operand(  task, B );
10    FLAOOC_Set_task_in_operand(  task, L );
11
12    FLAOOC_Push_task( task );
13
14  }
```

Figura 2.10: Rutinas intermedias para utilizar el sistema de caché.

memoria caché. La política de reemplazo implementada es la LRU y el número de bloques de la caché está en función del tamaño de la RAM y del tamaño de *tile*. En particular, la caché ocupa una parte importante de la memoria disponible, dejando el resto para el sistema operativo y el programa ejecutable.

Para esta nueva implementación se ha desarrollado además un entorno de ejecución, o *run-time*, que se encarga de determinar, en cada momento, qué *tiles* se deben leer y escribir. En primer lugar, el sistema ejecuta el código de manera simbólica, generando una lista de tareas pendientes de ejecución, y anotando para cada una de éstas los *tiles* de entrada y de salida. Para ello se han sustituido las llamadas a las rutinas de `libflame` (`FLA_Trsm`, `FLA_Syrk` y `FLA_Gemm`) por llamadas a unas funciones intermedias que realizan este trabajo. Por ejemplo, la llamada a `FLA_Trsm` del código de la Figura 2.3 (derecha) se sustituye por una llamada a la rutina `FLA_Trsm_OOCwrapper`, cuya implementación se muestra en la Figura 2.10. Esta rutina crea una nueva tarea en la línea 6, asocia los *tiles* de salida y de entrada en las líneas 8–10, y apila la tarea en la lista de tareas pendientes de ejecución en la línea 12.

Una vez se ha completado la ejecución simbólica del código, obtenemos la lista completa de las tareas que será necesario ejecutar para completar el proceso de factorización. A continuación, en una segunda etapa, las tareas se ejecutan en orden, haciendo un recorrido FIFO de la lista: el sistema toma la siguiente tarea, identifica los *tiles* que necesita para realizar la operación asociada y comprueba si se encuentran en la caché, leyéndolos desde el disco en caso contrario. Una vez los datos residen en memoria, se ejecuta la operación. Los *tiles* permanecen en la caché hasta que el sistema decide que deben ser reemplazados porque se necesita el espacio que están ocupando.

Con este modo de funcionamiento toda la gestión de la caché queda en manos del *run-time*. El resultado es un sistema que resulta transparente al programador y que puede utilizarse no sólo en el cálculo de la factorización de Cholesky sino también en cualquier otra operación de álgebra matricial para la que se disponga de un algoritmo por bloques y de los *wrappers* necesarios.

La sustitución de rutinas como `FLA_Trsm` por llamadas a sus correspondientes *wrappers* puede automatizarse (por ejemplo, mediante el uso de *scripts*), por lo que no complica en absoluto la programación. El desarrollo del propio *wrapper*, por ejemplo `FLA_Trsm_OOCwrapper`, también puede automatizarse, si bien, dado su escaso número, la implementación de los mismos se ha realizado de forma manual.

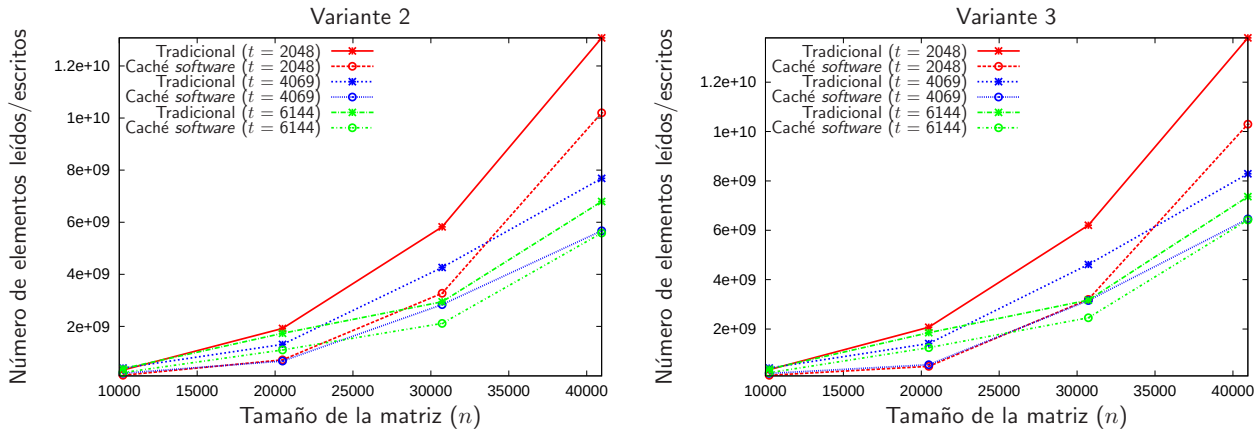


Figura 2.11: Coste teórico de la E/S de las variantes 2 y 3 de la factorización de Cholesky, con y sin el uso del sistema que gestiona la memoria disponible como una memoria caché *software*.

En la Figura 2.11 se puede observar el ahorro en el número de accesos a elementos que se produce gracias a la incorporación del sistema de caché *software* frente a las implementaciones iniciales (tradicionales). En los resultados no se incluyen los de la variante 1, pues hemos comprobado en el experimento anterior que únicamente es competitiva frente a las otras dos variantes cuando el tamaño de *tile* es muy grande. Sin embargo, los tamaños de *tile* muy grandes resultan incompatibles con un sistema de caché, ya que limitan demasiado el número de bloques que puede contener la caché, eliminando su utilidad. Puesto que todas nuestras mejoras incorporan este sistema de caché, a partir de ahora se descarta la variante 1. Para obtener estas curvas se ha realizado una simulación que permite contabilizar el número de accesos de cada una de ellas. Se observa que al aumentar el tamaño del *tile* se reduce el número de accesos, aunque no son los tamaños de *tile* más grandes los que realizan un menor número de accesos, sobre todo cuando se trabaja con matrices grandes.

Si comparamos los mejores resultados de los esquemas con caché y sin caché (tradicional), es decir, los correspondientes al tamaño de *tile* óptimo para cada tamaño de matriz, podemos comprobar que el primero reduce sustancialmente el número de accesos a disco.

2.3.3. Actualizaciones en zigzag

Para sacar más partido a la caché *software* se propone una reordenación de las operaciones de manera que se incremente la localidad de referencia de los algoritmos. Veamos, a modo de ejemplo, las operaciones sobre *tiles* que se realizan en la variante 3 sobre una matriz de 7×7 *tiles*. Nos situamos en la cuarta iteración del bucle, en la que A_{11}, el bloque central del reparticionado 3×3

que se realiza al inicio de esta iteración, corresponde al *tile* $\bar{A}_{33}$:

$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{ccc|c|ccc} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} & \bar{A}_{04} & \bar{A}_{05} & \bar{A}_{06} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} & \bar{A}_{14} & \bar{A}_{15} & \bar{A}_{16} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} & \bar{A}_{24} & \bar{A}_{25} & \bar{A}_{26} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} & \bar{A}_{34} & \bar{A}_{35} & \bar{A}_{36} \\ \hline \bar{A}_{40} & \bar{A}_{41} & \bar{A}_{42} & \bar{A}_{43} & \bar{A}_{44} & \bar{A}_{45} & \bar{A}_{46} \\ \bar{A}_{50} & \bar{A}_{51} & \bar{A}_{52} & \bar{A}_{53} & \bar{A}_{54} & \bar{A}_{55} & \bar{A}_{56} \\ \bar{A}_{60} & \bar{A}_{61} & \bar{A}_{62} & \bar{A}_{63} & \bar{A}_{64} & \bar{A}_{65} & \bar{A}_{66} \end{array}\right).$$

La lista de tareas que se llevan a cabo en esta iteración es la siguiente:

$$\begin{array}{lrcl}
\textsc{syrk} & (1) & \bar{A}_{33} & := & \bar{A}_{33} - \bar{A}_{30}\bar{A}_{30}^{\mathrm{T}} \\
& (2) & \bar{A}_{33} & := & \bar{A}_{33} - \bar{A}_{31}\bar{A}_{31}^{\mathrm{T}} \\
& (3) & \bar{A}_{33} & := & \bar{A}_{33} - \bar{A}_{32}\bar{A}_{32}^{\mathrm{T}} \\
\\
\textsc{chol} & (4) & \bar{A}_{33} & := & \{L\backslash\bar{A}\}_{33} = \textsc{chol}(\bar{A}_{33}) \\
\\
\textsc{gemm} & (5) & \bar{A}_{43} & := & \bar{A}_{43} - \bar{A}_{40}\bar{A}_{30}^{\mathrm{T}} \\
& (6) & \bar{A}_{43} & := & \bar{A}_{43} - \bar{A}_{41}\bar{A}_{31}^{\mathrm{T}} \\
& (7) & \bar{A}_{43} & := & \bar{A}_{43} - \bar{A}_{42}\bar{A}_{32}^{\mathrm{T}} \\
\\
& (8) & \bar{A}_{53} & := & \bar{A}_{53} - \bar{A}_{50}\bar{A}_{30}^{\mathrm{T}} \\
& (9) & \bar{A}_{53} & := & \bar{A}_{53} - \bar{A}_{51}\bar{A}_{31}^{\mathrm{T}} \\
& (10) & \bar{A}_{53} & := & \bar{A}_{53} - \bar{A}_{52}\bar{A}_{32}^{\mathrm{T}} \\
\\
& (11) & \bar{A}_{63} & := & \bar{A}_{63} - \bar{A}_{60}\bar{A}_{30}^{\mathrm{T}} \\
& (12) & \bar{A}_{63} & := & \bar{A}_{63} - \bar{A}_{61}\bar{A}_{31}^{\mathrm{T}} \\
& (13) & \bar{A}_{63} & := & \bar{A}_{53} - \bar{A}_{62}\bar{A}_{32}^{\mathrm{T}} \\
\\
\textsc{trsm} & (14) & \bar{A}_{43} & := & \bar{A}_{43}\textsc{tril}(A_{33})^{-\mathrm{T}} \\
& (15) & \bar{A}_{53} & := & \bar{A}_{53}\textsc{tril}(A_{33})^{-\mathrm{T}} \\
& (16) & \bar{A}_{63} & := & \bar{A}_{63}\textsc{tril}(A_{33})^{-\mathrm{T}}
\end{array}$$

Estas operaciones se ilustran en la Figura 2.12(a), en la que se ha señalado con el número de la operación, los *tiles* que son datos de entrada (lecturas desde disco) para cada una de ellas. En azul se marcan los *tiles* que deben escribirse al finalizar cada operación. Las flechas rojas muestran el sentido en que se recorren los *tiles*.

Si en la actualización (5) se produce un fallo de caché por $\bar{A}_{30}$, las siguientes actualizaciones mediante GEMM producirán también fallos de caché por los *tiles* de esa misma fila, tanto al actualizar $\bar{A}_{43}$, como $\bar{A}_{53}$ y $\bar{A}_{63}$. Estos fallos de caché serán menos probables si las actualizaciones de la (5) a la (7) se realizan en orden inverso, al igual que las actualizaciones de la (11) a la (13). Recorriendo en zigzag los *tiles* de la partición A_{20} en la GEMM es posible que se eviten algunos fallos de caché, ahorrando así accesos a disco. En cuanto a la última operación, la TRSM, ya que la GEMM anterior ha recorrido los *tiles* de la partición A_{20} de arriba a abajo, disminuirán los fallos de caché si los *tiles*

de la partición A_{21} se recorren de abajo a arriba. La reordenación resultante de realizar el recorrido en zigzag se muestra en la Figura 2.12(b).

Para la variante 2 también se ha realizado un recorrido en zigzag. En este caso, los *tiles* que se recorren son los del bloque A_{10}, tanto en la operación TRSM como en la SYRK posterior, operaciones que se muestran en la Figura 2.7.

Las gráficas de la Figura 2.13 reflejan, mediante simulación, el ahorro que se produce en el número de accesos de las variantes 2 y 3 gracias al recorrido en zigzag de los *tiles*. La ganancia que esta modificación es capaz de aportar teóricamente es de nuevo significativa.

2.3.4. Solapamiento de cálculos y accesos a disco

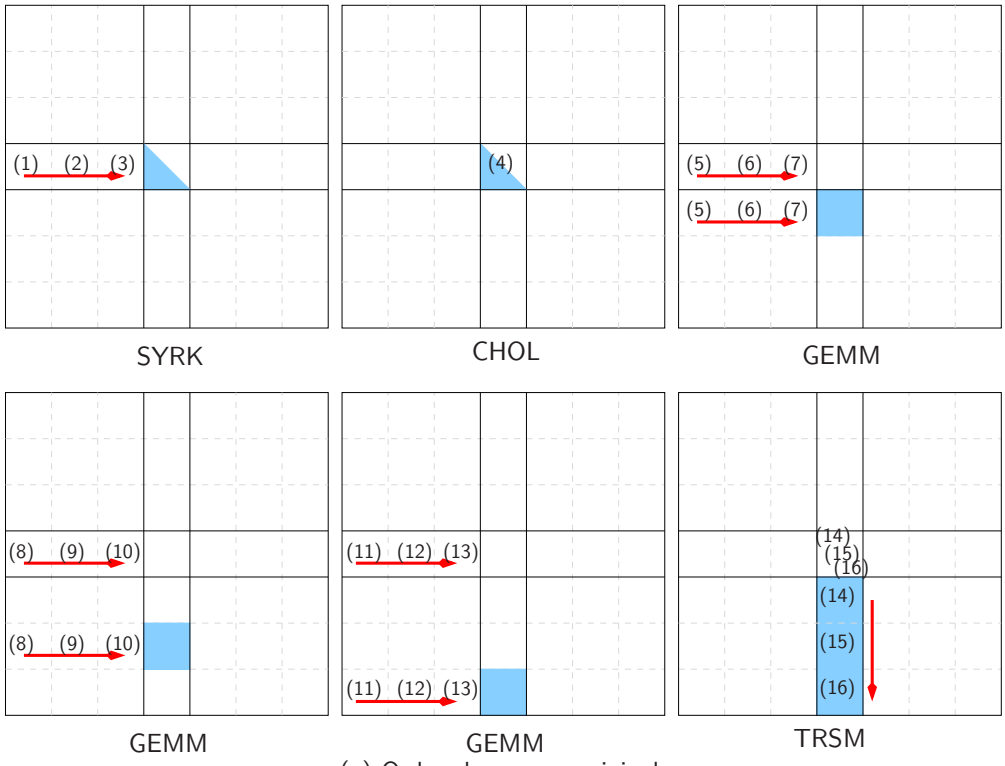
El *run-time* que gestiona la caché *software* y la reorganización de los recorridos en zigzag explotan la localidad de referencia para reducir las transferencias de datos entre el disco y la memoria, posibilitando una mejora de las prestaciones. Aún así, sigue habiendo tiempos de inactividad (sin cálculos) debidos a las esperas para que se completen los accesos a datos en disco. A continuación se propone introducir una nueva mejora en el *run-time* que gestiona la caché, de modo que sea capaz de solapar cálculos y comunicaciones, sin que sea necesario que el programador haga uso explícito de rutinas de E/S asíncrona en los códigos.

Cuando el *run-time* ejecuta el código de manera simbólica, genera una lista de tareas, indicando para cada una de éstas los *tiles* de entrada y de salida. En una segunda etapa las tareas se ejecutan de manera secuencial siguiendo la lista en orden FIFO. El orden en que las tareas aparecen en dicha lista, junto con la direccionalidad de los operandos (entrada o salida), definen el orden y la dirección en que deben transferirse los *tiles* entre la memoria y el disco. Puesto que la lista se genera antes de comenzar los cálculos, en cualquier momento de la factorización se conoce qué datos serán necesarios más adelante. Si estos datos son transferidos desde el disco con antelación, cuando sean necesarios ya residirán en memoria, reduciendo así los tiempos de espera.

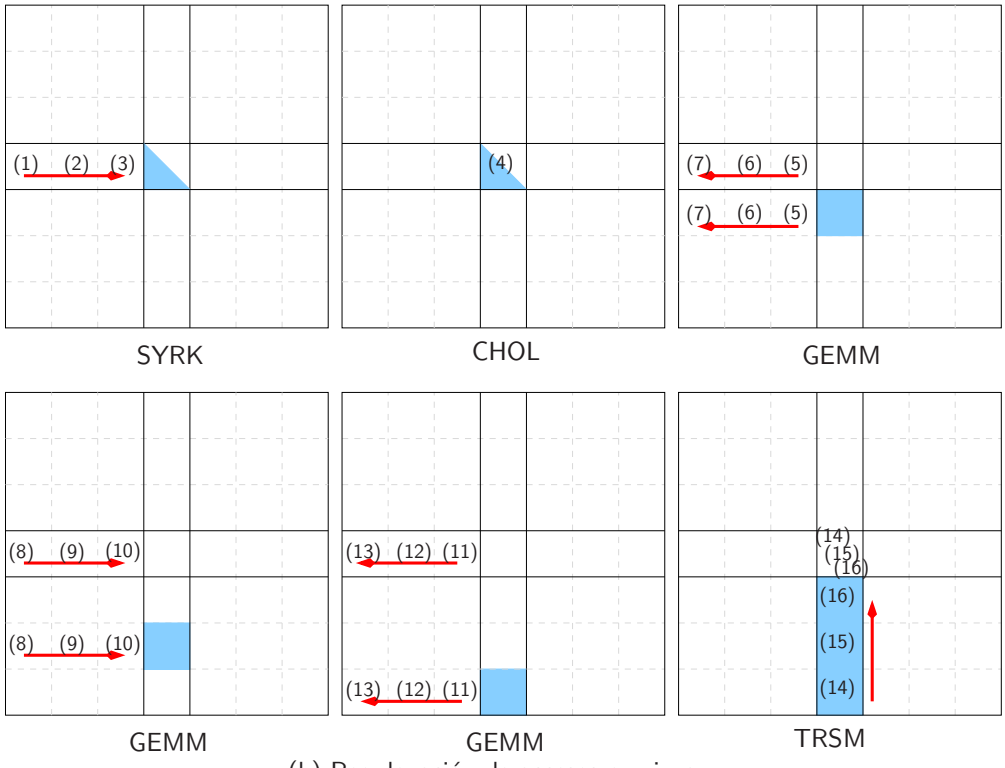
La implementación de esta técnica se consigue del siguiente modo. Cuando la ejecución comienza, una *hebra* explora la lista de tareas pendientes. Para cada tarea de la lista, si hay espacio suficiente en la caché *software*, esta hebra exploradora lee los *tiles* necesarios a memoria, si no residen ya en ésta, pasando la tarea a otra lista en donde se encuentran las tareas preparadas para su ejecución. Una segunda hebra, la ejecutora, es la que va tomando las tareas de esta segunda lista para ejecutarlas. Cuando la caché está llena y es necesario hacer espacio para un nuevo *tile*, la hebra exploradora reemplaza alguno de los que no están involucrados en ninguna tarea de la lista de pendientes de ejecución. Si esto no es posible, la hebra exploradora espera a que la hebra ejecutora complete el procesamiento de nuevas tareas.

Este funcionamiento corresponde a una típica interacción entre un productor y un consumidor: una hebra produce datos (la exploradora) y la otra hebra los consume (la ejecutora). Dado que la caché *software* tiene una capacidad limitada, se realiza una sincronización para detener a la hebra productora cuando la caché esté llena (y no se pueda reemplazar ningún *tile*) y bloquear a la hebra consumidora cuando la caché esté vacía.

Mediante el paralelismo que proporcionan las hebras, es posible solapar cálculo y transferencias con el disco, y ocultar casi totalmente los tiempos de espera (la latencia) que producen las lecturas y escrituras de datos, de modo que las prestaciones no se vean afectadas por el hecho de trabajar con matrices que residen en el disco. Las hebras también proporcionan un mecanismo de E/S



(a) Orden de accesos original.



(b) Reordenación de accesos en zigzag.

Figura 2.12: Reordenación de las operaciones de actualización sobre *tiles* para aumentar la localidad temporal en el acceso a los datos.

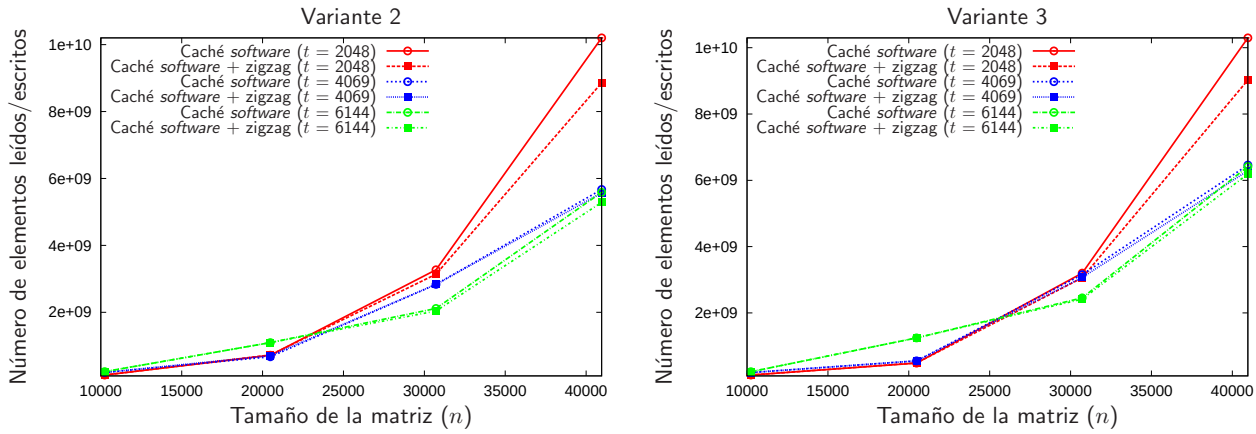


Figura 2.13: Coste teórico de la E/S de las variantes 2 y 3 de la factorización de Cholesky, con y sin el recorrido en zigzag de los *tiles*.

asíncrona que queda oculto al desarrollador de los códigos de computación matricial, aislándolo de la complejidad de su uso.

Las figuras de la 2.14 a la 2.19 muestran un ejemplo de traza de la factorización de Cholesky OOC de una matriz de 4×4 *tiles*,

$$A = \begin{pmatrix} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix},$$

con el algoritmo correspondiente a la variante 3 que realiza el solapamiento de cálculos y accesos a disco.

En este caso no se realiza ninguna simulación del número de lecturas/escrituras, puesto que el solapamiento no reduce su cantidad respecto a las mejoras aportadas anteriormente, sino que realiza al mismo tiempo los cálculos y las operaciones de E/S; en cambio, cabe esperar que el efecto del solapamiento se deje notar en la fase experimental.

2.4. Experimentos sobre una arquitectura monoprocesador

Como ya se ha expuesto en la introducción del Capítulo 1, las técnicas OOC pueden ser necesarias en entornos especializados donde, por motivos técnicos, la cantidad de memoria RAM es muy reducida, como sucede en los sistemas empotrados. Otra razón en favor de las técnicas OOC es que son más económicas, ya que reemplazan parcialmente el almacenamiento en RAM por el disco. Por estos motivos consideramos que resulta interesante experimentar nuestros códigos sobre sistemas monoprocesador.

Los experimentos que se presentan en esta sección han sido realizados sobre la estación de trabajo ROPE (en la Tabla 1.1 se muestran las características de ésta y otras máquinas usadas en la evaluación experimental de este capítulo), que dispone de un único procesador de propósito general.

En esta primera arquitectura se aprovecha la existencia de implementaciones optimizadas de las bibliotecas BLAS y LAPACK para el cálculo de los cuatro núcleos básicos que operan sobre

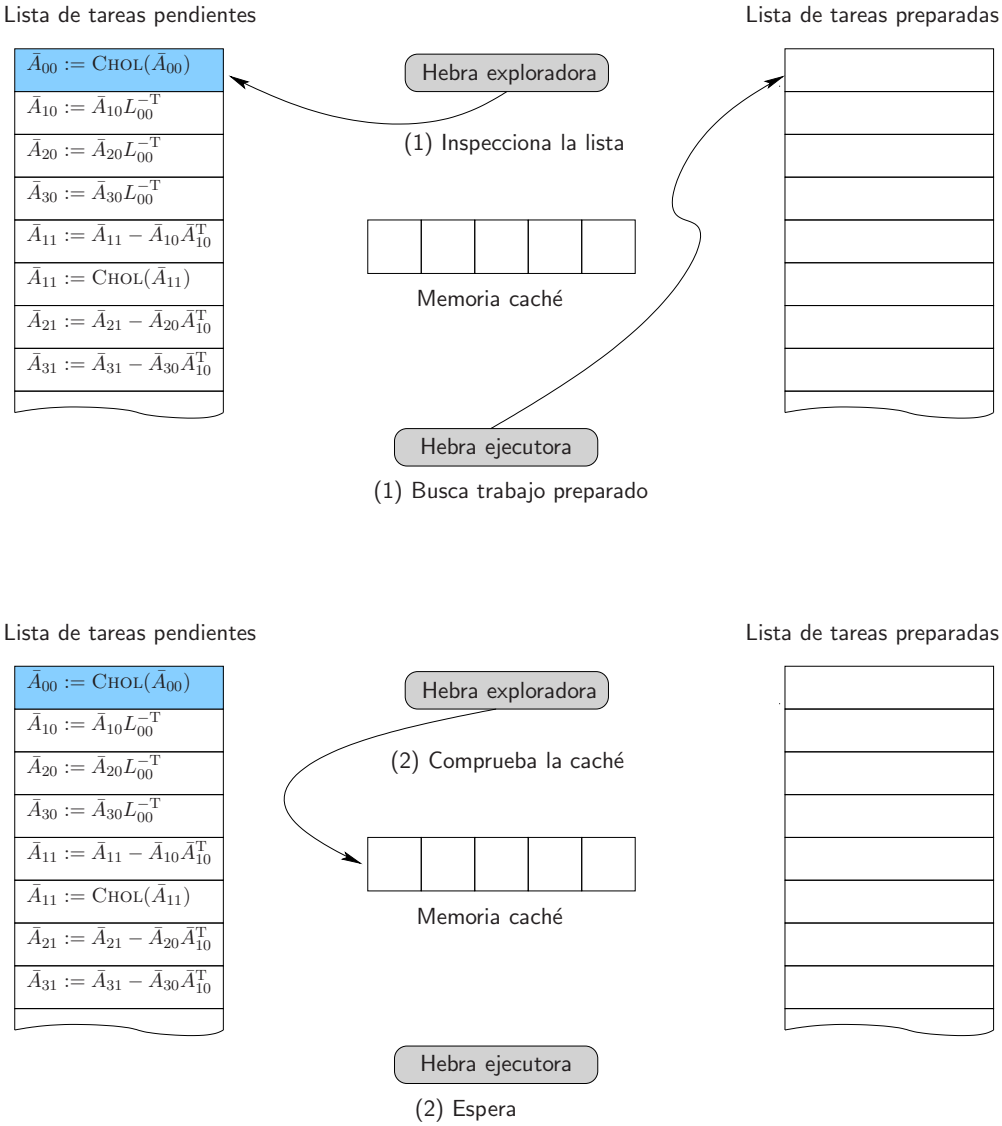


Figura 2.14: La hebra exploradora inspecciona la lista de tareas pendientes y comprueba si los *tiles* implicados en la tarea de la cabeza de la lista están disponibles en la memoria caché. La hebra ejecutora espera a que haya trabajo preparado para su ejecución.

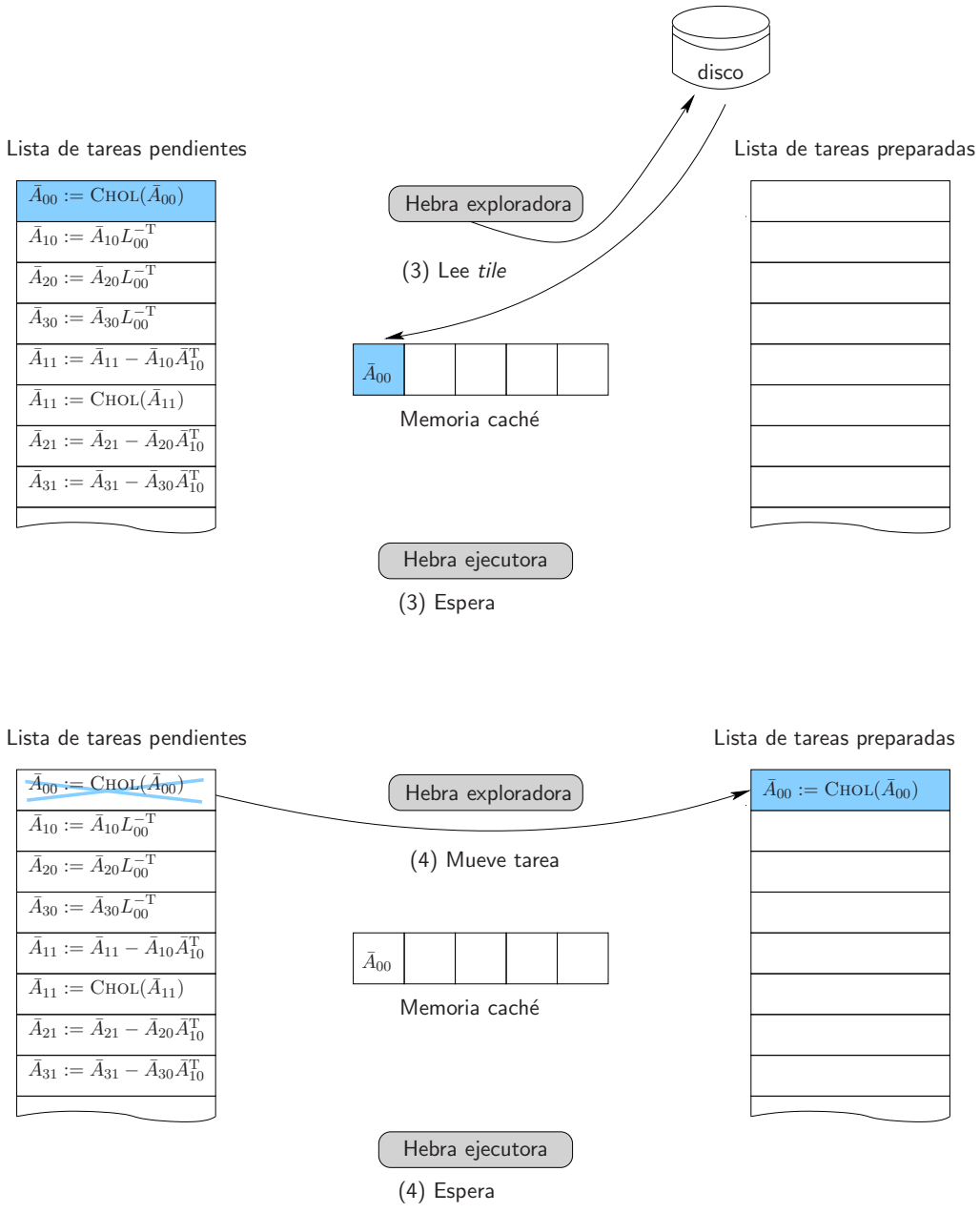


Figura 2.15: La hebra exploradora lee del disco el *tile* requerido por la primera tarea pendiente y pasa dicha tarea a la lista de tareas preparadas: la tarea está lista para su ejecución, ya que los datos que necesita están en memoria.

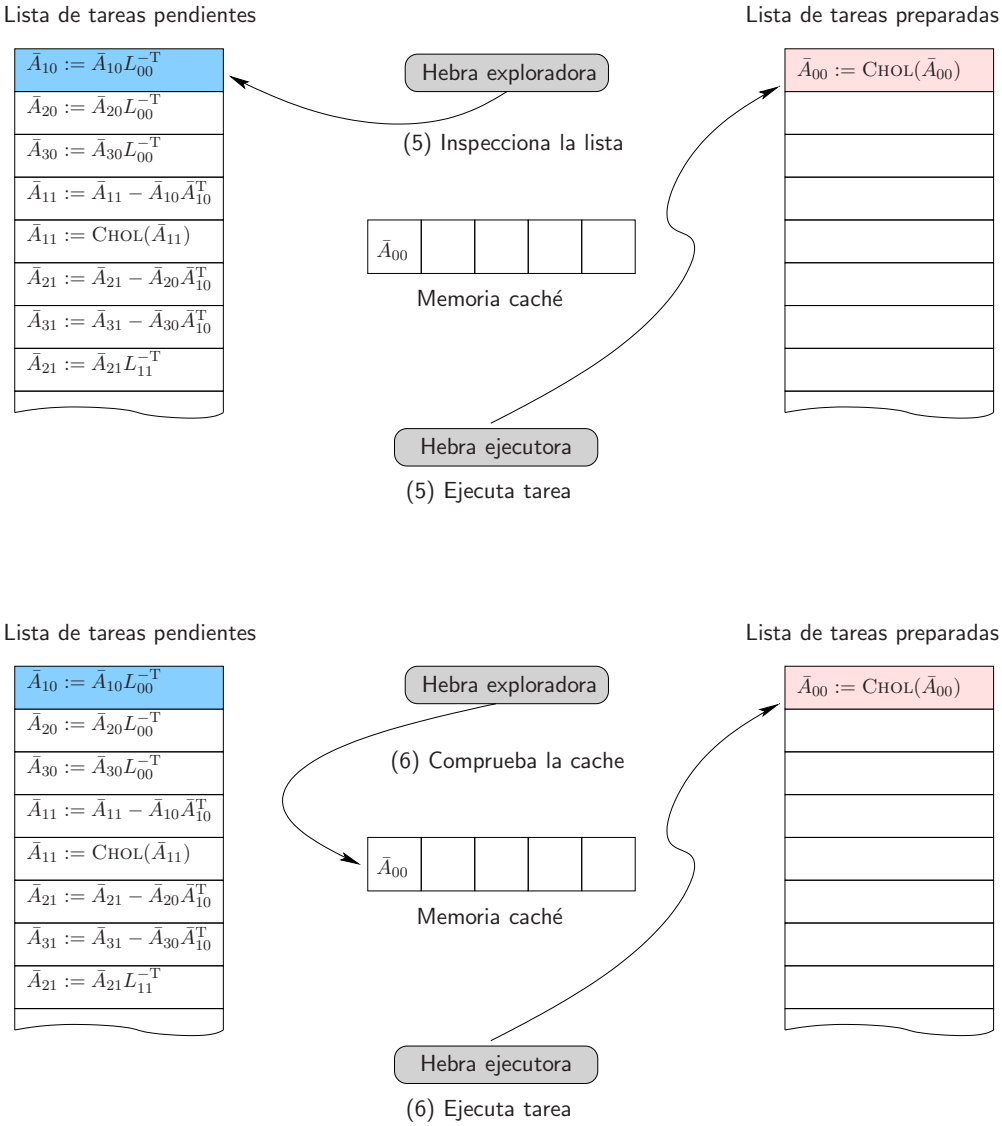


Figura 2.16: Mientras la hebra exploradora continúa encargándose de la lista de tareas pendientes, la hebra ejecutora toma la tarea que se encuentra en la cabeza de la lista de tareas preparadas para su ejecución y comienza a realizar los cálculos.

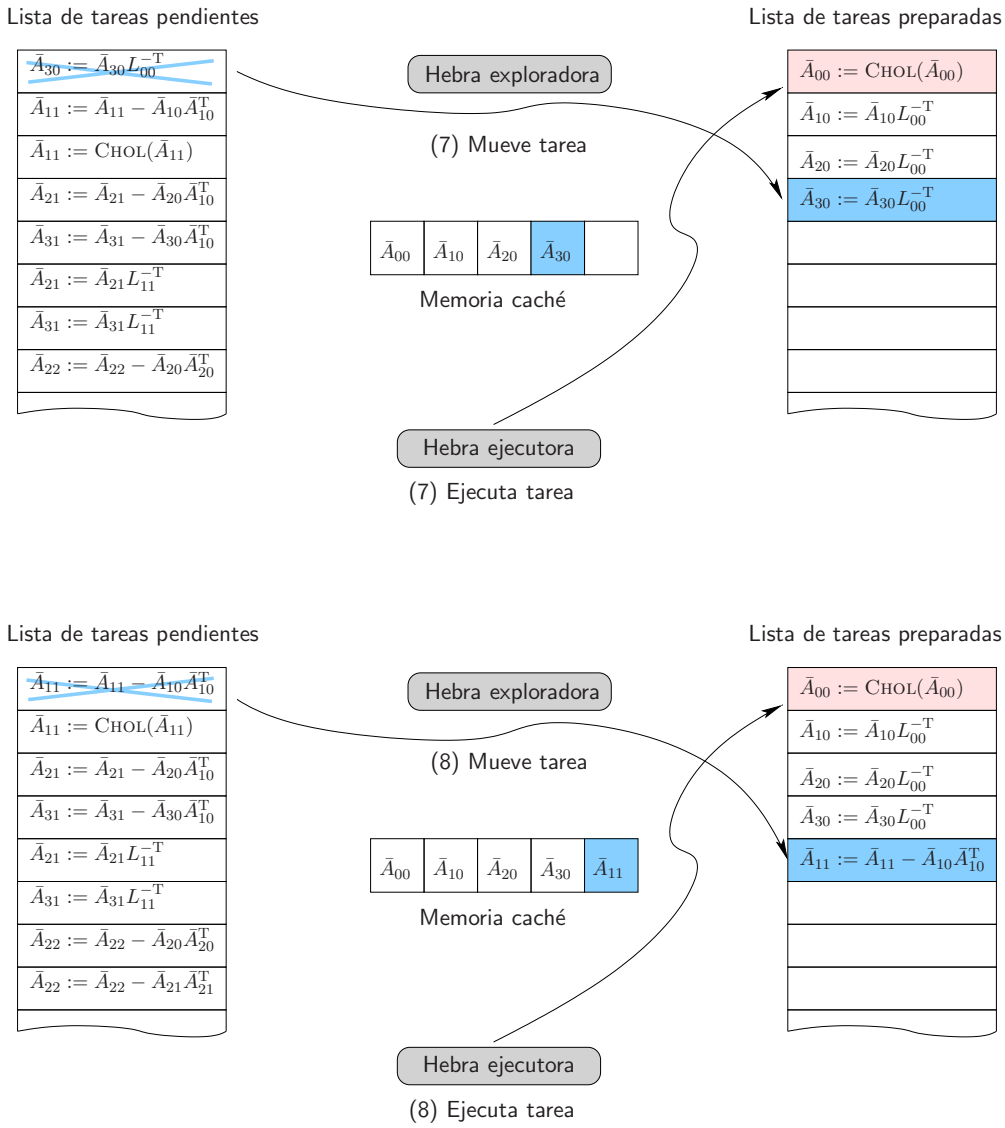


Figura 2.17: Mientras la hebra ejecutora se encarga de los cálculos, la hebra exploradora va llenando la memoria caché con los *tiles* con los que trabajarán las siguientes tareas listas para su ejecución. En la parte superior de la figura se observa que se han cargado los *tiles* $\bar{A}_{10}$, $\bar{A}_{20}$ y $\bar{A}_{30}$, y se han llevado las tareas correspondientes a la lista de preparadas.

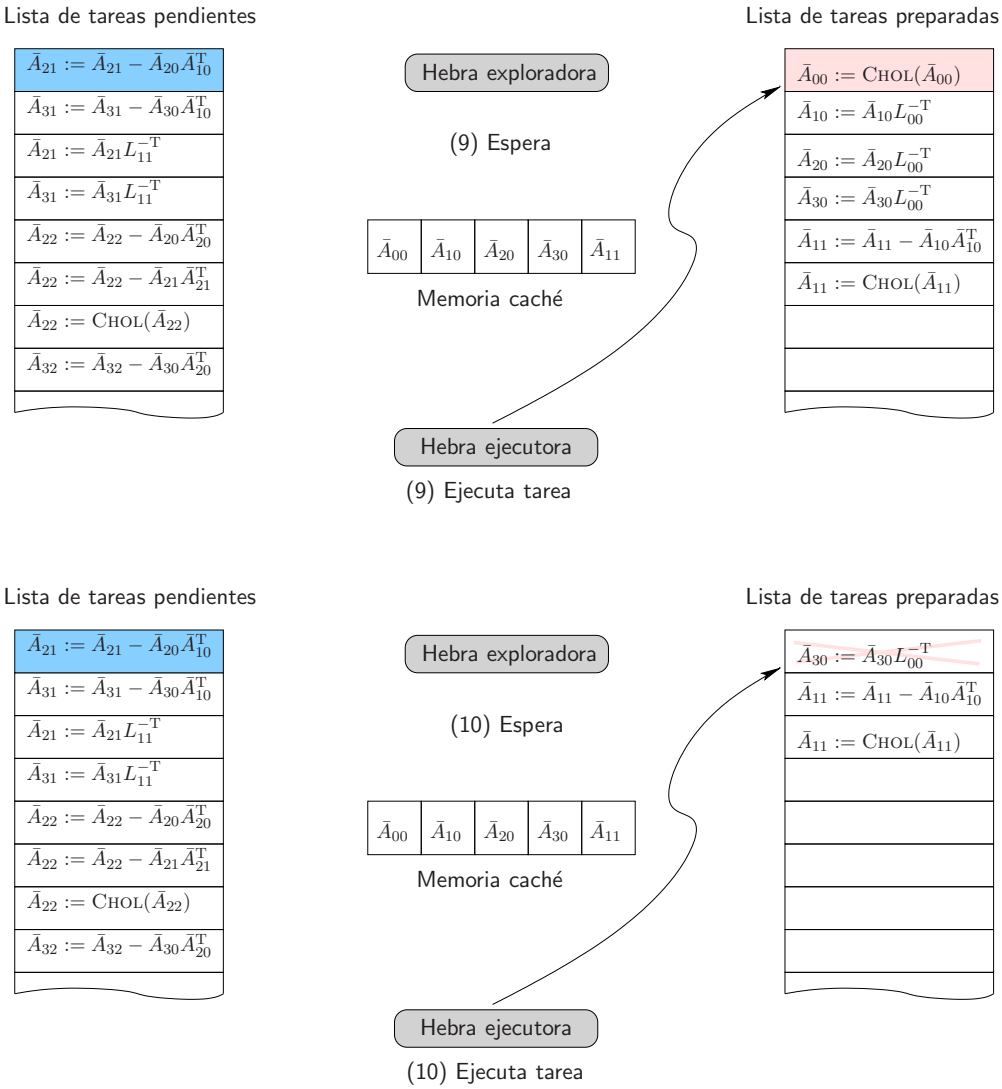


Figura 2.18: Cuando la memoria caché está llena y todos los *tiles* que contiene se necesitan para tareas preparadas, la hebra exploradora pasa a un estado de espera. Esta espera finalizará cuando la hebra exploradora detecte que algún *tile* no es necesario en memoria.

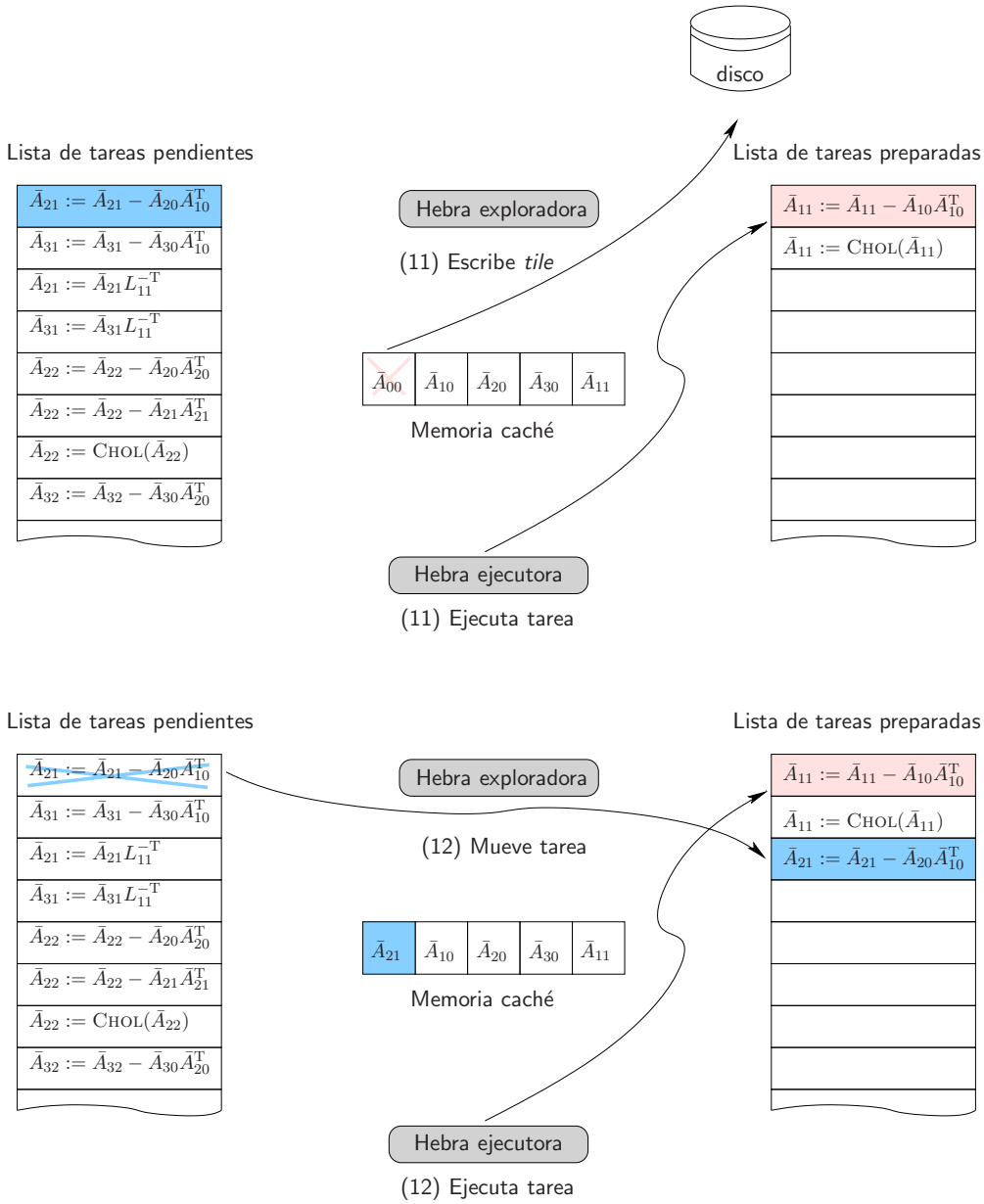


Figura 2.19: En cuanto la hebra exploradora detecta que un *tile* ya no es necesario en memoria, lo escribe en disco (si ha sido modificado) y reanuda su actividad.

los *tiles in-core* (en la Tabla 1.2 se muestra la versión de las bibliotecas usadas en la evaluación experimental de este capítulo). Denominaremos a éstos como POTRF (factorización de Cholesky de un *tile*), TRSM (resolución de un sistema triangular en el que tanto la matriz de coeficientes como la de términos independientes son *tiles*), GEMM (producto de *tiles*) y SYRK (actualización de rango t de la parte triangular inferior de un *tile*).

2.4.1. Variantes algorítmicas

Para poder llevar a cabo la ejecución de las tres variantes de los algoritmos OOC tradicionales (sin gestión de caché) es preciso ajustar, en primer lugar, el tamaño de los *tiles*, de modo que quepan tres de ellos en memoria en todo momento. La cuestión que se plantea es escoger el tamaño de *tile* óptimo, por lo que se han realizado ejecuciones de las tres variantes con distintos tamaños de *tile* y de matriz. Las prestaciones obtenidas se muestran en la Figura 2.20. En ella observamos un comportamiento similar para las tres variantes, que se asemeja al coste teórico que cabría esperar a partir de los resultados en la Figura 2.9 (izquierda). Al igual que en teoría, en la práctica la variante 1 es la que ofrece peores prestaciones (en términos de una tasa de GFLOPS más reducida), pero el rendimiento de las tres variantes tiende a igualarse a medida que crece el tamaño de *tile*.

La Figura 2.21 permite comparar las tres variantes para distintos tamaños de matriz. Para cada tamaño se ha escogido el tamaño de *tile* óptimo de cada variante. Tal y como se esperaba, las variantes 2 y 3 ofrecen mejores prestaciones, si bien la diferencia es menor de lo esperado.

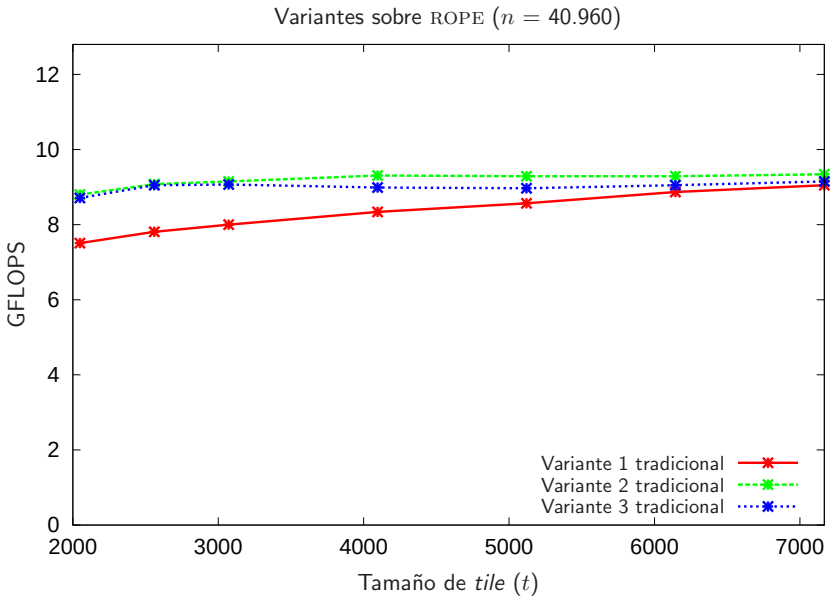


Figura 2.20: Ejecución sobre ROPE de la implementación OOC tradicional de las tres variantes para la factorización de Cholesky. Efecto del tamaño de *tile* sobre las prestaciones para una matriz de tamaño $n = 40.960$.

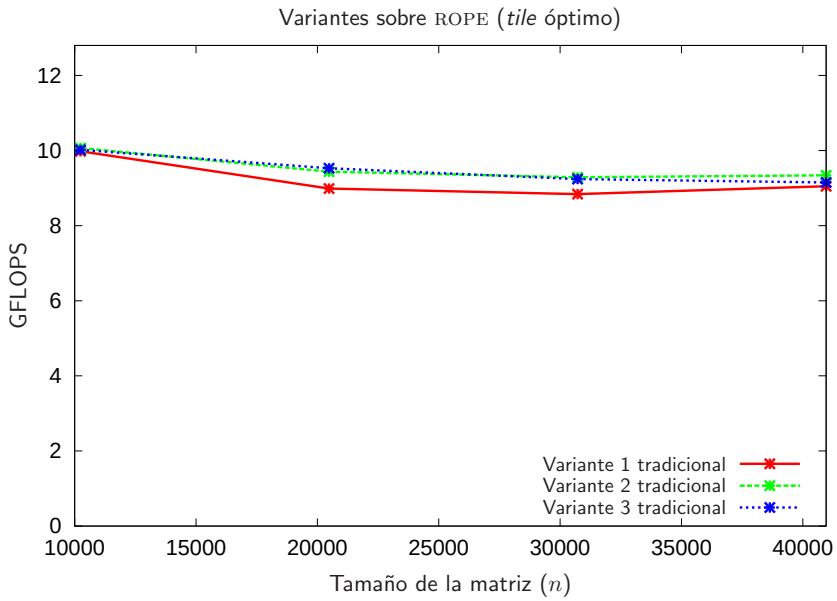


Figura 2.21: Ejecución sobre ROPE de la implementación OOC tradicional de las tres variantes para la factorización de Cholesky.

2.4.2. Uso de una caché software

En el apartado anterior se ha comprobado que el tamaño de *tile* no resulta determinante sobre las prestaciones obtenidas en las implementaciones OOC de las variantes 2 y 3. Esto confirma que no es necesario ocupar toda la memoria con los *tiles* con los que se opera y, por lo tanto, es posible utilizar la RAM como si de una memoria caché se tratara. Esto no es posible para la variante 1, que necesita unos tamaños de *tile* grandes para ser competitiva. Por esta razón se descarta esta variante a partir de este experimento. Si se saca buen partido de la caché, será posible ahorrar accesos a disco, por lo que la E/S tendrá un impacto menor y podrán mejorar las prestaciones.

La gráfica de la Figura 2.22 expone cómo se incrementa el rendimiento para las variantes 2 y 3 cuando se realiza la gestión de memoria como una caché. Evidentemente, cuantos más *tiles* quepan en la caché, mayor probabilidad de acierto en el acceso a la caché. Sin embargo, para un tamaño de caché determinado, un mayor tamaño de *tile* permitirá un menor número de bloques de caché y, por lo tanto, menor ahorro y menos ganancia. Es por esto que la mejora obtenida es algo discreta, ya que el ahorro de accesos no es muy significativo para el tamaño de *tile* óptimo.

2.4.3. Actualizaciones en zigzag

Para intentar ahorrar accesos a disco en las actualizaciones de los *tiles* éstas se han reordenado, realizando recorridos en zigzag. La ligera mejora obtenida se puede observar en la gráfica de la Figura 2.23, que muestra las prestaciones obtenidas para las variantes 2 y 3 implementadas con el *run-time* de gestión de memoria y con la reordenación de accesos para evitar fallos de caché. En la gráfica se recogen también las prestaciones cuando no se realiza la reordenación.

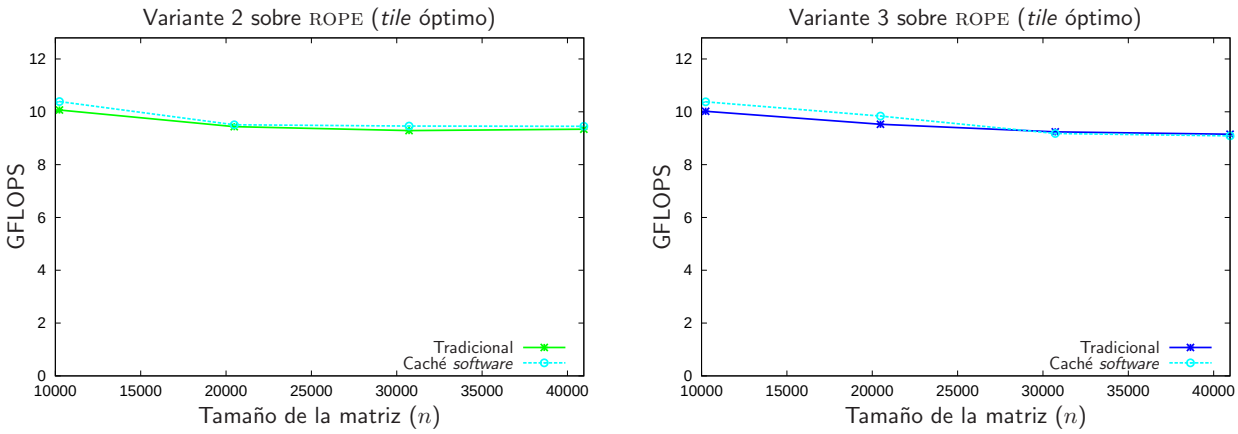


Figura 2.22: Ejecución sobre ROPE de las implementaciones OOC tradicional y con gestión de memoria caché de las variantes 2 y 3 para la factorización de Cholesky.

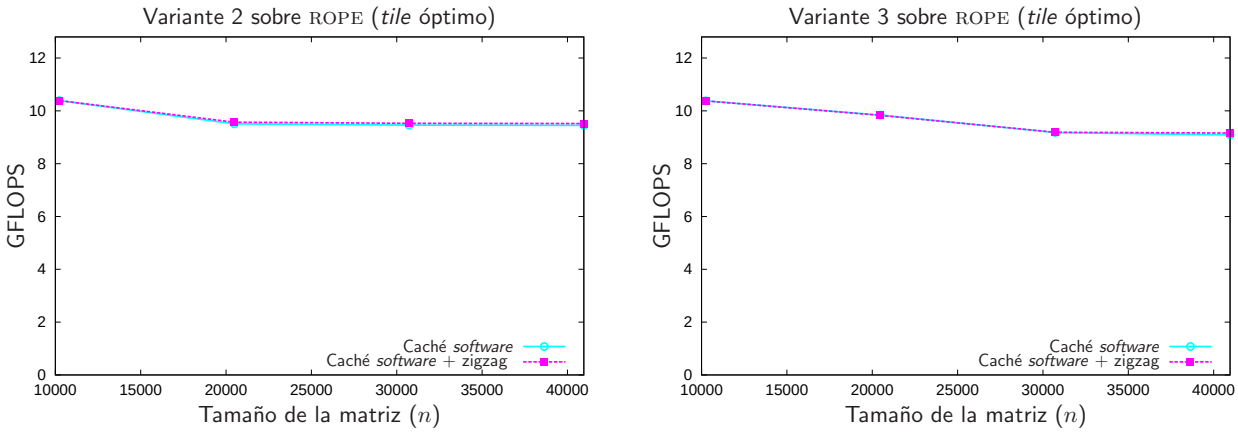


Figura 2.23: Ejecución sobre ROPE de las implementaciones OOC con gestión de memoria y con reordenación de operaciones (zigzag) de las variantes 2 y 3 para la factorización de Cholesky.

Como en el caso anterior, la ganancia aportada por esta modificación es exigua, pues no consigue reducir el número de accesos a disco de manera significativa para el tamaño de *tile* óptimo.

2.4.4. Solapamiento de cálculos y accesos a disco

Las gráficas de la Figura 2.24 ilustran el efecto del tamaño del *tile* en las prestaciones obtenidas sobre un procesador de las variantes 2 y 3, cuando se realiza solapamiento entre cálculos y accesos a disco. Resulta interesante observar que las prestaciones ahora sí crecen notablemente, situándose por encima del 85 % de la velocidad pico de la máquina. Como se puede observar, el rendimiento crece con el tamaño de la matriz.

La Figura 2.25 permite comparar todas las implementaciones realizadas de las variantes 2 y 3 de la factorización de Cholesky OOC. En estas gráficas se han incluido las prestaciones obtenidas al realizar la factorización mediante la rutina POTRF de BLAS disponible en la biblioteca MKL [38].

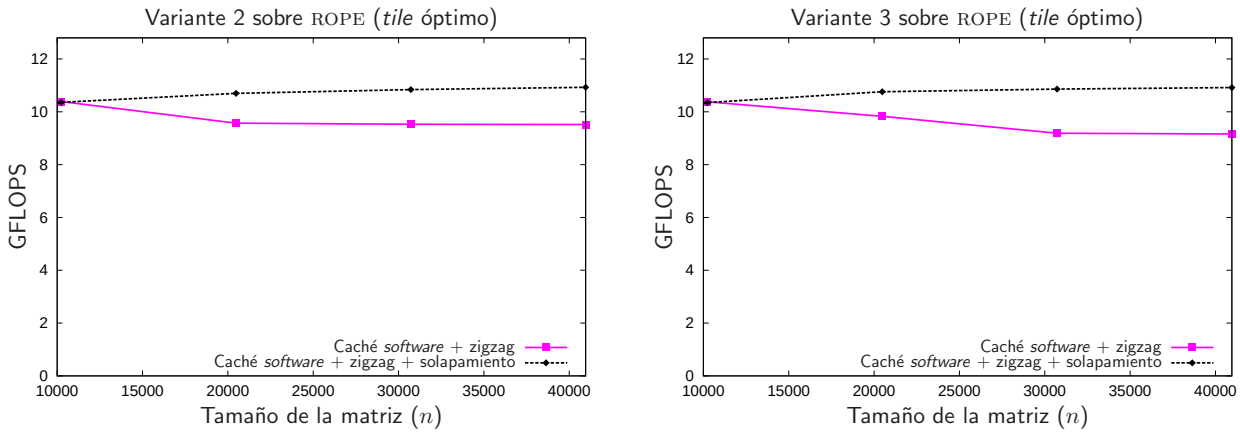


Figura 2.24: Ejecución sobre ROPE de las implementaciones OOC con reordenación de operaciones (zigzag) y con solapamiento de cálculos y E/S de las variantes 2 y 3 para la factorización de Cholesky.

En este caso, cuando la matriz no cabe en memoria, se hace uso de la memoria virtual, decayendo las prestaciones de una manera muy pronunciada. Además, el tamaño de matriz que se puede factorizar está limitado por el tipo de datos de las variables que se utilizan para indexar los elementos de la matriz.

Aunque el sistema de caché *software* y los recorridos en zigzag no presentan una mejora notable sobre la implementación tradicional de cada una de las variantes escogidas, sí son la base imprescindible para la implementación que realiza el solapamiento de la E/S con los cálculos. Es esta última mejora la que sí consigue aumentar significativamente las prestaciones, eliminando el impacto de las esperas a causa de las lecturas y escrituras en disco. Para un tamaño de matriz $n = 40.960$, la variante 2 mejora su rendimiento en un 15 % mientras que la variante 3 lo hace en casi un 20 %. Cabe destacar que las prestaciones obtenidas aumentan conforme crece el tamaño de la matriz. Además, resulta especialmente significativo que el algoritmo OOC que incluye todas las mejoras (caché *software*, recorrido en zigzag y solapamiento de cálculo y E/S) supera en buena medida el rendimiento de la implementación *in-core* de la rutina de factorización de MKL. Esto es debido al almacenamiento en RAM por *tiles*, que se deriva de la utilización de los algoritmos OOC, y que exhibe una mayor localidad de referencia que el almacenamiento tradicional por columnas utilizado por MKL.

2.5. Experimentos sobre arquitecturas multihebra

La mayor parte de los procesadores actuales disponen de varios núcleos, que pueden utilizarse para aumentar la productividad del equipo (en términos de trabajos procesados por unidad de tiempo) o para reducir el tiempo de respuesta de una determinada aplicación. En nuestro caso, utilizaremos los núcleos para reducir el tiempo de procesamiento de las cuatro operaciones computacionales básicas que operan sobre los *tiles*. Como se comentó en la sección anterior, estos núcleos computacionales básicos corresponden, en el caso de la factorización de Cholesky, a las tareas POTRF (factorización de Cholesky de un *tile*), TRSM (resolución de un sistema triangular en el que

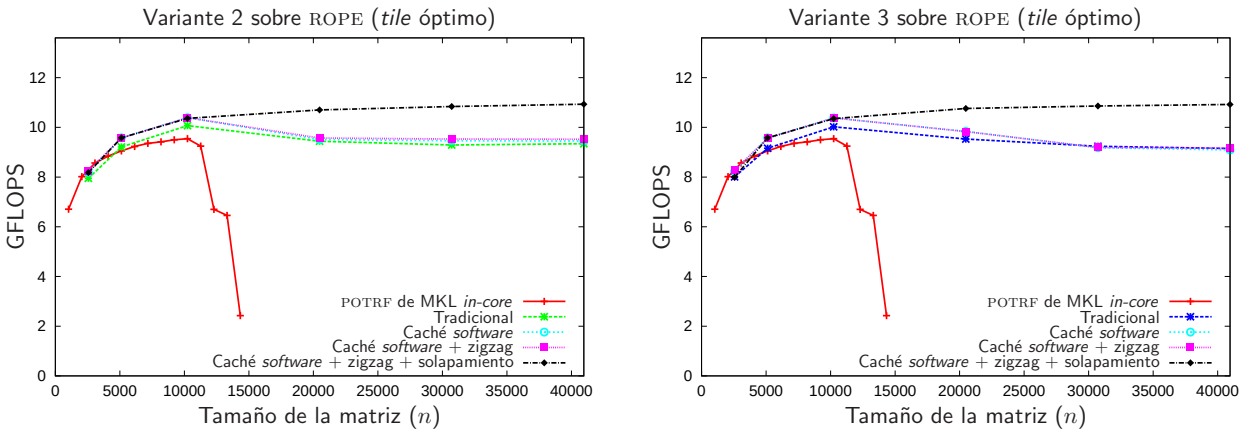


Figura 2.25: Ejecución sobre ROPE de las distintas implementaciones OOC de las variantes 2 y 3 de la factorización de Cholesky.

tanto la matriz de coeficientes como la de términos independientes son *tiles*), GEMM (producto de *tiles*) y SYRK (actualización de rango t de la parte triangular inferior de un *tile*).

En nuestra aproximación paralela, para explotar la multiplicidad de los recursos *hardware*, hemos optado por ejecutar las tareas en el mismo orden en que aparecen en la lista de tareas pendientes, extrayendo el paralelismo únicamente dentro de cada una de las tareas. Para la paralelización hacemos uso de una implementación multihebra de BLAS, optimizada para explotar de manera eficiente el paralelismo que ofrece el *hardware* de los procesadores multinúcleo. Así, cuando se requiere la ejecución de una tarea que involucra una operación de BLAS (como TRSM, GEMM o SYRK), se lanzan tantas hebras como núcleos, que colaboran para ejecutar la tarea en paralelo. De este modo, se consigue un doble nivel de paralelismo o paralelismo anidado: paralelismo entre hebra exploradora y hebra ejecutora, y paralelismo dentro de la hebra ejecutora gracias al uso de BLAS multihebra (Figura 2.26). En nuestra evaluación hemos usado la implementación multihebra de la biblioteca MKL de INTEL. Para la ejecución paralela del núcleo computacional POTRF (de LAPACK) hemos considerado dos opciones. Inicialmente utilizamos la implementación optimizada disponible en la propia biblioteca MKL pero, debido a las bajas prestaciones paralelas de esta implementación y a la importancia de este núcleo computacional en el rendimiento global de la aplicación, decidimos emplear una versión paralelizada mediante una aproximación basada en el flujo de datos (*data-flow*), descrita en [52].

Existe otra alternativa de paralelización, basada en el cálculo concurrente de varias operaciones de la lista de tareas pendientes, siempre que no existan dependencias entre éstas, usando tantos núcleos como tareas. Esta es, básicamente, la aproximación que utilizan los algoritmos *data-flow* a nivel de bloque para incrementar el rendimiento en el cálculo de factorizaciones matriciales *in-core* frente a las paralelizaciones tradicionales basadas en el uso de BLAS multihebra, como la de MKL. Sin embargo, esta misma aproximación no resulta claramente beneficiosa cuando el tamaño del problema es reducido o cuando el tamaño de *tile* es grande. Además, los discos habitualmente empleados no ofrecen buen rendimiento cuando se realizan múltiples peticiones de E/S. Concretamente, el que el tamaño de los *tiles* es relativamente grande (el tamaño de *tile* óptimo en nuestros experimentos habitualmente estaba en el rango 4.000–5.000) y el que el número de invocaciones a

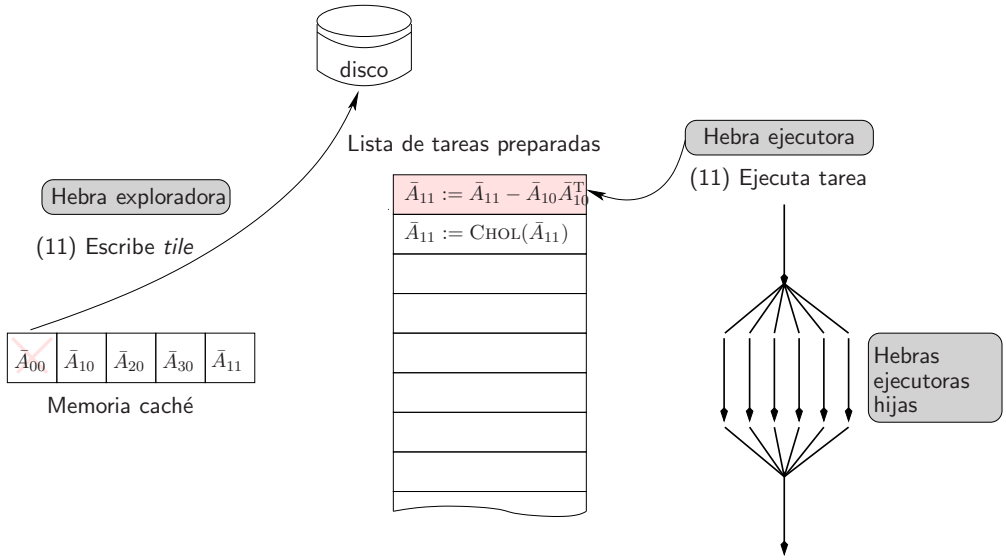


Figura 2.26: Paralelismo anidado en dos niveles. Primer nivel: la hebra exploradora se encarga de la E/S y, en paralelo, la hebra ejecutora se encarga de los cálculos. Segundo nivel: la hebra ejecutora realiza los cálculos en paralelo mediante el BLAS multihebra.

núcleos del BLAS es mucho más elevado que el de las invocaciones a POTRF, nos llevaron a descartar esta segunda opción.

El riesgo que corremos al utilizar varios núcleos para incrementar la velocidad de cómputo es que el problema pase a estar limitado por la E/S, en lugar de estar limitado por el cálculo. Esto puede ocurrir, aun cuando el número de operaciones aritméticas sigue siendo de $O(n^3)$ y el número de E/S de $O(n^2)$, si la relación entre las velocidades de cálculo y de transferencia con el disco no es muy elevada y el tamaño del problema no es suficientemente grande. El objetivo de esta evaluación es comprobar, pues, si éste es el caso cuando utilizamos un procesador con varios núcleos, incrementando la velocidad de cálculo pero manteniendo constante la ratio de E/S.

Para realizar la evaluación, se ha repetido el mismo estudio gradual de la sección anterior, introduciendo las mejoras una tras otra (uso de caché, recorrido en zigzag, solapamiento de cálculos y accesos a disco), determinando los tamaños de *tile* óptimos en cada caso. Además, se realizó un estudio inicial de las prestaciones de las implementaciones multihebra y *data-flow* de la factorización de Cholesky de un *tile* *in-core*, para determinar cuál de estas opciones sería la escogida para realizar la factorización de los *tiles* diagonales en los algoritmos OOC. Por simplicidad, todos estos resultados se recogen en una única figura para la variante 3. Como en la evaluación sobre un sistema monoprocesador, los resultados de la variante 2 son prácticamente idénticos, por lo que no se recogen en esta memoria, mientras que la variante 1 se descartó prácticamente al principio de la evaluación, debido a su peor rendimiento.

Los resultados obtenidos para la variante 3 se muestran en la Figura 2.27. En particular, en la figura se ilustra el rendimiento para matrices *in-core*, tanto por parte de la rutina de BLAS POTRF de MKL, como por la versión paralela *data-flow*, que consigue un nivel más alto de paralelismo mediante la planificación dinámica de tareas. Respecto al trabajo OOC, y como cabría esperar, las mejores prestaciones se consiguen mediante la implementación que combina todas las

mejoras: uso de caché *software*, recorrido en zigzag y solapamiento, llegando incluso a superar las prestaciones de la rutina *in-core* basada en el código *data-flow*. Así, mientras que esta última alcanza los 92,7 GFLOPS con las mayores matrices *in-core*, la mejor implementación OOC llega a los 97 GFLOPS con matrices cuatro veces más grandes, lo que supone un 75,8 % de la velocidad pico de la máquina (128 GFLOPS). En cambio, la versión optimizada de MKL no supera los 52 GFLOPS trabajando con matrices que caben en RAM.

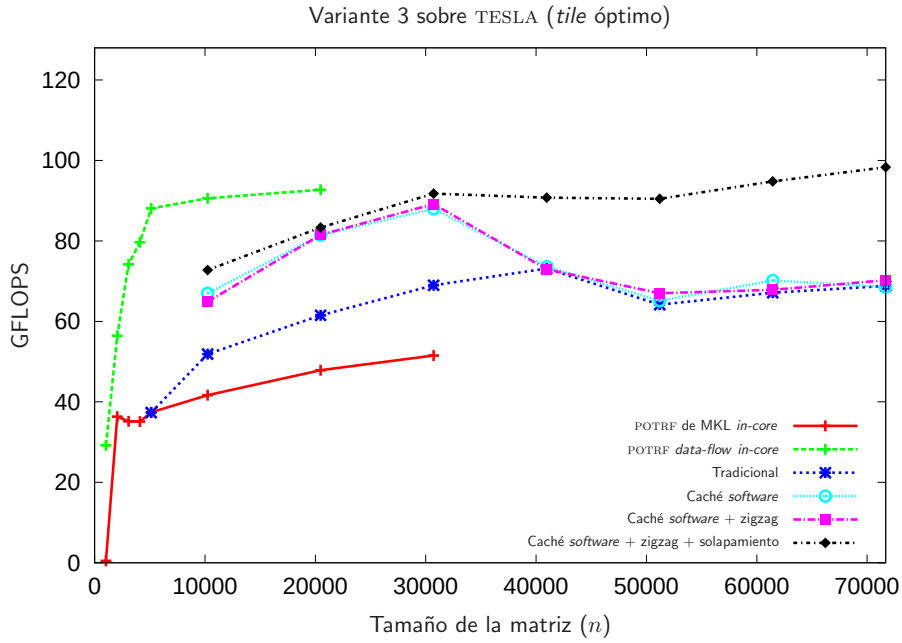


Figura 2.27: Ejecución sobre TESLA de distintas implementaciones de la variante 3 de la factorización de Cholesky.

La Figura 2.28 ofrece los mismos resultados para TESLA2. De nuevo, las mejores prestaciones están alrededor del 76 % de la velocidad pico de la máquina (181 GFLOPS). A diferencia de lo que sucede en TESLA, la mejora que introduce el solapamiento tiene ahora menos impacto porque TESLA2 dispone de seis discos duros con *striping* y más rápidos (15.000 rpm frente a las 7.200 rpm del disco de TESLA). De nuevo, los códigos OOC superan las prestaciones de la rutina *in-core* de MKL, que sólo consigue 72 GFLOPS, menos de un 40 % de la velocidad pico de la máquina.

2.6. Experimentos sobre una arquitectura multihebra con un acelerador gráfico

En esta sección damos un paso más en la aceleración de los cómputos que aparecen en la factorización de Cholesky, utilizando para ello tanto los múltiples núcleos del procesador como los de un acelerador *hardware* de tipo GPU (procesador gráfico). En particular, de las cuatro operaciones computacionales que aparecen durante la factorización de Cholesky, descargaremos sobre la GPU los tres pertenecientes a BLAS: TRSM, GEMM y SYRK, mientras que para el núcleo computacional restante (POTRF) describiremos dos opciones de ejecución en breve. Para realizar

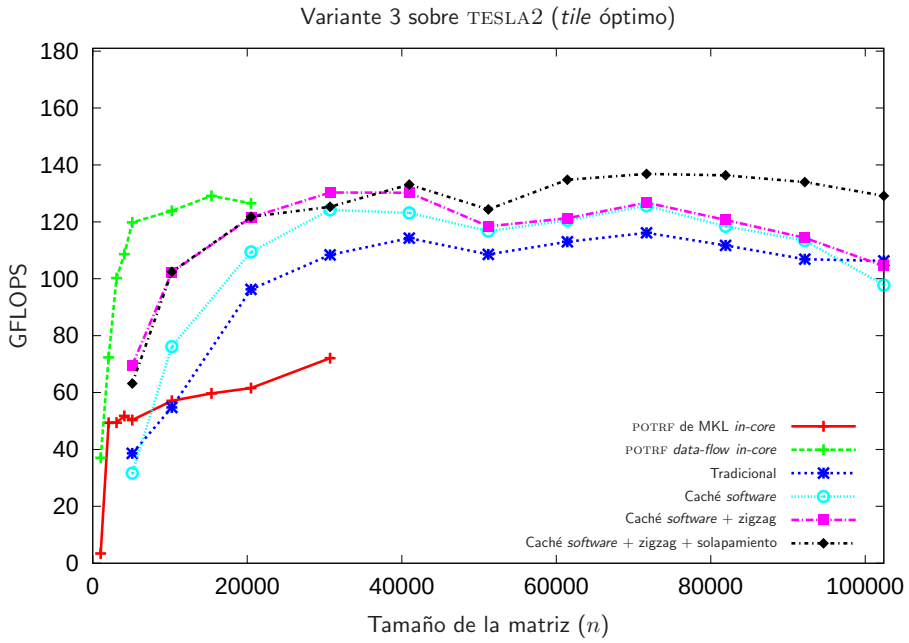


Figura 2.28: Ejecución sobre TESLA2 de distintas implementaciones de la variante 3 de la factorización de Cholesky.

esta asignación de las tareas, en la que algunas deben ejecutarse sobre la GPU, se ha llevado a cabo una adaptación del *run-time* en la que la CPU sigue siendo la encargada de planificar las tareas, y la comunicación entre CPU y GPU se realiza transfiriendo datos entre sus memorias. De este modo, cuando la CPU decide que una tarea que está preparada (sus datos están *in-core*) se ejecute en la GPU, copia los datos a la memoria de la GPU, ordena a ésta que realice los cálculos, y una vez se completa la operación, trae de vuelta los resultados a la memoria principal. Los experimentos realizados en esta sección se han llevado a cabo sobre ZAPE.

En la GPU las operaciones se ejecutan mediante la implementación de BLAS optimizada propia de NVIDIA: CUBLAS. Esta biblioteca no posee rutinas para realizar la factorización de Cholesky (POTRF), por lo que debemos encontrar alternativas para llevarla a cabo cuando se trata de factorizar los *tiles* diagonales. Una primera alternativa consiste en realizar esta factorización haciendo uso de los múltiples núcleos de la CPU, tal y como se ha hecho en la sección anterior. Puesto que se dispone de una GPU, una alternativa diferente consiste en realizar una factorización híbrida del *tile*, en donde participan tanto la CPU como la GPU. Para ello, se realiza un particionado en bloques del *tile*, de manera que cada bloque diagonal se factoriza en la CPU, mientras que las restantes actualizaciones sobre el *tile* se realizan en la GPU, mediante llamadas a rutinas de CUBLAS. Hay que tener en cuenta que, para la factorización híbrida de un *tile*, se emplea la variante 1 de la factorización de Cholesky por sus mejores prestaciones.

Una vez más, para realizar la evaluación se ha desarrollado un estudio gradual en el que las mejoras se han introducido una a una y los tamaños de *tile* óptimos se han determinado experimentalmente para cada tamaño de problema, implementación y variante. En este caso se ha realizado un estudio inicial de las prestaciones de las implementaciones *in-core* correspondientes a la facto-

rización de Cholesky. Por una parte, se ha estudiado la implementación que empleaba únicamente los núcleos de la CPU, y por otra parte, la factorización híbrida que utilizaba tanto la CPU como la GPU, con el objetivo de determinar cuál de estas opciones sería la escogida para realizar la factorización de los *tiles* diagonales en los algoritmos OOC. Esta segunda alternativa ha sido claramente la ganadora. Todos estos resultados se recogen en la Figura 2.29 para la variante 3. Los resultados de la variante 2 fueron muy similares mientras que la variante 1 quedó descartada de inicio por su peor rendimiento. En la figura se muestra el rendimiento para matrices *in-core* de la implementación híbrida que realiza la factorización de la matriz, así como las prestaciones que se obtienen al incorporar las sucesivas mejoras a los algoritmos OOC. Una vez más, las mejores prestaciones se consiguen mediante la implementación que combina las tres mejoras, que incluso supera las prestaciones de la implementación híbrida *in-core*.

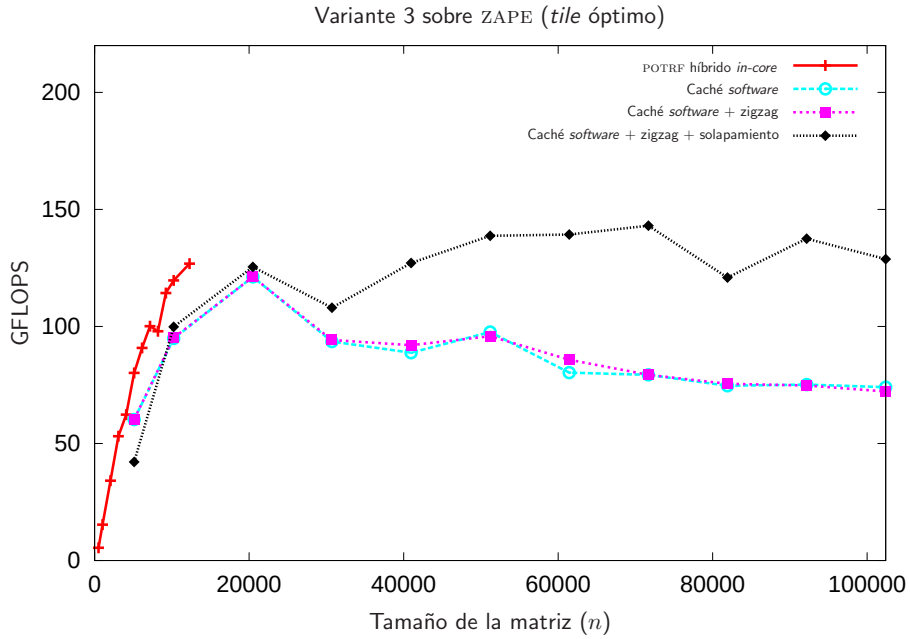


Figura 2.29: Ejecución sobre ZAPE de distintas implementaciones de la variante 3 de la factorización de Cholesky.

2.7. Resumen y conclusiones

Este capítulo ha introducido las principales aportaciones de la tesis en el campo del uso de técnicas OOC en computación matricial, mediante un ejemplo de operación simple como la factorización de Cholesky. En particular, en el capítulo se describen estas aportaciones de forma progresiva:

- Uso de una caché *software* para mejorar la eficiencia de los algoritmos OOC. La caché *software* realiza una función análoga a la caché *hardware* entre el procesador y la memoria principal, reduciendo el número de lecturas y escrituras de datos desde/en disco.

- Incorporación de un entorno de ejecución (*run-time*), a cargo de la E/S, que aisla a la biblioteca de computación matricial del acceso a disco y gestiona la caché de manera transparente al programador.
- Reorganización de los accesos, siguiendo un patrón zigzagueante para incrementar la localidad de referencia.
- Solapamiento de cálculo y accesos a disco, para ocultar la latencia de este último, mediante una prelectura de datos desde el disco a partir de la lista de tareas futuras de la descomposición que se encuentran pendientes de ejecución.
- Modificación del *run-time* para que gestione el solapamiento de cálculo y E/S de manera transparente a los códigos.

El entorno de ejecución supone un avance importante desde el punto de vista de la programabilidad, permitiendo obtener rutinas OOC directamente a partir de algoritmos orientados a bloques, sin necesidad de realizar ninguna modificación sobre éstos, descargando al programador de la tarea de introducir en los códigos operaciones de E/S que organicen el trasiego de datos entre el disco y la memoria principal.

Las implementaciones correspondientes a estas mejoras se han evaluado en tres arquitecturas diferentes, representativas de los sistemas de cálculo actuales: ROPE, una ordenador personal con un único procesador INTEL PENTIUM; TESLA y TESLA2, estaciones de trabajo con dos procesadores INTEL XEON que disponen de cuatro núcleos cada uno; y ZAPE, una estación de trabajo con un procesador AMD PHENOM de cuatro núcleos conectado mediante un bus PCI-e a una GPU GEFORCE 9800 GX2 de NVIDIA. En todos los casos, el rendimiento de la mejor implementación OOC que incorpora las distintas mejoras propuestas iguala o incluso llega a superar el que consigue una implementación eficiente *in-core* para el mayor tamaño de problema que es posible almacenar en la RAM. La conclusión más directa es que, para el cálculo de la factorización de Cholesky en estas tres arquitecturas, las técnicas aportadas en este trabajo hacen que la latencia del disco sea irrelevante.

Capítulo 3

Factorización LU en Disco

En este capítulo se aborda la factorización LU de una matriz de gran dimensión que reside en disco, al no ser posible almacenarla completamente en la memoria principal del sistema. Si la factorización de Cholesky, tratada en el Capítulo 2, es la operación de descomposición que se lleva a cabo en la resolución de sistemas de ecuaciones lineales cuando la matriz de coeficientes es simétrica y definida positiva, la factorización LU juega el mismo papel cuando la matriz de coeficientes no tiene una estructura particular. Nuestro trabajo analiza, desde el punto de vista teórico y práctico, la implementación *Out-of-Core* (OOC) de dos variantes algorítmicas para el cálculo de esta factorización, incorporando un tipo especial de pivotamiento denominado pivotamiento incremental.

Son dos las arquitecturas destino de los algoritmos OOC desarrollados: una plataforma de altas prestaciones con un único procesador de propósito general y un procesador multinúcleo con varios núcleos (*cores*) de propósito general que comparten una memoria común.

El capítulo está estructurado del modo siguiente. En la primera sección se define la factorización LU, se justifica su utilidad, y se ofrecen dos formulaciones recursivas, una escalar y otra por bloques, para su cálculo. En la Sección 3.2 se justifica la necesidad del pivotamiento y se presentan tres variantes algorítmicas escalares para el cálculo de la factorización de forma iterativa con pivotamiento parcial, y las respectivas versiones por bloques. La técnica del pivotamiento incremental se introduce en la Sección 3.3, como una alternativa al pivotamiento parcial, junto a dos variantes algorítmicas orientadas a bloques. Las implementaciones de estas variantes en forma de algoritmos OOC se discuten en la Sección 3.4, que reitera la utilidad de las aportaciones principales de la tesis. Las Secciones 3.5 y 3.6 recogen la evaluación del impacto experimental que suponen las propuestas realizadas y, finalmente, la Sección 3.7 resume el contenido y las aportaciones del capítulo.

3.1. Factorización LU

Una de las estrategias que se utilizan habitualmente para resolver sistemas de ecuaciones lineales densos de la forma $Ax = b$ (o $AX = B$) comienza con la factorización de la matriz coeficiente A, de manera que ésta se descompone en el producto de dos matrices triangulares, que se utilizan después para obtener la solución del sistema inicial resolviendo sendos sistemas triangulares. La factorización LU y la factorización de Cholesky, esta última estudiada en el Capítulo 2, son descomposiciones de este tipo.

La factorización de Cholesky se utiliza cuando la matriz coeficiente A es densa y simétrica definida positiva. En cambio, la factorización LU, combinada con alguna estrategia de pivotamiento

(normalmente un pivotamiento parcial de filas), es el método más común para resolver sistemas lineales cuando la matriz coeficiente A es densa, pero no presenta ninguna estructura particular. En cualquier caso, para poder calcular estas factorizaciones es necesario que la matriz coeficiente sea invertible (ver definición en la Sección 2.1).

Para llevar a cabo la factorización LU de una matriz $A \in \mathbb{R}^{n\times n}$ se aplica una secuencia de $n-1$ transformaciones de Gauss [73], que descomponen la matriz en el producto $A = LU$, donde $L \in \mathbb{R}^{n\times n}$ es triangular inferior unidad (los elementos de su diagonal son todos iguales a 1) y $U \in \mathbb{R}^{n\times n}$ es triangular superior. A continuación se definen las transformaciones de Gauss y se establece formalmente la relación entre el proceso de eliminación Gaussiana y la factorización LU de una matriz.

Definición 3 *Una transformación de Gauss es una matriz que tiene la forma*

$$L_k = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & l_{21} & I_{n-k-1} \end{array}\right), \quad 0 \le k < n. \tag{3.1}$$

Una de las propiedades más interesantes de las transformaciones de Gauss es su facilidad de inversión y aplicación. Así, la inversa de la transformación de Gauss en (3.1) viene dada por

$$L_k^{-1} = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & -l_{21} & I_{n-k-1} \end{array}\right), \quad 0 \le k \le n,$$

mientras que, por ejemplo, al aplicar la inversa de L_0 a una matriz A tenemos que

$$L_0^{-1}A = \left(\begin{array}{c|c} 1 & 0 \\ \hline -l_{21} & I_{n-1} \end{array}\right)\left(\begin{array}{c|c} \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline a_{21}-\alpha_{11}l_{21} & A_{22}-l_{21}a_{12}^{\mathrm{T}} \end{array}\right). \tag{3.2}$$

Sin embargo, la utilidad principal de las transformaciones de Gauss reside en su capacidad de anular los elementos subdiagonales de una columna de una matriz. Por ejemplo, tomando $l_{21} := a_{21}/\alpha_{11}$ en (3.2), se obtiene $a_{21} - \alpha_{11}l_{21} = 0$, de modo que se anulan los elementos bajo la diagonal de la primera columna de A.

Otra propiedad interesante es que las transformaciones de Gauss se pueden acumular fácilmente, tal y como se muestra a continuación:

$$\left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^{\mathrm{T}} & 1 & 0 \\ \hline L_{20} & 0 & I_{n-k-1} \end{array}\right) L_k = \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^{\mathrm{T}} & 1 & 0 \\ \hline L_{20} & l_{21} & I_{n-k-1} \end{array}\right),$$

donde $L_{00} \in \mathbb{R}^{k\times k}$.

Con estas herramientas, ya estamos en disposición de describir el uso de las transformaciones de Gauss para triangularizar una matriz (obteniendo en la práctica la factorización LU). En concreto, si después de k pasos se ha sobreescrito la matriz inicial A con $L_{k-1}^{-1}\cdots L_1^{-1}L_0^{-1}A$, habiendo escogido

las transformaciones de Gauss $L_j, 0 \leq j < k$, de modo que anulen los elementos subdiagonales de la columna j de $L_{j-1}^{-1} \cdots L_1^{-1} L_0^{-1} A$, se tiene

$$L_{k-1}^{-1} \cdots L_1^{-1} L_0^{-1} A = \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & A_{BR} \end{array} \right),$$

donde $U_{TL} \in \mathbb{R}^{k \times k}$ es triangular superior. Reparticionando

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & u_{12} & U_{02} \\ \hline 0 & \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline 0 & a_{21} & A_{22} \end{array} \right) \quad \text{y tomando} \quad L_k = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & l_{21} & I_{n-k-1} \end{array} \right),$$

donde $l_{21} := a_{21}/\alpha_{11}$, se tiene

$$\begin{aligned} L_k^{-1}(L_{k-1}^{-1} \cdots L_1^{-1} L_0^{-1} A) &= \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & -l_{21} & I_{n-k-1} \end{array} \right) \left(\begin{array}{c|c|c} U_{00} & u_{12} & U_{02} \\ \hline 0 & \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline 0 & a_{21} & A_{22} \end{array} \right) \\ &= \left(\begin{array}{c|c|c} U_{00} & u_{12} & U_{02} \\ \hline 0 & \mu_{11} & u_{12}^{\mathrm{T}} \\ \hline 0 & 0 & A_{22} - l_{21} u_{12}^{\mathrm{T}} \end{array} \right). \end{aligned}$$

Este razonamiento inductivo se puede utilizar para confirmar que se puede reducir una matriz a la forma triangular superior aplicando sucesivamente las transformaciones de Gauss que anulan sus elementos subdiagonales, lo que es equivalente a realizar una factorización LU. De hecho, si $L_{n-1}^{-1} \cdots L_1^{-1} L_0^{-1} A = U$ entonces $A = LU$, donde $L = L_0 L_1 \cdots L_{n-1}$ se construye de manera trivial a partir de las columnas de las correspondientes transformaciones de Gauss.

Una vez obtenida la factorización LU de la matriz A, se puede obtener la solución del sistema lineal $Ax = b$ resolviendo dos sistemas triangulares: en primer lugar se obtiene la solución del sistema $Ly = b$ mediante eliminación progresiva y, a continuación, se resuelve el $Ux = y$ mediante sustitución regresiva. Los procedimientos de eliminación progresiva y de sustitución regresiva son muy sencillos [32], y presentan un coste de orden menor frente al de la propia factorización, por lo que su cálculo, cuando la matriz reside en disco, no se aborda explícitamente en nuestro trabajo.

Para concluir esta introducción, y antes de presentar los algoritmos para el cálculo de la factorización LU de una matriz, se ofrece un resultado en forma de teorema que establece las condiciones necesarias para la existencia de esta factorización.

Definición 4 *Dada la partición de la matriz A como $A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right)$, donde $A_{TL} \in \mathbb{R}^{k \times k}$, se denominan menores principales de A a las submatrices A_{TL}, $1 \leq k \leq n$.*

El siguiente teorema establece las condiciones para que exista la factorización LU de una matriz.

Teorema 2 (Factorización LU) *Una matriz $A \in \mathbb{R}^{n \times n}$ tiene una factorización LU si sus $n-1$ primeros menores principales son invertibles. Si la factorización LU existe y A es invertible, entonces dicha factorización es única.*

Golub y Van Loan [32] dan una demostración de este teorema.

3.1.1. Algoritmo escalar

A continuación se obtiene, mediante un proceso de derivación formal, un algoritmo para calcular la factorización LU de una matriz $A \in \mathbb{R}^{n \times n}$. Durante este proceso (y en todos los que presentaremos en este capítulo en forma de algoritmos), los elementos bajo la diagonal de la matriz L sobreescriben el triángulo inferior estricto de A, mientras que los elementos de la matriz U sobreescriben el triángulo superior. La diagonal de la matriz L no se almacena puesto que sus elementos son todos la unidad. Consideremos las siguientes particiones de la matriz de coeficientes densa A y de sus factores L y U:

$$A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline a_{21} & A_{22} \end{array}\right), \quad L = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right), \quad U = \left(\begin{array}{c|c} \mu_{11} & u_{12}^{\mathrm{T}} \\ \hline 0 & U_{22} \end{array}\right),$$

siendo α_{11} y μ_{11} escalares; a_{21}, l_{21} y u_{12} vectores de $n-1$ elementos; y A_{22}, L_{22} y U_{22} matrices de tamaño $(n-1) \times (n-1)$. A partir de la expresión $A = LU$ se tiene que:

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} \mu_{11} & u_{12}^{\mathrm{T}} \\ \hline 0 & U_{22} \end{array}\right) = \left(\begin{array}{c|c} \mu_{11} & u_{12}^{\mathrm{T}} \\ \hline \mu_{11} l_{21} & l_{21} u_{12}^{\mathrm{T}} + L_{22} U_{22} \end{array}\right),$$

de donde se derivan las siguientes expresiones:

$$\begin{array}{rcl} \mu_{11} & = & \alpha_{11}, \\ u_{12}^{\mathrm{T}} & = & a_{12}^{\mathrm{T}}, \\ l_{21} & = & a_{21}/\mu_{11}, \\ A_{22} - l_{21} u_{12}^{\mathrm{T}} & = & L_{22} U_{22}. \end{array}$$

Se obtiene así una formulación recursiva de un algoritmo escalar para calcular la factorización LU, en la que se sobreescriben los elementos de A con las correspondientes entradas de los factores L y U, tal y como se muestra a continuación:

1. Particionar $A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline a_{21} & A_{22} \end{array}\right)$.
2. $\alpha_{11} := \mu_{11} = \alpha_{11}$.
3. $a_{12}^{\mathrm{T}} := u_{12}^{\mathrm{T}} = a_{12}^{\mathrm{T}}$.
4. $a_{21} := l_{21} = a_{21}/\mu_{11}$.
5. $A_{22} := A_{22} - l_{21} u_{12}^{\mathrm{T}}$.
6. Continuar recursivamente con $A = A_{22}$ en el paso 1.

Éste es un algoritmo orientado hacia la derecha (*right–looking*) ya que, en cada iteración, se calcula una nueva columna del factor L, se "mira" a la derecha para obtener una nueva fila del factor U y se actualiza la submatriz que queda encerrada a la derecha y por debajo de la nueva columna y la nueva fila, respectivamente. Los pasos 2 y 3, aunque no conllevan operaciones aritméticas, se han incluido en el algoritmo para mostrar cómo queda la matriz A sobreescrita con sus factores L y U. El paso 4 calcula los parámetros que definen una transformación de Gauss que es la transformación que permite anular los elementos que se encuentran por debajo de la diagonal.

El algoritmo escalar que obtiene la factorización LU tiene un coste de $2n^3/3$ operaciones aritméticas en coma flotante (*flops*). Al igual que en el algoritmo de la factorización de Cholesky, el mayor volumen de cálculos de este nuevo algoritmo está en la actualización de rango 1, que realiza $O(n^2)$ operaciones aritméticas sobre $O(n^2)$ datos. De nuevo, para obtener códigos que ofrezcan un buen rendimiento, es necesario reformular el algoritmo en términos de productos de matrices (o actualizaciones de rango k), de modo que se incremente la relación entre el número de operaciones aritméticas y el volumen de datos, haciendo posible una mayor reutilización de datos en los niveles de la jerarquía de memoria más próximos al procesador. El algoritmo por bloques que se presenta a continuación persigue este objetivo.

3.1.2. Algoritmo por bloques

Un algoritmo por bloques es un algoritmo que se basa en operaciones matriz–matriz, por lo que el particionado de las matrices debe realizarse de modo que en cada paso se trabaje con submatrices, en lugar de operar con escalares y vectores, como era el caso del algoritmo escalar. Por lo tanto, partiremos de particionados de A, L y U como los siguientes:

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right), \quad L = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right), \quad U = \left(\begin{array}{c|c} U_{11} & U_{12} \\ \hline 0 & U_{22} \end{array}\right),$$

donde A_{11}, L_{11} y U_{11} son submatrices (o bloques) de tamaño $b \times b$ (con b mucho menor que el tamaño de A), y A_{22}, L_{22} y L_{22} son matrices con $(n-b) \times (n-b)$ elementos. A partir de la expresión $A = LU$, con el particionado anterior, tenemos que:

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} U_{11} & U_{12} \\ \hline 0 & U_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11}U_{11} & L_{11}U_{12} \\ \hline L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{array}\right),$$

de donde se obtienen las expresiones:

$$\begin{aligned} L_{11}U_{11} &= A_{11}, \\ U_{12} &= L_{11}^{-1}A_{12}, \\ L_{21} &= A_{21}U_{11}^{-1}, \\ A_{22} - L_{21}U_{12} &= L_{22}U_{22}. \end{aligned}$$

Así pues, el algoritmo recursivo por bloques que calcula la factorización LU de A, sobreescribiendo ésta con los factores L y U, puede formularse como sigue:

1. Particionar $A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)$, siendo A_{11} de tamaño $b \times b$ (o menor, si A no es suficientemente grande).
2. $A_{11} := \{L\backslash U\}_{11}$ tal que $A_{11} = L_{11}U_{11}$. (Es decir, obtener la factorización LU de A_{11}, sobreescribiendo este bloque con los correspondientes elementos de L_{11} y U_{11}.)
3. $A_{12} := U_{12} = L_{11}^{-1}A_{12}$.
4. $A_{21} := L_{21} = A_{21}U_{11}^{-1}$.
5. $A_{22} := A_{22} - L_{21}U_{12}$.

6. Continuar recursivamente con $A = A_{22}$ en el paso 1.

Este algoritmo por bloques está orientado hacia la derecha, al igual que el algoritmo escalar. Así, tras factorizar el bloque diagonal A_{11}, se actualizan los bloques que se encuentran a la derecha y por debajo de éste. El coste de esta formulación por bloques es también de $2n^3/3$ operaciones aritméticas.

En este algoritmo, tanto el cálculo de los paneles A_{12} y A_{21}, como la actualización de la submatriz A_{22}, se pueden llevar a cabo con rutinas de BLAS 3. La mayor parte de los cálculos se realizan, precisamente, en esta última operación, un producto de matrices que conlleva $O(n^2b)$ operaciones sobre $O(n^2)$ datos. Si $b \gg 1$, mediante el algoritmo por bloques se consigue, por tanto, aumentar la relación entre operaciones y datos (en un factor proporcional a b), aumentando en consecuencia la reutilización de éstos.

3.2. Factorización LU con pivotamiento parcial

La solución del sistema lineal $Ax = b$ existe y es única cuando la matriz A es invertible. Sin embargo, éste no es el único requisito que debe cumplir A para que exista su factorización LU. Por ejemplo, si A es invertible pero algún elemento de su diagonal es nulo, la factorización no se puede llevar a cabo, ya que aparece una división por cero durante el propio proceso. En este caso, el requisito que se incumple es el de que los $n-1$ primeros menores principales de A sean invertibles.

El pivotamiento de filas es una estrategia que se puede utilizar para soslayar este problema: si la fila con el elemento diagonal nulo se intercambia con otra fila situada más abajo en la matriz y que no tenga nulo el elemento correspondiente, la factorización puede proseguir. Hay que tener en cuenta que los intercambios realizados deberán reflejarse también sobre el vector b antes de resolver el sistema lineal.

Otro problema que podemos encontrar al realizar la factorización LU, y que también se evita mediante el pivotamiento de filas, sucede cuando, durante el proceso de factorización, aparece un elemento diagonal mucho menor (en magnitud) que el resto de los elementos que se encuentran por debajo en su misma columna. Cuando esto sucede, y aún cumpliéndose los requisitos para poder calcular la factorización LU, la magnitud de los elementos del factor U crece considerablemente, lo que conlleva errores de redondeo importantes cuando se trabaja en aritmética finita [73].

Un pivotamiento de filas que deje en la diagonal el elemento más grande en magnitud de entre los situados en y por debajo de la diagonal de la columna que se está tratando en cada iteración, garantiza que todos los elementos de L son menores o iguales a 1, modera el crecimiento de los elementos de U y, con ello, limita los errores de redondeo. A esta técnica de pivotamiento de filas se le denomina pivotamiento parcial, para distinguirlo del pivotamiento completo, que involucra tanto a filas como a columnas.

3.2.1. Algoritmo escalar

La introducción del pivotamiento parcial en el algoritmo escalar recursivo que obtiene la factorización LU de una matriz conlleva la inserción de dos nuevas operaciones tras el paso inicial en el que se realiza el particionado: un primer paso en el que se debe encontrar el elemento de mayor magnitud de la primera columna de la submatriz con la que se trabaja, y un segundo paso en el que se permutan la primera fila y la fila del elemento de mayor magnitud encontrado.

El algoritmo resultante se muestra a continuación:

1. Particionar $A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline a_{21} & A_{22} \end{array}\right)$.
2. $\pi :=$ índice del mayor elemento (en valor absoluto) de $\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$.
3. Intercambiar la fila $\left(\begin{array}{c|c} \alpha_{11} & a_{12}^{\mathrm{T}} \end{array}\right)$ con los correspondientes elementos de la fila cuyo índice es π.
4. $\alpha_{11} := \mu_{11} = \alpha_{11}$.
5. $a_{12}^{\mathrm{T}} := u_{12}^{\mathrm{T}} = a_{12}^{\mathrm{T}}$.
6. $a_{21} := l_{21} = a_{21}/\mu_{11}$.
7. $A_{22} := A_{22} - l_{21}u_{12}^{\mathrm{T}}$.
8. Continuar recursivamente con $A = A_{22}$ en el paso 1.

La misma secuencia de permutaciones que se efectúa al calcular la factorización LU con pivotamiento parcial debe aplicarse sobre el vector de términos independientes b. Habitualmente, esta aplicación se realiza tras el cálculo de la factorización LU, por lo que se hace necesario guardar la información que permita recuperar esta secuencia. En la práctica, un vector de n elementos es suficiente para mantener esta información.

Formalmente, las permutaciones de una matriz se capturan mediante el siguiente par de definiciones.

Definición 5 *Una matriz de permutación $P \in \mathbb{R}^{n\times n}$ es equivalente a la matriz identidad con algunas de sus filas y/o columnas intercambiadas.*

Definición 6 *Si P es la matriz identidad con las filas i y j intercambiadas, entonces aplicar P sobre una matriz A como PA realiza una permutación de las filas i y j de A.*

El algoritmo que se acaba de mostrar es el que se encuentra implementado en LINPACK [49]. Una particularidad de este algoritmo es que la aplicación de las transformaciones de Gauss y de las permutaciones sobre b debe hacerse de manera intercalada, exactamente del mismo modo que se ha realizado sobre A para obtener U. Así, si $P_k \in \mathbb{R}^{n\times n}$ y $L_k \in \mathbb{R}^{n\times n}$ son, respectivamente, la matriz de permutación y la transformación de Gauss aplicadas durante la etapa k, se tiene que $A = (P_0L_0P_1L_1 \cdots P_{n-1}L_{n-1})U$ y, por lo tanto, la solución del sistema lineal $Ax = b$ puede obtenerse de $Ux = (L_{n-1}^{-1}P_{n-1} \cdots L_1^{-1}P_1L_0^{-1}P_0)b$. Desgraciadamente, intercalar las permutaciones con la aplicación de las transformaciones de Gauss no permite el desarrollo de un algoritmo por bloques equivalente, ya que cada transformación de Gauss se aplica mediante una actualización de rango 1 (GER de BLAS 2).

Otro modo de realizar el pivotamiento de filas es el que se implementa en LAPACK [48]. En esta biblioteca, cada intercambio afecta a todos los elementos de las filas implicadas. Así, en el paso 3 del algoritmo, los elementos que están a la izquierda del elemento α_{11} (y que no aparecen explícitamente, debido a la formulación recursiva del algoritmo), junto con $\left(\begin{array}{c|c} \alpha_{11} & a_{12}^{\mathrm{T}} \end{array}\right)$, son permutados con todos los elementos de la fila cuyo índice es π. Hay que tener en cuenta que, al sobreescribir A con los factores triangulares resultantes de la factorización, estos elementos corresponden a entradas de L. Realizar el pivotamiento sobre las filas completas es equivalente a agrupar las permutaciones, de modo que no es necesario aplicarlas de manera intercalada con las transformaciones de Gauss.

En concreto, si la matriz $P = P_{n-1} \cdots P_1 P_0$ contiene las permutaciones realizadas durante la factorización, la solución del sistema lineal $Ax = b$ se obtiene resolviendo los sistemas $Ly = Pb$ y $Ux = y$. Como se puede ver, las permutaciones se realizan en primer lugar sobre la parte derecha b, y después se lleva a cabo la resolución de sendos sistemas triangulares, tal y como se haría si no hubiera necesidad de realizar el pivotamiento.

En lo que respecta a la estabilidad numérica, ésta depende del factor de crecimiento [59]

$$\rho = \frac{\|L\| \, \|U\|}{\|A\|},$$

que viene determinado básicamente por el tamaño del problema y la estrategia de pivotamiento. El factor de crecimiento indica en qué medida crecen los valores numéricos de los factores triangulares durante el proceso de eliminación. Aunque la incorporación del pivotamiento de filas no garantiza la estabilidad numérica del algoritmo de la factorización LU, en la práctica esta combinación raramente falla (sólo para matrices especialmente construidas para este propósito se ha podido comprobar que el método produce un crecimiento desmesurado de las entradas del factor U [36, 32]). En consecuencia, el pivotamiento parcial de filas es la técnica habitualmente aplicada en las bibliotecas de resolución de sistemas lineales densos, por su sencillez y menor coste frente a otras como el pivotamiento completo.

Adaptar la formulación recursiva del algoritmo anterior para este nuevo modo de pivotamiento no resulta un paso directo, puesto que el pivotamiento afecta a la submatriz sobre la que se realizan los cálculos en cada iteración (A_{22} de la iteración anterior), así como a los elementos que han sido ya factorizados a la izquierda en iteraciones anteriores. En este caso resulta más adecuada la formulación del algoritmo en notación FLAME, tanto para el algoritmo escalar como para el algoritmo por bloques.

3.2.2. Algoritmos en notación FLAME

Al seguir el proceso de derivación formal de algoritmos propuesto por FLAME para la factorización LU se obtienen cinco variantes algorítmicas. Sólo en tres de estas cinco variantes la combinación con el pivotamiento parcial de filas da lugar a algoritmos correctos [72]. Los algoritmos escalares y por bloques de estas tres variantes se muestran en la Figura 3.1 (izquierda y derecha, respectivamente). En estos algoritmos, $n(\cdot)$ denota el número de columnas de una matriz y $P(\pi_1)$ denota la matriz de permutación que, al aplicarla a otra matriz, intercambia la primera fila de ésta por la fila π_1. Si $p_1 = (\pi_1, \pi_2, \ldots, \pi_b)$, $P(p_1)$ denota la matriz de permutación que, aplicada a otra matriz, intercambia la primera fila de ésta por la fila π_1, la segunda fila por la fila π_2, y así sucesivamente hasta cambiar la fila b por la fila π_b. La secuencia de permutaciones aplicadas en los algoritmos queda recogida en el vector resultado p. La función *PivIndex* devuelve el índice del elemento de mayor magnitud del vector que se le pasa como parámetro. La función TRILU$(\cdot)$ devuelve la matriz triangular inferior que se almacena en la matriz se le pasa como parámetro, poniendo a unos la diagonal (el resultado es triangular inferior unidad). Su utilización indica, en nuestro caso, que la operación $A_{12} := \text{TRILU}(A_{11})^{-\text{T}} A_{12}$ consiste en la resolución de un sistema triangular, que corresponde a la operación TRSM de BLAS ($A := L^{-1}A$). Las demás operaciones de cálculo de los algoritmos por bloques son todas del tipo GEMM de BLAS ($C := C + A \cdot B$). Nótese que el algoritmo escalar de la variante 1 sigue el esquema del algoritmo recursivo del Apartado 3.2.1, esta vez realizando el pivotamiento sobre filas completas.

Algorithm: $[A, p] := \text{LU\_PIV\_UNB}(A)$

Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \; p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$

where A_{TL} is 0×0 and p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

where α_{11} and π_1 are 1×1

Variante 1:

$$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) := P(\pi_1) \left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

$$a_{21} := a_{21}/\alpha_{11}$$

$$A_{22} := A_{22} - a_{21} a_{12}^T$$

Variante 2:

$$\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) := P(p_0) \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$$a_{01} := \text{TRILU}(A_{00})^{-1} a_{01}$$

$$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$$

$$a_{21} := a_{21} - A_{20} a_{01}$$

$$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c} a_{10}^T & \alpha_{11} \\ \hline A_{20} & a_{21} \end{array}\right) := P(\pi_1) \left(\begin{array}{c|c} a_{10}^T & \alpha_{11} \\ \hline A_{20} & a_{21} \end{array}\right)$$

$$a_{21} := a_{21}/\alpha_{11}$$

Variante 3:

$$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$$

$$a_{21} := a_{21} - A_{20} a_{01}$$

$$a_{12}^T := a_{12}^T - a_{10}^T A_{02}$$

$$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) := P(\pi_1) \left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

$$a_{21} := a_{21}/\alpha_{11}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Algorithm: $[A, p] := \text{LU\_PIV\_BLK}(A)$

Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \; p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$

where A_{TL} is 0×0 and p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

where A_{11} is $b \times b$ and p_1 has b elements

Variante 1:

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LU\_PIV\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1) \left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$$

$$A_{12} := \text{TRILU}(A_{11})^{-1} A_{12}$$

$$A_{22} := A_{22} - A_{21} A_{12}$$

Variante 2:

$$\left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right) := P(p_0) \left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right)$$

$$A_{01} := \text{TRILU}(A_{00})^{-1} A_{01}$$

$$A_{11} := A_{11} - A_{10} A_{01}$$

$$A_{21} := A_{21} - A_{20} A_{01}$$

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LU\_PIV\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$$\left(\begin{array}{c} A_{10} \\ \hline A_{20} \end{array}\right) := P(p_1) \left(\begin{array}{c} A_{10} \\ \hline A_{20} \end{array}\right)$$

Variante 3:

$$A_{11} := A_{11} - A_{10} A_{01}$$

$$A_{21} := A_{21} - A_{20} A_{01}$$

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LU\_PIV\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1) \left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$$

$$A_{12} := A_{12} - A_{10} A_{02}$$

$$A_{12} := \text{TRILU}(A_{11})^{-1} A_{12}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Figura 3.1: Variantes de los algoritmos escalares (izquierda) y por bloques (derecha) para calcular la factorización LU con pivotamiento parcial.

Como paso previo a la implementación OOC de los algoritmos por bloques es conveniente obtener algoritmos orientados a bloques, que son aquellos en los que cada operación involucra, en general, a unos pocos bloques cuadrados (aproximadamente del mismo tamaño). En este tipo de algoritmos, la matriz a factorizar se trata como una matriz de submatrices (bloques). Estas submatrices serán después los *tiles* en los algoritmos OOC. Debido a la introducción del pivotamiento, la formulación por bloques descrita en esta sección no desemboca fácilmente en los deseados algoritmos orientados a bloques, ya que tanto la columna que se debe utilizar para decidir qué fila permutar, como las filas que se permutan, se extienden a través de distintos bloques, pudiendo llegar a involucrar al bloque diagonal y a todos los subdiagonales en cada etapa. Esta falta de localidad de referencia espacial es especialmente perjudicial cuando estamos tratando con el disco y el subsistema de E/S.

Por otra parte, una solución OOC basada en algoritmos que incorporan un pivotamiento parcial sufre de problemas de escalabilidad (a medida que el tamaño del problema crece, el rendimiento no solo no aumenta sino que mengua). Esto es debido a que son algoritmos que trabajan sobre paneles (secciones verticales) de la matriz de modo que el pivotamiento parcial tenga cabida dentro del panel. Así, al aumentar el tamaño de la matriz, el panel deberá tener menos columnas, de modo que siga cabiendo en memoria. Sin embargo, estrechar el panel limita el uso de operaciones de BLAS 3, lo que provoca una pérdida de prestaciones.

Puesto que la factorización LU con pivotamiento parcial es un algoritmo poco escalable y que resulta difícil de formular como un algoritmo orientado a bloques, que además requiere un gran número de actualizaciones con una carga computacional mínima (permutar filas), nos proponemos utilizar una estrategia de pivotamiento diferente para llevar a cabo la factorización OOC de modo más eficiente, garantizando una aceptable calidad numérica.

3.3. Factorización LU con pivotamiento incremental

Joffrain et al. [39] presentan un algoritmo para calcular la factorización LU de una matriz con *pivotamiento incremental*. El algoritmo surgió ante la cuestión de cómo actualizar una factorización LU existente cuando cambian algunas de las filas y/o columnas de la matriz que da origen a la misma. Este problema aparece cuando es necesario calcular la factorización LU de la matriz A, particionada como

$$A = \left(\begin{array}{c|c} B & C \\ \hline D & E \end{array} \right), \tag{3.3}$$

para diferentes valores de C, D y E. La cuestión se transforma en cómo reutilizar la factorización de B para obtener la de A para diferentes C, D y E. En las aplicaciones que dan lugar a esta cuestión, la dimensión de B es normalmente mucho mayor que la de las demás submatrices, por lo que resulta muy conveniente factorizar B una sola vez y actualizar la factorización de A cuando cambian C, D y E.

En concreto, algunas aplicaciones del campo de los métodos de elementos de contorno (*boundary element methods*) conllevan la resolución de grandes sistemas lineales densos. En muchas de estas aplicaciones, el objetivo es optimizar una característica de un objeto, como puede ser la forma de alguno de sus componentes. Si los grados de libertad del componente cuya forma se quiere optimizar se colocan al final de todos los grados de libertad, la matriz presenta la estructura de (3.3). La

búsqueda de la forma óptima de este componente requiere entonces la solución de sistemas lineales de la forma (3.3) para diferentes bloques C, D y E [30].

3.3.1. Algoritmo básico

A continuación se presenta el algoritmo básico que calcula la factorización LU con pivotamiento incremental de la matriz A en (3.3) [39]:

- **Paso 1**: Factorizar B. En este paso se calcula la factorización LU con pivotamiento parcial de B. El pivotamiento se realiza sobre filas completas de la matriz, al estilo de LAPACK. Los factores triangulares sobreescriben la propia matriz B y, además, se hace necesario el uso de un vector p para almacenar los índices que definen la secuencia de permutaciones aplicada durante esta factorización. Denotamos los cálculos realizados en este primer paso como

$$[B, p] := [\{L\backslash U\}, p] = \mathrm{LU}(B)$$

- **Paso 2**: Actualizar C consistentemente con la factorización de B. Para ello se debe llevar a cabo un proceso de eliminación progresiva con el factor L obtenido en el paso anterior. Además, puesto que en la factorización de B se ha realizado el pivotamiento al estilo de LAPACK, todas las permutaciones contenidas en p se podrán aplicar sobre C al principio de esta etapa. Es decir, $C := L^{-1} \cdot P(p) \cdot C$, que denotamos a partir de ahora como

$$C := \textsc{Apply\_L}(B, p, C)$$

- **Paso 3**: Factorizar $\begin{pmatrix} U \\ D \end{pmatrix}$. En este paso se calcula la factorización LU con pivotamiento parcial, esta vez al estilo de LINPACK, permutando sólo los elementos que se encuentran a la derecha del elemento diagonal. De este modo se preserva la estructura triangular superior de U. Consideraremos que el vector r almacena los índices de las permutaciones empleadas en esta factorización. En nuestra notación esto se refleja como

$$\left[\begin{pmatrix} U \\ D \end{pmatrix}, \bar{L}, r\right] := \left[\begin{pmatrix} \{\bar{L}\backslash\bar{U}\} \\ \check{L} \end{pmatrix}, r\right] = \mathrm{LU}_{\mathrm{TD}}^{\mathrm{LIN}} \begin{pmatrix} U \\ D \end{pmatrix}$$

  En este paso, $\bar{U}$ sobreescribe la parte triangular superior de B, que contenía U tras el paso 1. La matriz triangular inferior $\bar{L}$ que se obtiene en esta factorización debe almacenarse por separado, ya que tanto la matriz L, obtenida en el paso 1, como la matriz $\bar{L}$, serán necesarias para resolver el sistema lineal una vez completado el cálculo de la factorización.

- **Paso 4**: Actualizar $\begin{pmatrix} C \\ E \end{pmatrix}$ consistentemente con la factorización de $\begin{pmatrix} U \\ D \end{pmatrix}$. Esta actualización se realiza mediante eliminación progresiva, intercalando además las permutaciones definidas en r al estilo LINPACK durante este proceso. En la notación empleada, este paso se refleja como

$$\begin{pmatrix} C \\ E \end{pmatrix} := \textsc{Apply\_L}_{\mathrm{TD}}^{\mathrm{LIN}} \left(\begin{pmatrix} \bar{L} \\ D \end{pmatrix}, r, \begin{pmatrix} C \\ E \end{pmatrix}\right)$$

- **Paso 5**: Factorizar E. Por último, se realiza una factorización LU con pivotamiento parcial estilo LAPACK de E. El vector s almacenará los índices que definen la secuencia de permutaciones, en tanto que los factores triangulares sobreescriben las entradas de E.

$$[E, s] := [\{\tilde{L}\backslash\tilde{U}\}, s] = \mathrm{LU}(E)$$

El algoritmo presentado lleva a cabo una secuencia de permutaciones de filas diferente a las permutaciones que se realizan en una factorización LU con pivotamiento parcial, por lo que la estabilidad numérica de este algoritmo también será diferente. En el Apartado 3.3.3 se ofrecen algunas observaciones en este sentido.

La segunda columna de la Tabla 3.1 muestra el coste teórico de cada paso del algoritmo. Por simplicidad, se considera que las submatrices B, C, D y E son todas de tamaño $t \times t$.

Paso	*Coste teórico* Algoritmo básico	*Coste teórico* Formulación BLAS 3
LU	$\frac{2}{3}t^3$	$\frac{2}{3}t^3$
APPLY_L	t^3	t^3
$\mathrm{LU}_{\mathrm{TD}}^{\mathrm{LIN}}$	$\frac{5}{3}t^3$	$t^3 + \frac{2}{3}t^2b$
$\mathrm{APPLY\_L}_{\mathrm{TD}}^{\mathrm{LIN}}$	$3t^3$	$2t^3 + tb^2$
LU	$\frac{2}{3}t^3$	$\frac{2}{3}t^3$
Total	$7t^3$	$\frac{16}{3}t^3 + \frac{2}{3}t^2b + tb^2$

Tabla 3.1: Coste teórico en *flops* de cada paso del algoritmo para la factorización LU con pivotamiento incremental. Se ha considerado que las submatrices B, C, D y E son todas de tamaño $t \times t$. El valor b corresponde al tamaño algorítmico de bloque considerado en la factorización BLAS 3 de los módulos $\mathrm{LU}_{\mathrm{TD}}^{\mathrm{LIN}}$ y $\mathrm{APPLY\_L}_{\mathrm{TD}}^{\mathrm{LIN}}$.

El coste total del algoritmo básico es de $7t^3$ *flops*, frente al coste de realizar la factorización LU con pivotamiento parcial, que es de $\frac{16}{3}t^3$ *flops*, lo que supone un coste adicional de $\frac{5}{3}t^3$ *flops*. Este coste adicional se puede reducir utilizando sendas implementaciones por bloques de los algoritmos empleados en las etapas $\mathrm{LU}_{\mathrm{TD}}^{\mathrm{LIN}}$ y $\mathrm{APPLY\_L}_{\mathrm{TD}}^{\mathrm{LIN}}$ (denominados módulos en adelante), que se presentan en el siguiente apartado. Si bien el sobrecoste en el que se puede llegar a incurrir no es importante cuando C, D y E en (3.3) son bloques de dimensión pequeña comparada con la de B, reducir este coste, haciéndolo despreciable, es fundamental en el algoritmo OOC, donde C, D y E tendrán la misma dimensión que B.

3.3.2. Formulación BLAS 3 del algoritmo básico

Tanto el módulo LU como el módulo APPLY_L, correspondientes a los pasos 1 y 2 del algoritmo básico para la factorización LU con pivotamiento incremental, pueden ser implementados como algoritmos por bloques que hagan uso de BLAS 3. El módulo LU se puede implementar mediante cualquiera de las variantes por bloques de la Figura 3.1, que fundamentalmente hacen uso de las

Algorithm: $[C] := \text{APPLY\_L}(B, C)$

Partition $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array}\right)$
where C_L has 0 columns
while $n(C_L) < n(C)$ **do**
Determine block size b
Repartition
$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$
where C_1 has b columns

$C_1 := \text{TRILU(B)}^{-1} C_1$

Continue with
$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$
endwhile

Figura 3.2: Algoritmo por bloques que actualiza la matriz C consistentemente con la factorización LU con pivotamiento parcial de la matriz B.

operaciones TRSM y GEMM. La Figura 3.2 muestra una implementación para el módulo APPLY_L, que únicamente utiliza operaciones TRSM. El detalle de cómo se realiza el pivotamiento no se incluye en los algoritmos que se presentan en adelante para dar una mayor claridad a los mismos.

Los módulos $\text{LU}_{\text{TD}}^{\text{LIN}}$ y $\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$, que calculan y aplican una factorización LU con pivotamiento estilo LINPACK, no pueden implementarse haciendo uso de operaciones de BLAS 3, ya que tanto el cálculo de las transformaciones de Gauss como su aplicación (operación GER de BLAS 2), se intercalan con las permutaciones de filas. En [39] se proponen sendos algoritmos por bloques para implementar estas operaciones, que se muestran en las Figuras 3.3 y 3.4.

La clave para obtener un algoritmo por bloques para el módulo de factorización $\text{LU}_{\text{TD}}^{\text{LIN}}$ está en avanzar en la factorización por paneles de columnas, factorizando cada panel con un algoritmo que realiza el pivotamiento estilo LAPACK (Figura 3.1), pero aplicando las permutaciones resultantes de la factorización del panel actual sólo a aquellos paneles a la derecha de éste, tal y como se hace en el pivotamiento estilo LINPACK. Una vez factorizado un panel de b columnas, la aplicación de los factores obtenidos a los paneles de la derecha se realiza con operaciones de BLAS 3 (TRSM y GEMM). Como resultado, se obtiene una factorización con un estilo híbrido LAPACK–LINPACK que tiene la forma

$$L_{M-1}^{-1} P_{M-1} \ \cdots \ L_1^{-1} P_1 L_0^{-1} P_0 \left(\frac{U}{D}\right) = \left(\frac{\bar{U}}{0}\right).$$

En el algoritmo se hace uso de una matriz auxiliar $\bar{L}$ con b columnas y tantas filas como A, destinada a guardar la parte triangular inferior de los factores L_{11}. El objetivo es preservar la parte estrictamente triangular inferior de A, que contiene el factor L obtenido en el paso 1 de la factorización LU con pivotamiento incremental.

Considerando que las submatrices B, C, D y E son todas de tamaño $t \times t$ y que cada panel tiene b columnas (por simplicidad se asume que b es un múltiplo de t: $M = t/b$), se obtiene que el algoritmo $\text{LU}_{\text{TD}}^{\text{LIN}}$ de la Figura 3.3 tiene un coste de $t^3 + \frac{2}{3} t^2 b$ *flops*. Gracias a que se sigue este proceso por paneles, también es posible obtener un algoritmo por bloques para aplicar la factorización de

Algorithm: $\left[\left(\frac{U}{D}\right), \bar{L}\right] := \mathrm{LU}_{\mathrm{TD}}^{\mathrm{LIN}}\left(\frac{U}{D}\right)$

Partition $U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $\bar{L} \rightarrow \left(\frac{\bar{L}_T}{\bar{L}_B}\right)$

where U_{TL} is 0×0, D_L has 0 columns, $\bar{L}_T$ has 0 rows

while $n(U_{TL}) < n(U)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right), \left(\frac{\bar{L}_T}{\bar{L}_B}\right) \rightarrow \left(\frac{\bar{L}_0}{\frac{\bar{L}_1}{\bar{L}_2}}\right)$$

where U_{11}, $\bar{L}_1$ are $b \times b$, D_1 has b columns

---

$$\left(\frac{\{\bar{L}_1 \backslash U_{11}\}}{D_1}\right) := \mathrm{LU\_PIV\_BLK}\left(\frac{U_{11}}{D_1}\right)$$

$U_{12} := \bar{L}_1^{-1} U_{12}$

$D_2 := D_2 - D_1 U_{12}$

Continue with

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right), \left(\frac{\bar{L}_T}{\bar{L}_B}\right) \leftarrow \left(\frac{\bar{L}_0}{\frac{\bar{L}_1}{\bar{L}_2}}\right)$$

endwhile

Figura 3.3: Algoritmo por bloques que calcula la factorización LU con pivotamiento parcial de una matriz de la forma $(U^{\mathrm{T}}, D^{\mathrm{T}})^{\mathrm{T}}$ cuyo bloque U es triangular superior. Esta estructura triangular superior se mantiene gracias a la factorización con pivotamiento estilo LINPACK (no se permutan las filas de la matriz a la izquierda de A_{11}). Se procede por paneles de columnas realizando factorizaciones con pivotamiento estilo LAPACK, lo que posibilita el uso de BLAS 3.

$\left(\frac{U}{D}\right)$ a $\left(\frac{C}{E}\right)$, tal y como se muestra en el algoritmo APPLY_L$_{\mathrm{TD}}^{\mathrm{LIN}}$ de la Figura 3.4. En este caso, los cálculos se realizan todos en términos de operaciones de BLAS 3 TRSM y GEMM, y se consigue reducir el coste de este algoritmo a solo $2t^3 + tb^2$ *flops*.

El coste adicional en que se incurre con los algoritmos LU$_{\mathrm{TD}}^{\mathrm{LIN}}$ y APPLY_L$_{\mathrm{TD}}^{\mathrm{LIN}}$, frente a la factorización LU con pivotamiento parcial, es de $\frac{2}{3}t^2b + tb^2$ *flops*, un término de orden menor que resulta despreciable cuando $b \ll t$ (ver tercera columna de la Tabla 3.1) [39].

3.3.3. Algoritmos FLAME orientados a bloques

Los algoritmos presentados en los Apartados 3.3.1 y 3.3.2 proporcionan los módulos básicos que nos permitirán construir algoritmos orientados a bloques para el cálculo de la factorización LU con pivotamiento incremental. Para ilustrar cómo pueden utilizarse estos módulos básicos en

Algorithm: $\left(\frac{C}{E}\right) := \text{APPLY\_L}^{\text{LIN}}_{\text{TD}}\left(\left(\frac{\bar{L}}{D}\right), \left(\frac{C}{E}\right)\right)$

Partition $\bar{L} \rightarrow \left(\frac{\bar{L}_T}{\bar{L}_B}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$

where $\bar{L}_T$ is $b \times b$, C_T has 0 rows, D_L has 0 columns

while $n(D_L) < n(D)$ **do**

Determine block size b

Repartition

$$\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \rightarrow \left(\frac{\bar{L}_0}{\frac{\bar{L}_1}{\bar{L}_2}}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right), \left(\frac{C_T}{C_B}\right) \rightarrow \left(\frac{C_0}{\frac{C_1}{C_2}}\right)$$

where $\bar{L}_1$, C_1 have b rows, D_1 has b columns

$C_1 := \bar{L}_1^{-1} C_1$

$E := E - D_1 C_1$

Continue with

$$\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \leftarrow \left(\frac{\bar{L}_0}{\frac{\bar{L}_1}{\bar{L}_2}}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right), \left(\frac{C_T}{C_B}\right) \leftarrow \left(\frac{C_0}{\frac{C_1}{C_2}}\right)$$

endwhile

Figura 3.4: Algoritmo por bloques que actualiza la matriz $(C^{\mathrm{T}}, E^{\mathrm{T}})^{\mathrm{T}}$ consistentemente con la factorización LU con pivotamiento parcial de la matriz $(\bar{L}^{\mathrm{T}}, D^{\mathrm{T}})^{\mathrm{T}}$ realizada con el algoritmo $\text{LU}^{\text{LIN}}_{\text{TD}}$ de la Figura 3.3.

la factorización, se desarrolla en detalle un ejemplo para una matriz formada por 4×4 bloques:

$$A = \begin{pmatrix} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix}.$$

En la primera iteración del algoritmo orientado a bloques se efectúa el siguiente particionado (se indica qué papel toma cada bloque de la matriz en relación con la estructura de (3.3)):

$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \overset{\text{1ª iteración}}{\left(\begin{array}{c|ccc} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right)} \rightarrow \left(\begin{array}{c|ccc} B & C & C & C \\ \hline D & E & E & E \\ D & E & E & E \\ D & E & E & E \end{array}\right).$$

De este modo, inicialmente calculamos la factorización LU del bloque B (paso 1), aplicamos repetidamente el correspondiente factor L a cada uno de los bloques C (esto es, repetimos el paso 2 tres veces, para $\bar{A}_{01}$, $\bar{A}_{02}$ y $\bar{A}_{03}$), calculamos las sucesivas factorizaciones de $\left(\frac{U}{D}\right)$ para cada uno de

los bloques D (paso 3 repetido en tres ocasiones), y actualizamos de manera consistente los bloques de tipo $\left(\begin{array}{c} C \\ \hline E \end{array}\right)$ (paso 4 repetido nueve veces, una por cada bloque de tipo E).

Este procedimiento repetitivo aplicado para la primera iteración es el equivalente del problema de la actualización de una factorización LU, pues se está utilizando la factorización de B para diferentes valores de C, D y E, que en este caso son diferentes bloques de la matriz.

Este mismo proceso se efectúa nuevamente a continuación para la segunda iteración, en la que el particionado es

$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \overset{\text{2ª iteración}}{\left(\begin{array}{c|c|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \hline \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right)} \rightarrow \left(\begin{array}{c|c|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \hline \bar{A}_{10} & B & C & C \\ \hline \bar{A}_{20} & D & E & E \\ \hline \bar{A}_{30} & D & E & E \end{array}\right).$$

Seguidamente, en la tercera iteración,

$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \overset{\text{3ª iteración}}{\left(\begin{array}{c|c|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \hline \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right)} \rightarrow \left(\begin{array}{c|c|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \hline \bar{A}_{20} & \bar{A}_{21} & B & C \\ \hline \bar{A}_{30} & \bar{A}_{31} & D & E \end{array}\right).$$

Y, en la última iteración,

$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \overset{\text{4ª iteración}}{\left(\begin{array}{c|c|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right)} \rightarrow \left(\begin{array}{c|c|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & B \end{array}\right).$$

Un detalle de implementación importante es que, cuando se lleva a cabo el paso 3 en cada iteración mediante el módulo $\text{LU}_{\text{TD}}^{\text{LIN}}$, es necesario guardar cada factor $\bar{L}$ obtenido para cada bloque D, ya que más tarde será necesario para resolver el sistema lineal $Ax = b$. Por lo tanto, en nuestro algoritmo orientado a bloques deberemos utilizar un espacio adicional con este propósito.

Las Figuras 3.5 y 3.6 capturan el proceso global en forma de un algoritmo orientado a bloques para la factorización LU con pivotamiento incremental. En los algoritmos se han sombreado los módulos básicos que llevan a cabo cada uno de los pasos del algoritmo básico (en su formulación BLAS 3): la factorización LU de una matriz densa (LU), la aplicación de las transformaciones de Gauss resultantes de esta factorización (APPLY_L), la factorización LU de una matriz con 2×1 bloques, en la que el bloque superior es triangular y el inferior es denso ($\text{LU}_{\text{TD}}^{\text{LIN}}$), y la aplicación de las transformaciones de Gauss correspondientes a esta última factorización sobre una matriz con 2×1 bloques ($\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$). La implementación de estos módulos es la que se ha ilustrado en las Figuras 3.1–3.4.

De nuevo, el detalle de cómo se realiza el pivotamiento no se ha incluido para dar una mayor claridad a los algoritmos. Este detalle no modifica el análisis del coste computacional de este

Tipo de paso	*Coste teórico* Factorización LU con pivotamiento incremental
LU	$n\frac{2t^2}{3}$
APPLY_L	$n^2\frac{t}{2} - n\frac{t^2}{2}$
$\text{LU}_{\text{TD}}^{\text{LIN}}$	$n^2(\frac{t}{2} + \frac{b}{3}) - n(\frac{t^2}{2} + \frac{bt}{3})$
$\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$	$n^3(\frac{2}{3} + \frac{b^2}{3t^2}) - n^2(t + \frac{b^2}{2t}) + n(\frac{t^2}{3} + \frac{b^2}{6})$
Total	$n^3(\frac{2}{3} + \frac{b^2}{3t^2}) + n^2(\frac{b}{3} - \frac{b^2}{2t}) + n(\frac{b^2}{6} - \frac{bt}{3})$

Tabla 3.2: Coste teórico en *flops* de la variante 1 del algoritmo orientado a bloques para la factorización LU con pivotamiento incremental de una matriz $n \times n$, siendo t el tamaño de los bloques A_{ij} y b el tamaño algorítmico de bloque utilizado internamente en los módulos $\text{LU}_{\text{TD}}^{\text{LIN}}$ y $\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$.

algoritmo, que se muestra en la Tabla 3.2. Además, puesto que cada acción asociada al pivotamiento (encontrar el índice y permutar filas) va seguida de una operación algebraica que afecta a los bloques implicados en el pivotamiento, el no incluir estas operaciones en los algoritmos tampoco invalida las consideraciones sobre el coste teórico de la E/S que realizaremos más adelante. Es importante destacar aquí que, cuando $t \ll n$ y $b \ll t$, el coste total de este procedimiento se reduce a $\frac{2}{3}n^3$ operaciones aritméticas, resultando equivalente al coste de calcular la factorización LU con pivotamiento parcial de una matriz de dimensión $n \times n$.

En lo que respecta a la estabilidad numérica, la factorización LU con pivotamiento incremental aplica pivotamiento parcial en la factorización de la submatriz B y, posteriormente, también en la factorización de $\begin{pmatrix} U \\ D \end{pmatrix}$, por lo que puede considerarse como una variante por bloques del pivotamiento por pares (*pairwise pivoting* [74]). En [65] se demuestra, mediante modelos estadísticos y un gran número de experimentos, que en la media, el factor de crecimiento para el pivotamiento completo es $\rho_c \approx n^{1/2}$, para el pivotamiento parcial es $\rho_p \approx n^{2/3}$ y para el pivotamiento por pares es $\rho_w \approx n$. De este estudio deducen sus autores que, en la práctica, el pivotamiento parcial y el pivotamiento por pares son numéricamente estables, esperándose que el pivotamiento por pares exhiba un comportamiento numérico ligeramente peor que el pivotamiento parcial. Por lo tanto, es razonable pensar que la factorización LU con pivotamiento incremental responda a un factor de crecimiento que se encuentre entre el del pivotamiento parcial y el del pivotamiento por pares. En [39] se presenta un experimento en el que se obtienen evidencias que apoyan esta hipótesis, observándose que el pivotamiento incremental tiene un factor de crecimiento menor que el del pivotamiento por pares y muy próximo al factor de crecimiento del pivotamiento parcial. En este sentido, resulta ilustrativo apuntar que, cuando $t = 1$, el pivotamiento incremental se transforma en pivotamiento por pares, mientras que, cuando $t = n$, se transforma en pivotamiento parcial.

La Figura 3.7 muestra una segunda variante algorítmica para la factorización LU con pivotamiento incremental. Ésta es una variante orientada hacia la izquierda, a diferencia de la primera variante, que está orientada hacia la derecha. El desarrollo en cada iteración es el siguiente: en primer lugar se actualiza el panel que contiene el bloque A_{11} respecto de la factorización de la submatriz que queda a su izquierda, a continuación se factoriza el bloque A_{11} y, por último, se factoriza

Algorithm: $\left[A,\ \bar{L}\right] := \text{LU\_B\_VAR1}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $\bar{L} \rightarrow \left(\begin{array}{c|c} \bar{L}_{TL} & \bar{L}_{TR} \\ \hline \bar{L}_{BL} & \bar{L}_{BR} \end{array}\right)$

where A_{TL}, $\bar{L}_{TL}$ have 0×0 tiles

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} \bar{L}_{TL} & \bar{L}_{TR} \\ \hline \bar{L}_{BL} & \bar{L}_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} \bar{L}_{00} & \bar{L}_{01} & \bar{L}_{02} \\ \hline \bar{L}_{10} & \bar{L}_{11} & \bar{L}_{12} \\ \hline \bar{L}_{20} & \bar{L}_{21} & \bar{L}_{22} \end{array}\right)$$

where A_{11}, $\bar{L}_{11}$ are tiles

$$\begin{aligned} A_{11} &:= \text{LU}(A_{11}) \\ A_{12} &:= \text{LU\_B1}(A_{11}, A_{12}) \\ \left[A_{11},\ A_{21},\ \bar{L}_{21}\right] &:= \text{LU\_B2}(A_{11}, A_{21}) \\ \left[A_{12},\ A_{22}\right] &:= \text{LU\_B3}(\bar{L}_{21}, A_{12}, A_{21}, A_{22}) \end{aligned}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} \bar{L}_{TL} & \bar{L}_{TR} \\ \hline \bar{L}_{BL} & \bar{L}_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} \bar{L}_{00} & \bar{L}_{01} & \bar{L}_{02} \\ \hline \bar{L}_{10} & \bar{L}_{11} & \bar{L}_{12} \\ \hline \bar{L}_{20} & \bar{L}_{21} & \bar{L}_{22} \end{array}\right)$$

endwhile

Algorithm: $[C] := \text{LU\_B1}(B, C)$

Partition $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array}\right)$

where C_L is 0 tiles wide

while $n(C_L) < n(C)$ **do**

Repartition

$$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$$

where C_1 is a tile

$C_1 := \text{APPLY\_L}(B, C_1)$

Continue with

$$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$$

endwhile

Algorithm: $\left[B,\ D,\ \bar{L}\right] := \text{LU\_B2}(B, D)$

Partition $D \rightarrow \left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right)$, $\bar{L} \rightarrow \left(\begin{array}{c} \bar{L}_T \\ \hline \bar{L}_B \end{array}\right)$

where D_T, $\bar{L}_T$ are 0 tiles high

while $m(D_T) < m(D)$ **do**

Repartition

$$\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right), \left(\begin{array}{c} \bar{L}_T \\ \hline \bar{L}_B \end{array}\right) \rightarrow \left(\begin{array}{c} \bar{L}_0 \\ \hline \bar{L}_1 \\ \hline \bar{L}_2 \end{array}\right)$$

where D_1, $\bar{L}_1$ are tiles

$$\left[\left(\begin{array}{c} B \\ \hline D_1 \end{array}\right), \bar{L}_1\right] := \text{LU}^{\text{LIN}}_{\text{TD}}\left(\left(\begin{array}{c} B \\ \hline D_1 \end{array}\right)\right)$$

Continue with

$$\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \leftarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right), \left(\begin{array}{c} \bar{L}_T \\ \hline \bar{L}_B \end{array}\right) \leftarrow \left(\begin{array}{c} \bar{L}_0 \\ \hline \bar{L}_1 \\ \hline \bar{L}_2 \end{array}\right)$$

endwhile

Figura 3.5: Algoritmo orientado a bloques para calcular la factorización LU. Variante 1 orientada hacia la derecha.

Algorithm: $[C,\ E] := \text{LU\_B3}(\bar{L}, C, D, E)$

Partition $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array}\right)$, $E \rightarrow \left(\begin{array}{c|c} E_L & E_R \end{array}\right)$
where C_L, E_L are 0 tiles wide

while $n(C_L) < n(C)$ **do**

Repartition
$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$, $\left(\begin{array}{c|c} E_L & E_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} E_0 & E_1 & E_2 \end{array}\right)$
where C_1 is a tile, E_1 is 1 tile wide

$[C_1, E_1] := \text{LU\_B4}(\bar{L}, C_1, D, E_1)$

Continue with
$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$, $\left(\begin{array}{c|c} E_L & E_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} E_0 & E_1 & E_2 \end{array}\right)$

endwhile

Algorithm: $[C,\ E] := \text{LU\_B4}(\bar{L}, C, D, E)$

Partition $\bar{L} \rightarrow \left(\begin{array}{c} \bar{L}_T \\ \hline \bar{L}_B \end{array}\right)$, $D \rightarrow \left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right)$, $E \rightarrow \left(\begin{array}{c} E_T \\ \hline E_B \end{array}\right)$
where $\bar{L}_T$, D_T, E_T are 0 tiles high

while $m(D_T) < m(D)$ **do**

Repartition
$\left(\begin{array}{c} \bar{L}_T \\ \hline \bar{L}_B \end{array}\right) \rightarrow \left(\begin{array}{c} \bar{L}_0 \\ \hline \bar{L}_1 \\ \hline \bar{L}_2 \end{array}\right)$, $\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right)$, $\left(\begin{array}{c} E_T \\ \hline E_B \end{array}\right) \rightarrow \left(\begin{array}{c} E_0 \\ \hline E_1 \\ \hline E_2 \end{array}\right)$
where $\bar{L}_1$, D_1, E_1 are tiles

$\left(\begin{array}{c} C \\ \hline E_1 \end{array}\right) := \text{APPLY\_L}^{\text{LIN}}_{\text{TD}}\left(\left(\begin{array}{c} \bar{L}_1 \\ \hline D_1 \end{array}\right), \left(\begin{array}{c} C \\ \hline E_1 \end{array}\right)\right)$

Continue with
$\left(\begin{array}{c} \bar{L}_T \\ \hline \bar{L}_B \end{array}\right) \leftarrow \left(\begin{array}{c} \bar{L}_0 \\ \hline \bar{L}_1 \\ \hline \bar{L}_2 \end{array}\right)$, $\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \leftarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right)$, $\left(\begin{array}{c} E_T \\ \hline E_B \end{array}\right) \leftarrow \left(\begin{array}{c} E_0 \\ \hline E_1 \\ \hline E_2 \end{array}\right)$

endwhile

Figura 3.6: Algoritmo orientado a bloques para calcular la factorización LU. Variante 1 orientada hacia la derecha (continuación).

Algorithm: $\left[A,\ \bar{L}\right] := \text{LU\_B\_VAR2}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $\bar{L} \rightarrow \left(\begin{array}{c|c} \bar{L}_{TL} & \bar{L}_{TR} \\ \hline \bar{L}_{BL} & \bar{L}_{BR} \end{array}\right)$

where A_{TL}, $\bar{L}_{TL}$ have 0×0 tiles

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} \bar{L}_{TL} & \bar{L}_{TR} \\ \hline \bar{L}_{BL} & \bar{L}_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} \bar{L}_{00} & \bar{L}_{01} & \bar{L}_{02} \\ \hline \bar{L}_{10} & \bar{L}_{11} & \bar{L}_{12} \\ \hline \bar{L}_{20} & \bar{L}_{21} & \bar{L}_{22} \end{array}\right)$$

where A_{11}, $\bar{L}_{11}$ are tiles

$$\left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right) := \text{LU\_B5}\left(\left(\begin{array}{c} A_{00} \\ \hline A_{10} \\ \hline A_{20} \end{array}\right), \left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right), \left(\begin{array}{c} \bar{L}_{00} \\ \hline \bar{L}_{10} \\ \hline \bar{L}_{20} \end{array}\right)\right)$$

$$A_{11} := \text{LU}(A_{11})$$

$$\left[A_{11},\ A_{21},\ \bar{L}_{21}\right] := \text{LU\_B2}(A_{11}, A_{21})$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} \bar{L}_{TL} & \bar{L}_{TR} \\ \hline \bar{L}_{BL} & \bar{L}_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} \bar{L}_{00} & \bar{L}_{01} & \bar{L}_{02} \\ \hline \bar{L}_{10} & \bar{L}_{11} & \bar{L}_{12} \\ \hline \bar{L}_{20} & \bar{L}_{21} & \bar{L}_{22} \end{array}\right)$$

endwhile

Algorithm: $[B] := \text{LU\_B5}(A, B, \bar{L})$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$, $\bar{L} \rightarrow \left(\begin{array}{c|c} \bar{L}_{TL} & \bar{L}_{TR} \\ \hline \bar{L}_{BL} & \bar{L}_{BR} \end{array}\right)$

where A_{TL}, $\bar{L}_{TL}$ have 0×0 tiles, B_T is 0 tiles high

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right), \left(\begin{array}{c|c} \bar{L}_{TL} & \bar{L}_{TR} \\ \hline \bar{L}_{BL} & \bar{L}_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} \bar{L}_{00} & \bar{L}_{01} & \bar{L}_{02} \\ \hline \bar{L}_{10} & \bar{L}_{11} & \bar{L}_{12} \\ \hline \bar{L}_{20} & \bar{L}_{21} & \bar{L}_{22} \end{array}\right)$$

where A_{11}, B_1, $\bar{L}_{11}$ are tiles

$$B_1 := \text{APPLY\_L}(A_{11},\ B_1)$$

$$[B_1,\ B_2] := \text{LU\_B4}(\bar{L}_{21},\ B_1,\ A_{21},\ B_2)$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right), \left(\begin{array}{c|c} \bar{L}_{TL} & \bar{L}_{TR} \\ \hline \bar{L}_{BL} & \bar{L}_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} \bar{L}_{00} & \bar{L}_{01} & \bar{L}_{02} \\ \hline \bar{L}_{10} & \bar{L}_{11} & \bar{L}_{12} \\ \hline \bar{L}_{20} & \bar{L}_{21} & \bar{L}_{22} \end{array}\right)$$

endwhile

Figura 3.7: Algoritmo orientado a bloques para calcular la factorización LU. Variante 2 orientada hacia la izquierda.

Algoritmo	*Lecturas*	*Escrituras*
Variante 1	$\frac{2s^3+s^2-s}{2}$	$\frac{4s^3+3s^2-s}{6}$
Variante 2	$\frac{2s^3+s^2-s}{2}$	$\frac{4s^3+3s^2-s}{6}$

Tabla 3.3: Número de lecturas y escrituras de *tiles* (coste teórico de la E/S) de las dos variantes obtenidas para la factorización LU con pivotamiento incremental. (El número de elementos leídos/ escritos puede obtenerse fácilmente multiplicando estos valores por el tamaño del *tile*: $t \times t$.)

el panel de bloques que contiene al bloque A_{11} y los bloques que se encuentran por debajo de éste. En este último paso de cada iteración se generan los bloques de la matriz $\bar{L}$, que son utilizados en el primer paso de las siguientes iteraciones para actualizar el panel que se va a factorizar.

3.4. Implementaciones OOC para la factorización LU

Para transformar los algoritmos a bloques en algoritmos OOC consideraremos que cada bloque es un *tile*, es decir, un bloque que constituye la unidad elemental de transferencia de datos entre disco y memoria principal. A partir de los algoritmos orientados a bloques resulta sencillo saber, en cada momento y sin manejar subíndices, qué *tiles* se deben leer del disco para trabajar con ellos y cuáles escribir después de ser actualizados.

En la Figura 3.8 se muestran dos iteraciones del desarrollo de la variante 1 orientada a bloques LU_B_VAR1 de la Figura 3.5, concretamente la segunda y la tercera iteración. Cada iteración contiene los cuatro pasos del algoritmo básico presentado en el Apartado 3.3.1, y en cada paso se han sombreado los *tiles* que son modificados. Los cuatro pasos son: obtener la factorización LU del *tile* diagonal (LU), actualizar los *tiles* a la derecha de éste con respecto a la factorización obtenida en el paso anterior (LU_B1), obtener la factorización LU del panel de *tiles* $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ (LU_B2) y, por último, actualizar los *tiles* a la derecha de este panel con respecto a la factorización recién calculada (LU_B3). El espacio adicional $\bar{L}$ que se necesita cuando se factorizan los paneles $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ tiene una estructura triangular inferior. Si el tamaño de *tile* es t y $s = n/t$, este espacio adicional será de tamaño $(s-1)t \times (s-1)b$, siendo b el tamaño de bloque con que se trabaja en los módulos $\text{LU}_{\text{TD}}^{\text{LIN}}$ y $\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$. La estructura de cada columna de bloques de $\bar{L}$ se observa en las dos iteraciones del desarrollo del algoritmo mostradas en la figura. Nótese que puede ser almacenada de manera compactada ya que de cada bloque sólo se almacena la parte triangular inferior.

En la Figura 3.9 se muestra el desarrollo de una iteración (la tercera) de la variante 2 orientada a bloques LU_B_VAR2 de la Figura 3.7.

A continuación, analizamos el comportamiento de las dos variantes de la factorización LU con pivotamiento incremental desde el punto de vista del coste teórico de la E/S. Para calcular este coste se tiene en cuenta que toda la E/S es realizada por los módulos básicos LU, APPLY_L, $\text{LU}_{\text{TD}}^{\text{LIN}}$ y $\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$. La Tabla 3.3 recoge el coste de la E/S de ambas variantes, en función del tamaño en *tiles* de la matriz ($s = n/t$). Como se puede apreciar, el coste es exactamente el mismo para ambas variantes.

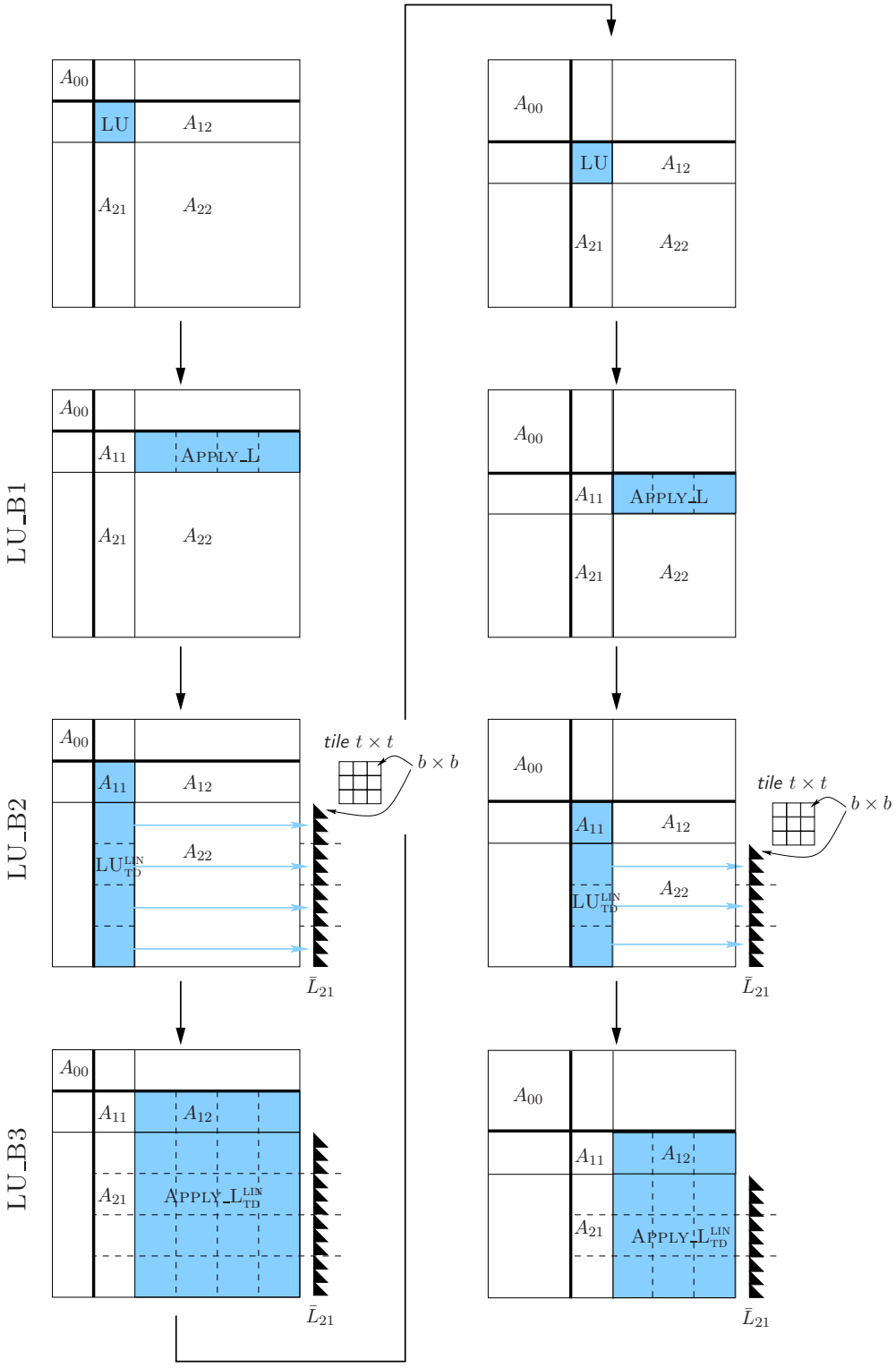


Figura 3.8: Desarrollo de la segunda (izquierda) y tercera (derecha) iteración de la variante 1 orientada a bloques para la factorización LU con pivotamiento incremental OOC.

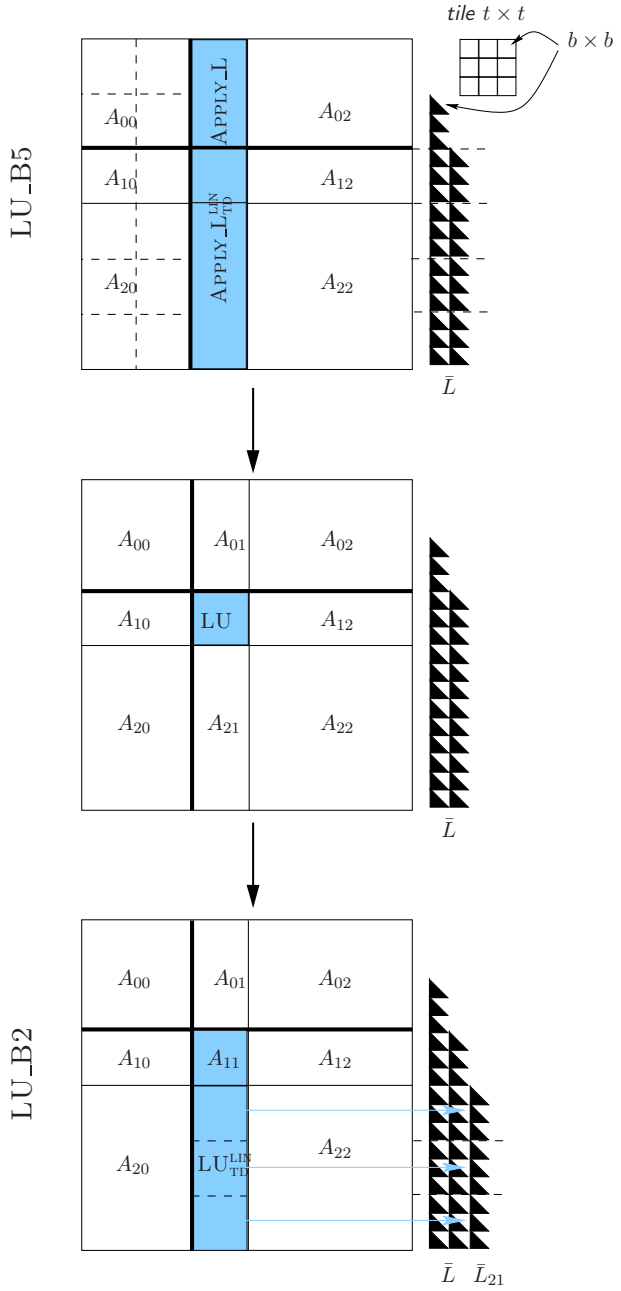


Figura 3.9: Desarrollo de la tercera iteración de la variante 2 orientada a bloques para la factorización LU con pivotamiento incremental OOC.

En las implementaciones realizadas de las dos variantes para calcular la factorización LU con pivotamiento incremental OOC, se han llevado a cabo distintas optimizaciones que dan título al apartado en que se presentan a continuación.

3.4.1. Uso de una caché software

A partir del *run-time* desarrollado para la factorización de Cholesky OOC (expuesto en el Apartado 2.3.2), se han desarrollado las funciones intermedias necesarias para poder utilizarlo en la factorización LU OOC. Recordemos que el *run-time* es el encargado de determinar qué *tiles* se deben leer y escribir en cada momento desde/en disco. En una primera etapa, el sistema ejecuta el código de manera simbólica, generando una lista de tareas pendientes de ejecución, y anotando para cada una de éstas los *tiles* de entrada y de salida. Una vez finalizada la ejecución simbólica del código, se obtiene la lista completa de las tareas que será necesario ejecutar para completar el proceso de factorización (equivalente a un desenrollado completo de los bucles que comprende el algoritmo). A continuación, en una segunda etapa, las tareas se ejecutan en orden, haciendo un recorrido secuencial de la lista: el sistema toma la siguiente tarea, identifica los *tiles* que necesita para realizar la operación asociada y comprueba si se encuentran en la caché, leyéndolos desde el disco en caso contrario. Una vez los datos residen en memoria, se ejecuta la operación. Los *tiles* permanecen en la caché hasta que el sistema decide que deben ser reemplazados porque se necesita reutilizar el espacio que están ocupando.

En la Figura 3.10 se puede observar el ahorro en el número de accesos a elementos que se produce gracias a la incorporación del sistema de caché *software* en la variante 2, que llega a ser de casi el 40 % para $n = 40.960$. Los datos de la gráfica de esta figura se han obtenido a partir del coste teórico del algoritmo (sin optimizaciones) mediante una simulación del cálculo de lecturas y escrituras de *tiles* cuando se utiliza el sistema de caché.

3.4.2. Solapamiento de cálculos y accesos a disco

Al igual que en la factorización de Cholesky OOC (Apartado 2.3.4), otra optimización que es posible introducir en los algoritmos OOC para la factorización LU con pivotamiento incremental es la que permite el solapamiento de cálculos y accesos a disco. En particular, cuando el *run-time* ejecuta el código de manera simbólica, genera la lista de tareas, indicando para cada una de éstas los *tiles* de entrada y de salida. El orden en que las tareas aparecen en dicha lista, junto con la direccionalidad de los operandos (entrada o salida), definen el orden y la dirección en que deben transferirse los *tiles* entre la memoria y el disco. Puesto que la lista se genera antes de comenzar los cálculos, en cualquier momento de la factorización se conoce qué datos serán necesarios más adelante, de modo que si los datos son transferidos desde el disco con antelación, cuando sean necesarios, ya residirán en memoria. El resultado es un mecanismo de *prefetch* con una tasa de aciertos plena.

Además, mediante el paralelismo que proporcionan las hebras exploradora y ejecutora, es posible solapar cálculo y transferencias con el disco, y ocultar parcialmente los tiempos de espera que producen las lecturas y escrituras de datos, de modo que las prestaciones no estén afectadas por el hecho de trabajar con matrices que residen en el disco. El hecho de dejar el solapamiento de la E/S y los cálculos en manos del *run-time* hace transparente su uso al desarrollador de bibliotecas OOC.

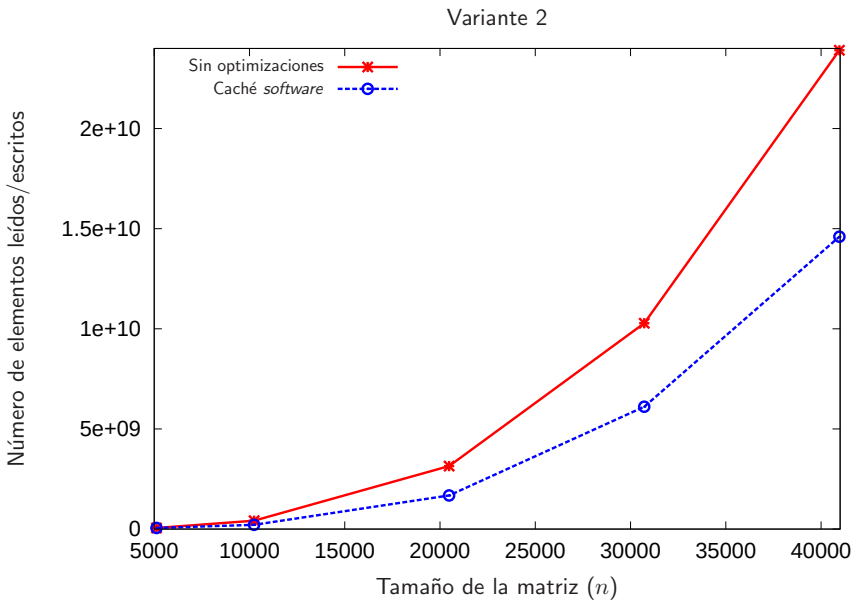


Figura 3.10: Coste teórico de la E/S de la variante 2 (*left–looking*) de la factorización de LU con pivotamiento incremental, con y sin el uso del sistema que gestiona la memoria disponible como una memoria caché *software*. El tamaño de *tile* es $t = 5.120$.

3.5. Experimentos sobre una arquitectura monoprocesador

En el cálculo de las prestaciones de esta sección y las siguientes, se ha considerado que el coste de la factorización LU de una matriz cuadrada de dimensión $n \times n$ es de $2n^3/3$ *flops*, ya sea con pivotamiento parcial como con pivotamiento incremental.

La Figura 3.11 recoge los resultados experimentales obtenidos sobre ROPE (en la Tabla 1.1 se muestran las características de ésta y otras máquinas usadas en la evaluación experimental de este capítulo) con las distintas implementaciones OOC desarrolladas para la factorización LU con pivotamiento incremental. Como referencia, en la figura se ha incluido la curva correspondiente a la implementación *in-core* de la rutina GETRF de MKL (en la Tabla 1.2 se muestra la versión de las bibliotecas usadas en la evaluación experimental de este capítulo) que calcula la factorización LU con pivotamiento parcial. Cuando la matriz cabe en memoria principal, esta rutina obtiene mejores prestaciones que nuestras implementaciones ya que con el pivotamiento parcial se realizan menos operaciones al calcular la factorización. Como se observa, estas prestaciones se deterioran en cuanto la matriz aumenta de tamaño y no cabe en memoria (se empieza a hacer uso de la memoria virtual).

En primer lugar, se han implementado las variantes 1 y 2 OOC, mostrando esta última un mejor comportamiento. Se observa que las prestaciones de ambas variantes decaen a partir de un determinado tamaño de matriz ($n \geq 20.480$), que coincide con el punto en que el disco baja su velocidad a menos de un 3 % de la que se consigue en el acceso a matrices más pequeñas.

Sobre la variante 2 se ha incorporado una primera optimización consistente en el uso de la caché *software* ya utilizada para la factorización de Cholesky. Con este mecanismo, las prestaciones mejoran, aunque lo hacen muy ligeramente, y de nuevo disminuyen en el mismo punto en que solían

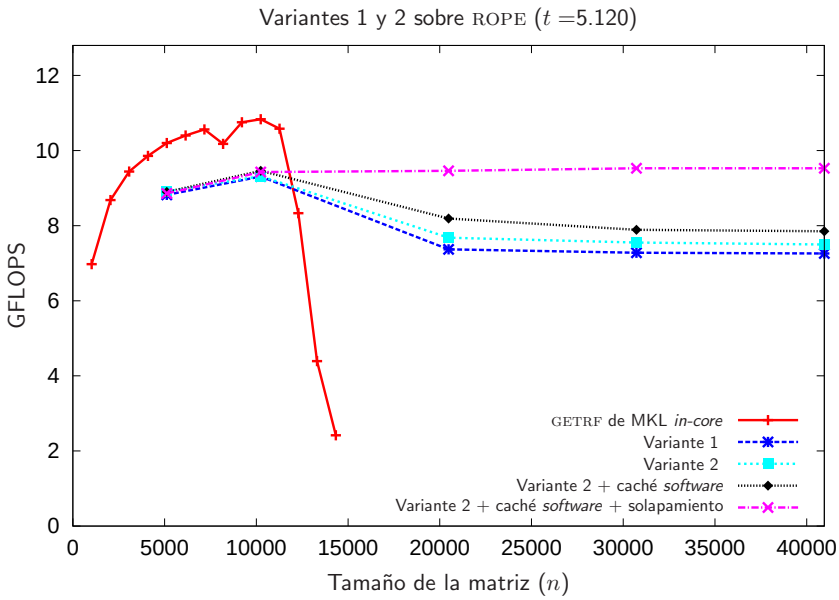


Figura 3.11: Ejecución sobre ROPE de distintas implementaciones OOC de las dos variantes de la factorización LU con pivotamiento incremental.

hacerlo. Este comportamiento negativo se evita gracias a la segunda optimización introducida, el solapamiento de la E/S con los cálculos. De este modo, se consigue ocultar el impacto negativo del acceso a disco. En la curva correspondiente a esta última implementación se observa que, mediante la combinación de estos dos mecanismos, se logran mantener unas prestaciones constantes de 9,5 GFLOPS, lo que corresponde a un 74,5 % de la velocidad pico de la máquina (12,8 GFLOPS) y un 88 % de las mejores prestaciones conseguidas por MKL para matrices *in-core*.

3.6. Experimentos sobre una arquitectura multihebra

Cuando se dispone de varios procesadores/núcleos, es posible desarrollar nuevas versiones de los códigos para explotar el paralelismo que ofrece el *hardware*, lanzando a ejecución tantas hebras como núcleos para llevar a cabo los cálculos en paralelo, explotando la multiplicidad de recursos computacionales (procesadores o núcleos). En la factorización LU con pivotamiento incremental OOC hemos implementado dos alternativas distintas para aprovechar el paralelismo de las operaciones, que se describen en los siguientes apartados.

3.6.1. Paralelización estándar con BLAS

La primera alternativa de paralelización consiste en utilizar implementaciones multihebra de las operaciones de LAPACK y de BLAS que realizan los cálculos en los módulos básicos de los algoritmos a bloques de las dos variantes.

El primero de estos módulos es la factorización LU de un *tile* (matriz densa), cuya implementación se puede realizar mediante cualquiera de los algoritmos por bloques de la Figura 3.1, que

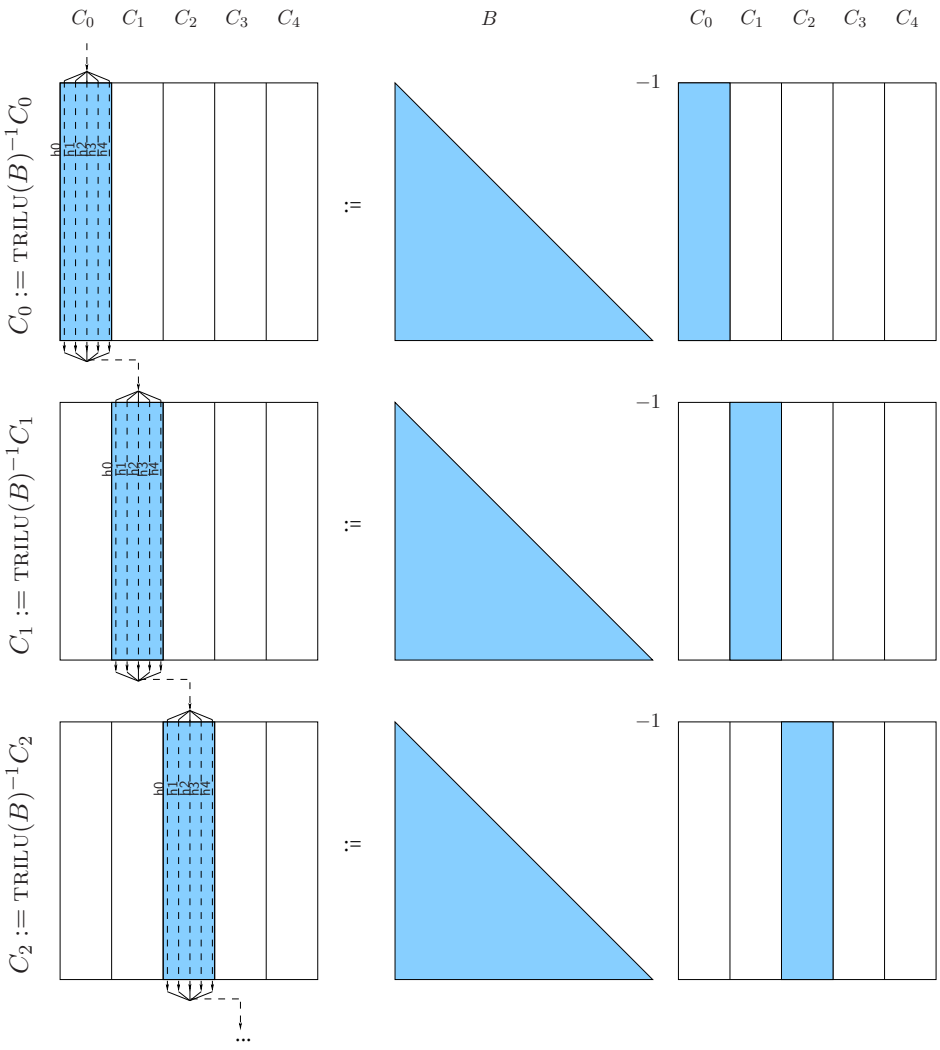


Figura 3.12: Paralelización estándar con BLAS en el módulo básico APPLY_L. Las hebras $h0, \ldots, h4$ trabajan en paralelo para resolver cada uno de los sistemas $\bar{C}_j := \text{TRILU}(B)^{-1}\bar{C}_j$. Tras la resolución de cada sistema hay una sincronización que limita el paralelismo.

calculan la factorización LU con pivotamiento parcial. Para la paralelización en este módulo se han probado dos alternativas: la rutina de LAPACK GETRF que proporciona la implementación multihebra de la biblioteca MKL, y una versión de esta misma rutina paralelizada mediante una aproximación basada en el flujo de datos (*data–flow*) [52].

El segundo paso del algoritmo básico de la factorización LU con pivotamiento incremental corresponde al módulo APPLY_L de la Figura 3.2, que lleva a cabo la aplicación sobre un *tile* de las transformaciones de Gauss resultantes de la factorización LU realizada en el primer paso. El tercer paso consiste en el cálculo de la factorización LU de una matriz de 2×1 *tiles* en la que el *tile* superior es triangular y el inferior es denso (módulo $\text{LU}_{\text{TD}}^{\text{LIN}}$ de la Figura 3.3), y el cuarto y último paso aplica las transformaciones de Gauss correspondientes a la factorización recién calculada por $\text{LU}_{\text{TD}}^{\text{LIN}}$, sobre una matriz de 2×1 *tiles* (módulo $\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$ de la Figura 3.4). La paralelización en estos tres últimos módulos se realiza mediante el uso de las implementaciones multihebra de las rutinas de BLAS TRSM y GEMM de MKL.

Los resultados obtenidos mediante esta optimización para las variantes 1 y 2 de la factorización LU con pivotamiento incremental OOC sobre TESLA y TESLA2 se muestran en las Figuras 3.14 y 3.15, respectivamente, en las curvas identificadas con la leyenda paralelización estándar. En ambas arquitecturas, la mejora en prestaciones es mínima, por lo que se propone otra alternativa de paralelización de nivel superior a la que proporciona BLAS multihebra. Como se aprecia en la Tabla 3.2, es el módulo $\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$ el que representa un mayor coste en el algoritmo por lo que el paralelismo de este módulo fija el rendimiento de todo el algoritmo paralelo. En consecuencia, la siguiente alternativa se centra precisamente en mejorar el rendimiento paralelo de este módulo.

3.6.2. Paralelización de nivel superior a BLAS

Puesto que en la paralelización estándar, cada llamada a BLAS multihebra supone un punto de sincronización para las hebras que colaboran en su ejecución, encontramos que en algunos módulos básicos aparece un elevado número de sincronizaciones que limitan el paralelismo. Aumentando la granularidad del paralelismo es posible evitar estos puntos de sincronización, aumentando así el nivel de paralelismo. Esto es posible hacerlo tanto en el módulo APPLY_L como en el módulo $\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$, que soporta la mayor carga computacional.

Como muestra la Figura 3.12, en cada ejecución del módulo APPLY_L(B, C) habrá tantas sincronizaciones como bloques de columnas contenga el *tile* C. Por ejemplo, si el tamaño de bloque es $b = t/5$, el *tile* C contiene cinco bloques de columnas de tamaño $t \times b$: $C = \begin{pmatrix} C_0 & C_1 & C_2 & C_3 & C_4 \end{pmatrix}$. Con la paralelización estándar ilustrada en el apartado anterior encontramos que, en cada iteración del bucle de la rutina APPLY_L(B, C), se resuelve un sistema triangular para un bloque de b columnas de C mediante una llamada a BLAS multihebra (TRSM). Puesto que el *tile* C tiene cinco bloques de columnas, habrá cinco sincronizaciones.

Una alternativa a esta paralelización, que evita los puntos de sincronización intermedios, consiste en lanzar tantas hebras paralelas como bloques de columnas tenga el *tile* C, y utilizar una implementación secuencial de BLAS para resolver el sistema triangular correspondiente a cada bloque de columnas de forma concurrente. Ésta es una paralelización de nivel superior a BLAS y de un grano más grueso. La Figura 3.13 muestra el efecto de esta optimización sobre el ejemplo de la Figura 3.12.

Cuando se realiza esta optimización en el módulo $\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$, se aplica sobre la operación GEMM $E := E - D_1 C_1$. En este caso, se particiona E en bloques de columnas y se trabaja concurrentemente

sobre cada uno de estos bloques E_j mediante la implementación de BLAS secuencial para realizar la operación $E_j := E_j - D_1 C_j$ (C_1 se particiona acorde a la partición de E).

Los resultados obtenidos mediante la paralelización de nivel superior a BLAS y el uso de la caché *software* para la variante 2 de la factorización LU con pivotamiento incremental OOC están incluidos en las Figuras 3.14 y 3.15. Aunque se produce una mejora en las prestaciones con la paralelización de nivel superior, hay un descenso de las mismas conforme aumenta el tamaño de la matriz, debido, en parte, a la disminución en la ganancia que produce el uso de la caché *software*, ya que la localidad temporal es menor.

En las mismas figuras se muestran los resultados obtenidos mediante la paralelización de nivel superior a BLAS, el uso de la caché *software* y el solapamiento de cálculos y accesos a disco para la variante 2. Se observa una mejora en las prestaciones, que se mantiene incluso para las matrices más grandes, ocultando la latencia: 61,8 GFLOPS frente a 42,6 GFLOPS en TESLA y 82 GFLOPS frente a 73,4 GFLOPS en TESLA2. Cabe destacar que la ganancia en TESLA es de un 15 % (sobre la velocidad pico de la máquina), mientras que en TESLA2 es de tan solo un 5 %. Esto es debido a que esta arquitectura dispone de un RAID con 6 discos, por lo que la latencia es, de por sí, menor.

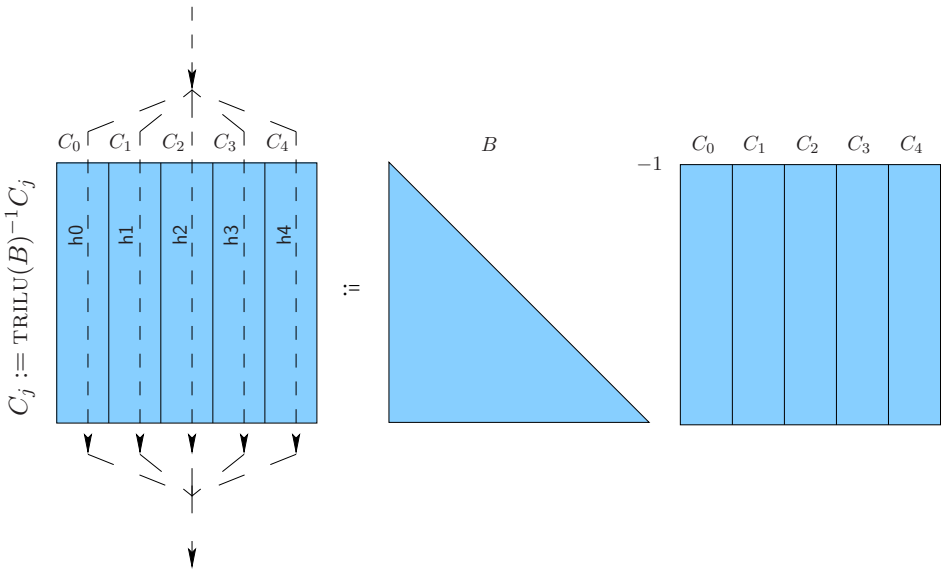


Figura 3.13: Paralelización de nivel superior a BLAS en el módulo básico APPLY_L. Las hebras $h0, \ldots, h4$ trabajan en paralelo para resolver el sistema $C := \text{TRILU}(B)^{-1} C$. La única sincronización tiene lugar tras la resolución del sistema.

3.7. Resumen y conclusiones

En este capítulo se han aplicado las técnicas OOC aportadas en esta tesis para la resolución de una operación más compleja que la factorización de Cholesky (tratada en el Capítulo 2), la factorización LU con pivotamiento incremental. Al igual que en el capítulo anterior, se han aplicado las técnicas de manera progresiva, para evaluar el impacto que producen sobre las prestaciones que se obtienen. Aparte del uso de la caché *software* y del *run-time* que permite el solapamiento de la

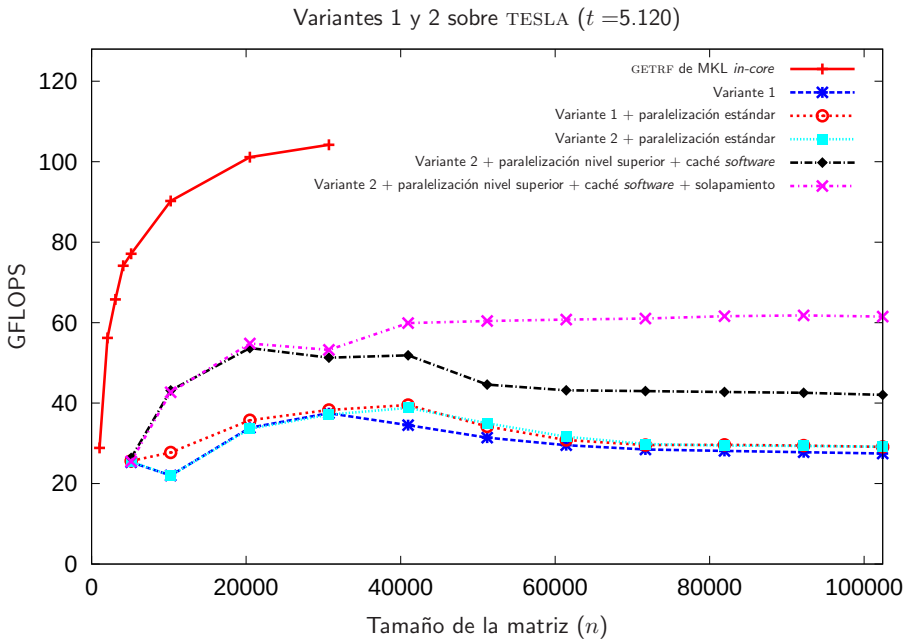


Figura 3.14: Ejecución sobre TESLA de distintas implementaciones OOC de las dos variantes de la factorización LU con pivotamiento incremental.

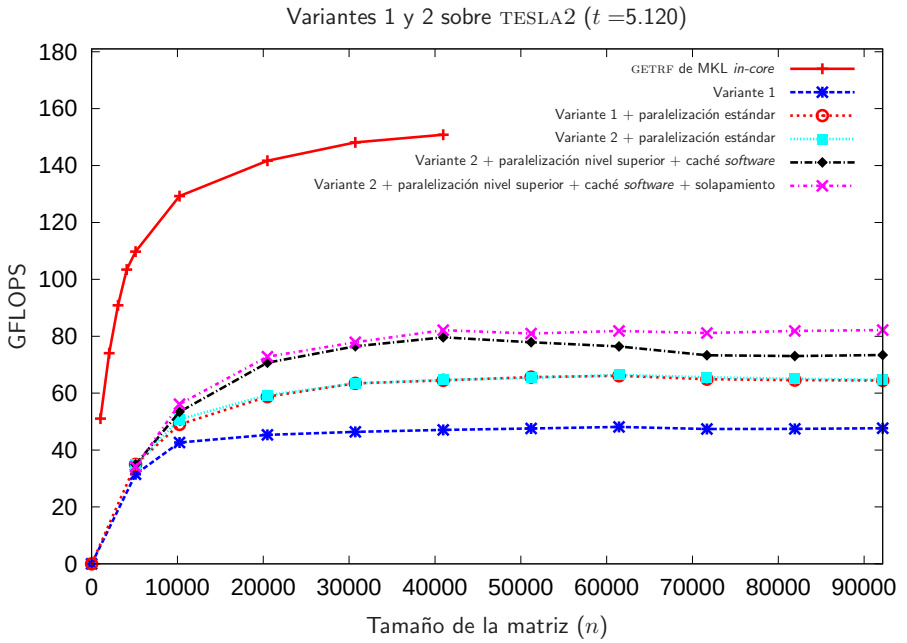


Figura 3.15: Ejecución sobre TESLA2 de distintas implementaciones OOC de las dos variantes de la factorización LU con pivotamiento incremental.

E/S y los cálculos, se ha realizado una nueva aportación mediante la paralelización de los códigos en un nivel superior a BLAS, consiguiendo así una mejora en el rendimiento.

Las implementaciones realizadas se han evaluado en dos arquitecturas diferentes, representativas de los sistemas de cálculo actuales: ROPE, un ordenador personal con un único procesador INTEL PENTIUM, y TESLA y TESLA2, estaciones de trabajo con dos procesadores INTEL XEON que disponen de cuatro núcleos cada uno. La diferencia principal entre TESLA y TESLA2 es que la primera dispone de un solo disco a 7.200 r.p.m. mientras que TESLA2 dispone de un RAID de seis discos a 15.000 r.p.m.

A diferencia de la experimentación realizada sobre la factorización de Cholesky, para este problema no hemos trabajado sobre GPU debido a que la biblioteca CUBLAS no proporciona las rutinas necesarias para implementar eficientemente las operaciones de los módulos $\text{LU}_{\text{TD}}^{\text{LIN}}$ y $\text{APPLY\_L}_{\text{TD}}^{\text{LIN}}$ sobre matrices de 2×1 bloques.

Los resultados obtenidos son satisfactorios, ya que se logra mantener las prestaciones conforme aumenta el tamaño de la matriz, ocultando en gran medida la latencia en el acceso al disco, aún cuando disminuye su velocidad. El rendimiento es siempre menor que el obtenido en la factorización de Cholesky puesto que el algoritmo de la factorización LU tiene una mayor complejidad debido a la realización del pivotamiento, y debido a que el tamaño de los datos de entrada es también mayor, ya que en esta ocasión la matriz no es simétrica y por lo tanto se almacenan y se procesan todos sus elementos.

Capítulo 4

Factorización QR en Disco

Una vez tratadas las factorizaciones de Cholesky (Capítulo 2) y LU (Capítulo 3), abordamos en este capítulo la factorización QR de una matriz que, por su tamaño, no es posible almacenar completamente en la memoria principal del sistema y, por tanto, reside inicialmente en disco. Esta operación de descomposición está asociada a la resolución por mínimos cuadrados de sistemas de ecuaciones lineales sobredeterminados. Como en los capítulos anteriores, nuestro trabajo en este capítulo analiza, desde el punto de vista teórico y práctico, la implementación *Out-of-Core* (OOC) de dos variantes algorítmicas para el cálculo de esta factorización. La solución propuesta calcula la factorización mediante un algoritmo alternativo al tradicional, basado en el problema de la actualización de una factorización cuando cambian algunas de las filas/columnas de la matriz de coeficientes. Esta solución, aunque conlleva un mayor coste computacional, da lugar a algoritmos escalables.

Las arquitecturas destino de los algoritmos OOC desarrollados son dos: una plataforma de altas prestaciones con un único procesador de propósito general y un procesador multinúcleo con varios núcleos (*cores*) de propósito general que comparten una memoria común.

El capítulo está estructurado del modo siguiente. En la primera sección se define la factorización QR, se justifica su utilidad, y se ofrecen dos algoritmos, uno escalar y otro por bloques, para su cálculo mediante transformaciones de Householder. Puesto que estos algoritmos trabajan sobre paneles de la matriz (secciones verticales) no son escalables desde el punto de vista de su implementación OOC, lo que hace necesario plantear algoritmos orientados a bloques que después sean fácilmente adaptables para trabajar OOC. Para ello se plantea un algoritmo basado en el problema de la actualización de la factorización QR de una matriz cuando cambian algunas de sus filas y columnas. En la Sección 4.2 se discuten las implementaciones de las dos variantes orientadas a bloques presentadas en la sección anterior. Las Secciones 4.3 y 4.4 recogen la evaluación del impacto experimental que suponen las propuestas realizadas sobre una arquitectura monoprocesador y una arquitectura multihebra, respectivamente. Finalmente, se incluye una sección donde se resume el contenido y las aportaciones del capítulo.

4.1. Factorización QR

Uno de los métodos que se utilizan habitualmente para resolver sistemas de ecuaciones lineales sobredeterminados es el de mínimos cuadrados. Dado el sistema $Ax = b$, con $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ y $m \geq n$, su resolución mediante este método consiste en encontrar el vector $x \in \mathbb{R}^n$ que

minimiza $\| Ax - b \|_2$. Los procedimientos más fiables para resolver este problema conllevan la reducción de la matriz A a alguna forma canónica mediante transformaciones ortogonales. Entre estos procedimientos el más comúnmente utilizado es la factorización QR [32], que descompone la matriz $A \in \mathbb{R}^{m\times n}$ en el producto $A = QR$, donde $Q \in \mathbb{R}^{m\times m}$ es ortogonal y $R \in \mathbb{R}^{m\times n}$ es triangular superior. Una vez factorizada la matriz en el producto $A = QR$, la resolución del problema de mínimos cuadrados se lleva a cabo calculando, en primer lugar, $y := Q^{\mathrm{T}}b$ y resolviendo, después, el sistema triangular superior $Rx = y$. El cálculo de estas dos últimas operaciones no se aborda explícitamente en nuestro trabajo ya que su coste computacional presenta un coste de orden menor frente al de la propia factorización.

Existen varios métodos para calcular la factorización QR. Así, los hay que se basan en transformaciones de Householder, otros se basan en rotaciones de Givens, y por último están los métodos que realizan una ortogonalización vía Gram-Schmidt o vía Gram-Schmidt modificado [32]. Cuando se trabaja con matrices densas, el método a elegir depende en gran medida de cómo se va a utilizar después la factorización, de la estabilidad del sistema y de la dimensión de la matriz. Cuando $m \gg n$, el método basado en transformaciones de Householder es muy conveniente.

4.1.1. Transformaciones de Householder

Definición 7 *Dado el vector $x \in \mathbb{R}^m$ y la partición $x = \begin{pmatrix} \chi_0 \\ x_1 \end{pmatrix}$, donde χ_0 es el primer elemento de x, el vector de Householder asociado a x se define como*

$$u = \begin{pmatrix} 1 \\ x_1/\nu_0 \end{pmatrix},$$

donde $\nu_0 = \chi_0 + signo(\chi_0)\|x\|_2$.

Definición 8 *Si $\sigma = \frac{2}{u^{\mathrm{T}}u}$, a la transformación $I_m - \sigma uu^{\mathrm{T}}$ se le denomina transformación o reflector de Householder.*

Al aplicar la transformación de Householder asociada a x se tiene que $(I_m - \sigma uu^{\mathrm{T}})x = \eta e_0$; es decir, en el resultado se anulan todos los elementos de x excepto el primero, que toma el valor $\eta = -\mathrm{signo}(\chi_0)\|x\|_2$.

Utilizaremos la notación $[u, \eta, \sigma] := h(x)$ para referirnos al cálculo de η, u y σ a partir del vector x, y denominaremos $H(x)$ a la transformación correspondiente, $H(x) = (I - \sigma uu^{\mathrm{T}})$. Por simplicidad, obviaremos el subíndice que indica el tamaño de la matriz identidad cuando quede determinado por el contexto. Una característica importante de $H(x)$ es que es ortogonal y simétrica ($H(x)^{-1} = H(x)^{\mathrm{T}}$ y $H(x)^{\mathrm{T}} = H(x)$ respectivamente).

4.1.2. Algoritmo escalar en notación FLAME

La factorización QR puede obtenerse mediante el algoritmo escalar (orientado a la derecha) mostrado en la Figura 4.1. En este algoritmo, las transformaciones de Householder se van calculando sucesivamente para anular los elementos subdiagonales de la matriz A. Los vectores de Householder sobreescriben los elementos que se han anulado (es decir, la parte estrictamente triangular inferior de A) y, al finalizar el proceso, el factor triangular R queda contenido sobre la parte triangular superior de la matriz A. Los escalares σ se almacenan en el vector s de n elementos. El coste de este algoritmo es de $2n^2(m - n/3)$ *flops*.

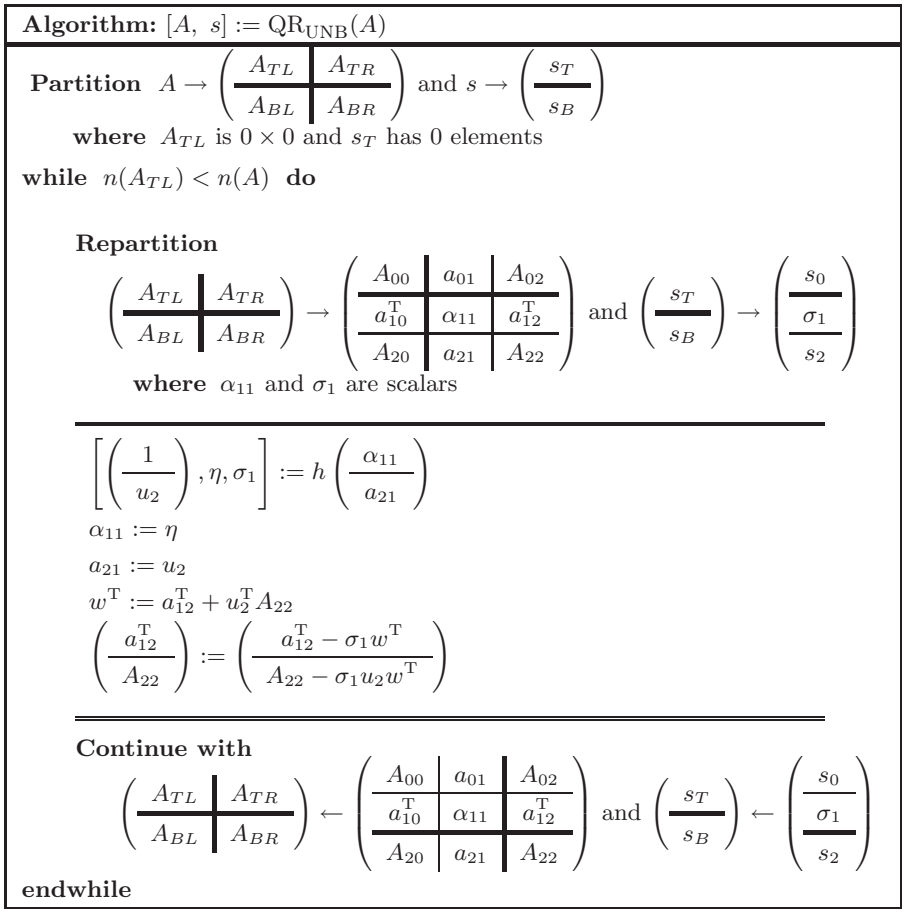


Figura 4.1: Algoritmo escalar para calcular la factorización QR mediante transformaciones de Householder.

Si se desea calcular la matriz Q explícitamente, se debe acumular el producto $H_0 \cdot H_1 \cdots H_{n-1} = Q$, siendo H_k la $(k+1)$-ésima transformación de Householder calculada mediante el algoritmo anterior. Puesto que en los problemas de mínimos cuadrados no es habitual la necesidad de formar la matriz Q, no trataremos este aspecto en adelante. La realización del cálculo $y := Q^{\mathrm{T}} b$ en la resolución del problema de mínimos cuadrados tras la factorización no requiere la formación de la matriz Q, ya que las transformaciones ortogonales que conlleva se realizan directamente a partir de los vectores de Householder almacenados en A, y de modo más eficiente que a partir de Q.

4.1.3. Algoritmo por bloques en notación FLAME

Para obtener un algoritmo por bloques que calcule la factorización QR utilizamos la representación WY compacta [12, 57], que acumula productos de transformaciones de Householder, de manera que su aplicación se puede realizar mediante productos de matrices (en el algoritmo escalar las transformaciones se aplican mediante productos matriz-vector).

Hay dos detalles que juegan un papel clave en la construcción del algoritmo por bloques de la Figura 4.2:

Algorithm: $[A,\ S] := \mathrm{QR}_{\mathrm{BLK}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ and $S \rightarrow \left(\begin{array}{c} S_T \\ \hline S_B \end{array}\right)$

where A_{TL} is 0×0 and S_T has 0 rows

while $n(A_{TL}) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} S_T \\ \hline S_B \end{array}\right) \rightarrow \left(\begin{array}{c} S_0 \\ \hline S_1 \\ \hline S_2 \end{array}\right)$$

where A_{11} is $b \times b$ and S_1 has b rows

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right),\ s_1\right] := \left[\left(\begin{array}{c} \{U\backslash R\}_{11} \\ \hline U_{21} \end{array}\right),\ s_1\right] = \mathrm{QR}_{\mathrm{UNB}}\left(\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)\right)$$

Compute S_1 from $\left[\left(\begin{array}{c} \{U\backslash R\}_{11} \\ \hline U_{21} \end{array}\right),\ s_1\right]$

$$\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) := \left(I + \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right) S_1 \left(\begin{array}{c|c} U_{11}^{\mathrm{T}} & U_{21}^{\mathrm{T}} \end{array}\right)\right) \left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right)$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} S_T \\ \hline S_B \end{array}\right) \leftarrow \left(\begin{array}{c} S_0 \\ \hline S_1 \\ \hline S_2 \end{array}\right)$$

endwhile

Figura 4.2: Algoritmo por bloques para calcular la factorización QR mediante transformaciones de Householder.

- Si $u_0, \ldots, u_{b-1}$ son los primeros b vectores de Householder calculados en la factorización y $\sigma_0, \ldots, \sigma_{b-1}$ son los escalares correspondientes, de modo que $H_j = (I - \sigma_j u_j u_j^{\mathrm{T}})$, $j = 0, \ldots, b-1$, entonces

$$H_0 \cdot H_1 \cdots H_{b-1} = I + U_b S_b U_b^{\mathrm{T}}, \tag{4.1}$$

  donde U_b es una matriz trapezoidal inferior unidad de tamaño $n \times b$, S_b es una matriz triangular superior de tamaño $b \times b$, y bajo la diagonal de la columna $(j+1)$ de U_b se encuentra u_j, para $j = 0, \ldots, b-1$.

- La factorización QR de las k primeras columnas de A da lugar a los mismos k vectores u_k y a los mismos valores en la parte triangular superior de esas k columnas que la factorización QR de toda la matriz.

Asumiendo que $b \ll m, n$, el coste del algoritmo por bloques es igual al coste del algoritmo escalar, es decir, $2n^2(m - n/3)$ *flops*. El algoritmo devuelve en la matriz $S \in \mathbb{R}^{n \times b}$ las matrices triangulares superiores "S_1", de tamaño $b \times b$ cada una de ellas, que agrupan parte de las transformaciones de Householder correspondientes a cada bloque de columnas de la matriz A. Este algoritmo, que se desvía de las implementaciones tradicionales como la de LAPACK, no necesita volver a calcular

dichas matrices al aplicar las transformaciones de Householder al vector de términos independientes del sistema.

Aunque los algoritmos por bloques son deseables para obtener buenas prestaciones, no son escalables cuando las matrices se particionan en paneles (secciones verticales), como sucede en el algoritmo de la Figura 4.2. Nuestro objetivo es, por tanto, utilizar algoritmos orientados a bloques (algoritmos en los que el bloque es la unidad básica de cálculo), ya que son escalables, permiten un mayor grado de paralelismo y pueden ser directamente transformados en algoritmos OOC con tan solo plantear una equivalencia entre el bloque y el *tile* (la unidad de almacenamiento en disco).

4.1.4. Algoritmo orientado bloques en notación FLAME

Gunter y van de Geijn [33] plantean un algoritmo por bloques para calcular la actualización de una factorización QR cuando se añaden más ecuaciones lineales al sistema inicial. Sería el caso en que, por ejemplo, se ha obtenido la factorización QR de una matriz $A = QR$ y se necesita calcular la factorización

$$\left(\frac{A}{B}\right) = \bar{Q}\bar{R}. \tag{4.2}$$

Si la factorización de A se ha realizado mediante el algoritmo de la Figura 4.2, se tiene que la parte triangular superior de A se ha sobreescrito con la matriz R y las transformaciones de Householder se han almacenado de manera compacta en su parte triangular inferior esctricta y en la matriz S. Ya que la factorización QR de

$$\left(\frac{R}{B}\right) \tag{4.3}$$

produce la misma matriz triangular superior $\bar{R}$ que la factorización de (4.2), si además no es necesario formar explícitamente la matriz $\bar{Q}$ y es suficiente con almacenar los vectores de Householder que calculan la factorización QR de A y la posterior factorización de (4.3), se dispone de una estrategia para calcular la factorización QR de un sistema que ha sido ampliado.

Este es básicamente el mismo procedimiento seguido por Joffrain et al. [39] para obtener un algoritmo que calcula la factorización LU de una matriz con pivotamiento incremental, a partir del problema de cómo actualizar una factorización LU existente cuando cambian algunas de las filas y/o columnas de la matriz que da origen a la misma. Extendiendo ambos análisis al caso que nos ocupa, el problema consiste en calcular la factorización QR de la matriz A, particionada como

$$A = \left(\begin{array}{c|c} B & C \\ \hline D & E \end{array}\right), \tag{4.4}$$

para diferentes valores de C, D y E. La cuestión se transforma en cómo reutilizar la factorización de B para obtener la de A para diferentes C, D y E.

A continuación se presenta el algoritmo básico que calcula la factorización QR de la matriz A en (4.4):

- **Paso 1**: Factorizar B. En este paso se calcula la factorización QR de B mediante transformaciones de Householder.

$$[B,\ S] := \mathrm{QR}(B)$$

- **Paso 2**: Actualizar C consistentemente con la factorización de B.

$$C := \text{APPLY\_Q}(B, S, C)$$

- **Paso 3**: Factorizar $\begin{pmatrix} R \\ D \end{pmatrix}$. En este paso se calcula la factorización QR mediante transformaciones de Householder, preservando la estructura triangular superior de R (que se almacena en B).

$$\left[\begin{pmatrix} R \\ D \end{pmatrix}, \bar{S} \right] := \text{QRTD} \begin{pmatrix} R \\ D \end{pmatrix}$$

- **Paso 4**: Actualizar $\begin{pmatrix} C \\ E \end{pmatrix}$ consistentemente con la factorización de $\begin{pmatrix} R \\ D \end{pmatrix}$.

$$\begin{pmatrix} C \\ E \end{pmatrix} := \text{APPLY\_QTD} \left(\begin{pmatrix} R \\ D \end{pmatrix}, \bar{S}, \begin{pmatrix} C \\ E \end{pmatrix} \right)$$

- **Paso 5**: Factorizar E mediante transformaciones de Householder.

$$\left[E, \tilde{S} \right] := \text{QR}(E)$$

Los módulos QR y APPLY_Q se pueden implementar mediante los algoritmos de las Figuras 4.2 y 4.3, respectivamente. Ya que éstas son operaciones habituales de álgebra lineal, existen implementaciones secuenciales eficientes para las mismas en bibliotecas como `libflame` y LAPACK (rutinas GEQRF y ORMQR).

En cuanto a las implementaciones de los módulos QRTD y APPLY_QTD, cabe destacar el hecho de que el bloque superior de la matriz de 2×1 con la que trabajan es triangular superior, por lo que han de explotar esta estructura y mantenerla. La Figura 4.4 muestra cómo hacerlo en el caso del módulo QRTD. Asumiendo que R y C son ambas de dimensión $t \times t$ y que $b \ll t$, calcular QRTD requiere $2t^3$ *flops*, que es bastante menor que los $8t^3/3$ *flops* que se requieren si no se tiene en cuenta la estructura de la matriz R. El procedimiento del módulo APPLY_QTD se muestra en la Figura 4.5; en este caso, con las mismas consideraciones de tamaño que en QRTD, el coste se reduce de $8t^3$ a $4t^3$ *flops* cuando se tiene en cuenta la estructura del bloque superior de la matriz [34].

Una vez establecidos los módulos básicos del algoritmo, estamos en disposición de obtener un algoritmo orientado a bloques que dé lugar, más tarde, al algoritmo OOC. Consideramos un particionado de la matriz $A \in \mathbb{R}^{m \times n}$ en bloques cuadrados (serán los *tiles* cuando trabajemos OOC) de tamaño $t \times t$ (por sencillez, supondremos que ambas dimensiones de la matriz, m y n, son múltiplos de t). Las Figuras 4.6 y 4.7 muestran un algoritmo orientado a bloques orientado hacia la derecha que calcula la factorización QR de una matriz. En los algoritmos de las figuras aparecen sombreados los módulos básicos identificados en la resolución del problema de actualización de una factorización QR: la factorización QR de una matriz densa (QR), la aplicación de la transformación ortogonal resultante de esta factorización (APPLY_Q), la factorización QR de una matriz con 2×1 bloques, en la que el bloque superior es triangular superior y el inferior es denso (QRTD), y la

Algorithm: $C := \text{APPLY\_Q}(B,\ S,\ C)$

Partition $B \rightarrow \left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline B_{BL} & B_{BR} \end{array}\right)$, $S \rightarrow \left(\begin{array}{c} S_T \\ \hline S_B \end{array}\right)$, $C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$

where B_{TL} is 0×0, S_T and C_T have 0 rows

while $n(B_{TL}) < n(B)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline B_{BL} & B_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_{00} & B_{01} & B_{02} \\ \hline B_{10} & B_{11} & B_{12} \\ \hline B_{20} & B_{21} & B_{22} \end{array}\right), \left(\begin{array}{c} S_T \\ \hline S_B \end{array}\right) \rightarrow \left(\begin{array}{c} S_0 \\ \hline S_1 \\ \hline S_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$$

where B_{11} is $b \times b$, S_1 and C_1 have b rows

$$\left(\begin{array}{c} C_1 \\ \hline C_2 \end{array}\right) := \left(I + \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right) S_1 \left(\begin{array}{c|c} U_{11}^{\mathrm{T}} & U_{21}^{\mathrm{T}} \end{array}\right)\right) \left(\begin{array}{c} C_1 \\ \hline C_2 \end{array}\right)$$

Continue with

$$\left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline B_{BL} & B_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_{00} & B_{01} & B_{02} \\ \hline B_{10} & B_{11} & B_{12} \\ \hline B_{20} & B_{21} & B_{22} \end{array}\right), \left(\begin{array}{c} S_T \\ \hline S_B \end{array}\right) \leftarrow \left(\begin{array}{c} S_0 \\ \hline S_1 \\ \hline S_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$$

endwhile

Figura 4.3: Algoritmo por bloques para actualizar la matriz C respecto a la factorización QR de la matriz B (almacenada en B y S). En el algoritmo, U_{11} se refiere a la parte triangular inferior de B_{11}, mientras que U_{21} se refiere a B_{21}.

Algorithm: $\left[\left(\begin{array}{c} R \\ \hline D \end{array}\right), \bar{S}\right] := \text{QRTD}\left(\left(\begin{array}{c} R \\ \hline D \end{array}\right)\right)$

Partition $R \rightarrow \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $\bar{S}^{(\text{TD})} \rightarrow \left(\begin{array}{c} \bar{S}_T \\ \hline \bar{S}_B \end{array}\right)$

where R_{TL} is 0×0, D_L has 0 columns, $\bar{S}_T$ has 0 rows

while $m(R_{TL}) < m(R)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} R_{00} & R_{01} & R_{02} \\ \hline R_{10} & R_{11} & R_{12} \\ \hline R_{20} & R_{21} & R_{22} \end{array}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right), \left(\begin{array}{c} \bar{S}_T \\ \hline \bar{S}_B \end{array}\right) \rightarrow \left(\begin{array}{c} \bar{S}_0 \\ \hline \bar{S}_1 \\ \hline \bar{S}_2 \end{array}\right)$$

where R_{11} is $b \times b$, D_1 has b columns, $\bar{S}_1$ has b rows

$$\left[\left(\begin{array}{c} R_{11} \\ \hline D_1 \end{array}\right), \bar{S}_1\right] := \text{QR}\left(\left(\begin{array}{c} R_{11} \\ \hline D_1 \end{array}\right)\right)$$

$$\left(\begin{array}{c} R_{12} \\ \hline D_2 \end{array}\right) := \text{APPLY\_Q}\left(\left(\begin{array}{c} R_{11} \\ \hline D_1 \end{array}\right), \bar{S}_1, \left(\begin{array}{c} R_{12} \\ \hline D_2 \end{array}\right)\right)$$

Continue with

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} R_{00} & R_{01} & R_{02} \\ \hline R_{10} & R_{11} & R_{12} \\ \hline R_{20} & R_{21} & R_{22} \end{array}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right), \left(\begin{array}{c} \bar{S}_T \\ \hline \bar{S}_B \end{array}\right) \leftarrow \left(\begin{array}{c} \bar{S}_0 \\ \hline \bar{S}_1 \\ \hline \bar{S}_2 \end{array}\right)$$

endwhile

Figura 4.4: Algoritmo por bloques para el módulo QRTD que calcula la factorización QR de una matriz de la forma $(R^{\text{T}}, D^{\text{T}})^{\text{T}}$ cuyo bloque R es triangular superior.

Algorithm: $\left[\left(\frac{C}{E}\right)\right] := \text{APPLY\_QTD}\left(\left(\frac{R}{D}\right), \bar{S}^{(\text{TD})}, \left(\frac{C}{E}\right)\right)$

Partition $R \rightarrow \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $\bar{S}^{(\text{TD})} \rightarrow \left(\frac{\bar{S}_T}{\bar{S}_B}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$

where R_{TL} is 0×0, D_L has 0 columns, $\bar{S}_T$ and C_T have 0 rows

while $m(R_{TL}) < m(R)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} R_{00} & R_{01} & R_{02} \\ \hline R_{10} & R_{11} & R_{12} \\ \hline R_{20} & R_{21} & R_{22} \end{array}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

$$\left(\frac{\bar{S}_T}{\bar{S}_B}\right) \rightarrow \left(\frac{\bar{S}_0}{\bar{S}_1}{\bar{S}_2}\right), \left(\frac{C_T}{C_B}\right) \rightarrow \left(\frac{C_0}{C_1}{C_2}\right)$$

where R_{11} is $b \times b$, D_1 has b columns, $\bar{S}_1$ and C_1 have b rows

$$\left(\frac{C_1}{E}\right) := \text{APPLY\_Q}\left(\left(\frac{R_{11}}{D_1}\right), \bar{S}_1, \left(\frac{C_1}{E}\right)\right)$$

Continue with

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} R_{00} & R_{01} & R_{02} \\ \hline R_{10} & R_{11} & R_{12} \\ \hline R_{20} & R_{21} & R_{22} \end{array}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

$$\left(\frac{\bar{S}_T}{\bar{S}_B}\right) \leftarrow \left(\frac{\bar{S}_0}{\bar{S}_1}{\bar{S}_2}\right), \left(\frac{C_T}{C_B}\right) \leftarrow \left(\frac{C_0}{C_1}{C_2}\right)$$

endwhile

Figura 4.5: Algoritmo por bloques para el módulo APPLY_QTD que actualiza la matriz $(C^{\mathrm{T}}, E^{\mathrm{T}})^{\mathrm{T}}$ consistentemente con la factorización QR de la matriz $(R^{\mathrm{T}}, D^{\mathrm{T}})^{\mathrm{T}}$ realizada con el algoritmo QRTD de la Figura 4.4.

Algorithm: $[A,\ S] := \text{QR\_B\_VAR1}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $S \rightarrow \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline S_{BL} & S_{BR} \end{array}\right)$

where A_{TL}, S_{TL} have 0×0 tiles

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline S_{BL} & S_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} S_{00} & S_{01} & S_{02} \\ \hline S_{10} & S_{11} & S_{12} \\ \hline S_{20} & S_{21} & S_{22} \end{array}\right)$$

where A_{11}, S_{11} are tiles

$[A_{11},\ S_{11}] := \text{QR}(A_{11})$

$A_{12} := \text{QR\_B1}(A_{11}, S_{11}, A_{12})$

$[A_{11},\ A_{21},\ S_{21}] := \text{QR\_B2}(A_{11}, A_{21})$

$[A_{12},\ A_{22}] := \text{QR\_B3}(S_{21}, A_{11}, A_{12}, A_{21}, A_{22})$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline S_{BL} & S_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} S_{00} & S_{01} & S_{02} \\ \hline S_{10} & S_{11} & S_{12} \\ \hline S_{20} & S_{21} & S_{22} \end{array}\right)$$

endwhile

Algorithm: $[C] := \text{QR\_B1}(B, \bar{S}, C)$

Partition $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array}\right)$

where C_L is 0 tiles wide

while $n(C_L) < n(C)$ **do**

Repartition

$$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$$

where C_1 is a tile

$C_1 := \text{APPLY\_Q}(B, \bar{S}, C_1)$

Continue with

$$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$$

endwhile

Algorithm: $\left[B,\ D,\ \bar{S}\right] := \text{QR\_B2}(B, D)$

Partition $D \rightarrow \left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right)$, $\bar{S} \rightarrow \left(\begin{array}{c} \bar{S}_T \\ \hline \bar{S}_B \end{array}\right)$

where D_T, $\bar{S}_T$ are 0 tiles high

while $m(D_T) < m(D)$ **do**

Repartition

$$\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right), \left(\begin{array}{c} \bar{S}_T \\ \hline \bar{S}_B \end{array}\right) \rightarrow \left(\begin{array}{c} \bar{S}_0 \\ \hline \bar{S}_1 \\ \hline \bar{S}_2 \end{array}\right)$$

where D_1, $\bar{S}_1$ are tiles

$$\left[\left(\begin{array}{c} B \\ \hline D_1 \end{array}\right), \bar{S}_1\right] := \text{QRTD}\left(\left(\begin{array}{c} B \\ \hline D_1 \end{array}\right)\right)$$

Continue with

$$\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \leftarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right), \left(\begin{array}{c} \bar{S}_T \\ \hline \bar{S}_B \end{array}\right) \leftarrow \left(\begin{array}{c} \bar{S}_0 \\ \hline \bar{S}_1 \\ \hline \bar{S}_2 \end{array}\right)$$

endwhile

Figura 4.6: Algoritmo orientado a bloques para calcular la factorización QR. Variante 1 orientada hacia la derecha.

Algorithm: $[C,\ E] := \text{QR\_B3}(\bar{S}, R, C, D, E)$

Partition $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array}\right)$, $E \rightarrow \left(\begin{array}{c|c} E_L & E_R \end{array}\right)$
where C_L, E_L are 0 tiles wide

while $n(C_L) < n(C)$ **do**

Repartition
$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$, $\left(\begin{array}{c|c} E_L & E_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} E_0 & E_1 & E_2 \end{array}\right)$
where C_1 is a tile, E_1 is 1 tile wide

$[C_1, E_1] := \text{QR\_B4}(\bar{S}, R, C_1, D, E_1)$

Continue with
$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$, $\left(\begin{array}{c|c} E_L & E_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} E_0 & E_1 & E_2 \end{array}\right)$

endwhile

Algorithm: $[C,\ E] := \text{QR\_B4}(\bar{S}, R, C, D, E)$

Partition $\bar{S} \rightarrow \left(\begin{array}{c} \bar{S}_T \\ \hline \bar{S}_B \end{array}\right)$, $D \rightarrow \left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right)$, $E \rightarrow \left(\begin{array}{c} E_T \\ \hline E_B \end{array}\right)$
where $\bar{S}_T$, D_T, E_T are 0 tiles high

while $m(D_T) < m(D)$ **do**

Repartition
$\left(\begin{array}{c} \bar{S}_T \\ \hline \bar{S}_B \end{array}\right) \rightarrow \left(\begin{array}{c} \bar{S}_0 \\ \hline \bar{S}_1 \\ \hline \bar{S}_2 \end{array}\right)$, $\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right)$, $\left(\begin{array}{c} E_T \\ \hline E_B \end{array}\right) \rightarrow \left(\begin{array}{c} E_0 \\ \hline E_1 \\ \hline E_2 \end{array}\right)$
where $\bar{S}_1$, D_1, E_1 are tiles

$\left(\begin{array}{c} C \\ \hline E_1 \end{array}\right) :=$ APPLY_QTD $\left(\left(\begin{array}{c} R \\ \hline D_1 \end{array}\right), \bar{S}_1, \left(\begin{array}{c} C \\ \hline E_1 \end{array}\right)\right)$

Continue with
$\left(\begin{array}{c} \bar{S}_T \\ \hline \bar{S}_B \end{array}\right) \leftarrow \left(\begin{array}{c} \bar{S}_0 \\ \hline \bar{S}_1 \\ \hline \bar{S}_2 \end{array}\right)$, $\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \leftarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right)$, $\left(\begin{array}{c} E_T \\ \hline E_B \end{array}\right) \leftarrow \left(\begin{array}{c} E_0 \\ \hline E_1 \\ \hline E_2 \end{array}\right)$

endwhile

Figura 4.7: Algoritmo orientado a bloques para calcular la factorización QR. Variante 1 orientada hacia la derecha (continuación).

Algorithm: $[A,\ S] := \text{QR\_B\_VAR2}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $S \rightarrow \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline S_{BL} & S_{BR} \end{array}\right)$

where A_{TL}, S_{TL} have 0×0 tiles

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline S_{BL} & S_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} S_{00} & S_{01} & S_{02} \\ \hline S_{10} & S_{11} & S_{12} \\ \hline S_{20} & S_{21} & S_{22} \end{array}\right)$$

where A_{11}, S_{11} are tiles

$$\left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right) := \text{LU\_B5}\left(\left(\begin{array}{c} A_{00} \\ \hline A_{10} \\ \hline A_{20} \end{array}\right), \left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right), \left(\begin{array}{c} S_{00} \\ \hline S_{10} \\ \hline S_{20} \end{array}\right)\right)$$

$$[A_{11}, S_{11}] := \text{QR}(A_{11})$$

$$[A_{11},\ A_{21},\ S_{21}] := \text{LU\_B2}(A_{11}, A_{21})$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline S_{BL} & S_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} S_{00} & S_{01} & S_{02} \\ \hline S_{10} & S_{11} & S_{12} \\ \hline S_{20} & S_{21} & S_{22} \end{array}\right)$$

endwhile

Algorithm: $[B] := \text{QR\_B5}(A, B, \bar{S})$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$, $\bar{S} \rightarrow \left(\begin{array}{c|c} \bar{S}_{TL} & \bar{S}_{TR} \\ \hline \bar{S}_{BL} & \bar{S}_{BR} \end{array}\right)$

where A_{TL}, $\bar{S}_{TL}$ have 0×0 tiles, B_T is 0 tiles high

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right), \left(\begin{array}{c|c} \bar{S}_{TL} & \bar{S}_{TR} \\ \hline \bar{S}_{BL} & \bar{S}_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} \bar{S}_{00} & \bar{S}_{01} & \bar{S}_{02} \\ \hline \bar{S}_{10} & \bar{S}_{11} & \bar{S}_{12} \\ \hline \bar{S}_{20} & \bar{S}_{21} & \bar{S}_{22} \end{array}\right)$$

where A_{11}, B_1, $\bar{S}_{11}$ are tiles

$$B_1 := \text{APPLY\_QR}(A_{11},\ \bar{S}_{11},\ B_1)$$

$$[B_1,\ B_2] := \text{QR\_B4}(\bar{S}_{21},\ A_{11},\ B_1,\ A_{21},\ B_2)$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right), \left(\begin{array}{c|c} \bar{S}_{TL} & \bar{S}_{TR} \\ \hline \bar{S}_{BL} & \bar{S}_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} \bar{S}_{00} & \bar{S}_{01} & \bar{S}_{02} \\ \hline \bar{S}_{10} & \bar{S}_{11} & \bar{S}_{12} \\ \hline \bar{S}_{20} & \bar{S}_{21} & \bar{S}_{22} \end{array}\right)$$

endwhile

Figura 4.8: Algoritmo orientado a bloques para calcular la factorización QR. Variante 2 orientada hacia la izquierda.

aplicación de las transformaciones correspondientes a esta última factorización sobre una matriz 2×1 (APPLY_QTD).

Para reducir el número de operaciones de E/S, planteamos en la Figura 4.8 un algoritmo orientado hacia la izquierda. Tanto éste como el algoritmo orientado hacia la derecha, tienen una estructura similar a las dos variantes orientadas a bloques para la factorización LU. Esto es así porque ambos algoritmos se han desarrollado a partir de la solución al problema de la actualización de una factorización existente cuando cambian algunas filas y columnas de la matriz de coeficientes. El motivo para obtener los algoritmos a partir de esta solución es el hecho de que los algoritmos originales de factorización trabajan sobre paneles de la matriz y, por lo tanto, no son escalables. Los nuevos algoritmos, aunque realizan más operaciones, sí son escalables.

4.2. Implementaciones OOC para la factorización QR

De nuevo, realizamos la transformación de los algoritmos a bloques en algoritmos OOC considerando que cada bloque es un *tile*. En esta sección no hacemos un estudio en detalle de las dos variantes algorítmicas orientadas a bloques obtenidas para la factorización QR puesto que su estructura, en esencia, es la misma que la de las variantes algorítmicas obtenidas para la factorización LU con pivotamiento incremental (ver Sección 3.4). En su lugar, nos detenemos en el estudio de las diferencias, que vienen dadas por las operaciones de cálculo que conllevan cada una de las factorizaciones.

Una primera diferencia se encuentra en los módulos básicos que se utilizan en ambas factorizaciones. En la factorización QR, estos módulos se denominan QR, APPLY_Q, QRTD y APPLY_QTD y sus implementaciones se muestran en las Figuras de la 4.2 a la 4.5. La segunda diferencia se encuentra en la estructura y el uso de la matriz auxiliar $\bar{S}$, que almacena de manera compacta las transformaciones de Householder tal y como se detalla a continuación.

La Figura 4.9 muestra gráficamente dos iteraciones de la variante 1 orientada a bloques hacia la derecha, QR_B_VAR1, concretamente la segunda y la tercera iteración. En la figura aparecen sombreados los *tiles* que son actualizados en cada paso. Al inicio de cada iteración, en el primer paso, se calcula la factorización QR del *tile* A_{11} mediante un algoritmo que procede por paneles de b columnas (el tamaño de cada panel es $t \times b$). La factorización de cada uno de estos paneles da lugar a un bloque de tamaño $b \times b$ triangular superior que corresponde a la matriz S_b de (4.1). En el ejemplo de la figura, el tamaño de *tile* es $t = 3b$, por lo que la factorización QR de A_{11} da lugar a tres bloques triangulares superiores de tamaño $b \times b$, que pueden almacenarse de manera compacta en S tal y como se muestra en la figura. El segundo paso de cada iteración (QR_B1) consiste en aplicar las transformaciones de Householder correspondientes a la factorización del *tile* A_{11} a los *tiles* que quedan a la derecha de éste en A_{12}. Por ejemplo, en la segunda iteración hay cuatro *tiles* en A_{12}, por lo que se invocará cuatro veces al módulo APPLY_Q. En el tercer paso (QR_B2), se anulan los elementos de los *tiles* que se encuentran por debajo del A_{11} (A_{21}). Para ello, se invoca al módulo QRTD con cada *tile* de A_{21}. Este módulo realiza la factorización QR de una matriz de 2×1 *tiles*: el *tile* superior es A_{11} y el inferior es uno de los *tiles* de A_{21} cuyos elementos se debe anular. El módulo QRTD también procede por paneles de b columnas, pero en este caso cada panel es de tamaño $2t \times b$. Puesto que el *tile* superior ya es triangular superior, los cálculos se realizan respetando los elementos nulos. Continuando con el ejemplo de la figura, ya que hay tres paneles de b columnas en cada una de estas matrices, en su factorización se generan tres nuevos bloques de tamaño $b \times b$ que se almacenan en $\bar{S}$. El cuarto y último paso del algoritmo (QR_B3) consiste

en la aplicación de las transformaciones de Householder del paso anterior (QR_B2), a los *tiles* que quedan a la derecha de A_{11} y A_{21}. Este paso se lleva a cabo mediante el módulo básico APPLY_QTD, que también actualiza matrices de 2×1 *tiles*. En la figura se muestra, en cada iteración, cómo se forma y cómo se utiliza cada columna de *tiles* de la matriz $\bar{S}$.

La Figura 4.10 muestra el desarrollo de la tercera iteración del algoritmo orientado a bloques hacia la izquierda QR_B_VAR2. De nuevo, aparecen sombreados los *tiles* que se actualizan en cada paso. En la figura se observa cómo se construye la matriz $\bar{S}$ y cómo se utilizan sus bloques $b \times b$. En el primer paso (QR_B5) se aplican las transformaciones de Householder producidas en iteraciones anteriores del algoritmo, tanto las transformaciones que anulan elementos en los *tiles* diagonales (mediante APPLY_Q), como las transformaciones que anulan los elementos en los *tiles* subdiagonales (mediante APPLY_QTD). En el segundo paso se realiza la factorización del *tile* diagonal A_{11}. Siguiendo con el mismo ejemplo utilizado para el algoritmo orientado hacia la derecha, tenemos que $t = 3b$, por lo que esta factorización da lugar a tres bloques $b \times b$ en $\bar{S}$. El tercer y último paso de esta segunda variante algorítmica (QR_B2) coincide con el tercer paso de la primera variante, la eliminación de los elementos de los *tiles* de A_{21}. En la figura se observa cómo se va construyendo la matriz $\bar{S}$ a partir de los bloques $b \times b$ triangulares superiores generados en las distintas factorizaciones.

4.2.1. Uso de una caché software

Se han realizado implementaciones OOC de ambas variantes algorítmicas realizando la gestión de la memoria como si se tratara de una caché. Esto ha requerido la creación de una nueva función para cada variante, que se encarga de generar la lista de tareas que llevan a cabo la factorización QR en cada caso. Puesto que los módulos básicos QRTD y APPLY_QTD no se encuentran implementados en las bibliotecas de álgebra lineal, también hemos llevado a cabo su implementación *in-core*.

Una vez realizadas estas implementaciones, el código se ha integrado con el *run-time* ya desarrollado, que es el encargado de la gestión de la caché *software*. Las gráficas de la Figura 4.11 muestran el ahorro, en cuanto al número de accesos a elementos, que supone la gestión de la memoria por parte del *run-time*. A la vista de los resultados, se confirma que la variante 2 orientada hacia la izquierda requiere menos E/S en la implementación tradicional, en la que los *tiles* se leen de disco cuando son necesarios en los cálculos y se escriben en disco después de ser actualizados. La integración con el *run-time* conlleva un ahorro en los accesos a elementos, que es más notorio en la variante 2, por lo que se espera obtener mejores prestaciones para ésta.

4.2.2. Solapamiento de cálculos y accesos a disco

La siguiente mejora introducida es la que proporciona el *run-time* mejorado que, mediante dos hebras especializadas, consigue solapar los accesos a disco con los cálculos, ocultando así la latencia de acceso al disco. El patrón de acceso a los *tiles* se hace en zigzag para mejorar la localidad en las referencias. Este patrón de acceso requiere una reorganización de las operaciones de actualización similar a la realizada en la factorización de Cholesky (ver Apartado 2.3.3 del Capítulo 2).

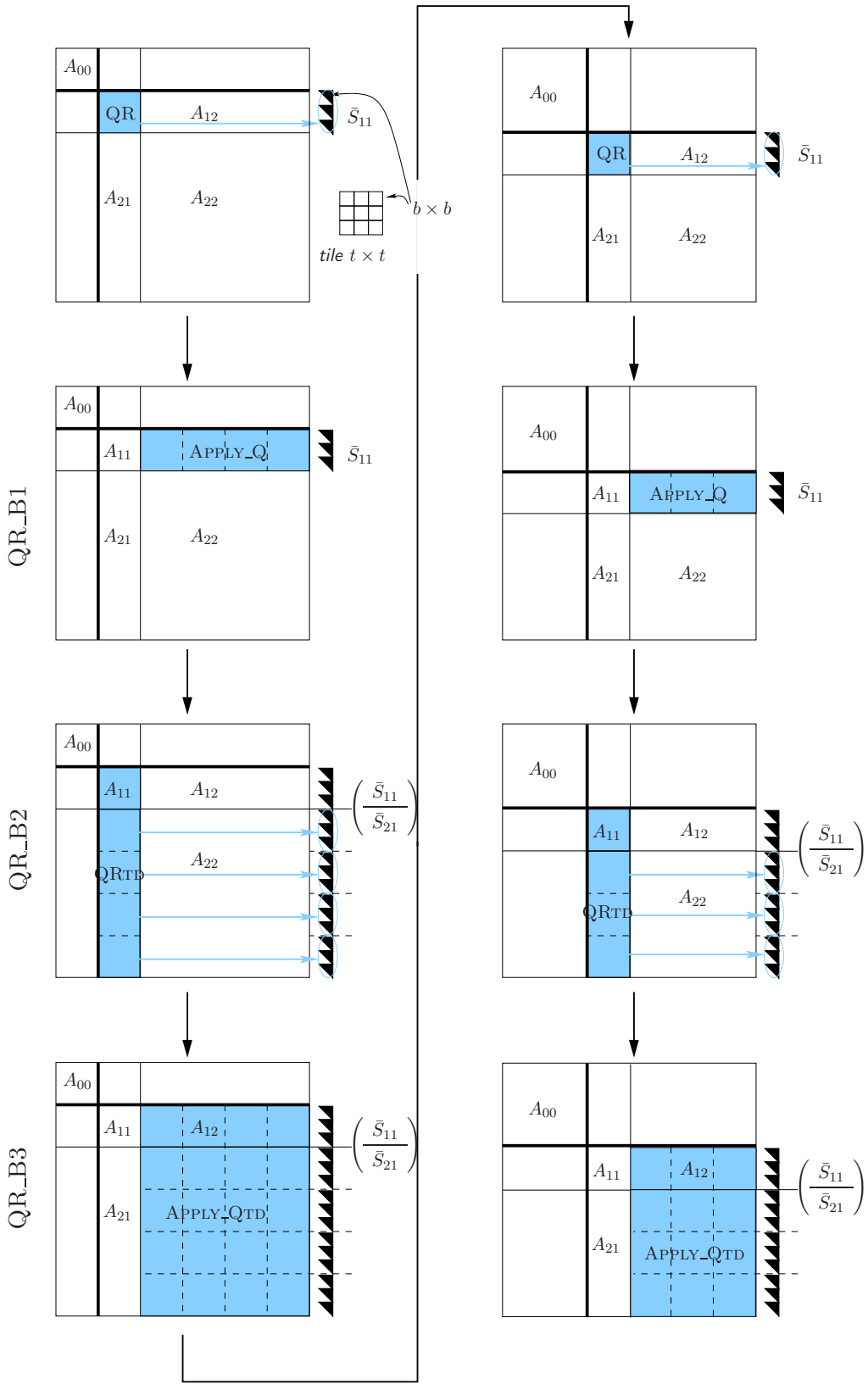


Figura 4.9: Desarrollo de la segunda (izquierda) y tercera (derecha) iteración de la variante 1 orientada a bloques para la factorización QR OOC.

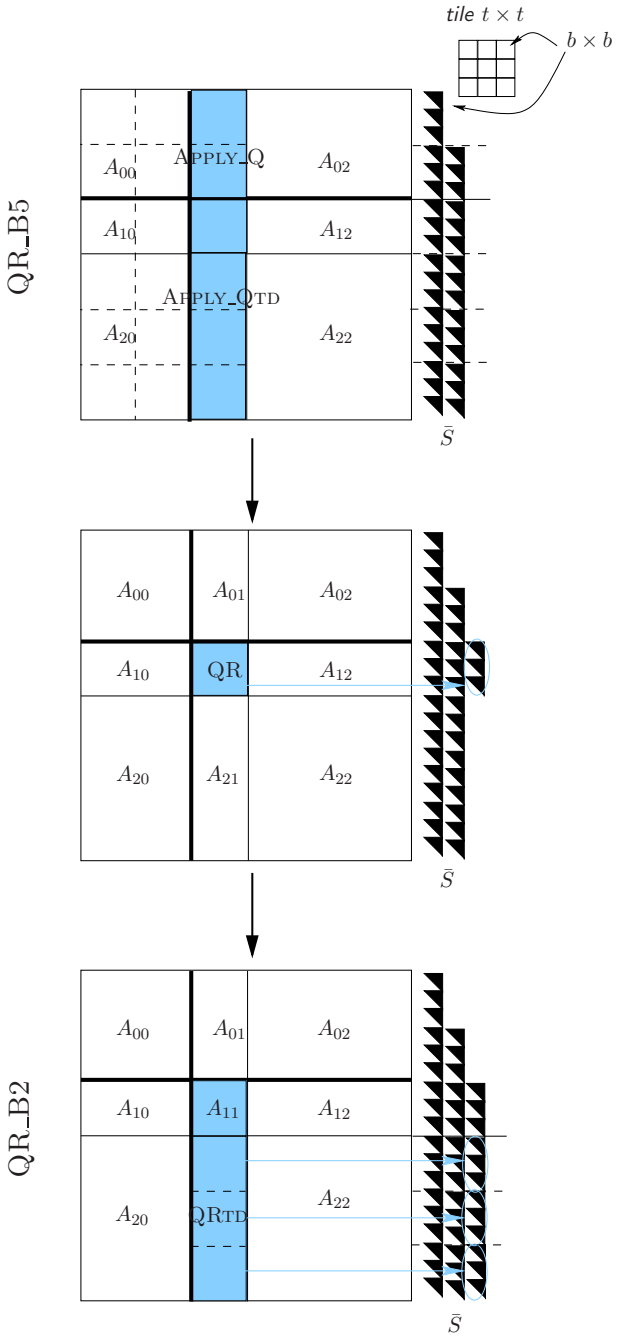


Figura 4.10: Desarrollo de la tercera iteración de la variante 2 orientada a bloques para la factorización QR OOC.

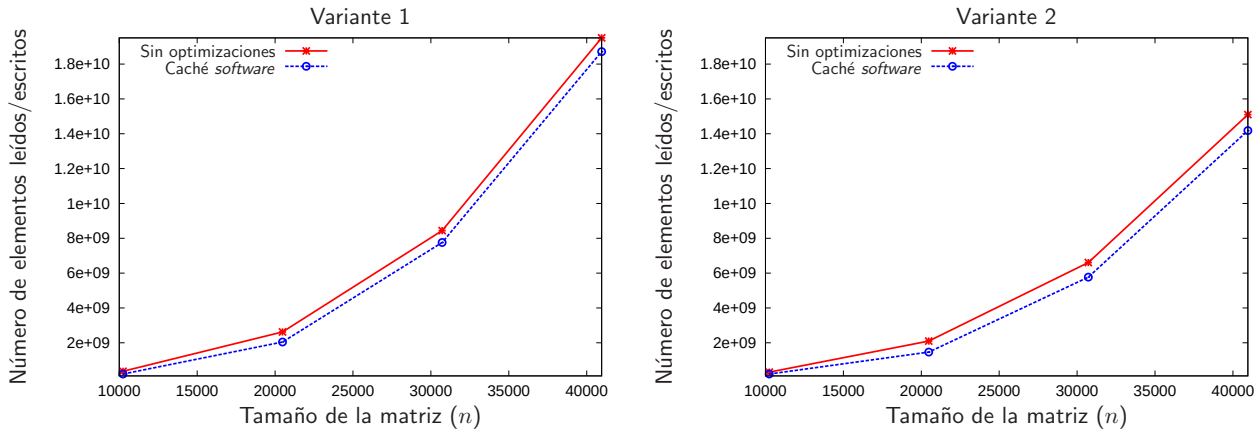


Figura 4.11: Coste teórico de la E/S de las variantes 1 y 2 de la factorización QR OOC, con y sin el uso del sistema que gestiona la memoria disponible como una memoria caché *software*. El tamaño de *tile* es t =5.120.

4.3. Experimentos sobre una arquitectura monoprocesador

En el cálculo de las prestaciones de esta sección y las siguientes, se ha considerado que el coste de la factorización QR de una matriz cuadrada de dimensión $n \times n$ es de $4n^3/3$ *flops*.

La Figura 4.12 recoge los resultados experimentales obtenidos sobre ROPE (en la Tabla 1.1 se muestran las características de ésta y otras máquinas usadas en la evaluación experimental de este capítulo) con las distintas implementaciones OOC desarrolladas para la factorización QR. A diferencia de nuestras predicciones, observamos que las prestaciones en GFLOPS son prácticamente las mismas en ambas variantes, lo que nos lleva a concluir que estamos ocultado la latencia en gran medida. Resulta interesante observar que las prestaciones se mantienen cuando aumenta el tamaño de la matriz, siendo de 9,2 GFLOPS para las matrices más grandes, lo que supone alrededor del 72 % de la velocidad pico de la máquina (12,8 GFLOPS). Además, este rendimiento se encuentra muy próximo a las prestaciones de la implementación *in-core* de la factorización QR que proporciona la biblioteca MKL (rutina GEQRF), que son de 10,8 GFLOPS (las implementaciones OOC alcanzan el 90 % de esta cifra).

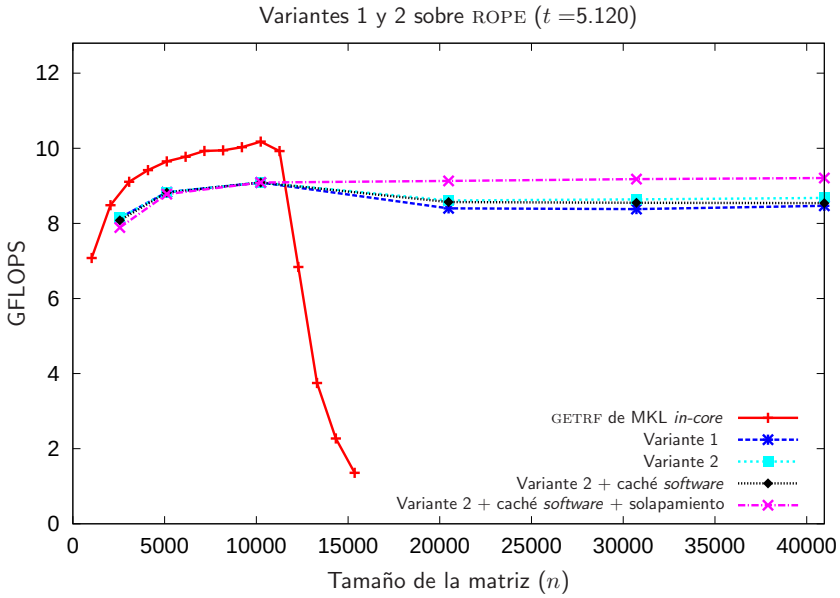


Figura 4.12: Ejecución sobre ROPE de distintas implementaciones OOC de las dos variantes de la factorización QR.

4.4. Experimentos sobre una arquitectura multihebra

Cuando trabajamos con arquitecturas multihebra tenemos dos alternativas diferentes para explotar el paralelismo, análogas a las ya presentadas para la factorización LU con pivotamiento incremental. La diferencia entre ambas alternativas reside en el nivel en el que conseguimos el paralelismo: a nivel de las operaciones de BLAS o bien a un nivel superior, que permite realizar operaciones de grano más grueso y evitar puntos de sincronización que se dan trabajando en

niveles inferiores. En los dos apartados siguientes realizaremos una breve reseña de estas dos aproximaciones, que ya se comentaron más extensamente en los Apartados 3.6.1 y 3.6.2 del capítulo anterior.

4.4.1. Paralelización estándar con BLAS

Un modo habitual de obtener el paralelismo en arquitecturas multihebra es utilizando las implementaciones paralelas de las rutinas de LAPACK y de BLAS que aparecen en los módulos básicos QR y APPLY_Q cuando operan sobre los *tiles* en memoria. Este tipo de paralelización, que tan solo requiere el uso de bibliotecas con implementaciones multihebra, provoca múltiples sincronizaciones entre llamadas a las rutinas de cálculo. Otra opción que hemos utilizado a este mismo nivel es el algoritmo de flujo de datos con planificación dinámica de Quintana Ortí et al. [51].

Otro inconveniente que encontramos en este nivel de paralelización es que, aunque la paralelización de los módulos QRTD y APPLY_QTD también se puede llevar a cabo enlazando con las versiones paralelas de las rutinas de BLAS, no se espera obtener buenas prestaciones, ya que la anchura de los paneles con que se trabaja es pequeña.

4.4.2. Paralelización de nivel superior a BLAS

Puesto que la aplicación de las transformaciones de Householder sólo se realiza desde la izquierda en la variante 2, la actualización de distintos bloques de columnas resulta ser independiente. Esto nos permite realizar una paralelización de nivel superior a BLAS, dividiendo el *tile* a actualizar en tantos bloques de columnas (paneles) como hebras se ejecuten en paralelo, y utilizando una implementación secuencial de BLAS para llevar a cabo el trabajo asociado a cada hebra.

4.4.3. Resultados

La Figura 4.13 muestra las prestaciones en GFLOPS, obtenidas sobre TESLA, de distintas implementaciones de las variantes 1 y 2 de la factorización QR OOC.

En la figura se observa que la factorización QR *in-core* realizada con la rutina de MKL GEQRF consigue 74 GFLOPS. Nuestra mejor implementación, que corresponde a la que hace uso de la caché *software* y el solapamiento, además de la paralelización de nivel superior a BLAS, consigue alrededor de 65 GFLOPS. Teniendo en cuenta que el algoritmo OOC implementado realiza más operaciones que la factorización QR tradicional, podemos afirmar que se está ocultando, en gran medida, la latencia de acceso a disco. Hay que destacar la escalabilidad del algoritmo, ya que al aumentar el tamaño del problema, las prestaciones se mantienen.

La Figura 4.14 muestra las prestaciones en GFLOPS de las mismas implementaciones, esta vez sobre TESLA2. Al igual que sucedía en la factorización LU, la mejora que aporta el solapamiento es menor en esta plataforma ya que se dispone de su sistema de almacenamiento más rápido.

4.5. Resumen y conclusiones

En este capítulo se ha mostrado cómo hemos aplicado las técnicas OOC aportadas en esta tesis para el tercer tipo de factorización de matrices que restaba tratar, la factorización QR. La técnica utilizada en este ocasión, al igual que en la factorización LU, es una técnica de diseño de algoritmos OOC de transformación del flujo de datos. Los algoritmos obtenidos, aunque realizan

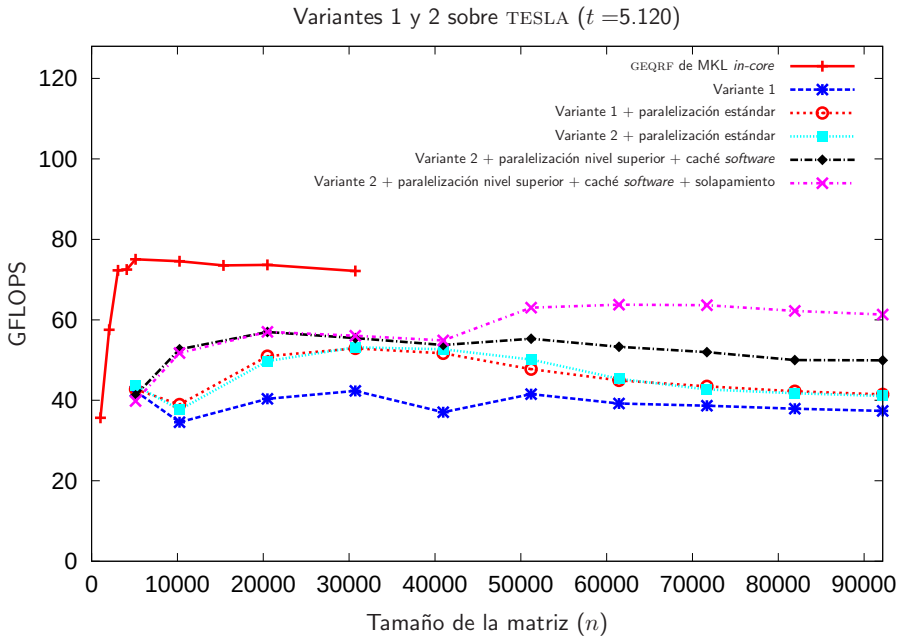


Figura 4.13: Ejecución sobre TESLA de distintas implementaciones OOC de las dos variantes de la factorización QR.

más operaciones que el algoritmo tradicional, son más adecuados para OOC, ya que evitan trabajar sobre paneles de la matriz. Con los nuevos algoritmos, los cálculos sobre paneles se reducen a cálculos sobre matrices de 2×1 *tiles*, lo que permite trabajar OOC ocultando la latencia.

Del mismo modo que se ha hecho en capítulos anteriores, las técnicas se han ido aplicando progresivamente para apreciar, de este modo, el impacto de las mejoras que cada conlleva. Estas técnicas son: el uso de la memoria como una caché *software*, la paralelización de nivel superior a BLAS y el solapamiento de la E/S y los cálculos.

Las implementaciones realizadas se han evaluado en dos arquitecturas: ROPE, un sistema monoprocesador, y TESLA y TESLA2 arquitecturas multihebra con ocho núcleos (TESLA dispone de un solo disco, mientras que TESLA2 dispone de un RAID con seis discos más rápidos). Los resultados obtenidos muestran que, mediante las técnicas utilizadas, es posible ocultar la latencia en el acceso al disco, obteniéndose un rendimiento muy próximo al que proporcionan las rutinas *in-core* de las bibliotecas más populares de álgebra lineal. La evaluación no se ha llevado a cabo sobre una arquitectura con una GPU al no disponer de las rutinas de cálculo necesarias para la implementación de los módulos básicos QRTD y APPLY_QTD

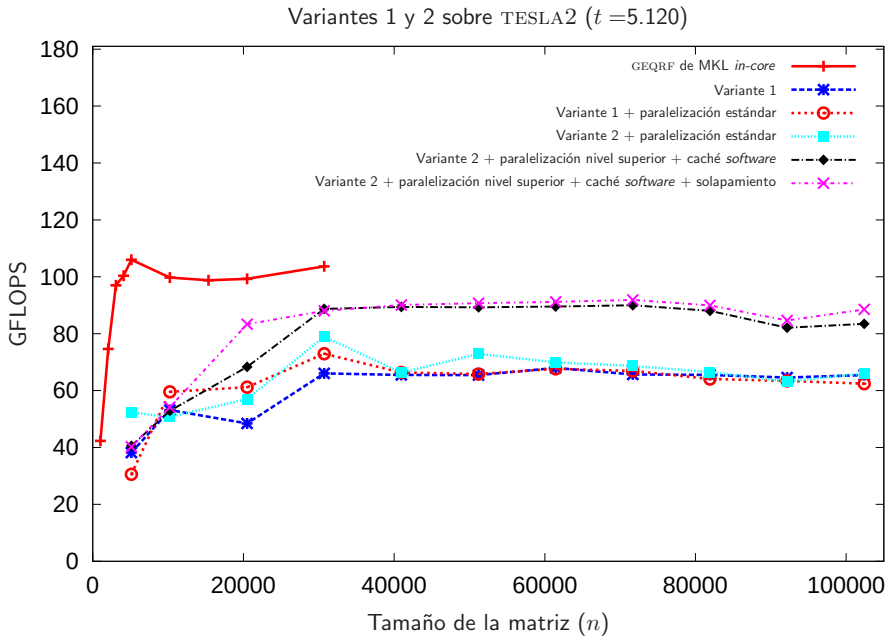


Figura 4.14: Ejecución sobre TESLA2 de distintas implementaciones OOC de las dos variantes de la factorización QR.

Capítulo 5

Conclusiones

5.1. Conclusiones y aportaciones de la tesis

Como se expuso en el Capítulo 1, diversos trabajos en el campo de la modelización de aplicaciones científicas, tecnológicas e industriales requieren la resolución de sistemas de ecuaciones lineales y problemas lineales de mínimos cuadrados densos de gran dimensión.

Una de estas aplicaciones es el estudio de la dispersión acústica, que analiza la presión que ejerce una onda acústica incidente sobre un objeto. En este tipo de problemas, resulta particularmente eficiente el método de elementos de contorno (*Boundary Element Method*) [30, 45], que permite formulaciones numéricas muy precisas. Sin embargo, este método conlleva un elevado coste computacional ya que el sistema de ecuaciones lineales que genera es denso, no simétrico y de muy gran escala. Por ejemplo, cuando el objeto que se analiza corresponde a un modelo geométrico de un avión de la serie Airbus 300, para obtener la distribución de la presión acústica a una frecuencia de 500 Hz, se genera una malla de 253.836 triángulos (cada triángulo corresponde a una incógnita del problema). Si se aumenta la frecuencia a 1 KHz, la malla consta de 1.009.391 triángulos.

Otra aplicación la encontramos en la creación de un modelo de un campo gravitatorio a partir de un sistema de ecuaciones lineales [34]. La parte más importante de este proceso se encuentra en una estimación por mínimos cuadrados para ajustar el modelo a un conjunto de observaciones en las que existen errores. El proceso que se sigue estima los valores de los parámetros del modelo de modo que el error entre las observaciones realizadas y las calculadas se minimice. Los datos provienen de la monitorización de la posición absoluta y relativa de dos satélites, en cuya trayectoria influye el campo gravitatorio de la tierra, por lo que el volumen de información es elevado. Concretamente, en un día se realizan unas 66.000 observaciones parciales, que ocupan en disco 5 GBytes, por lo que un modelo anual de un campo gravitatorio conlleva una estimación por mínimos cuadrados de cientos de miles de parámetros a partir de un volumen de datos del orden de TBytes.

Ante tales necesidades, el objetivo planteado en esta tesis ha sido diseñar, desarrollar y evaluar una colección de rutinas altamente eficientes para resolver sistemas de ecuaciones lineales y problemas lineales de mínimos cuadrados de dimensión elevada (matrices con decenas de miles de filas/columnas) sobre arquitecturas actuales, haciendo uso de técnicas OOC.

Las técnicas OOC extienden la jerarquía de memoria para abarcar el nivel del almacenamiento secundario, haciendo posible la resolución de sistemas de ecuaciones lineales densos de gran dimensión en plataformas con una memoria principal de tamaño reducido. En esta tesis se explotan, además, las características de los procesadores actuales como las arquitecturas multihebra y los

procesadores gráficos. Los crecientes avances en este tipo de arquitecturas ofrecen unas halagüeñas previsiones de futuro, siendo posible pensar en utilizar equipos de sobremesa para la resolución de problemas de muy elevada dimensión en unos pocos años.

Una de las estrategias que se aplican habitualmente para resolver sistemas de ecuaciones lineales, cuando la matriz de coeficientes es densa, comienza con la factorización de esta matriz, descomponiéndola en el producto de dos matrices que se utilizan más tarde para calcular la solución del sistema inicial. Dependiendo de la estructura y propiedades de la matriz de coeficientes, disponemos de diferentes herramientas para obtener su factorización. Así, la factorización de Cholesky se utiliza cuando la matriz de coeficientes es simétrica y definida positiva, la factorización LU se utiliza cuando no presenta una estructura particular, y la factorización QR cuando es rectangular, con más filas que columnas (es decir, se trata de un sistema de ecuaciones lineales sobredeterminado).

El diseño y la implementación de algoritmos OOC para calcular las tres factorizaciones citadas se han tratado en los tres capítulos anteriores. En cada uno de ellos se han ofrecido conclusiones específicas, a partir del análisis del correspondiente estudio experimental. A continuación, se ofrecen las conclusiones generales del trabajo desarrollado, destacando cuáles han sido las aportaciones originales de la tesis.

1. Una propiedad importante de los algoritmos de factorización de matrices densas es que presentan una alta reutilización de datos. En consecuencia, podemos considerar la RAM como una memoria caché de los datos en disco y utilizar los algoritmos de factorización por bloques habituales para ocultar la latencia del disco.

   La experimentación realizada nos permite concluir que no es necesario el diseño de algoritmos específicos para gestionar el trasiego de datos entre RAM y disco, aunque la variante algorítmica óptima no tiene porqué coincidir con la que mejores resultados obtiene en la ordenación de las transferencias de datos entre procesador y caché.

2. Se ha implementado un *run-time* para realizar la gestión de la RAM como una caché *software*. El sobrecoste que supone manejar esta caché queda amortizado por el número de cálculos que se realizan a nivel de *tile* (unidad de transferencia de datos entre disco y memoria principal).

3. Una segunda propiedad de los algoritmos de factorización de matrices densas es que son códigos iterativos, compuestos por bucles donde, una vez escogido el tamaño del bloque, quedan fijadas automáticamente todas las operaciones que habrá que ejecutar para calcular la factorización. Esta característica nos permite desenrollar completamente los bucles del código, generando una lista de tareas que contiene toda la información sobre los datos que serán necesarios en cada ejecución del algoritmo (qué y cuándo). Gracias a ello, es posible implementar un mecanismo de *prefetch* perfecto, integrado en el *run-time*, de modo que resulta transparente a la biblioteca de rutinas de factorización OOC. De este modo, el *run-time* proporciona un mecanismo de E/S asíncrona transparente a la biblioteca.

4. En cuanto a la programabilidad, entendida como la facilidad de adaptación de otro *software* para hacer uso de nuestras técnicas OOC, hay que destacar que, con la solución propuesta en esta tesis, el programador de rutinas de álgebra lineal densa sólo tiene que preocuparse de la especificación de los cálculos a realizar, en tanto que toda la E/S es gestionada por el *run-time* de forma transparente.

   Cuando las rutinas se programan con `libflame`, la portabilidad es completa, no siendo necesario ningún cambio en el código para trabajar sobre disco (en modo OOC).

5. Se ha demostrado la validez de la solución propuesta mediante la implementación de tres factorizaciones de matrices densas. Los códigos desarrollados muestran su eficiencia gracias a una extensa evaluación realizada sobre varias plataformas representativas de la tecnología actual en computación.

En resumen, se ha diseñado e implementado un *run-time* que permite la resolución de problemas de gran dimensión utilizando técnicas OOC, a partir de los algoritmos tradicionales, obteniendo prestaciones similares, y en algunos casos superiores, a las de las implementaciones *in-core* existentes para los problemas de álgebra lineal con los que se ha trabajado en esta tesis.

Llegados a este punto, hacemos una reflexión sobre qué colectivos y qué tipo de aplicaciones pueden beneficiarse de los resultados de esta tesis:

- Científicos e ingenieros no acostumbrados a interactuar con grandes *clusters*, o que no tienen acceso a los mismos, cuyas aplicaciones no presentan fuertes restricciones en los tiempos de respuesta. Aunque hay bibliotecas como PLAPACK y ScaLAPACK, que permiten resolver problemas de las dimensiones tratadas en esta tesis haciendo uso de un *cluster* —posiblemente en menor tiempo—, para un investigador que no tiene profundos conocimientos de arquitecturas/programación distribuidas, interactuar con estas bibliotecas no es nada trivial, especialmente a la hora de introducir los datos y recoger los resultados.

- Sistemas en los que existen limitaciones en el tamaño de la memoria RAM disponible. Este puede ser el caso de los sistemas empotrados, en los que los criterios económicos tienen un fuerte peso. La aplicación de los resultados de esta tesis puede ayudar a resolver problemas de dimensión mayor de la que es capaz de almacenar la RAM del sistema, cumpliendo con las restricciones económicas.

- Relacionado con el caso anterior, los resultados de la tesis pueden ser igualmente válidos en sistemas de propósito general, pues demuestran que el uso del disco como dispositivo de almacenamiento no tiene porqué penalizar el tiempo de respuesta de ciertos cálculos. Esto puede suponer que es posible hacer uso de un equipamiento con una cantidad menor de RAM, reduciéndose en consecuencia los costes económicos. En una gran instalación de cálculo, como el *cluster* de un centro de proceso de datos, por ejemplo, cualquier reducción en el coste por nodo se multiplica linealmente con el número de nodos, pudiendo ser importante.

- Entornos en los que no hay posibilidad de acceder a un *cluster* de computadores. Esta situación se da a bordo de los vehículos espaciales, que habitualmente están dotados de tecnología estándar multinúcleo para la resolución de problemas computacionales surgidos de experimentos realizados en el espacio. En este tipo de entornos, realizar los cálculos a bordo del propio vehículo espacial puede ser más eficiente que descargar un gran volumen de datos a la tierra para realizar los cálculos remotamente.

5.2. Publicaciones

Los resultados del trabajo desarrollado en esta tesis han generado las siguientes publicaciones:

1. "Solving 'large' dense matrix problems on multi-core processors", Marqués, M., Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R.A. *10th IEEE International Workshop*

on Parallel and Distributed Scientific and Engineering Computing, PDSEC, Roma (Italia), 2009, pp. 1–8. IEEE Computer Society, Washington, DC. ISBN: 978-1-4244-3750-4.

2. "Using graphics processors to accelerate the solution of out-of-core linear systems", Marqués, M., Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R.A. *Proceedings of the 8th International Symposium on Parallel and Distributed Computing, ISPDC, Lisboa (Portugal), 2009, pp. 169–176.* IEEE Computer Society, Washington, DC. ISBN: 978-0-7695-3680-4.

3. "Out-of-core computation of the QR factorization on multi-core processors", Marqués, M., Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R.A. *Lecture Notes In Computer Science Vol. 5704, Proceedings of the 15th international Euro-Par Conference on Parallel Processing, Delft (Holanda), 2009, pp. 809–820.* H. Sips, D. Epema, and H. Lin, (Eds.) Springer-Verlag, Berlin, Heidelberg. ISBN: 978-3-642-03868-9.

4. "Solving out-of-core linear systems on desktop computers", Marqués, M., Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R.A. *Proceedings of the 9th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE, Gijón (España), 2009, Vol. II, pp. 660–664.* ISBN: 978-84-612-9727-6.

5. "Solución de sistemas de ecuaciones lineales densos de gran dimensión sobre computadores personales", Marqués, M., Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R.A. *Actas del Congreso de Métodos Numéricos en Ingeniería 2009, METNUM, Barcelona (España), 2009, página 342.* Sociedad Española de Métodos Numéricos en Ingeniería. ISBN: 978-84-96736-66-5.

6. "Using desktop computers to solve large-scale dense linear algebra problems", Marqués, M., Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R.A. *Journal of Supercomputing, 2009 (aceptado y pendiente de publicación).* ISSN: 0920-8542.

7. "Out-of-core solution of linear systems on graphics processors", Castillo, M., Igual, F.D., Marqués, M., Mayo R., Quintana, E.S., Quintana, G., Rubio R., van de Geijn, R.A. *International Journal of Parallel, Emergent and Distributed Systems, Vol. 24(6), 2009, pp. 521–538.* ISSN: 1744-5760.

8. "Specialized spectral division algorithms for generalized eigenproblems via the inverse-free iteration", Marqués, M., Quintana-Ortí, E.S., Quintana-Ortí, G. *Lecture Notes in Computer Science Vol. 4699, PARA'06 Applied Computing: State-of-the-Art in Scientific Computing, Umeå (Suecia). 2007, pp. 157–166.* B. Kågström, E. Elmroth, J. Dongarra, J. Waśniesky (Eds.). Springer-Verlag, Berlin, Heidelberg. ISBN: 978-3-540-75754-2.

9. "A Run-Time System for Programming Out-of-Core Matrix Algorithms-by-Tiles on Multithreaded Architectures", Quintana-Ortí, G., Igual, F.D., Marqués M., Quintana-Ortí, E.S., van de Geijn, R.A. *FLAME Working Note #43. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-10-10.* March 31, 2010. Este trabajo se encuentra en proceso de revisión en *ACM Transactions on Mathematical Software.*

5.3. Líneas abiertas de investigación

La tesis cubre con sus objetivos la resolución eficiente de sistemas de ecuaciones lineales y problemas lineales de mínimos cuadrados densos de gran dimensión sobre procesadores actuales y arquitecturas multihebra. En esta misma línea de trabajo pueden identificarse los siguientes problemas no resueltos hasta la fecha, que constituyen líneas abiertas de investigación:

1. A fin de demostrar la utilidad de las aportaciones del trabajo desarrollado, queda abierta la tarea de detectar un mayor número de aplicaciones en las que aparezcan problemas de computación matricial densos de gran dimensión. Se dispone actualmente de un conjunto de aplicaciones de este tipo, que dan lugar a sistemas de ecuaciones lineales densos con estructura general o simétrica definida positiva. La intención es conocer si existen otras aplicaciones que den lugar a problemas de este tipo y en los que la biblioteca podría aportar una nueva vía de solución.

2. También queda abierta la aplicación de las técnicas OOC desarrolladas en esta tesis, como el uso de una caché *software* o el mecanismo de E/S asíncrona, a la resolución de otras operaciones de álgebra lineal como el cálculo de valores propios y el cálculo de valores singulares.

3. La colección de rutinas desarrollada en esta tesis hace posible una nueva alternativa de solución de sistemas de ecuaciones lineales y problemas de mínimos cuadrados de dimensión muy elevada (millones de filas/columnas), basada en *clusters* de computadores equipados con aceleradores *hardware* y que hagan uso de técnicas OOC. Para implementar esta solución se propone abordar la combinación de las rutinas desarrolladas con la biblioteca de paso de mensajes PLAPACK.

4. Otra alternativa para la resolución de problemas de álgebra lineal densa de dimensión muy elevada en *clusters* de computadores se puede apoyar en la integración de la biblioteca desarrollada con sistemas de ficheros paralelos.

Apéndice A

Interfaz de Programación para el Manejo de Matrices OOC

Para manipular las matrices OOC se propone una formulación orientada a objetos similar a la utilizada en PLAPACK y en FLAME [71, 75]. La API está formada por una serie de rutinas que se describen a continuación.

Para crear una matriz OOC en disco se utiliza la siguiente rutina:

```
FLAOOC_Obj_create( FLA_Matrixtype matrixtype, FLA_Datatype datatype,
                   dim_t m, dim_t n, dim_t mt, dim_t nt,
                   char *file_name, FLA_Obj *Aooc );
```

Propósito: Crea un nuevo objeto que describe una matriz `m` × `n` denominada `Aooc`, con elementos de tipo `datatype` (`FLA_INT`, `FLA_REAL`, `FLA_DOUBLE`, etc.), y reserva el espacio asociado a la matriz en disco. El tipo de matriz `matrixtype` permite ahorrar la mitad del espacio cuando la matriz es triangular o simétrica, escogiendo `FLA_LOWER_TRIANGULAR` o `FLA_UPPER_TRIANGULAR`; para matrices densas se debe escoger `FLA_DENSE`. La matriz se particiona en *tiles* de tamaño `mt`×`nt`, almacenando los elementos de cada *tile* contiguos en disco y por columnas. Cada *tile* se almacena en un fichero cuyo nombre es `file_name` seguido por el número del *tile* (el número del *tile* (i, j) viene dado por la expresión $j * m/mt + i$).

El tamaño de los *tiles*, dado por `mt` y `nt`, debe afinarse para optimizar las prestaciones, en función de las dimensiones del problema, el tamaño de la memoria RAM y el número de *tiles* que se pueden mantener en ella durante la ejecución del programa.

Una vez creada la matriz OOC se pueden escribir los datos (transferirlos de memoria a disco) mediante la siguiente rutina:

```
FLAOOC_Copy_submatrix_to_global( FLA_Trans trans, dim_t m, dim_t n,
                                 void *X, dim_t ldim,
                                 dim_t i, dim_t j, FLA_Obj Aooc );
```

Propósito: Copia el contenido de una matriz convencional `X` almacenada por columnas y con dimensión principal `ldim`, en la submatriz de tamaño `m`×`n` que comienza en la entrada (`i`,`j`) de `Aooc`. El parámetro `trans` se puede utilizar para trasponer la matriz durante la copia.

Suponiendo que `FLA_Trans` es `FLA_NO_TRANSPOSE` y dada la matriz `X` de dimensión `m`×`n`, una llamada a la rutina anterior equivale a la siguiente asignación en notación de MATLAB:

```
Aooc( i:i+m-1, j:j+n-1 ) = X;
```

donde `Aooc` es un objeto OOC.

Para transferir los datos en sentido opuesto, se debe utilizar la siguiente rutina:

```
FLAOOC_Copy_global_to_submatrix( FLA_Trans trans, dim_t i, dim_t j, FLA_Obj Aooc,
                                 dim_t m, dim_t n, void *X, dim_t ldim );
```

Propósito: Copia el contenido de la submatriz de tamaño `m`×`n` de `Aooc`, cuyo elemento de la esquina superior izquierda es la entrada (`i`,`j`), en una matriz convencional `X` almacenada por columnas y con dimensión principal `ldim`. El parámetro `trans` se utiliza para trasponer la matriz durante la copia.

Los algoritmos OOC son algoritmos orientados a *tiles*: las operaciones básicas se realizan sobre *tiles*. Cuando se lanza una operación sobre un *tile*, éste debe cargarse en un objeto *in-core* (leyéndolo del disco) para trabajar sobre él. El objeto *in-core* debe crearse en primer lugar mediante la rutina:

```
FLAOOC_create_conf_to_tile( FLA_Obj Tile, FLA_Obj *Ainc )
```

Propósito: Crea un objeto *in-core* `Ainc` del mismo tamaño que el *tile* `Tile`.

Una vez creado el objeto que alojará al *tile*, se carga éste desde el disco mediante la siguiente rutina:

```
FLAOOC_OOC_to_INC( FLA_Obj Tile, FLA_Obj Ainc );
```

Propósito: Copia los datos del *tile* `Tile` en el objeto *in-core* `Ainc`.

Para devolver a disco un objeto *in-core* `Ainc` que corresponde a un *tile* `Tile`, se debe utilizar la rutina:

```
FLAOOC_INC_to_OOC( FLA_Obj Ainc, FLA_Obj Tile );
```

Propósito: Copia a disco los datos del objeto *in-core* `Ainc` correspondientes al *tile* `Tile`.

Para liberar el espacio en disco asociado a una matriz OOC se debe llamar a la siguiente rutina:

```
FLAOOC_Obj_free( FLA_Obj *Aooc );
```

Propósito: Libera todos los recursos utilizados para almacenar los datos asociados a la matriz `Aooc` en disco.

Bibliografía

[1] Advanced Micro Devices Inc. *ACML.* <http://www.amd.com/acml>.

[2] Agarwal, V., Hrishikesh, M. S., Kecler, S. W., and Burger, D. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *27th Annual Int. Symp. on Computer Architectures* (2005), pp. 248–259.

[3] Agullo, E., Guermouche, A., and L'Excellent, J. A preliminary out-of-core extension of a parallel multifrontal solver. In *EuroPar'06 Parallel Processing* (2006), vol. 4128 of *Lecture Notes in Computer Science*, pp. 1053–1063.

[4] Andersen, B. S., Gunnels, J. A., Gustavson, F. G., Reid, J. K., and Waśniewski, J. A fully portable high performance minimal storage hybrid format Cholesky algorithm. *ACM Trans. Math. Softw. 31*, 2 (2005), 201–227.

[5] Andersen, B. S., Waśniewski, J., and Gustavson, F. G. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Trans. Math. Softw. 27*, 2 (2001), 214–244.

[6] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J. J., Croz, J. D., Hammarling, S., Greenbaum, A., McKenney, A., and Sorensen, D. *LAPACK Users' guide (third ed.).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[7] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, University of California at Berkeley, Electrical Engineering and Computer Sciences, 2006.

[8] Baboulin, M. *Solving large dense linear least squares problems on parallel distributed computers. Application to the Earth's gravity field computation.* Ph.D. dissertation, INPT, March 2006. TH/PA/06/22.

[9] Barron, D. W., and Swinnerton-Dyer, H. P. F. Solution of simultaneous linear equations using a magnetic-tape store. *Computer Journal 3*, 1 (Apr. 1960), 28–33.

[10] Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Softw. 31*, 1 (2005), 1–26.

[11] Bientinesi, P., Gunter, B. C., and van de Geijn, R. A. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw. 35*, 1 (2008), 1–22.

[12] Bischof, C., and van Loan, C. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput. 8*, 1 (1987), 2–13.

[13] Borkar, S. Y., Dubey, P., Kahn, K. C., Kuck, D. J., Mulder, H., Pawlowski, S. S., and Rattner, J. R. Platform 2015: Intel processor and platform for the next decade. Intel white paper, R. M. Ramanathan and Vince Thomas, Intel Corporation, 2005.

[14] Brewer, O., Dongarra, J. J., and Sorensen, D. Tools to aid in the analysis of memory access patterns of FORTRAN programs. *Parallel Computing 9*, 1 (1988), 25–35.

[15] Béreux, N. Out-of-core implementations of Cholesky factorization: loop-based versus recursive algorithms. Technical Report #602, Centre de Mathématiques Appliquées, Ecole Polytechnique, 2006.

[16] Chan, E., Quintana-Ortí, E. S., Quintana-Ortí, G., and van de Geijn, R. A. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007* (2007), P. B. Gibbons and C. Scheideler, Eds., ACM, pp. 116–125.

[17] Croz, J. D., Mayes, P., and Radicati, G. Factorization of Band Matrices Using Level 3 BLAS. In *Joint International Conference on Vector and Parallel Processing* (1990), no. 457 in Lecture Notes in Computer Science, Springer-Verlag, London, UK, pp. 222–231.

[18] Croz, J. J. D., Nugent, S. M., Reid, J. K., and Taylor, D. B. Solving large full sets of linear equations in a paged virtual store. *ACM Trans. Math. Softw. 7*, 4 (1981), 527–536.

[19] Demmel, J. W., Dongarra, J. J., Croz, J. D., Greenbaum, A., Hammarling, S., and Sorensen, D. Prospectus for the developement of a linear algebra library for high-performance computers. LAPACK Working Note #1 ANL/MCS-TM-07, Argonne National Laboratory, Argonne, IL, USA, 1987.

[20] Dongarra, J., and D'Azevedo, E. F. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. Tech. rep., Knoxville, TN, USA, 1997.

[21] Dongarra, J. J. *Oral history.* Philadelphia, PA, USA.

[22] Dongarra, J. J., Bunch, J., Moler, C., and Stewart, G. *LINPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1979.

[23] Dongarra, J. J., Croz, J. D., Hammarling, S., and Duff, I. S. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1 (1990), 1–17.

[24] Dongarra, J. J., Croz, J. D., Hammarling, S., and Hanson, R. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft. 14* (1988), 1–17.

[25] Dongarra, J. J., Croz, J. D., Hammarling, S., and Hanson, R. J. Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw. 14*, 1 (1988), 18–32.

[26] Dongarra, J. J., Cruz, J. D., Hammarling, S., and Duff, I. S. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw. 16*, 1 (1990), 18–28.

[27] Dongarra, J. J., and Walker, D. The design of linear algebra libraries for high performance computers. LAPACK Working Note #58 UT-CS-93-188, University of Tennessee, Knoxville, TN, USA, 1993.

[28] Dowd, K., and Severance, C. *High performance computing*, second ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[29] Elmroth, E., Gustavson, F., Jonsson, I., and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIREV: SIAM Review 46*, 1 (2004), 3–45.

[30] Geng, P., Oden, J., and van de Geijn, R. Massively parallel computation for acoustical scattering problems using boundary element methods. *Journal of Sound and Vibration 191*, 1 (1996), 145–165.

[31] Gilbert, J., and Toledo, S. High-performance out-of-core sparse LU factorization, 1999.

[32] Golub, G. H., and Loan, C. F. V. *Matrix computations*, third ed. John Hopkins, Baltimore MD, 1996.

[33] Gunter, B., and van de Geijn, R. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software 31*, 1 (Mar. 2005), 60–78.

[34] Gunter, B. C. *Computational methods and processing strategies for estimating Earth's gravity field*. PhD thesis, The University of Texas at Austin, 2004.

[35] Gunter, B. C., Reiley, W. C., and van de Geijn, R. A. Parallel out-of-core Cholesky and QR factorization with POOCLAPACK. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium* (Washington, DC, USA, 2001), IEEE Computer Society, p. 179.

[36] Higham, N. J. *Accuracy and Stability of Numerical Algorithms*, second ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[37] IBM. *ESSL*. <`http://www-03.ibm.com/systems/p/software/essl.html`>.

[38] Intel Corporation. *MKL*. <`http://www.intel.com`>.

[39] Joffrain, T., Quintana-Ortí, E. S., and van de Geijn, R. A. Rapid development of high-performance out-of-core solvers. In *PARA* (2004), J. Dongarra, K. Madsen, and J. Wasniewski, Eds., vol. 3732 of *Lecture Notes in Computer Science*, Springer, pp. 413–422.

[40] Kumar, R., Tullsen, D. M., and Jouppi, N. P. Core architecture optimization for heterogeneous chip multiprocessors. In *15th international conference on Parallel architectures and compilation technique – PACT'06* (2006), pp. 23–32.

[41] Kumar, R., Tullsen, D. M., Jouppi, N. P., and Ranganathan, P. Heterogeneous chip multiprocessors. *IEEE Computer 38*, 11 (2005), 32–38.

[42] Kumar, R., Zyuban, V., and Tullsen, D. M. Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling. In *32nd Int. Symp. on Computer Architecture, ISCA'05* (2005), pp. 408–419.

[43] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw. 5*, 3 (1979), 308–323.

[44] Low, T. M., and van de Geijn, R. A. An API for manipulating matrices stored by blocks. FLAME Working Note #12 TR-2004-15, The University of Texas at Austin, Department of Computer Sciences, May 2004.

[45] López-Fernández, J., Portugués, M., Taboada, J., Rice, H., and Obelleiro, F. Hp-fass: A hybrid parallel fast acoustic scattering solver. In *Proceedings of the Int. Conf. on Computational and Mathematical Methods in Science and Engineering* (2009), vol. 2, pp. 622–632.

[46] National Research Council of the National Academies. *Getting up to speed. The future of supercomputing*, first ed. The National Academies Press, Washington, DC, 2005.

[47] Netlib.org. *BLAS*. <`http://www.netlib.org/blas`>.

[48] Netlib.org. *LAPACK*. <`http://www.netlib.org/lapack`>.

[49] Netlib.org. *LINPACK*. <`http://www.netlib.org/linpack`>.

[50] Netlib.org. *ScaLAPACK*. <`http://www.netlib.org/scalapack`>.

[51] Quintana-Ortí, G., Quintana-Ortí, E., Geijn, R. V. D., Zee, F. V., and Chan, E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw. 36*, 3 (2009), 1–26.

[52] Quintana-Ortí, G., Quintana-Ortí, E. S., van de Geijn, R. A., Zee, F. G. V., and Chan, E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw. 36*, 3 (2009), 14:1–14:26.

[53] Reiley, W. C., and van de Geijn, R. A. POOCLAPACK: Parallel out-of-core linear algebra package. Tech. Rep. CS-TR-99-33, University of Texas at Austin, Austin, TX, USA, 1999.

[54] Rothberg, E., and Schreiber, R. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM J. Sci. Comput. 21*, 1 (1999), 129–144.

[55] Rotkin, V., and Toledo, S. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Trans. Math. Softw. 30*, 1 (2004), 19–46.

[56] Schafer, N., Serban, R., and Negrut, D. Implicit integration in molecular dynamics simulation. In *ASME International Mechanical Engineering Congress & Exposition* (2008). (IMECE2008-66438).

[57] Schreiber, R., and van Loan, C. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput. 10*, 1 (1989), 53–57.

[58] Smith, B. T., Boyle, J. M., Dongarra, J. J., Garbow, B. S., Ikebe, Y., Klema, V. C., and Moler, C. B. *Matrix Eigensystem Routines—EISPACK Guide*, second ed., vol. 6 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1976.

[59] Stewart, G. W. *Matrix Algorithms Volume 1: Basic Decompositions*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[60] SUN microsystems. *Sun Performance Library*. <http://docs.sun.com/app/docs/doc/819-5268>.

[61] Texas Advanced Computing Center. *Goto BLAS*. <http://www.tacc.utexas.edu/resources/software/#blas>.

[62] Toledo, S. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl. 18*, 4 (1997), 1065–1081.

[63] Toledo, S. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, J. Abello and J. S. Vitter, Eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999, pp. 161–180.

[64] Toledo, S., and Gustavson, F. G. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *In Fourth Workshop on Input/Output in Parallel and Distributed Systems* (1996), ACM Press, pp. 28–40.

[65] Trefethen, L., and Schreiber, R. S. Average-case stability of Gaussian elimination. *SIAM J. Matrix Anal. Appl. 11*, 3 (1990), 335–360.

[66] University of Tennessee. *BLASATLAS*. <http://math-atlas.sourceforge.net>.

[67] University of Texas. *FLAME Project*. <http://z.cs.utexas.edu/wiki/flame.wiki/>.

[68] University of Texas. *libFLAME Project*. <http://www.linearalgebrawiki.org/>.

[69] University of Texas. *PLAPACK*. <http://www.cs.utexas.edu/users/plapack>.

[70] University of Texas. *Spark: FLAME Code Generator*. <http://www.cs.utexas.edu/users/flame/Spark/>.

[71] van de Geijn, R. A., Alpatou, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., and Overfelt, J. *Using PLAPACK: parallel linear algebra package*. MIT Press, Cambridge, MA, USA, 1997.

[72] van de Geijn, R. A., and Quintana-Ortí, E. S. *The Science of Programming Matrix Computations*. <www.lulu.com>, 2008.

[73] Watkins, D. S. *Fundamentals of Matrix Computations*, second ed. John Wiley & Sons, Inc., New York, 2002.

[74] Wilkinson, J. H. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, England, 1965.

[75] Zee, F. G. V. `libflame`. The Complete Reference, 2008. <http://www.lulu.com>.

[76] Zhang, Y., Sarkar, T. K., van de Geijn, R. A., and Taylor, M. C. Parallel MoM using higher order basis function and PLAPACK in-core and out-of-core solvers for challenging EM simulations. In *IEEE AP-S & USNC/URSI Symposium* (2008).