UNIVERSITAT POLITÈCNICA DE VALÈNCIA

MÁSTER EN COMPUTACIÓN PARALELA Y DISTRIBUIDA

TRABAJO DE FIN DE MÁSTER

Aprovechamiento del paralelismo de tareas en factorizaciones de matrices jerárquicas sobre procesadores multinúcleo

Autora:
Rocío Carratalá Sáez

Tutores:
José Ignacio Aliaga Estellés
Pedro Alonso Jordá

11 de julio de 2016
Curso académico 2015/2016

Resumen

Pese a su corta historia, las matrices jerárquicas presentan ventajas sustanciales en el manejo de datos que tradicionalmente se representarían con matrices densas pero que presentan gran cantidad de datos nulos. Con el particionado adecuado y un grado de dispersión suficiente, las matrices jerárquicas logran los beneficios de los algoritmos que se aplican sobre matrices dispersas tanto a nivel de almacenamiento como de cómputo.

El presente documento presenta, inicialmente, las operaciones elementales del álgebra de las matrices jerárquicas, comparándolas con las comunmente conocidas e incluyendo algoritmos recursivos y secuenciales para su cálculo. A continuación, se explica una implementación propia del algoritmo de factorización LU, basada en el trabajo de Kriemann [2].

Dado que las librerías existentes, pese a que incluyen muchas de las operaciones sobre matrices jerárquicas necesarias, no son de código abierto, esta implementación conforma el cuerpo principal del Trabajo Final de Máster. Con ella comenzará mi doctorado, con la intención mejorarla y aplicar técnicas de paralelismo distintas a las que utilizan las librerías actuales, como OmpSs, para intentar lograr un mejor rendimiento.

Abstract

Although hierarchical matrices are still young, they present important advantages when applied to data management that traditionaly has been represented using dense matrices, but containing a big amount of zero values. When data presents a considerable sparse coefficient, using the appropiate partitioning hierarchical matrices show benefits in a similar way to sparse matrices, not only in storage but also performing operations.

The current document starts whith a summary of the basic hierarchical matrices algebra operations, comparing them with the ones that are commonly known and including recursive and also sequential algorithms to perform them. After that, I explain my own LU

factorization implementation, which is based in Kriemann's previous work [2].

Even though existing libraries include most of the operations that need to be performed when working with hierarchical matrices, they are not open source. That is why the main part of mi effort has been made in order to write my own LU factorization. With it, I will start my PhD, pretending to improve it and apply parallelization techniques different from the ones that existing libraries use, such as OmpSs, in order to achieve better performance.

Palabras clave

Matrices jerárquicas, factorización LU, OpenMP, sistemas con memoria compartida, computación de altas prestaciones.

Keywords

Hierarchical matrices, LU factorization, OpenMP, shared memory systems, high performance computing.

Índice general

1. Introducción y motivación **9**

1.1. Estado del arte 9

1.2. Motivación personal 11

1.3. Objetivos 12

1.4. Estructura del trabajo 13

2. Álgebra elemental de las matrices jerárquicas **15**

2.1. Las matrices jerárquicas: conceptos básicos 15

2.2. Almacenamiento de matrices jerárquicas 19

2.3. Operaciones algebraicas elementales con $\mathcal{H}$-matrices 21

2.3.1. Suma de matrices jerárquicas 21

2.3.2. Producto matriz-vector con matrices jerárquicas. 24

2.3.3. Producto matriz-matriz con matrices jerárquicas. 26

3. Factorización LU de matrices jerárquicas **29**

3.1. Factorización LU: conceptos generales 29

3.2. La factorización LU de matrices jerárquicas: dependencia de tareas 32

3.3. Mi implementación de la $\mathcal{H}$-factorización LU 34

3.3.1. Paralelización del algoritmo sobre memoria compartida, mediante OpenMP 37

3.4. Pruebas realizadas y resultados obtenidos 37

4. Conclusiones **49**

5. Trabajo futuro **51**

A. Código implementado **57**

A.1. Factorización LU: versión secuencial 57

A.2. Matrices para las pruebas 75

A.2.1. $matrix_1.txt$ 76

A.2.2. $matrix_2.txt$ 76

A.2.3. $matrix_3.txt$ 78

Índice de figuras

2.1. Árbol jerárquico para el conjunto de índices $\mathcal{I} = \{0, 1, 2, 3, 4, 5, 6, 7\}$. . . 17

2.2. Particionado de una matriz mediante el árbol jerárquico a bloques para el conjunto de índices $\mathcal{I} \times \mathcal{I} = \{0, 1, 2, 3, 4, 5, 6, 7\} \times \{0, 1, 2, 3, 4, 5, 6, 7\}$. 18

2.3. Ejemplo de descomposición en niveles de una matriz jerárquica pequeña. 21

2.4. Esquema de la suma de matrices jerárquicas. 23

2.5. Esquema del produto de una matriz jerárquica por un vector. 25

2.6. Esquema del produto de una matriz jerárquica por un vector. 28

3.1. Detalle de los pasos que conforman el cálculo de la factorización LU sobre una $\mathcal{H}$-matriz ejemplo. En rojo las operaciones LU, en verde las TRSM, en amarillo las actualizaciones y, debajo, en cada paso, las modificaciones sobre las matrices L y U. 36

5.1. A la izquierda, el esquema sobre una matriz jerárquica del proceso de TRSM desde el bloque A^1_{00} (formado por cuatro sub-bloques $A^2_{00}, A^2_{01}, A^2_{10}, A^2_{11}$, cuyas L, U ya han sido calculadas), al bloque A^1_{01} (también formado por cuatro sub-bloques $A^2_{02}, A^2_{03}, A^2_{12}, A^2_{13}$, cuyas U's se van a calcular). A la derecha, por colores, el desglose de operaciones (y su orden) a realizar en cada sub-bloque, en lugar de aplicar el TRSM directamente de la forma $U^1_{01} = (L^1_{00})^{-1} \cdot A^1_{01}$. 52

5.2. A la izquierda, el orden que actualmente sigue la paralelización con OpenMP (siempre y cuando se ejecuten en paralelo las TRSM sobre la fila y la columna a la vez), a la derecha el orden que puede lograrse si se ejecuta en paralelo con OmpSs [13] (se aprecia mayor paralelismo). 53

A.1. Visualización de la matriz matrix_1.txt. 76

A.2. Visualización de la matriz matrix_2.txt. 77

A.3. Visualización de la matriz matrix_3.txt. 78

Capítulo 1

Introducción y motivación

1.1. Estado del arte

Entre los años finales de la década de los noventa del siglo pasado y los primeros del actual, se formalizó la definición de lo que se denominan matrices jerárquicas o $\mathcal{H}$-matrices y se definieron, para estas, algunas operaciones elementales del álgebra de matrices tradicional. Principalmente a cargo de Steffen Börm, Lars Grasedyck y Wolfgang Hackbusch ([4], [3], [5]), nació la idea de "aproximar matrices densas con matrices dispersas", resultantes de la discretización de operadores no locales en métodos integrales sobre contornos, de inversas de operadores diferenciales parciales, o de las soluciones a problemas de control. Es justo esta idea la que motivó la existencia de las matrices jerárquicas, en torno a las cuales se desarrolla el presente documento.

Partiendo del trabajo de esos primeros autores, Ronald Kriemann perfeccionó, un poco después, los algoritmos descritos para la suma, producto matriz-vector, producto matriz-matriz o inversa ([1]) que se habían planteado, y desarrolló otras operaciones no tan elementales del álgebra de matrices, como la factorización LU ([2]).

Actualmente la librería HLib [11] proporciona rutinas para la construcción de matrices jerárquicas, funciones de discretización para rellenar dichas matrices mediante aproximaciones de los operadores FEM (*Finite Element Methods*) o BEM (*Boundary Elements Methods*), algoritmos aritméticos para la adición, multiplicación, inversión y factoriza-

ción de matrices jerárquicas, rutinas de conversión de matrices dispersas y densas en matrices jerárquicas y de estas en matrices $\mathcal{H}^2$, así como funciones *auxiliares* para mostrar estructuras de matrices, realizar cuadraturas numéricas o manejar archivos.

Esta librería fue escrita en C por Lars Grasedyck y Steffen Börm, incluyendo llamadas a Blas y Lapack para llevar a cabo las operaciones algebraicas de menor nivel (como, por ejemplo, la resolución de sistemas de ecuaciones o problemas de valores propios). No se trata de código abierto porque, como se explica en la web de la librería, ha sido parcialmente desarrollado en el Max Planck Institut en Leipzig, lo cual no permite utilizar una licencia de código abierto y convertirlo en software gratuito; no obstante, si se quiere el código para fines académicos y/o de investigación, este se proporciona tras firmarse un acuerdo de licencia.

Hoy en día, es sobradamente conocido que, tanto la capacidad de almacenamiento como el tamaño de los conjuntos de datos que conforman las entradas de los algoritmos a resolver, mantienen un crecimiento rápido y proporcional. A la vista de esta tendencia que previsiblemente se mantendrá, al menos en un futuro cercano, es importante disponer de algoritmos con complejidad lineal para cálculos a gran escala.

Tal como se expone y justifica en las secciones que conforman este texto, las matrices jerárquicas (o matrices $\mathcal{H}$) son una herramienta muy útil para representar matrices densas de una forma jerárquica, a bloques, con dispersión de datos y con costes de almacenamiento en memoria log-lineal. Su uso permite reducir el coste cuadrático o incluso cúbico de los métodos tradicionales de operaciones sobre matrices como la suma, el producto de una matriz por un vector o por otra matriz, la inversión o la factorización LU, lográndose su resolución con un coste lineal (o casi lineal[1]).

Con esto en mente y atendiendo a la proliferación de grandes centros de trabajo con arquitecturas multi-core y clusters paralelos, Krieman elaboró su propia librería para el manejo de matrices jerárquicas, a la que llamó HLibPro [12]. Esta librería es especialmente importante, no solo por las operaciones que son capaces de resolver sus funciones, también presentes en HLib, sino porque es capaz de realizar las ejecuciones en parale-

[1]Una complejidad $\mathcal{O}(Nlog^qN)$ para algún $q > 0$ será considerada *quasi-lineal*, dado que el valor del logaritmo crece lentamente y, a efectos prácticos, este coste es muy similar a $\mathcal{O}(N)$.

lo, lo cual es de vital importancia hoy en día. Redactada en C++ y, de igual modo que HLib, apoyada en Blas y Lapack para ciertas operaciones a bajo nivel, el paralelismo que ofrece para escenarios con memoria compartida se basa en la gestión de hilos y, para el caso de memoria distribuida, en MPI (*Message Passing Interface*). Esta librería ofrece una licencia para uso comercial y otra para uso académico, gratuita bajo la aceptación del correspondiente acuerdo de licencias y condiciones.

Es importante señalar que, dadas sus ventajas tanto a nivel de coste computacional como de almacenamiento, las matrices jerárquicas han despertado un gran interés, muy especialmente entre quienes trabajan con problemas de control o ecuaciones integrales. Esto ha motivado la aparición de varias librerías alternativas a HLib o HLibPro para trabajar con matrices jerárquicas, algunas especializadas en algún campo de aplicación, como por ejemplo BEM++ [14], la cual se ha diseñado específicamente para tratar problemas de integrales de contorno. Entre las restantes, pueden destacarse H2Lib [15] (desarrollada en la Universidad de Kiel, en lenguaje C), u otras opciones en C++, como AHMED [16], DMHM [17], STRUMPACK [18] o Dense_HODLR [19]. También en Matlab existen librerías disponibles, como Misc [20], HSS [21], RSVDPACK [22] o HMat [23], así como en Fortran (SSS_Toeplitz [24]) o Python (H2TOOLS [25]).

1.2. Motivación personal

Doce de los sesenta créditos que recoge el Máster en Computación Paralela y Distribuida de la Universitat Politécnica de Valencia corresponden al Trabajo de Fin de Máster (en adelante, TFM). Tras concluir las asignaturas que suman los cuarenta y ocho créditos restantes, cada alumno debe elaborar un trabajo de especialización, eligiendo, según su preferencia, o bien desarrollar una aplicación de tipo práctico, o bien un trabajo de investigación para iniciarse en dicho campo.

Siempre he sentido interés por todo lo relacionado con la investigación científica y, a lo largo de mis estudios del Grado en Matemática Computacional, pude conocerla de primera mano, tanto en el ámbito de las matemáticas como en el de la computación científica, gracias a becas de investigación y de colaboración. Además, al finalizar mis

estudios de grado empecé a trabajar con el equipo de investigación “High Performance Computing and Architectures” de la Universitat Jaume I de Castellón, compaginando esto con mis estudios de máster.

Tras la consecución de mi título de máster, tengo previsto iniciar mi doctorado, centrado en estudiar y desarrollar algoritmos sobre grafos para aplicaciones de procesamiento de grandes volúmenes de datos. Como preparación y punto de partida del mismo decidí, junto con mis tutores, que mi TFM recogiera un trabajo de investigación en torno al álgebra de las matrices jerárquicas. En concreto, el cuerpo principal de mi trabajo se centra en el estudio y desarrollo de un algoritmo que calcule una factorización LU de una matriz jerárquica, detallando tanto una versión secuencial como una paralela sobre memoria compartida.

Si bien a la vista de la cantidad de librerías disponibles para el manejo de matrices jerárquicas, puede considerarse innecesaria una re-implementación de la factorización LU, de nuevo cabe destacar que dichas librerías no son, en su mayoría, de código abierto. Esto impide editar los algoritmos para ajustarlos a nuevos planteamientos y, dado que los primeros meses de mi doctorado se centrarán en paralelizar la factorización LU de matrices jerárquicas con técnicas distintas a OpenMP [10] (por ejemplo, empleando OmpSs [13]), resulta conveniente proponer una implementación propia, lo cual garantiza, a su vez, la comprensión de la ingeniería de este tipo de matrices.

1.3. Objetivos

Los objetivos que se plantearon inicialmente al proponer el Trabajo de Fin de Máster incluían:

- Búsqueda, lectura y comprensión de bibliografía relacionada con las matrices jerárquicas, sus operaciones algebraicas elementales y su factorización LU.
- Implementación de una versión secuencial de la factorización LU para matrices jerárquicas en Matlab para verificar la correcta comprensión del proceso.
- Traducción a C del algoritmo secuencial que se ha implementado en Matlab, inclu-

yendo llamadas a rutinas de Blas y Lapack para operaciones como el producto de matrices o la resolución de sistemas de ecuaciones.

- Paralelización de la factorización LU en memoria compartida mediante OpenMP.
- Prueba de otras técnicas de paralelización como OmpSs y comparativa con los resultados obtenidos mediante OpenMP.

Tal como se detalla en las secciones correspondientes, se han logrado todos estos objetivos, a excepción del último, por no disponer de tiempo suficiente para desarrollarlo.

1.4. Estructura del trabajo

El documento se estructura a lo largo de cinco capítulos, incluyéndose tras ellos la bibliografía y un anexo con parte del código implementado y las matrices que se han utilizado para las pruebas.

El Capítulo 1 es introducción al texto que, a partir del breve repaso por la historia de las matrices jerárquicas presentado en la primera sección ("Estado del arte") y de la motivación personal explicada en la segunda sección, justifique su atractivo para la realización de este TFM. Además, el primer capítulo recoge un listado de los objetivos del Trabajo, así como un resumen de la estructura del mismo.

A continuación, el Capítulo 2 expone los conceptos básicos teóricos relacionados con las matrices jerárquicas, su almacenamiento y la descripción de tres de las operaciones del álgebra elemental de matrices: la suma, el producto matriz-vector y el producto matriz-matriz. Para cada una de estas operaciones se compara su definición en el ámbito de las $\mathcal{H}$-matrices con la que presentan para las matrices convencionales y se incluye una versión secuencial y otra paralela de los algoritmos que las resuelven.

El Capítulo 3 está dedicado a la factorización LU de las matrices jerárquicas, incluyendo una breve introducción que repasa qué es y cómo se calcula dicha factorización sobre las matrices convencionales y también sobre las jerárquicas, los esquemas de los algoritmos que la calculan en el caso de las $\mathcal{H}$-matrices y una explicación de mis implementaciones.

Los últimos dos capítulos están dedicados, respectivamente, a presentar las conclusiones derivadas de todo el trabajo de investigación e implementación que ha permitido la elaboración de este TFM y a presentar las líneas de trabajo futuro que previsiblemente desarrollaré durante los primeros meses de mi doctorado.

Espero que todo lo expuesto a lo largo del presente documento resulte tanto interesante como debidamente justificado y presentado.

Capítulo 2

Álgebra elemental de las matrices jerárquicas

2.1. Las matrices jerárquicas: conceptos básicos

Además de su definición, es importante saber cómo se construyen estas matrices. Estos dos aspectos se tratarán en esta sección como introducción a las matrices jerárquicas.

De manera intuitiva, podría definirse una matriz jerárquica como la estructura de datos compuesta por varios bloques en los que se agrupan los elementos que la conforman, según el particionado de su conjunto de índices. Este particionado conforma un árbol jerárquico ($\mathcal{H}$-tree) o *cluster tree*. Veamos una definición formal para este árbol.

Definición 2.1. *Sea el conjunto de índices I, con cardinalidad $n = \#I$. A un árbol $T_I = (V, E)$, con V el conjunto de nodos que lo componen y E el conjunto de aristas entre ellos, se le denomina árbol jerárquico o cluster tree sobre I cuando se cumplen, a la vez:*

- *I es la raíz de T_I y, además, $\emptyset \neq v \subset I$ para todo $v \in V$.*
- *Para todos los nodos $v \in V$, o bien $v = \dot{\cup}_{v' \in S(v)} v'$ siendo $\dot{\cup}$ la unión disjunta, o bien v es una hoja de T_I, i.e. $S(v) = \emptyset$.*

Donde $S(v)$ es el conjunto de nodos hijos del nodo v, i.e. $S(v) = \{v' \in V \mid (v, v') \in E\}$. Además, al conjunto de nodos hoja del árbol T_I se le denota mediante $\mathcal{L}(T_I)$ y cabe señalar que cada nodo de T_I es, a su vez, un cluster.

Para no incurrir en incongruencias y confusiones, es importante comentar el uso que va a hacerse de la notación referente a los árboles jerárquicos.

Notación 2.1. *A lo largo del capítulo, se escribirá $t \in T_I$ si $t \in V$, identificándose T_I con V y no siendo necesario el uso de E, dado que $S(\cdot)$ contiene toda la información sobre los nodos hijo y, por ende, sobre las aristas entre los nodos del árbol.*

Veamos ahora algunas anotaciones y observaciones para aclarar la estructura que define un árbol jerárquico y sus implicaciones.

Nota 2.1. *Dos nodos del árbol jerárquico se dice que están en el mismo nivel del árbol si están a la misma distancia de la raíz. De este modo, al subconjunto de nodos del mismo nivel l del árbol T_I se le representa $T_I^l := \{t \in T_I : level(t) = l\}$.*

Nota 2.2. *Las hojas del árbol jerárquico T_I tienen un tamaño máximo limitado $n_{min} > 0$, i.e. $\forall t \in \mathcal{L}(T_I)$ se cumple que $\#t \leq n_{min}$.*

Observación 2.1. *Cualquier árbol jerárquico T_I verifica la propiedad $\dot{\bigcup}_{v \in \mathcal{L}(T_I)} v = I$, es decir, las hojas del árbol jerárquico producen una partición para el conjunto de índices I.*

Observación 2.2. *Cualquier árbol jerárquico T_I verifica la propiedad $\dot{\bigcup}_{v \in \mathcal{L}(T_I)} v = I$, es decir, las hojas del árbol jerárquico producen una partición para el conjunto de índices I.*

Este límite se utiliza, en la práctica, para fijar cuándo detener el particionado, es decir, si para un vértice $t \in T_I$ su cardinalidad cumple que $\#t \leq n_{min}$, entonces t será una hoja en T_I y ya no se dividirá en más bloques. De esta forma, n_{min} se convierte en análogo al "tamaño mínimo de bloque" cuando se aplica el particionado sobre matrices densas.

La Figura 2.1, extraída de [3], ejemplifica un árbol jerárquico para el conjunto de índices $\mathcal{I} = \{0, 1, 2, 3, 4, 5, 6, 7\}$.

Retomándose el discurso en términos de matrices, es bien sabido que sus elementos vendrán determinados por dos índices que pertenecerán al conjunto $I \times J$ siendo I, J conjuntos de índices de la forma $I = \{i_0, i_1, ..., i_n\}$, $J = \{j_0, j_1, ..., j_m\}$, con $n, m \in \mathbb{N}$.

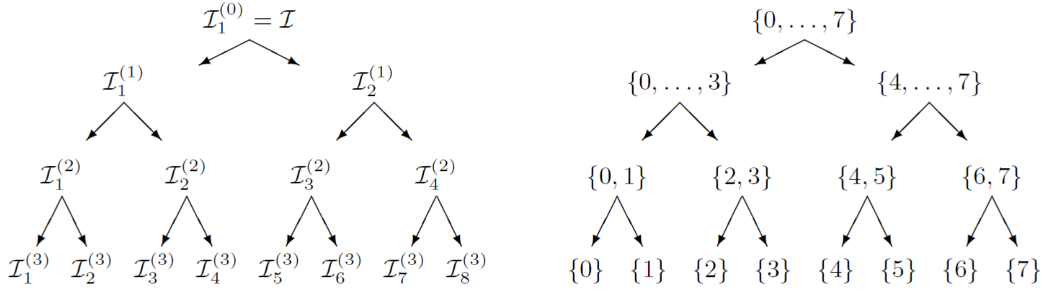


Figura 2.1: Árbol jerárquico para el conjunto de índices $\mathcal{I} = \{0, 1, 2, 3, 4, 5, 6, 7\}$.

Este conjunto $I \times J$, puede particionarse mediante la definición de un árbol jerárquico especial, de la forma $T_{I\times J}$, el cual define el subconjunto de índices a utilizar para la división en bloques de los índices de la matriz. Justo de este particionado derivan las matrices jerárquicas o $\mathcal{H}$-matrices.

Para que la $\mathcal{H}$-matriz resultante sea eficiente, el particionado se restringirá a un conjunto mínimo de clusters de bloques, $t \times s \in T_{I\times J}$, con $t, s \in T_I$ ó T_J de forma que $t \times s$ sea suficientemente pequeño, es decir, $t, s \in \mathcal{L}(T_{I\times J})$ o de un tamaño admisible. La condición clásica de admisibilidad viene dada por la expresión:

$$min(diam(t), diam(s)) \leq \eta dist(t, s)$$

Donde $diam(t) = max_{t_i,t_j \in t}||t_i - t_j||$, $diam(s) = max_{s_i,s_j \in s}||s_i - s_j||$, $dist(t, s) = min_{t_i \in t, s_i \in s}||t_i - s_i||$, y $\eta \in \mathbb{R}_{>0}$ es un parámetro fijado para acotar la admisibilidad.

Con el concepto de árbol jerárquico y la condición clásica de admisibilidad, puede definirse ya el particionado de índices que dará lugar a las matrices jerárquicas.

Definición 2.2. *Sea T_I un árbol jerárquico sobre I. Entonces se define el árbol jerárquico a bloques $T_{I\times I}$ recursivamente para un vértice $b = t \times s$, empezando con la raíz $I \times I$ de la forma:*

$$S(b) = \begin{cases} \emptyset & si\ b\ es\ admisible\ o\ \mathcal{S}(t) = \emptyset\ o\ \mathcal{S}(s) = \emptyset \\ \mathcal{S}' & en\ cualquier\ otro\ caso \end{cases}$$

Siendo $\mathcal{S}' = \{t' \times s' : t' \in \mathcal{S}(t), s' \in \mathcal{S}(s)\}$ repetidamente para todos los nodos de $\mathcal{S}(b)$.

Siguiendo esta definición, retomando el conjunto de índices planteado en la Figura 2.1, el conjunto $I \times I = \{0, 1, 2, 3, 4, 5, 6, 7\} \times \{0, 1, 2, 3, 4, 5, 6, 7\}$ puede particionarse tal

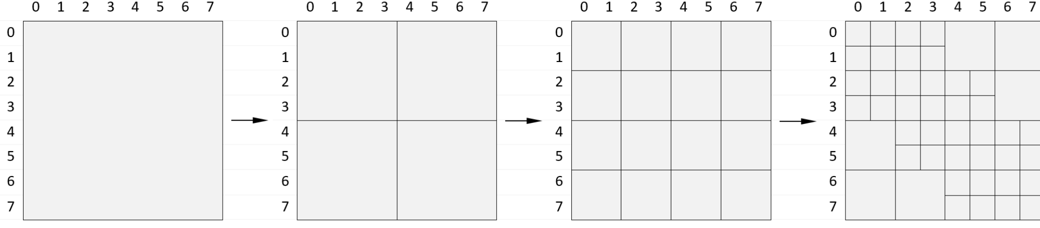


Figura 2.2: Particionado de una matriz mediante el árbol jerárquico a bloques para el conjunto de índices $\mathcal{I} \times \mathcal{I} = \{0, 1, 2, 3, 4, 5, 6, 7\} \times \{0, 1, 2, 3, 4, 5, 6, 7\}$.

como aparece en la Figura 2.2 hasta conseguir que todos los nodos sean admisibles.

La condición de admisibilidad garantiza que los bloques admisibles en $T_{I\times I}$ pueden aproximarse con una precisión $\epsilon \geq 0$ predefinida, mediante matrices de rango bajo con un rango máximo k. Precisamente el conjunto de dichas matrices conforma el conjunto de matrices jerárquicas.

Definición 2.3. *El conjunto de matrices jerárquicas $\mathcal{H}(T_{I\times I}, k)$ puede definirse, para un árbol jerárquico de bloques $T_{I\times I}$ sobre un árbol jerárquico T_I y para $k \in \mathbb{N}$, como:*

$$\mathcal{H}(T_{I\times I}, k) := \{\ M \in \mathbb{R}^{I\times I} \mid \forall\, t \times s \in \mathcal{L}(T_{I\times I}) :$$

$$rango(M|_{t\times s}) \leq k \vee \{t, s\} \cap \mathcal{L}(T_I) \neq \emptyset\ \}$$

En las $\mathcal{H}$-matrices, las matrices de rango bajo $M|_{t\times s}$ se pueden representar en la forma $M = AB^T$ donde $A \in \mathbb{R}^{t\times k}$ y $B \in \mathbb{R}^{s\times k}$. De este modo, se logra que el almacenamiento de las matrices jerárquicas sea de orden $\mathcal{O}(k\, n\, log\, n)$ (ver [5]). Asimismo, la complejidad de la suma de matrices sea de orden $\mathcal{O}(k^2\ n\ log\ n)$ y el producto, la inversión y la factorización LU de orden $\mathcal{O}(k^2\ n\ log^2\ n)$.

Cabe señalar que, en lugar de basarse en un rango k fijo, la construcción y las operaciones aritméticas de matrices $\mathcal{H}$ pueden hacerse atendiendo a la precisión ϵ. Para esto, en el momento de truncar las matrices de rango bajo, solamente se utilizan los valores singulares $s_l \geq \epsilon \cdot s_0$, con $l \leq 0 < k$, y los vectores correspondientes, asumiéndose que $s_0 \geq s_1 \geq s_2 \ldots$. Esto conlleva un rango variable en los sub-bloques de la matriz jerárquica y mayor eficiencia que las operaciones aritméticas basadas en rangos fijos en lugar de precisiones fijas (ver [1]).

2.2. Almacenamiento de matrices jerárquicas

La configuración de una matriz jerárquica $M \in \mathcal{H}(T)$ puede conseguirse construyendo la jerarquía asociada a los nodos del árbol jerárquico a bloques formado por sus índices, de forma que aquellos que sean admisibles se convierten en hoja y, lo que no, se dividan en la jerarquía de nodos necesaria hasta alcanzar un tamaño admisible o nodos hoja. El proceso recursivo para este propósito está detallado en el algoritmo 1.

Algorithm 1 Construcción recursiva de una matriz jerárquica

Entrada: $T_{I\times I}$ árbol jerárquico a bloques, $t \times s \in T_{I\times I}$.
Salida: $M \in \mathcal{H}(T)$.

```
 1: procedure C = BUILD_HMATRIX(t × s, T_{I×I})
 2:     if t × s ∈ L(T_{I×I}) then
 3:         if is_admissible(t × s) or size(t × s) ≤ n_min then
 4:             S(t × s) = ∅
 5:         end if
 6:     else
 7:         S(t × s) = {t' × s' | t' ∈ S(t), s' ∈ S(s)}
 8:         for all t' × s' ∈ S(t × s) do
 9:             Build_HMatrix (t' × s', T_{I×I})
10:         end for
11:     end if
12: end procedure
```

Es importante destacar que las matrices jerárquicas presentan beneficios en cuanto a costes de almacenamiento y de procesamiento cuando hay una cierta dispersión en los datos que representan. En esos casos, si el particionado es adecuado, no es necesario hacer un almacenamiento denso de los datos sino simplemente controlar en qué bloques todos los elementos son cero y evitándose su almacenamiento.

Para el control de esta dispersión se utiliza la constante de dispersión C_{sp} definida sobre los árboles jerárquicos a bloques $T_{I\times J}$, con independencia de la cardinalidad de los conjuntos de índices.

Definición 2.4. *Sea el árbol jerárquico a bloques $T_{I\times J}$ basado en T_I y T_J, con I, J conjuntos de índices. Se define la constante de dispersión C_{sp} de $T_{I\times J}$ como:*

$$C_{sp} := max\{ \ max_{r\in T_I} \#\{s \in T_j \mid r \times s \in T_{I\times J}\}, max_{s\in T_J} \#\{r \in T_I \mid r \times s \in T_{I\times J}\} \ \}$$

En [5] puede consultarse la demostración del lema que sigue.

Lema 2.1. *Sea $T_{I\times J}$ un árbol jerárquico a bloques basado en T_I y T_J, con I, J conjuntos de índices, con constante de dispersión C_{sp} y tamaño de bloque mínimo n_{min}. Entonces los requisitos de almacenamiento $N_{\mathcal{H},S_t}(T_{I\times J}, k)$ para una matriz jerárquica $M \in \mathcal{H}(T_{I\times J}, k)$, están acotados por:*

$$N_{\mathcal{H},S_t}(T_{I\times J}, k) \leq \#LC_{sp}max\{k, n_{min}\}(\#I + \#J)$$

El objetivo es, pues, construir un árbol jerárquico a bloques $T_{I\times J}$ tal que $depth(T_{I\times J}) = \mathcal{O}(logn)$, donde $n = \#I = \#J$.

En el caso que concierne a este trabajo, dado que la pretensión última es estudiar, analizar, comprender e implementar la factorización LU de matrices jerárquicas, en las secciones 3 y 4 asumiremos que las matrices $\mathcal{H}$ a factorizar vienen ya dadas mediante dos vectores: uno de ellos conteniendo los valores de la matriz - almacenados por columnas - y otro conteniendo los índices de inicio de los bloques de cada nivel, empezando por el nivel que contiene menos bloques, es decir, más elementos por bloque. Veamos un ejemplo para entender esta representación debidamente.

Ejemplo 2.1. *Sea una $\mathcal{H}$-matriz de tamaño 8×8 donde, para facilitar la comprensión, los valores almacenados coinciden con el orden que les corresponde en el almacenamiento, como la que aparece en la Figura 2.3, cuyo particionado del nivel cero (entendiendo este como la matriz sin particionar) se subdivide en hasta dos niveles anidados, de tamaños 4×4, 2×2, respectivamente. Para esta situación, la representación correspondiente se compondría de:*

- *Vector de valores:*[1] $[\,1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 12,\ ...,\ 61,\ 62,\ 63,\ 64\,]$
- *Vector de índices:* $[\,0,\ 0,\ 4,\ 32,\ 36,\ 0,\ 2,\ 16,\ 18,\ 36,\ 38,\ 52,\ 54\,]$

Puede apreciarse, atendiendo al segundo vector, que el nivel 0 está formado por un único bloque que se inicia en el índice 0, el nivel 1 se compone de cuatro bloques con inicios en los índices 0, 16, 32 y 36, respectivamente, el nivel 2 se compone de ocho bloques con

[1] Se compone de todos los valores que forman la matriz; en este caso todos los números naturales entre 1 y 64. Se ha configurado así el ejemplo para reflejar también, con dichos valores, el orden de almacenamiento de los valores de la matriz (por columnas y por bloques).

inicios en los índices 0, 2, 16, 18, 36, 38, 52 y 54, respectivamente.

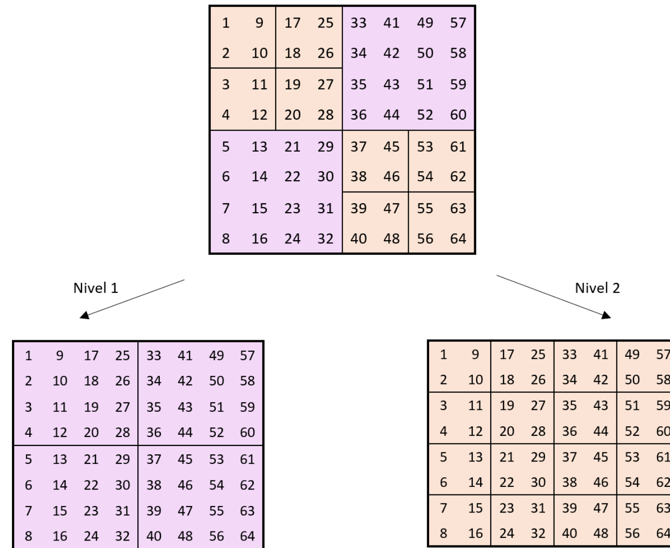


Figura 2.3: Ejemplo de descomposición en niveles de una matriz jerárquica pequeña.

2.3. Operaciones algebraicas elementales con $\mathcal{H}$-matrices

En esta sección del capítulo presento la definición y dos versiones de los algoritmos (recursiva y secuencial) que resuelven la suma de matrices jerárquicas, su producto por un vector y su producto por otra matriz jerárquica.

2.3.1. Suma de matrices jerárquicas

Una de las operaciones convencionales más elementales sobre matrices es la suma de las mismas. La resolución de la suma de matrices en el caso de matrices convencionales es sencilla.

Definición 2.5. *Sean las matrices $A, B, C \in \mathbb{R}^{n \times m}$, con $n, m \in \mathbb{N}$. Se define la suma de matrices $C = A + B$ donde $c_{ij} = a_{ij} + b_{ij}$ con $c_{ij} \in C$, $a_{ij} \in A$, $b_{ij} \in B$, $0 \leq i < n \in \mathbb{N}$, $0 \leq j < m \in \mathbb{N}$.*

En el caso de matrices jerárquicas, esta definición se mantiene, si bien puede llevarse a cabo el proceso, tal como puede intuirse, bloque a bloque.

Definición 2.6. *Sean* $A, B \in \mathcal{H}(T_{I\times J})$ *dos matrices jerárquicas sobre el conjunto de índices* $I \times J$*, entonces la suma de ambas puede expresarse como* $C = A + B$*, con* $C \in \mathcal{H}(T_{I\times J})$ *también sobre el conjunto de índices* $I \times J$*, donde* $C_{t\times s} = A_{t\times s} + B_{t\times s}$*, con* $t \times s \in T_{I\times J}$*.*

El algoritmo 2 describe el proceso a seguir para realizar la suma de matrices jerárquicas de forma recursiva, comprobándose si un cierto bloque a sumar contiene sub-bloques, revisando para ello si el conjunto de nodos hijos del mismo es vacío o no. Si el bloque actual no contiene sub-bloques (es nodo hoja) se realiza la suma y se devuelve el resultado; en caso contrario se realiza una llamada recursiva al mismo algoritmo.

Algorithm 2 Suma de matrices jerárquicas (versión recursiva)

Entrada: $A \in \mathcal{H}(T_{I\times J})$, $B \in \mathcal{H}(T_{I\times J})$.
Salida: $C \in \mathcal{H}(T_{I\times J})$.

```
1: procedure C = ADDHMATRICES(A, B, t × s)
2:     if S(t × s) ≠ ∅ then
3:         for t' × s' ∈ S(t × s) do
4:             C_{t'×s'} = AddHMatrices(A_{t'×s'}, B_{t'×s'}, t' × s')
5:         end for
6:     else C = A + B
7:     end if
8: end procedure
```

No obstante, la suma de matrices jerárquicas podría llevarse a cabo también de forma secuencial, lo cual facilita su paralelización si se tiene en cuenta, además, que no hay dependencias entre las sumas de bloques distintos.

Partiendo de las mismas matrices jerárquicas que en el caso del algoritmo recursivo, el esquema presentado en el algoritmo 3 plantea una implementación de sumas de matrices jerárquicas de forma secuencial.

En este segundo planteamiento, se recorren todos los bloques de todos los niveles de las matrices a sumar y se suman aquellos que son nodos hoja respecto al árbol jerárquico - tal como puede verse en la Figura 2.4 -, es decir, aquellos que en la representación matricial no contienen sub-bloques (por esto se obtiene de nuevo el conjunto de nodos hijo de cada bloque, mediante $S(t, s)$).

Si se dispone del conjunto de hojas de las matrices que intervienen en la suma (o si se

Algorithm 3 Suma de matrices jerárquicas (versión secuencial)

Entrada: $A \in \mathcal{H}(T_{I\times J})$, $B \in \mathcal{H}(T_{I\times J})$.
Salida: $C \in \mathcal{H}(T_{I\times J})$.

1: **procedure** C = ADDHMATRICES(A,B)
2: **for each** level l **do**
3: **for each** block $t \times s$ **in** l **do**
4: **if** $S(t \times s) = \emptyset$ **then**
5: $C_{t\times s} = A_{t\times s} + B_{t\times s}$
6: **end if**
7: **end for**
8: **end for**
9: **end procedure**

obtiene el mismo a partir de ellas), entonces serían necesarias menos comprobaciones y su ejecución se simplificaría, tal como ilustra el algoritmo 4.

Algorithm 4 Suma de matrices jerárquicas (versión secuencial conociendo el conjunto de hojas $\mathcal{L}(T_{I\times J})$)

Entrada: $A \in \mathcal{H}(T_{I\times J})$, $B \in \mathcal{H}(T_{I\times J})$, $\mathcal{L}(T_{I\times J})$.
Salida: $C \in \mathcal{H}(T_{I\times J})$.

1: **procedure** C = ADDHMATRICES($A, B, \mathcal{L}(T_{I\times J})$)
2: **for each** block $t \times s$ **in** $\mathcal{L}(T_{I\times J})$ **do**
3: $C_{t\times s} = A_{t\times s} + B_{t\times s}$
4: **end for**
5: **end procedure**

La paralelización de la suma de matrices jerárquicas resulta ahora trivial.

La Figura 2.4 ilustra la suma de matrices jerárquicas que se ha descrito en este apartado.

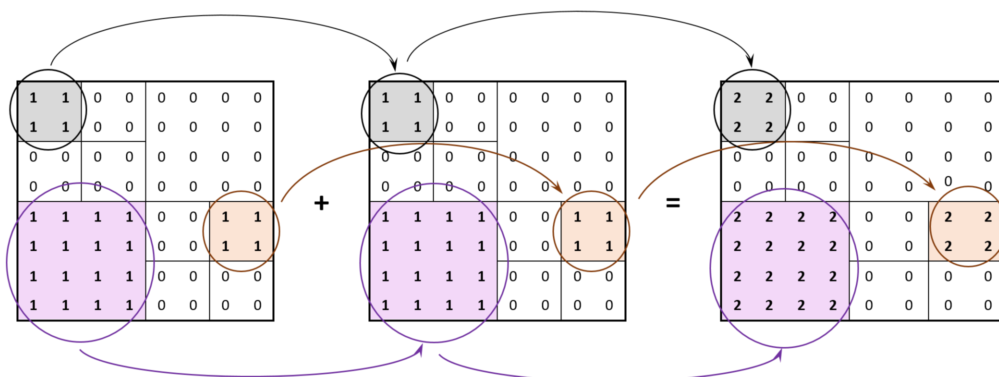


Figura 2.4: Esquema de la suma de matrices jerárquicas.

2.3.2. Producto matriz-vector con matrices jerárquicas.

Otra de las operaciones frecuentes en el álgebra lineal de las matrices comunes es su producto por un vector.

Definición 2.7. *Sea la matriz $A \in \mathbb{R}^{n \times m}$ y sea el vector $V \in \mathbb{R}^m$, con $n, m \in \mathbb{N}$. Se define el producto $W = A \times V$, con $W \in \mathbb{R}^n$, donde $w_i = a_{ij} \times v_j$ con $w_i \in W$, $a_{ij} \in A$, $v_j \in V$, $0 \leq i < n \in \mathbb{N}$, $0 \leq j < m \in \mathbb{N}$.*

La definición tradicional se mantiene también en el caso de matrices jerárquicas, pero puede expresarse según los bloques que componen la matriz jerárquica, como ilustran los algoritmos que siguen a la definición.

Definición 2.8. *Sea la matriz jerárquica $M \in \mathcal{H}(T_{I \times J})$ sobre el conjunto de índices $I \times J$ y sea el vector $V \in \mathbb{R}^J$, entonces el producto matriz por vector entre ambos tiene como resultado el vector $W = M \times V$, con $W \in \mathbb{R}^I$.*

El algoritmo 5 plantea la resolución recursiva de este producto. De igual modo que con la suma, este algoritmo se centra en comprobar si un bloque contiene sub-bloques (revisando la cardinalidad del conjunto de nodos hijos) para llamar a la misma función recursivamente en caso afirmativo y, en caso contrario, realizar el producto correspondiente y devolver el resultado.

Algorithm 5 Producto matriz $\mathcal{H}$ por vector (versión recursiva)

Entrada: $A \in \mathcal{H}(T)$, $V \in \mathbb{R}^J$.
Salida: $W \in \mathbb{R}^I$.

1: **procedure** W = MATVECPRODUCT($A, V, t \times s$)
2:     **if** $S(t \times s) \neq \emptyset$ **then**
3:         **for each** $t' \times s' \in S(t \times s)$ **do**
4:             $W_{t'} = MatVecProduct(A_{t' \times s'}, V_{s'})$
5:         **end for**
6:     **else** $W_t = M_{t \times s} \times V_s$
7:     **end if**
8: **end procedure**

Dado el interés en eliminar la recursividad planteando un algoritmo secuencial que pueda paralelizarse, el algoritmo 6 plantea una opción secuencial ya paralelizada para resolver

el producto de una matriz jerárquica por un vector. En ella, p representa el total de procesadores que van a intervenir en la ejecución, $I_{i,vec}$ corresponde a los índices del vector asociados al procesador i e $I_{i,mat}$ al conjunto de índices de la matriz que el procesador i utiliza.

Algorithm 6 Producto matriz $\mathcal{H}$ por vector (versión paralelizada)

Entrada: $A \in \mathcal{H}(T_{I\times J}), V \in \mathbb{R}^J$.
Salida: $W \in \mathbb{R}^I$.

1: **procedure** Y = MATVECPRODUCT(A,V)
2:     **for** $0 \leq i < p$ **do**
3:         $W_i' = A_i \cdot V$
4:     **end for**
5:     synch_all()
6:     **for** $0 \leq i < p$ **do**
7:         $W_i += \sum_{W_j' \in \{W_j' | I_{i,vec} \cap I_{j,mat} \neq \emptyset\}} W_j'|_{I_{i,vec}}$
8:     **end for**
9:     synch_all()
10: **end procedure**

Tal como ilustra la Figura 2.5, el algoritmo secuencial reparte los bloques de la matriz que corresponden a hojas entre los procesos que intervienen en el cálculo del producto. Cada proceso multiplica los valores que le han sido asignados por los elementos que corresponden en el vector por el que se quiere multiplicar la matriz y, finalmente, los resultados locales a cada proceso se suman para tener el resultado general final.

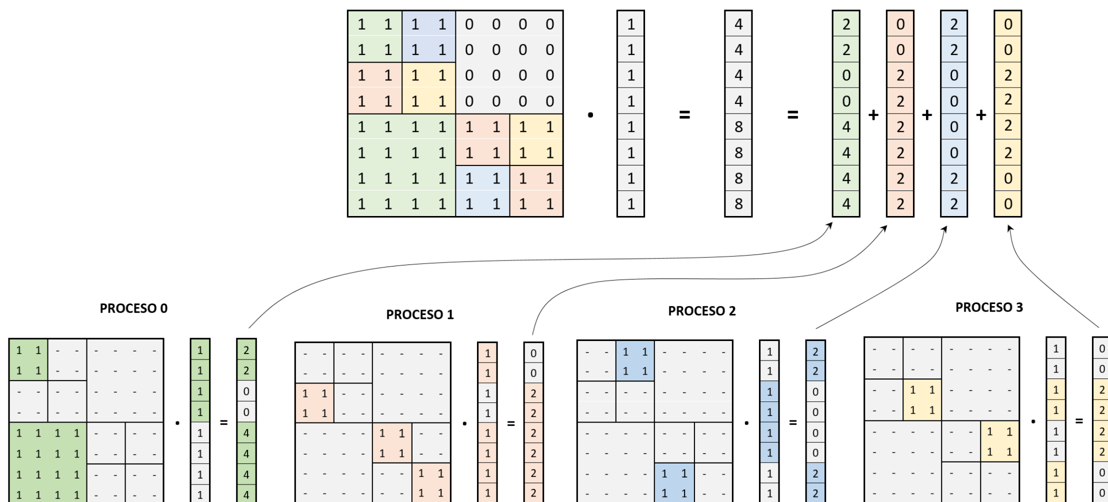


Figura 2.5: Esquema del produto de una matriz jerárquica por un vector.

2.3.3. Producto matriz-matriz con matrices jerárquicas.

Además de la suma de matrices y su producto por vectores, el producto entre matrices también es una operación habitual en el campo del álgebra de las matrices convencionales y también lo será en el caso de $\mathcal{H}$-matrices.

Definición 2.9. *Sean las matrices $A \in \mathbb{R}^{n\times k}$ y $B \in \mathbb{R}^{k\times m}$, con $n, k, m \in \mathbb{N}$. Se define el producto $C = A \times B$, con $C \in \mathbb{R}^{n\times m}$, donde $c_{ij} = \sum_{r=0}^{k-1} a_{ir} \times b_{rj}$ con $c_{ij} \in C$, $a_{ir} \in A$, $b_{rj} \in B$, $0 \leq i < n \in \mathbb{N}$, $0 \leq j < m \in \mathbb{N}$, $0 \leq r < k \in \mathbb{N}$.*

La misma definición se verifica en el caso de matrices jerárquicas, pero conviene reformularla en términos de los bloques que las componen.

Definición 2.10. *Sean los árboles jerárquicos a bloques $T = T_{I\times K}$, $T' = T_{K\times J}$ y sean las matrices jerárquicas $A \in \mathcal{H}(T)$, $B \in \mathcal{H}(T')$. Se define el producto de matrices $C = A \cdot B$ con $C \in \mathcal{H}(T'')$ a partir del producto de sus árboles jerárquicos $T \cdot T' = T'' = T_{I\times J}$ de forma que, para cada nivel $l = 0, ..., p-1$ y cada nodo $t \times s \in (T \cdot T')^{(l)}$, el conjunto de hijos de $t \times s$ se define:*

$$S(t \times s) := \{t' \times s' \mid \exists r \in T_J^{(l)},\ \exists r' \in T_J^{(l+1)} : t' \times r' \in S_T(t \times r), r' \times s' \in S_{T'}(r \times s)\}$$

Y, de esta forma, el producto de dos matrices jerárquicas queda, para cada bloque hoja, $t \times s \in \mathcal{L}(T \cdot T', i)$, como:

$$(A \cdot B)|_{t\times s} = \sum_{j=0}^{i} \sum_{r\in\mathcal{U}(t\times s,j)} A|_{t\times r} \cdot B|_{r\times s}$$

Donde $\mathcal{U}(t \times s, j)$ es:

$$\mathcal{U}(t \times s, j) = \{\ (r \in T_J^{(j)} \mid \mathcal{F}^j(t) \times r \in T)\ y\ (r \times \mathcal{F}^j(s) \in T')\ y$$
$$(\ (\mathcal{F}^j(t) \times r \in \mathcal{L}(T))\ o\ (r \times \mathcal{F}^j(s) \in \mathcal{L}(T'))\ )\}$$

Con $j \in \mathbb{N}_0$, siendo $\mathcal{F}^j(s) \in T(j)$ el vértice predecesor a $s \in T^{(i)}$ en el nivel $j \in \{0, ..., i\}$, con $i \in [\ 0,\ profundidad(T)\ ]$.

Este producto puede llevarse a cabo siguiendo el esquema presentado en el algoritmo 7.

Algorithm 7 Producto matriz $\mathcal{H}$ por matriz $\mathcal{H}$(versión recursiva)

Entrada: $A \in \mathcal{H}(T_{I\times J})$, $B \in \mathcal{H}(T_{J\times K})$.
Salida: $C \in \mathcal{H}(T_{I\times K})$.

1: **procedure** C = MATMATPRODUCT(A,B)
2: **if** A, B contain sub-blocks **then**
3: **for** $i, j, l \in TotalLevels$ **do**
4: $C_{i,j} = MatMatProduct(A_{i,l}, B_{l,j})$
5: **end for**
6: **else** $C = A \cdot B$
7: **end if**
8: **end procedure**

De nuevo, conviene disponer de una implementación secuencial para llevar a cabo la paralelización. Esta se expone en el algoritmo 8, donde $\mathcal{L}_{MM}$ representa el conjunto de bloques destino de los productos a realizar (nodos hoja de la matriz resultado donde se almacenarán los productos calculados) el cual se calcula aplicando la función `ObtainLMM` y en la que P_C es el conjunto de parejas de factores que contribuyen a la matriz C.

Algorithm 8 Producto matriz $\mathcal{H}$ por matriz $\mathcal{H}$(versión paralelizada)

Entrada: $A \in \mathcal{H}(T_{I\times J})$, $B \in \mathcal{H}(T_{J\times K})$, $\mathcal{L}_{MM}$, P_C.
Salida: $C \in \mathcal{H}(T_{I\times K})$.

1: **procedure** $C = PerformProduct(A, B, \mathcal{L}_{MM}, P_C)$
2: **for all** $C' \in \mathcal{L}_{MM}$ **do**
3: **for all** $(A', B') \in P_C$ **do**
4: $C' += A' \cdot B'$
5: **end for**
6: **end for**
7: synch_all()
8: **end procedure**

Algorithm 9 Producto matriz $\mathcal{H}$ por matriz $\mathcal{H}$(versión paralelizada)

Entrada: $A \in \mathcal{H}(T_{I\times J})$, $B \in \mathcal{H}(T_{J\times K})$.

Salida: $C \in \mathcal{H}(T_{I\times K})$, P_C.

1: **procedure** $\mathcal{L}_{MM}, P_C = ObtainLMM(A, B)$
2: **if** A,B contain sub-blocks **then**
3: **for** $i, j, l \in TotalLevels$ **do** $ObtainLMM(A_{il}, B_{lj}, \mathcal{L}_{MM}, P_C)$
4: **end for**
5: **else** $P_C := P_C \cup \{(A, B)\}$; $\mathcal{L}_{MM} := \mathcal{L}_{MM} \cup \{C\}$
6: **end if**
7: **end procedure**

Para aclarar el funcionamiento del algoritmo secuencial, presento a continuación un ejemplo de aplicación.

Ejemplo 2.2. *Sean las matrices jerárquicas $A, B, C \in \mathcal{H}(T_{I\times I})$, con $I = \{0, 1, 2, 3, 4, 5, 6, 7\}$, con la división en bloques que se describe en la Figura 2.6.*

Las parejas de bloques que intervienen en el producto (obviando los que son cero y no modifican el resultado del producto) son:

$$\mathcal{L}_{MM} = [\ \{A^1_{10}, B^1_{00}\}, \{A^1_{11}, B^1_{10}\}, \{A^2_{00}, B^2_{00}\}, \{A^2_{00}, B^2_{01}\}, \{A^2_{10}, B^2_{00}\}, \{A^2_{10}, B^2_{01}\},$$
$$\{A^2_{01}, B^2_{10}\}, \{A^2_{01}, B^2_{11}\}, \{A^2_{11}, B^2_{10}\}, \{A^2_{11}, B^2_{11}\}, \{A^2_{22}, B^2_{22}\}, \{A^2_{22}, B^2_{23}\}, \{A^2_{23}, B^2_{32}\},$$
$$\{A^2_{23}, B^2_{33}\}, \{A^2_{32}, B^2_{22}\}, \{A^2_{32}, B^2_{23}\}, \{A^2_{33}, B^2_{32}\}, \{A^2_{33}, B^2_{23}\}\ ]$$

Y los bloques destino de los productos son:

$$P_C = [\ C^1_{00},\ C^1_{10},\ C^2_{00},\ C^2_{01},\ C^2_{10},\ C^2_{11},\ C^2_{00},\ C^2_{01},\ C^2_{10},\ C^2_{11},$$
$$C^2_{22},\ C^2_{23},\ C^2_{22},\ C^2_{23},\ C^2_{32},\ C^2_{33},\ C^2_{32},\ C^2_{33}\ ]$$

Cada proceso realizará $size(\mathcal{L}_{MM})/p$ productos (siendo p el total de procesos que intervienen en el cálculo), procurándose en el reparto que el mayor número de productos con el mismo destino en C *sean realizados por el mismo proceso, reduciéndose así las colisiones al almacenar el resultado.*

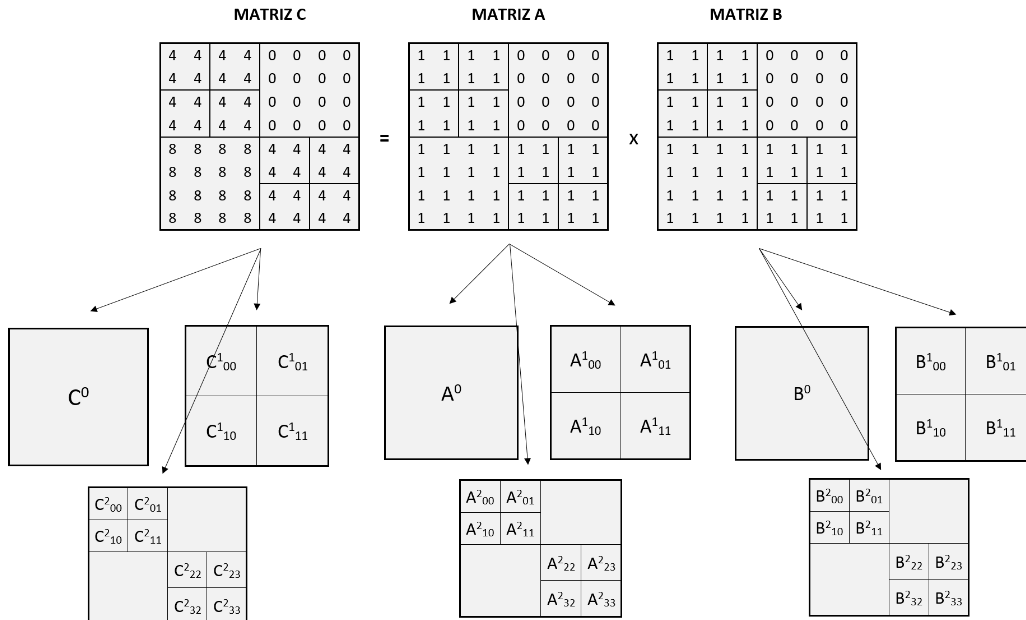


Figura 2.6: Esquema del produto de una matriz jerárquica por un vector.

Capítulo 3

Factorización LU de matrices jerárquicas

3.1. Factorización LU: conceptos generales

Antes de abordar la factorización LU en el ámbito de las matrices jerárquicas, en esta sección se hace un breve repaso de la misma sobre las matrices convencionales. Además, se incluye el algoritmo para la factorización LU con pivotamiento, así como la versión a bloques, ambos necesarios para facilitar la comprensión de la versión que se aplica en el caso de las $\mathcal{H}$-matrices.

Definición 3.1. *Sea la matriz $A \in \mathbb{R}^{m\times n}$. Se define la factorización LU de la matriz A como el producto $A = L \cdot U$ con $L \in \mathbb{R}^{m\times m}$ matriz triangular inferior unitaria, $U \in \mathbb{R}^{m\times n}$ matriz triangular superior.*

El teorema que sigue fija a qué matrices puede calculárseles esta factorización LU y cuándo es única.

Teorema 3.1. *Sea la matriz $A \in \mathbb{R}^{m\times n}$. Esta matriz tiene una factorización LU si $det(A(1:k, 1:k)) \neq 0$ para $k = 1 : n-1$. Además, si la factorización LU existe y la matriz A es no singular, entonces la factorización LU es única y el determinante de la matriz verifica $det(A) = u_{11} \cdot ... \cdot u_{nn}$, con $u_{ii} \in U$.*

La demostración de este teorema puede encontrarse en [6].

Ahora bien, ¿cómo se consiguen ambas matrices? Esto es posible resolviendo el sistema

$Ax = b$, de forma que:

Siendo $Ly = b, Ux = y$ entonces se puede expresar $Ax = LUx = Ly = b$.

La resolución del sistema indicado puede llevarse a cabo mediante transformaciones de Gauss, de forma que, si se sigue el algoritmo 10 se realizan n transformaciones de la forma $M_{n-1} \cdot M_{n-2} \cdot ... \cdot M_1 \cdot A = U$ (que se almacenan en la propia matriz A[1]) dando lugar a la matriz U triangular superior, siendo $L = M_1^{-1} \cdot M_2^{-1} \cdot ... \cdot M_{n-1}^{-1}$, es decir, $A(i,k))l_{ik}$ con $i > k$, $i,k \in \mathbb{N}$.

Algorithm 10 Transformaciones gaussianas sobre una matriz para conseguir la factorización LU (sobreescribe la matriz A).

Entrada: $A \in \mathbb{R}^{m \times n}$
Salida: $LU \in \mathbb{R}^{n \times n}$.

1: **procedure** $LU = GaussElimination(A)$
2: **while** $k < n - 1$ **and** $A(k, k \neq 0)$ **do**
3: $A(k+1:n,k) = A(k+1:n,k)/A(k,k)$
4: $A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k) \cdot A(k,k+1:n)$
5: $k = k + 1;$
6: **end while**
7: **end procedure**

Con el fin de paralelizar el algoritmo expuesto, debe limitarse la cantidad de dependencias que este presenta. La clave para ello es realizar el proceso a bloques, tal como se detalla en el algoritmo 11.

Algorithm 11 Factorización LU a bloques (con eliminación gaussiana)

Entrada: $A \in \mathbb{R}^{n \times n}$
Salida: $L, U \in \mathbb{R}^{n \times n}$.

1: **procedure** $L, U = BlockLU(A)$
2: $\lambda = 1$
3: **while do**$\lambda \leq n$
4: $\mu = min(n, \lambda + r - 1)$
5: Aplicar algoritmo 10 para sobreescribir $A(\lambda:\mu, \lambda:\mu)$ con su LU
6: Resolver $L_{A(\lambda:\mu,\lambda:\mu)} \cdot Z = A(\lambda:\mu, \mu+1:n)$; $A(\lambda:\mu, \mu+1:n) = Z$
7: Resolver $W \cdot U_{A(\lambda:\mu,\lambda:\mu)} = A(\lambda+1:n, \lambda:\mu)$; $A(\lambda+1:n, \lambda:\mu) = W$
8: $A(\mu+1:n, mu+1:n) = A(\mu+1:n, \mu+1:n) - W \cdot Z$
9: $\lambda = \mu + 1$
10: **end while**
11: **end procedure**

[1]Los multiplicadores M_k se almacenan en $A(k+1:n,k)$, es decir, $A(k+1:n,k) = -M_k(k+1:n,k)$.

En el caso de que las matrices no sean cuadradas, las modificaciones son ligeras, pero se mantiene la misma estructura en todos los algoritmos (esto puede verse con más detalle en [6]).

Antes de proceder a explicar la factorización LU sobre matrices jerárquicas, dado que es algo que surgirá en la implementación que he realizado, es importante destacar que se puede aplicar pivotamiento (concretamente, pivotamiento parcial) con el fin de que el cálculo de la LU sea lo más estable posible, evitando multiplicadores demasiado grandes.

Regla 3.1. *Si $A \in \mathbb{R}^{n \times n}$, entonces se puede aplicar el algoritmo 12 para las transformaciones Gaussianas $M_1, ..., M_{n-1}$ intercalando las permutaciones $E_1, ..., E_{n-1}$, de forma que $U = M_{n-1} \cdot E_{n-1} \cdot ... \cdot M_1 \cdot E_1 \cdot A$, siendo U triangular superior. Siguiendo este esquema no se aplican multiplicadores cuyo valor absoluto sea mayor que 1. Los valores $A(1 : k, k)$ se sobreescriben con $U(1 : k, k)$ con $k = 1 : n$, y los valores $A(k + 1 : n, k)$ se sobreescriben con $-M_k(k + 1 : n, k$. El vector de enteros $p(1 : n - 1)$ define las permutaciones; en particular, E_k intercambia las filas k y $p(k)$, con $k = 1 : n - 1$.*

Algorithm 12 Eliminación gaussiana con pivotamiento parcial

Entrada: $A \in \mathbb{R}^{n \times n}$

Salida: $L, U \in \mathbb{R}^{n \times n}$.

```
1: procedure L, U = GaussEliminationPartialPivoting(A)
2:     for k = 1 : n − 1 do
3:         Determinar μ con k ≤ μ ≤ n tal que |A(μ, k)| = ||A(k : n, k)||∞
4:         A(k, k : n) ↔ A(μ, k : n)
5:         p(k) = μ
6:         if A(k, k) ≠ 0 then
7:             A(k + 1 : n, k) = A(k + 1 : n, k)/A(k, k)
8:             A(k + 1 : n, k + 1 : n) = A(k + 1 : n, k + 1 : n) − A(k + 1 : n, k) · A(k, k + 1 : n)
9:         end if
10:    end for
11: end procedure
```

¿Dónde queda la matriz L ahora? Veamos el teorema que sigue.

Teorema 3.2. *Si se aplica la eliminación gaussiana con pivotamiento parcial para calcular la factorización LU mediante el algoritmo 12 detallado, de forma que:*

$$U = M_{n-1} \cdot E_{n-1} \cdot ... \cdot M_1 \cdot E_1 \cdot A$$

Entonces se tiene que $P \cdot A = L \cdot U$, donde $L = E_{n-1} \cdot ...E_1$ y L es una matriz triangular inferior unitaria y $|l_{ij}| \leq 1$. Además, la columna k-ésima de L (por debajo de la diagonal) contiene el vector de la eliminación gaussiana k-ésima permutado.

La demostración de este teorema puede verse en [6].

Cabe señalar que lo habitual será combinar el algoritmo a bloques con el de eliminación gaussiana con pivotamiento parcial para obtener el par de matrices L y U estables, sin valores extremadamente grandes (ni extremadamente pequeños) que las desfiguren.

3.2. La factorización LU de matrices jerárquicas: dependencia de tareas

En [7] se define por primera vez la factorización LU de matrices jerárquicas (a la que denominaré también $\mathcal{H}$-factorización LU) basada en tareas, para el caso de matrices densas. Kriemann, en [2] hace referencia a dicha definición y, además, ofrece una descripción detallada de las tareas que componen el proceso.

Esta sección se centrará pues, en presentar las tareas que componen el proceso de cálculo de la factorización LU de matrices jerárquicas, así como las dependencias existentes entre ellas.

La solución propuesta por [2] para las dependencias entre las tareas que conforman el proceso de cálculo de la factorización LU en matrices jerárquicas queda reflejada en el algoritmo 13.

En dicho algoritmo, deben tenerse en cuenta las siguientes equivalencias:

- FACTORIZE(t) equivale a LU($A|_{t \times t}$)
- SOLVE_UPPER(s,t) equivale a resolver $L|_{s \times t} \cdot U|_{t \times t} = A|_{s \times t}$, que es el proceso al que denomino TRSM sobre filas en mi implementación.
- SOLVE_LOWER(t,s) equivale a resolver $L|_{t \times t} \cdot U|_{t \times s} = A|_{t \times s}$, que es el proceso

al que denomino TRSM sobre columnas en mi implementación.

- UPDATE(r,t,s) equivale a $A|_{r\times s} := A|_{r\times s} - L|_{r\times t} \cdot U|_{t\times s}$

Algorithm 13 Algoritmo para el cálculo del Grafo Acíclico Dirigido que representa las dependencias entre las tareas que componen el proceso de cálculo de la factorización LU de una matriz jerárquica.

Entrada: $A|_{t\times t}$
Salida: $L|_{t\times t}, U|_{t\times t}$.

1: **procedure** $LU_DAG = LU_DAG(A)$
2: **for** $i = 0, ..., l-1$ **do**
3: $LU_DAG(A|_{t_i\times t_i})$
4: **task**(FACTORIZE(t_i))
5: **for** $s \in T_I^{level(t_i)}, s >_I t_i$ **do**
6: **if** $\{t_i \times t_i, s \times t_i\} \cap \mathcal{L}(T_{I\times I}) \neq \emptyset$ **then**
7: **task** (SOLVE_UPPER(s,t_i))
8: $\mathcal{T}_{solve} := \mathcal{T}_{solve} \cup \{s \times t_i\}$
9: **task**(FACTORIZE(t_i))→ **task**(SOLVE_UPPER(s, t_i))
10: **end if**
11: **if** $\{t_i \times t_i, t_i \times s\} \cap \mathcal{L}(T_{I\times I}) \neq \emptyset$ **then**
12: **task** (SOLVE_LOWER(t_i,s))
13: $\mathcal{T}_{solve} := \mathcal{T}_{solve} \cup \{t_i \times s\}$
14: **task**(FACTORIZE(t_i))→ **task**(SOLVE_UPPER(t_i,s))
15: **end if**
16: **end for**
17: **for** $r, s \in T_I^{level(t_i)}, r, s <_I$ **do**
18: **if** $\{r \times s, r \times t_i, t_i \times s\} \cap \mathcal{L}(T_{I\times I}) \neq \emptyset$ **then**
19: **task** (UPDATE(r,t_i,s))
20: $\mathcal{T}_{update}(r \times s) := \mathcal{T}_{update}(r \times s) \cup \{t_i\}$
21: **for all** $r' \times t' \in \mathcal{T}_{solve}, r' \times t' \subseteq r \times t_i$ **do**
22: **task**(SOLVE_UPPER(r',t'))→ **task**(UPDATE(r,t_i,s))
23: **end for**
24: **for all** $t' \times s' \in \mathcal{T}_{solve}, t' \times s' \subseteq t_i \times s$ **do**
25: **task**(SOLVE_LOWER(t',s'))→**task**(UPDATE(r,t_i,s))
26: **end for**
27: **end if**
28: **end for**
29: **end for**
30: **end procedure**

3.3. Mi implementación de la $\mathcal{H}$-factorización LU

Dado que en la referencia principal sobre la factorización LU de $\mathcal{H}$-matrices [2] no se especifica un esquema del algoritmo para la factorización LU de matrices jerárquicas, el algoritmo que se presenta en esta sección es propio. No obstante, en [2] se presentan versiones primitivas del mismo y, tal como se ha explicado en la sección anterior, un esquema detallado de cómo se forma el grafo acíclico de dependencia de las tareas que intervienen en la factorización LU, por lo que llegar al algoritmo 14 se ha hecho interpretando debidamente las versiones primitivas y dicho grafo de dependencias.

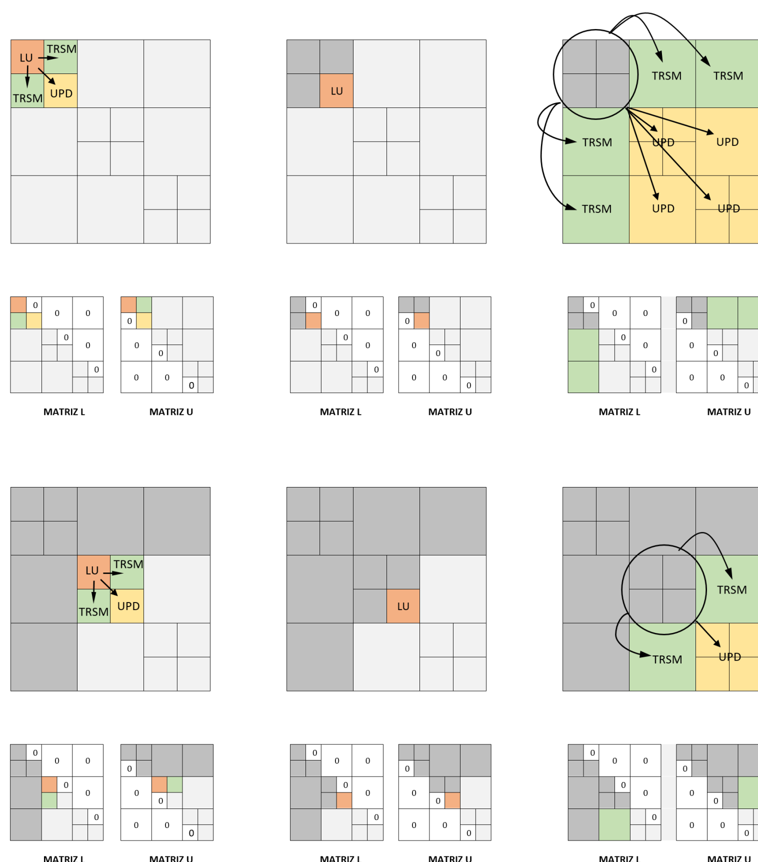
A grandes rasgos, el funcionamiento del algoritmo, una vez leída la información correspondiente a la matriz (número de niveles, tamaño de dichos niveles, número de bloques, inicios de dichos bloques y valores que forman la matriz), es el siguiente:

- Búsqueda del bloque más pequeño que hay en la diagonal de la matriz dada.
- Factorización LU de dicho bloque utilizando la rutina `dgetrf` de Lapack.
- Aplicar TRSM sobre los bloques que siguen al bloque sobre el que se acaba de hacer la LU, en su misma fila y de su mismo tamaño. La operación TRSM equivale a calcular, sobre dichos bloques, su matriz U correspondiente, resolviendo el sistema de ecuaciones que deriva de $U = L^{-1} \cdot A_{bloque}$, para evitar calcular la inversa. Dicho sistema de ecuaciones se resuelve con la rutina `dtrtrs` de Lapack, que calcula un sistema de ecuaciones a partir de una matriz triangular.
- Aplicar TRSM sobre los bloques que siguen al bloque sobre el que se acaba de hacer la LU, en su misma columna y de su mismo tamaño. La operación TRSM equivale a calcular, sobre dichos bloques, su matriz U correspondiente, resolviendo el sistema de ecuaciones que deriva de $L = A_{bloque} \cdot U^{-1}$, para evitar calcular la inversa. Dicho sistema de ecuaciones se resuelve de nuevo con la rutina `dtrtrs` de Lapack, que calcula un sistema de ecuaciones a partir de una matriz triangular (teniendo en cuenta que, en este caso, la matriz U debe transponerse para que, en la resolución del sistema, sea triangular inferior).
- Actualizar el contenido de los bloques de la matriz dada que quedan por debajo de la fila y a la derecha de la columa sobre las que se ha aplicado TRSM, es decir, se

actualiza el contenido de cada uno de los bloques diagonales que siguen al bloque sobre el que se ha calculado la LU y también a los bloques que hay en sus filas y columnas, a partir de ellos. Actualizar consiste en modificar el contenido de un sub-bloque de la matriz A de la forma: $A = A - L \cdot U$, siendo L y U los que se han calculado para su fila y columna en los TRSM inmediatamente anteriores. Para el producto de $L \cdot U$ en las actualizaciones, se utiliza la rutina `dgemm` de Blas.

- En el caso de que no se hayan aplicado TRSM, se busca si hay un bloque de tamaño mayor que contiene al bloque diagonal sobre el que se acaba de calcular la facotización LU, y que acabe en el mismo punto que él. Si es así, este bloque tiene ya, por operaciones que se han llevado a cabo anteriormente, sus L y U calculadas, y a partir de él, sobre los bloques de su mismo tamaño en la fila, columna y bloques restantes a partir del primer bloque siguiente en la diagonal, se aplica de nuevo el proceso de TRSM sobre filas, TRSM sobre columnas y actualizaciones.

Este proceso se repite para cada bloque diagonal, hasta que se completa el cálculo de todas las matrices L y U. Para aclarar más el funcionamiento del algoritmo, puede observarse el detalle de pasos que ejemplifica la Figura 3.1.



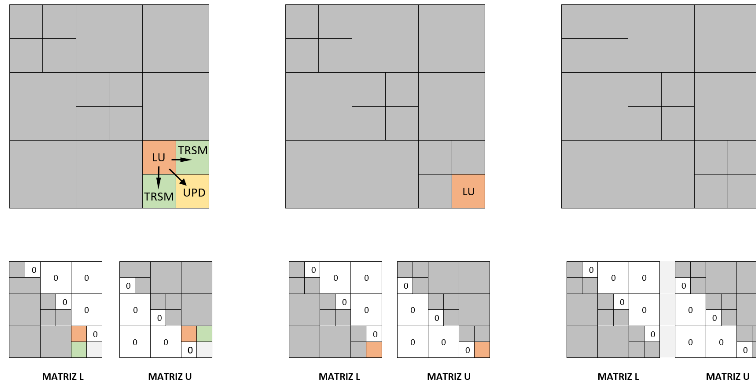


Figura 3.1: Detalle de los pasos que conforman el cálculo de la factorización LU sobre una $\mathcal{H}$-matriz ejemplo. En rojo las operaciones LU, en verde las TRSM, en amarillo las actualizaciones y, debajo, en cada paso, las modificaciones sobre las matrices L y U.

Cabe señalar que, provisionalmente, el almacenamiento de las matrices se ha hecho de forma "simpificada", tal como se detalla en la sección A.3 del anexo de este documento, pero en un futuro deberán probarse técnicas de almacenamiento de matrices dispersas.

Además, es importante resaltar que el algoritmo asume que el mayor particionado (aquel compuesto por los bloques con menos elementos) se da sobre los bloques diagonales de la matriz y, progresivamente, va disminuyendo sobre los bloques que ocupan las filas, de izquierda a derecha, y las columnas, de arriba a abajo. En el Capítulo 5 se explica que esta simplificación es temporal y, se añadirá en las funciones de TRSM, la capacidad de detectar si hay un particionado en sub-bloques sobre el bloque que actúan, actuando en consecuencia.

Asimismo, en la primera sección de este capítulo se ha predicho que el pivotamiento aparecería en esta implementación. Esto es cierto, puesto que Lapack aplica pivotamiento cuando calcula la LU mediante la rutina `dgetrf`. Para evitar los errores que derivan de obviar esto, tras cada uso de dicha rutina, es necesario contemplar el pivotamiento local que se aplica, lo cual conlleva aplicarlo también sobre los elementos de la matriz A que comparten fila con la LU calculada, así como aplicarlo también sobre la matriz que contiene las LU calculadas anteriormente, sometidas a un pivotamiento distinto. Además, tras multiplicar las matrices L y U resultado cuando concluye su cálculo, es necesario "deshacer" el pivotamiento sobre la matriz resultado del producto para que sus filas aparezcan en el orden original y esta coincida con la matriz dada.

En el anexo de este documento puede consultarse el código corespondiente a este algoritmo.

3.3.1. Paralelización del algoritmo sobre memoria compartida, mediante OpenMP

Para las primeras pruebas he paralelizado usando OpenMP aunque, como ya se ha explicado, no es el esquema de paralelización para el que se ha diseñado la implementación.

Las modificaciones sobre el código original han sido simples, consistiendo en añadir `#pragma omp parallel for private(...)` antes de los bucles `for` que distinguen sobre qué bloques en la fila y en la columna hay que aplicar las operaciones TRSM y también en las actualizaciones, de igual modo, precediendo al bucle que actualiza sobre la fila y sobre la columna de cada bloque diagonal.

Los resultados de la paralelización no se incluyen en la siguiente sección porque, tal como se ha hecho la implementación, es necesario recurrir a muchas secciones críticas, lo cual deriva en un beneficio escaso por paralelización. No obstante, estas pruebas han servido para proponer otra de las mejoras sobre el código: gestionar la memoria de modo distinto en los TRSM y actualizaciones para que no haya tantos conflictos.

3.4. Pruebas realizadas y resultados obtenidos

En primer lugar, es importante señalar que todas las pruebas se han realizado sobre el cluster Kahan de la UPV. Este es un cluster de pequeño tamaño, que se compone de:

- Frontend con un procesador Intel Core 2 Duo a 3GHz, con 4 GB de memoria.
- Seis nodos biprocesador conectados mediante una red Infiniband. Cada uno de ellos consta de:
  - Dos procesadores AMD Opteron 16 Core 6272, 2.1GHz, 16MB
  - 32GB de memoria DDR3 1600

- Disco 500GB, SATA 6 GB/s
- Controladora InfiniBand QDR 4X (40Gbps, con una tasa efectiva de 32Gbps)

Los nodos que componen el cluster están interconectados mediante una red Infiniband. En total se dispone de 12 procesadores, 192 núcleos y 192 GB. El pico teórico del rendimiento del sistema es de 2304 GFlops.

Sobre este cluster he hecho pruebas con ambos algoritmos, con tres matrices distintas cuya estructura detallada puede consultarse en el anexo de este documento.

Todos los resultados obtenidos han mostrado que el producto derivado de multiplicar la matriz L por la U calculadas coincide con la matriz original.

La compilación se lleva a cabo con `make`. El `Makefile` correspondiente contiene:

```
COMP=gcc
LIBS=-lm -llapack -lblas -lpthread
OMP=-fopenmp -liomp5

all: Sequential_LU OpenMP_LU

Sequential_LU: Sequential_LU.c
        $(COMP) $(LIBS) Sequential_LU.c -o Sequential_LU.o

OpenMP_LU: OpenMP_LU.c
        $(COMP) $(LIBS) $(OMP) OpenMP_LU.c -o OpenMP_LU.o

clean:
        rm -f *.o
```

A continuación muestro los resultados de las tres ejecuciones llevadas a cabo con el algoritmo secuencial.

- Ejecución 1: `./Sequential_LU.o matrix_1.txt`

```
Given file containing matrix information: matrix_1.txt

Obtaining matrix information... DONE

Obtaining levels indexes... DONE
```

```
Computing L and U matrices...

LU of block starting in index 0...
... end of LU
TRSM(s) (row) from block starting in index 0...
> TRSM number 1, of block starting in index 16
... end of TRSM (row)
TRSM(s) (col) from block starting in index 0...
> TRSM number 1, of block starting in index 2
... end of TRSM (col)
UPD from block starting in index 0...
> Updating block starting in index 18
... end of UPD

LU of block starting in index 18...
... end of LU
TRSM(s) (row) from block starting in index 18...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 18...
... end of TRSM (col)
No TRSM(s) have been performed --> Try TRSM(s) of bigger
levels blocks.
Prev level block start i: 0 (size = 4x4)
TRSM(s) (row) from block starting in index 0...
> TRSM number 1, of block starting in index 32
... end of TRSM (row)
TRSM(s) (col) from block starting in index 0...
> TRSM number 1, of block starting in index 4
... end of TRSM (col)
UPD from block starting in index 0...
> Updating block starting in index 36
... end of UPD

LU of block starting in index 36...
... end of LU
TRSM(s) (row) from block starting in index 36...
> TRSM number 1, of block starting in index 52
... end of TRSM (row)
TRSM(s) (col) from block starting in index 36...
> TRSM number 1, of block starting in index 38
... end of TRSM (col)
UPD from block starting in index 36...
> Updating block starting in index 54
... end of UPD

LU of block starting in index 54...
... end of LU
TRSM(s) (row) from block starting in index 54...
```

```
... end of TRSM (row)
TRSM(s) (col) from block starting in index 54...
... end of TRSM (col)
No TRSM(s) have been performed --> Try TRSM(s) of bigger
levels blocks.
Prev level block start i: 36 (size = 4x4)
TRSM(s) (row) from block starting in index 36...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 36...
... end of TRSM (col)
No TRSM(s) have been performed --> Try TRSM(s) of bigger
levels blocks.
Prev level block start i: 0 (size = 8x8)
TRSM(s) (row) from block starting in index 0...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 0...
... end of TRSM (col)

DONE

Computed matrix LU is:

9.000000 1.000000 1.000000 1.000000 -1.000000 1.000000 1.000000 10.000000
0.111111 9.888889 0.888889 0.888889 1.111111 0.888889 0.888889 -0.111111
0.111111 0.089888 8.808989 4.808989 1.011236 0.808989 0.808989 -0.101124
0.111111 -0.112360 0.112245 8.448980 1.122449 -1.102041 0.897959 -0.112245
0.111111 0.696629 0.030612 0.014493 8.289855 0.260870 0.231884 -0.028986
0.111111 0.089888 0.091837 0.043478 0.104895 7.755245 0.671329 -0.083916
0.111111 0.089888 0.091837 -0.193237 0.136946 0.062669 8.834385 0.895702
0.111111 0.089888 0.091837 0.043478 0.104895 0.097385 0.068590 7.862820

L * U =

9.000000 1.000000 1.000000 1.000000 -1.000000 1.000000 1.000000 10.000000
1.000000 10.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 9.000000 5.000000 1.000000 1.000000 1.000000 1.000000
1.000000 -1.000000 1.000000 9.000000 1.000000 -1.000000 1.000000 1.000000
1.000000 7.000000 1.000000 1.000000 9.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 8.000000 1.000000 1.000000
1.000000 1.000000 1.000000 -1.000000 1.000000 1.000000 9.000000 2.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 9.000000

Free memory... DONE
```

- Ejecución 2: `./Sequential_LU.o matrix_2.txt`

```
Given file containing matrix information: matrix_2.txt
```

```
Obtaining matrix information... DONE

Obtaining levels indexes... DONE

Computing L and U matrices...

LU of block starting in index 0...
... end of LU
TRSM(s) (row) from block starting in index 0...
> TRSM number 1, of block starting in index 36
... end of TRSM (row)
TRSM(s) (col) from block starting in index 0...
> TRSM number 1, of block starting in index 3
... end of TRSM (col)
UPD from block starting in index 0...
> Updating block starting in index 39
... end of UPD

LU of block starting in index 39...
... end of LU
TRSM(s) (row) from block starting in index 39...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 39...
... end of TRSM (col)
No TRSM(s) have been performed --> Try TRSM(s) of bigger levels blocks.
Prev level block start i: 0 (size = 6x6)
TRSM(s) (row) from block starting in index 0...
> TRSM number 1, of block starting in index 72
... end of TRSM (row)
TRSM(s) (col) from block starting in index 0...
> TRSM number 1, of block starting in index 6
... end of TRSM (col)
UPD from block starting in index 0...
> Updating block starting in index 78
... end of UPD

LU of block starting in index 78...
... end of LU
TRSM(s) (row) from block starting in index 78...
> TRSM number 1, of block starting in index 114
... end of TRSM (row)
TRSM(s) (col) from block starting in index 78...
> TRSM number 1, of block starting in index 81
... end of TRSM (col)
UPD from block starting in index 78...
> Updating block starting in index 117
... end of UPD

LU of block starting in index 117...
```

```
... end of LU
TRSM(s) (row) from block starting in index 117...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 117...
... end of TRSM (col)
No TRSM(s) have been performed --> Try TRSM(s) of bigger levels blocks.
Prev level block start i: 78 (size = 6x6)
TRSM(s) (row) from block starting in index 78...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 78...
... end of TRSM (col)
No TRSM(s) have been performed --> Try TRSM(s) of bigger levels blocks.
Prev level block start i: 0 (size = 12x12)
TRSM(s) (row) from block starting in index 0...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 0...
... end of TRSM (col)

DONE

Computed matrix LU is:

43.000000 38.000000 81.000000 99.000000 89.000000 99.000000 61.000000 89.000000 78.000000
83.000000 78.000000 44.000000
0.069767 20.348837 86.348837 74.093023 47.790698 35.093023 37.744186 28.790698 87.558140
12.209302 50.558140 47.930233
0.441860 -0.382857 72.268571 -12.377143 68.971429 64.691429 63.497143 17.697143 4.057143
45.000000 16.891429 98.908571
1.976744 -3.445714 2.759390 174.759073 -179.576184 -176.285997 -106.739069 -112.559026 209.319048
-227.172531 -4.587175 -104.750138
1.697674 0.024000 -0.796806 -0.719334 -163.458471 -176.174054 -71.650184 -176.650374 57.283102
-174.756718 -105.472527 9.612544
0.883721 -0.716571 0.405345 -0.116609 0.485157 75.351600 25.716089 29.383852 59.783972
-13.546300 45.087295 -12.508760
0.418605 2.510857 -3.081790 -0.982053 0.673255 0.534798 77.042264 16.671157 -36.970049 -4.914464
-2.147140 156.398706
1.348837 -0.798857 0.189911 -0.371910 0.958978 0.698040 -0.572709 89.170947 -49.023520 27.995531
89.708975 50.283604
1.720930 0.373714 -1.572816 -0.678240 1.011507 0.144626 0.378334 -0.149196 -55.520115 -35.938037
36.690942 -54.696875
2.232558 -2.252571 0.535083 -0.260362 0.689508 -0.942431 0.787975 0.378348 -3.385147 -171.671184
147.684479 -324.300160
2.162791 -3.891429 2.543765 1.056861 -0.268114 -0.664076 0.806078 0.417207 -1.257208 0.641519
-86.722191 -1.037790
0.627907 1.923429 -2.365431 -0.943886 0.368033 -0.122024 0.052641 -1.199057 0.321345 0.381788
-0.152530 275.498825

L * U =
```

```
3.000000 23.000000 92.000000 81.000000 54.000000 42.000000 42.000000 35.000000 93.000000 18.000000
56.000000 51.000000
43.000000 38.000000 81.000000 99.000000 89.000000 99.000000 61.000000 89.000000 78.000000 83.000000
78.000000 44.000000
19.000000 9.000000 75.000000 3.000000 90.000000 95.000000 76.000000 46.000000 5.000000 77.000000
32.000000 100.000000
73.000000 65.000000 82.000000 54.000000 63.000000 68.000000 59.000000 42.000000 38.000000 94.000000
18.000000 82.000000
38.000000 19.000000 39.000000 9.000000 14.000000 99.000000 56.000000 22.000000 71.000000 11.000000
34.000000 49.000000
85.000000 5.000000 62.000000 81.000000 22.000000 77.000000 59.000000 13.000000 73.000000 19.000000
22.000000 90.000000
74.000000 73.000000 58.000000 99.000000 19.000000 34.000000 52.000000 31.000000 23.000000 10.000000
52.000000 14.000000
58.000000 35.000000 54.000000 7.000000 5.000000 67.000000 9.000000 73.000000 27.000000 49.000000
91.000000 40.000000
18.000000 67.000000 28.000000 94.000000 11.000000 25.000000 72.000000 79.000000 68.000000 20.000000
63.000000 93.000000
96.000000 39.000000 25.000000 2.000000 62.000000 30.000000 100.000000 70.000000 48.000000 90.000000
11.000000 92.000000
27.000000 63.000000 46.000000 69.000000 94.000000 69.000000 36.000000 1.000000 63.000000 10.000000
40.000000 72.000000
93.000000 3.000000 23.000000 79.000000 36.000000 53.000000 98.000000 85.000000 24.000000 5.000000
6.000000 62.000000

Free memory... DONE
```

- Ejecución 3: `./Sequential_LU.o matrix_3.txt`

```
Given file containing matrix information: matrix_3.txt

Obtaining matrix information... DONE

Obtaining levels indexes... DONE

Computing L and U matrices...

LU of block starting in index 0...
... end of LU
TRSM(s) (row) from block starting in index 0...
> TRSM number 1, of block starting in index 24
... end of TRSM (row)
TRSM(s) (col) from block starting in index 0...
> TRSM number 1, of block starting in index 2
... end of TRSM (col)
UPD from block starting in index 0...
> Updating block starting in index 26
... end of UPD
```

```
LU of block starting in index 26...
... end of LU
TRSM(s) (row) from block starting in index 26...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 26...
... end of TRSM (col)
No TRSM(s) have been performed --> Try TRSM(s) of bigger levels blocks.
Prev level block start i: 0 (size = 4x4)
TRSM(s) (row) from block starting in index 0...
> TRSM number 1, of block starting in index 48
> TRSM number 2, of block starting in index 96
... end of TRSM (row)
TRSM(s) (col) from block starting in index 0...
> TRSM number 1, of block starting in index 4
> TRSM number 2, of block starting in index 8
... end of TRSM (col)
UPD from block starting in index 0...
> Updating block starting in index 52
diag_start = 52
U_idx = 48 49 50 51 60 61 62 63 72 73 74 75 84 85 86 87
L_idx = 4 5 6 7 16 17 18 19 28 29 30 31 40 41 42 43
> Updating block starting in index 100
diag_start = 52
U_idx = 48 49 50 51 60 61 62 63 72 73 74 75 84 85 86 87
L_idx = 4 5 6 7 16 17 18 19 28 29 30 31 40 41 42 43
> Updating block starting in index 56
> Updating block starting in index 104
... end of UPD

LU of block starting in index 52...
... end of LU
TRSM(s) (row) from block starting in index 52...
> TRSM number 1, of block starting in index 76
... end of TRSM (row)
TRSM(s) (col) from block starting in index 52...
> TRSM number 1, of block starting in index 54
... end of TRSM (col)
UPD from block starting in index 52...
> Updating block starting in index 78
... end of UPD

LU of block starting in index 78...
... end of LU
TRSM(s) (row) from block starting in index 78...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 78...
... end of TRSM (col)
No TRSM(s) have been performed --> Try TRSM(s) of bigger levels blocks.
```

```
Prev level block start i: 52 (size = 4x4)
TRSM(s) (row) from block starting in index 52...
> TRSM number 1, of block starting in index 100
... end of TRSM (row)
TRSM(s) (col) from block starting in index 52...
> TRSM number 1, of block starting in index 56
... end of TRSM (col)
UPD from block starting in index 52...
> Updating block starting in index 104
... end of UPD

LU of block starting in index 104...
... end of LU
TRSM(s) (row) from block starting in index 104...
> TRSM number 1, of block starting in index 128
... end of TRSM (row)
TRSM(s) (col) from block starting in index 104...
> TRSM number 1, of block starting in index 106
... end of TRSM (col)
UPD from block starting in index 104...
> Updating block starting in index 130
... end of UPD

LU of block starting in index 130...
... end of LU
TRSM(s) (row) from block starting in index 130...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 130...
... end of TRSM (col)
No TRSM(s) have been performed --> Try TRSM(s) of bigger levels blocks.
Prev level block start i: 104 (size = 4x4)
TRSM(s) (row) from block starting in index 104...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 104...
... end of TRSM (col)
No TRSM(s) have been performed --> Try TRSM(s) of bigger levels blocks.
Prev level block start i: 0 (size = 12x12)
TRSM(s) (row) from block starting in index 0...
... end of TRSM (row)
TRSM(s) (col) from block starting in index 0...
... end of TRSM (col)

DONE

Computed matrix LU is:

43.000000 38.000000 81.000000 99.000000 89.000000 99.000000 61.000000 89.000000 78.000000
83.000000 78.000000 44.000000
0.069767 20.348837 86.348837 74.093023 47.790698 35.093023 37.744186 28.790698 87.558140
```

```
12.209302 50.558140 47.930233
0.441860 -0.382857 72.268571 -12.377143 68.971429 64.691429 63.497143 17.697143 4.057143
45.000000 16.891429 98.908571
1.697674 0.024000 -0.796806 -125.710176 -34.283182 -49.365510 5.130877 -95.682818 -93.287246
-11.343750 -102.172816 84.962900
1.976744 -3.445714 2.759390 -1.390174 -227.235787 -244.912666 -99.606255 -245.574833 79.633504
-242.942322 -146.625211 13.363114
0.883721 -0.716571 0.405345 0.162106 0.232382 75.351600 25.716089 29.383852 59.783972 -13.546300
45.087295 -12.508760
1.348837 -0.798857 0.189911 0.517019 0.317916 0.698040 -44.122833 79.623218 -27.850423 30.810091
90.938662 -39.287415
1.720930 0.373714 -1.572816 0.942873 0.049371 0.144626 -0.660603 45.602729 -80.591131 -21.620900
82.568823 -28.981441
2.232558 -2.252571 0.535083 0.361949 0.235625 -0.942431 -1.375873 3.430179 378.387683 73.258252
-102.376256 48.476872
0.418605 2.510857 -3.081790 1.365225 -0.497757 0.534798 -1.746086 3.414273 0.500970 86.002080
-73.985465 162.464589
2.162791 -3.891429 2.543765 -1.469221 0.863997 -0.664076 -1.407481 3.567969 0.707989
0.132682 -86.722191 -1.037790
0.627907 1.923429 -2.365431 1.312167 -0.679148 -0.122024 -0.091916 -2.164889 -0.364800
-1.490286 -0.152530 275.498825

L * U =

3.000000 23.000000 92.000000 81.000000 54.000000 42.000000 42.000000 35.000000 93.000000 18.000000
56.000000 51.000000
43.000000 38.000000 81.000000 99.000000 89.000000 99.000000 61.000000 89.000000 78.000000 83.000000
78.000000 44.000000
19.000000 9.000000 75.000000 3.000000 90.000000 95.000000 76.000000 46.000000 5.000000 77.000000
32.000000 100.000000
73.000000 65.000000 82.000000 54.000000 63.000000 68.000000 59.000000 42.000000 38.000000 94.000000
18.000000 82.000000
38.000000 19.000000 39.000000 9.000000 14.000000 99.000000 56.000000 22.000000 71.000000 11.000000
34.000000 49.000000
85.000000 5.000000 62.000000 81.000000 22.000000 77.000000 59.000000 13.000000 73.000000 19.000000
22.000000 90.000000
74.000000 73.000000 58.000000 99.000000 19.000000 34.000000 52.000000 31.000000 23.000000 10.000000
52.000000 14.000000
58.000000 35.000000 54.000000 7.000000 5.000000 67.000000 9.000000 73.000000 27.000000 49.000000
91.000000 40.000000
18.000000 67.000000 28.000000 94.000000 11.000000 25.000000 72.000000 79.000000 68.000000 20.000000
63.000000 93.000000
96.000000 39.000000 25.000000 2.000000 62.000000 30.000000 100.000000 70.000000 48.000000 90.000000
11.000000 92.000000
27.000000 63.000000 46.000000 69.000000 94.000000 69.000000 36.000000 1.000000 63.000000 10.000000
40.000000 72.000000
93.000000 3.000000 23.000000 79.000000 36.000000 53.000000 98.000000 85.000000 24.000000 5.000000
6.000000 62.000000
Free memory... DONE
```

Algorithm 14 Algoritmo secuencial para la factorización LU de matrices jerárquicas

Entrada: $A \in \mathbb{H}^{I \times I}$
Salida: $L, U \in \mathbb{H}^{I \times J}$.

```
 1: procedure L,U = HLU(A)
 2:     for t × s in diag(A) do
 3:         trsms_row = 0; trsms_col = 0;
 4:         Lapack_LU(t × s)
 5:         for each next block t' × s' of size == size(t × s) in its row do
 6:             TRSM(t' × s') ; trsms_row ++;
 7:         end for
 8:         for each next block t' × s' of size == size(t × s) in its column do
 9:             TRSM(t' × s') ; trsms_col ++;
10:         end for
11:         if (min(trsms_row, trsms_col)>0) then
12:             for each next block t' × s' in diagonal do
13:                 TRSM(t' × s')
14:                 for each next block t' × s' of size == size(t × s) in its row do
15:                     TRSM(t' × s')
16:                 end for
17:                 for each next block t' × s' of size == size(t × s) in its column do
18:                     TRSM(t' × s')
19:                 end for
20:             end for
21:         elseIf there's a bigger block t' × s' containing current diagonal block that is
    a leaf
22:             trsms_row = 0; trsms_col = 0;
23:             for each next block t'' × s'' of size == size(t' × s') in its row do
24:                 TRSM(t'' × s'') ; trsms_row ++;
25:             end for
26:             for each next block t'' × s'' of size == size(t' × s') in its column do
27:                 TRSM(t'' × s'') ; trsms_col ++;
28:             end for
29:             if (min(trsms_row, trsms_col)>0) then
30:                 for each next block t'' × s'' in diagonal do
31:                     TRSM(t'' × s'')
32:                     for each next block t'' × s'' of size == size(t' × s') in its row do
33:                         TRSM(t'' × s'')
34:                     end for
35:                     for each next block t'' × s'' of size == size(t' × s') in its column
    do
36:                         TRSM(t'' × s'')
37:                     end for
38:                 end for
39:             end if
40:         end if
41:     end for
42: end procedure
```

Capítulo 4

Conclusiones

En primer lugar, debo decir que estoy satisfecha con los resultados obtenidos durante la realización de este Trabajo de Fin de Máster. Si bien es cierto que me hubiese gustado disponer de más tiempo para completar los aspectos que han quedado pendientes y poder presentar comparativas del rendimiento del algoritmo implementado con distintas paralelizaciones, la complejidad de los algoritmos sobre matrices jerárquicas es mayor que la de los algoritmos convencionales, y ha sido especialmente gratificante lograr una implementación válida de la factorización LU.

Gracias a las competencias adquiridas en el máster, he podido aplicar técnicas de paralelismo como loop-unroll que me han permitido mejorar el rendimiento de mi algoritmo. Además, haber trabajado en varias asignaturas con las librerías Blas [8] y Lapack [9] ha ayudado a simplificar el código y también a reducir sustancialmente los tiempos de resolución de procesos como el producto de matrices o la resolución de sistemas de ecuaciones.

También cabe señalar que, a mi llegada al máster, eran escasos los códigos que había programado en C, por lo que notaba una cierta "torpeza" al programar en este lenguaje, que ha ido desapareciendo y me ha permitido traducir mi código, inicialmente en Matlab, a C sin grandes retrasos ni dificultades.

Aunque no haya podido comparar el rendimiento de mi implementación con otras técnicas de paralelismo, sí que es cierto que [1] y [2] evidencia buenos resultados en el uso de

los algoritmos específicos de matrices jerárquicas, lo cual, combinado con que he podido obtener factorizaciones LU correctas con mi código, me anima a continuar trabajando en ello durante mi doctorado, pues este tipo de matrices se postulan como una buena herramienta ante matrices con grandes cantidades de datos y un alto índice de dispersión en los mismos.

Con este TFM concluyen mis estudios del Máster en Computación Paralela y Distribuida, que me han aportado un bagaje que no tenía en este campo y me han servido para reafirmar mi motivación a realizar mi doctorado. Muy especialmente el trabajo de investigación e implementación que ha hecho posible este TFM, ha evidenciado que mi capacidad de análisis de textos científicos, así como mi habilidad para desarrollar nuevos algoritmos, ha crecido durante estos meses. Así pues, este TFM es el fin de mis estudios de máster pero el inicio de los que deseo y espero que sean cuatro años intensos e interesantes dedicados a estudiar y desarrollar algoritmos en torno a las matrices jerárquicas y sus ámbitos de aplicación.

Capítulo 5

Trabajo futuro

El objetivo a lograr durante mis primeros meses de doctorado es conseguir una implementación robusta de la factorización LU para matrices jerárquicas que funcione ante cualquier partición válida y, además, de forma paralela. Por tanto, el trabajo futuro más inmediato se centrará en mejorar el algoritmo que presento en este documento.

Por un lado, es necesario ampliar la funcionalidad del algoritmo actual, de forma que, recibiendo un vector que funcione como "mapa de bits", en el momento de aplicar TRSM o una actualización sobre un bloque, se pueda detectar si dicho bloque tiene todos sus elementos nulos, en cuyo caso no se llevará a cabo ninguna acción sobre él. Además, tampoco se almacenarán sus valores, consiguiéndose así las ventajas de almacenamiento que ofrece el uso de matrices jerárquicas.

Por otro lado, si cuando se aplica TRSM o actualización sobre un bloque, este contiene sub-bloques, pueden aplicarse las operaciones de TRSM o actualización en cada uno de dichos sub-bloques, permitiendo un mayor paralelismo (ver Figura 5.1). Esta mejora permite iniciar antes parte de los TRSM sobre el siguiente bloque, aumentando el paralelismo, y también fragmentar más las tareas, con el mismo objetivo.

Asimismo, una vez comprobado el correcto funcionamiento del algoritmo, es conveniente que la LU sobreescriba progresivamente a la matriz A dada, con el fin de ahorrar espacio. Actualmente esto no es así para poder realizar comprobaciones al finalizar la ejecución.

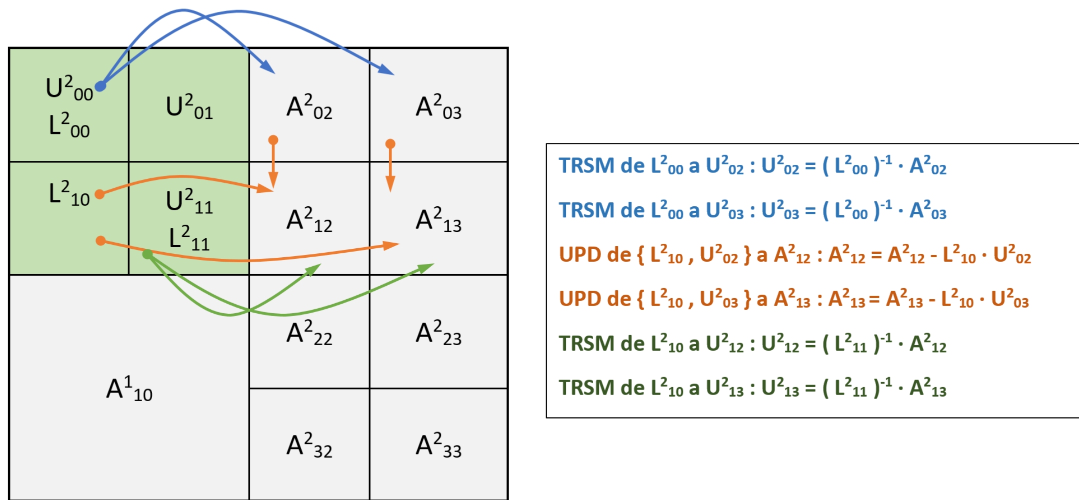


Figura 5.1: A la izquierda, el esquema sobre una matriz jerárquica del proceso de TRSM desde el bloque A^1_{00} (formado por cuatro sub-bloques $A^2_{00}, A^2_{01}, A^2_{10}, A^2_{11}$, cuyas L, U ya han sido calculadas), al bloque A^1_{01} (también formado por cuatro sub-bloques $A^2_{02}, A^2_{03}, A^2_{12}, A^2_{13}$, cuyas U's se van a calcular). A la derecha, por colores, el desglose de operaciones (y su orden) a realizar en cada sub-bloque, en lugar de aplicar el TRSM directamente de la forma $U^1_{01} = (L^1_{00})^{-1} \cdot A^1_{01}$.

Además, el algoritmo está actualmente paralelizado mediante OpenMP, pero el objetivo es paralelizarlo empleando otras técnicas, como por ejemplo OmpSs [13], que permiten un mayor paralelismo (ver Figura 5.2). También es deseable comprobar, con una misma técnica de paralelización, el mayor grado de paralelismo que se puede conseguir sin incurrir en incorrecciones en el resultado; de hecho, actualmente mediante OpenMP se ha implementado una paralelización básica que debe mejorarse, o bien englobando todas las operaciones de TRSM sobre fila y columna en una misma paralelización, o bien paralelizando por un lado los TRSM que se aplican sobre la fila y, por otro, los TRSM sobre las columas propagándose a su vez las actualizaciones conjuntamente.

Cabe añadir que la paralelización en OpenMP ha permitido detectar que debe modificarse el acceso a memoria en las funciones de TRSM y actualización para reducir los conflictos en los accesos llevados a cabo por distintos hilos, permitiendo así que la paralelización derive en beneficios perceptibles.

Cuando se hayan aplicado todas las mejoras y se disponga de alternativas de paralelización, se deben llevar a cabo pruebas con matrices de distintos tamaños, con particionados también diferentes y variando los grados de dispersión de las mismas, con el fin de poder comparar el rendimiento de cada versión.

<table>
<tr><td>1</td><td>2</td><td>4</td><td>4</td><td colspan="2" rowspan="2">9</td><td colspan="2" rowspan="2">9</td></tr>
<tr><td>2</td><td>3</td><td>5</td><td>5</td></tr>
<tr><td>4</td><td>5</td><td>6</td><td>7</td><td>10</td><td>10</td><td colspan="2" rowspan="2">10</td></tr>
<tr><td>4</td><td>5</td><td>7</td><td>8</td><td>11</td><td>11</td></tr>
<tr><td colspan="2" rowspan="2">9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>15</td><td>15</td></tr>
<tr><td>10</td><td>11</td><td>13</td><td>14</td><td>16</td><td>16</td></tr>
<tr><td colspan="2" rowspan="2">9</td><td colspan="2" rowspan="2">10</td><td>15</td><td>16</td><td>17</td><td>18</td></tr>
<tr><td>15</td><td>16</td><td>18</td><td>19</td></tr>
</table>

<table>
<tr><td>1</td><td>2</td><td>3</td><td>3</td><td colspan="2" rowspan="2">5</td><td colspan="2" rowspan="2">7</td></tr>
<tr><td>2</td><td>3</td><td>4</td><td>4</td></tr>
<tr><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>7</td><td colspan="2" rowspan="2">8</td></tr>
<tr><td>3</td><td>4</td><td>6</td><td>7</td><td>8</td><td>8</td></tr>
<tr><td colspan="2" rowspan="2">5</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>11</td></tr>
<tr><td>7</td><td>8</td><td>10</td><td>11</td><td>12</td><td>12</td></tr>
<tr><td colspan="2" rowspan="2">7</td><td colspan="2" rowspan="2">8</td><td>11</td><td>12</td><td>13</td><td>14</td></tr>
<tr><td>11</td><td>12</td><td>14</td><td>15</td></tr>
</table>

Figura 5.2: A la izquierda, el orden que actualmente sigue la paralelización con OpenMP (siempre y cuando se ejecuten en paralelo las TRSM sobre la fila y la columna a la vez), a la derecha el orden que puede lograrse si se ejecuta en paralelo con OmpSs [13] (se aprecia mayor paralelismo).

Asimismo, entre otros, una vez lograda una implementación robusta de la factorización LU que incluya las modificaciones detalladas, puede ser interesante que mi trabajo se encamine a implementar las operaciones del Capítulo 2 si los beneficios de paralelizar OmpSs son sustanciales, o a implementar otras operaciones como, por ejemplo, la inversa de matrices jerárquicas (ver [1]).

Por último, cabe destacar que el hilo conductor durante mi doctorado serán los algoritmos sobre grafos para aplicaciones de procesamiento de grandes volúmenes de datos. Esto motivó elegir trabajar con las matrices jerárquicas en mi TFM, pues parecen susceptibles de aplicarse en ese ámbito.

Bibliografía

[1] R. Kriemann, “*Parallel $\mathcal{H}$ matrix arithmetics on shared memory systems*”, Computing 74, 2005, pp. 273-297.

[2] R. Kriemann, “*$\mathcal{H}$-LU Factorization on Many-Core Systems*”, Computing and Visualization in Science, vol. 16, num. 3, 2013, pp. 105-117.

[3] S. Börm, L. Grasedyck, W. Hackbusch, “*Hierarchical matrices*”, Technical report, Lecture Note 21, MPI Leipzig, 2003.

[4] W. Hackbusch, “*A sparse matrix arithmetic based on $\mathcal{H}$-Matrices. Part I: Introduction to $\mathcal{H}$-Matrices*”, Computing 62, 1999, pp. 89-108.

[5] L. Grasedyck, W. Hackbrusch, “*Construction and arithmetics of $\mathcal{H}$-matrices*”, Computing 70, 2003, pp. 295-334.

[6] G. H. Golub, C. F. Van Loan, “*Matrix Computations*”, John Hopkins Studies in Mathematical Sciences, 3 edition, 1996.

[7] M. Izadi, “*Hierarchical Matrix Tchniques on Massively Parallel Computers*”, PhD thesis, University of Leipzig, 2012.

[8] Web oficial de la librería Blas, http://www.netlib.org/blas/ [Última consulta: 11 de julio, 2016]

[9] Web oficial de la librería Lapack para las rutinas con doble precisión, http://www.netlib.org/lapack/double/ [Última consulta: 11 de julio, 2016]

[10] Web oficial de la interfaz OpenMP, http://openmp.org/ [Última consulta: 11 de julio, 2016]

[11] Web oficial de la librería HLib, www.hlib.org/ [Última consulta: 11 de julio, 2016]

[12] Web oficial de la librería HLibPro, http://www.hlibpro.com/doc/2.4/index.html [Última consulta: 11 de julio, 2016]

[13] Web oficial de OmpSs, https://pm.bsc.es/ompss [Última consulta: 11 de julio, 2016]

[14] Web oficial de la libreía BEM++, https://github.com/bempp/bempp [Última consulta: 11 de julio, 2016]

[15] Web oficial de la librería H2Lib, http://www.h2lib.org/ [Última consulta: 11 de julio, 2016]

[16] Repositorio con la librería AHMED, https://github.com/xantares/ahmed, [Última consulta: 11 de julio, 2016]

[17] Repositorio con la librería DMHM, https://bitbucket.org/poulson/dmhm, [Última consulta: 11 de julio, 2016]

[18] Web oficial de la librería STRUMPACK, http://portal.nersc.gov/project/sparse/strumpack/, [Última consulta: 11 de julio, 2016]

[19] Repositorio con la librería Dense_HODLR, https://github.com/amiraa127/Dense_HODLR, [Última consulta: 11 de julio, 2016]

[20] Web oficial de la librería Misc, http://amath.colorado.edu/faculty/martinss/2014_CBMS/codes.html, [Última consulta: 11 de julio, 2016]

[21] Web oficial con la librería HSS http://www.math.purdue.edu/x̃iaj/, [Última consulta: 11 de julio, 2016]

[22] Repositorio con la librería RSVDPACK, https://github.com/sergeyvoronin/LowRankSVDCodes

[23] Repositorio con la librería HMat, https://github.com/YingzhouLi/HMat, [Última consulta: 11 de julio, 2016]

[24] Web oficial con la librería SSS_Toeplitz, http://www.math.purdue.edu/x̃iaj/frame-right.html, [Última consulta: 11 de julio, 2016]

[25] Repositorio con la librería H2TOOLS, https://bitbucket.org/muxas/h2tools, [Última consulta: 11 de julio, 2016]

Apéndice A

Código implementado

En este anexo presento el código implementado, eliminando el contenido de algunas funciones triviales (como la lectura de las matrices desde fichero o los “print” de matrices y vectores que se utilizan para comprobar el resultado) y también las matrices que he usado para las pruebas, con una imagen que muestra a qué equivale el contenido del fichero que las representa.

Todo el código implementado durante la realización de este TFM puede obtenerse en mi repositorio “TFM” en GitHub, en la dirección https://github.com/rociocs/TFM. Asimismo, las actualizaciones con las mejoras que vaya haciendo sobre él también irán apareciendo progresivamente.

A.1. Factorización LU: versión secuencial

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stddef.h>
4 #include <string.h>
5 #include <math.h>
6 #include <sys/time.h>
7 #include <malloc.h>
```

```
8   #include <mkl.h>
9
10  void PrintVector(int *v, int size)
11  {
12          ...
13  }
14
15  void PrintMatrix(double *A, int size)
16  {
17          ...
18  }
19
20  void ObtainMatrixData(char *file_name, int *n_levels, int **
       levels_sizes, int *n_blocks, int **block_indexes, double **A)
21  {
22          ...
23  }
24
25  void GetBlocksIndexes( int **levels_start_i, int **levels_end_i,
        int n_levels, int n_blocks, int *block_indexes)
26  {
27      int i, j;
28      *levels_start_i = (int *) malloc(n_levels * sizeof(int));
29      *levels_end_i = (int *) malloc(n_levels * sizeof(int));
30      (*levels_start_i)[0] = 0;
31      j = 1;
32      for (i=1; i<n_blocks; i++)
33      {
34          if(block_indexes[i] == 0)
35          {
36              (*levels_start_i)[j] = i;
37              (*levels_end_i)[j-1] = i-1;
38              j++;
39          }
```

```
40      }
41      (*levels_end_i)[j-1] = i-1;
42  }
43  
44  void GetIndexes(int **indexes, int start, int size, int big_size
    )
45  {
46      int i, j, arrayIndex;
47      arrayIndex = 0;
48      *indexes = (int *) malloc((size*size) * sizeof(int));
49      for (i=0; i<size; i++)
50      {
51          j = start + big_size*i;
52          while (j <= start + big_size*i + size -1)
53          {
54              (*indexes)[arrayIndex] = j;
55              j++;
56              arrayIndex++;
57          }
58      }
59  }
60  
61  void ApplyPivot(double **A, int *ipiv, int n, int m, int resto)
62  {
63      int i, j; double aux;
64      for(i=0; i<n; i++)
65      {
66          if((ipiv[i]-1) != i)
67          {
68              for(j=0; j<m; j++)
69              {
70                  aux = (*A)[(i+resto)+m*j];
71                  (*A)[(i+resto)+m*j] = (*A)[(ipiv[i]-1)+m*j];
72                  (*A)[(ipiv[i]-1)+m*j] = aux;
```

```
73              }
74          }
75      }
76  }
77
78  void ReversePivot(double **A, int *ipiv, int n, int m, int resto
    )
79  {
80      int i, j; double aux;
81      for(i=n-1; i>=0; i--)
82      {
83          if((ipiv[i]-1) != i)
84          {
85              for(j=0; j<m; j++)
86              {
87                  aux = (*A)[(i+resto)+m*j];
88                  (*A)[(i+resto)+m*j] = (*A)[(ipiv[i]-1)+m*j];
89                  (*A)[(ipiv[i]-1)+m*j] = aux;
90              }
91          }
92      }
93  }
94
95  void TRSM_row( int first_block_start_i, int size_bl_start, int
    bl_start_lvl, int *first_block_indexes, int n_levels, int *
    levels_sizes, int *block_indexes, double *A, double **LU, int
     *levels_start_i, int *levels_end_i, int *TRSM_performed )
96  {
97      printf("\tTRSM(s) (row) from block starting in index %d...\n
        ", first_block_indexes[0]);
98      int i, j, k, next_in_row, next_row_start, *trsm_bl_indexes,
        N, NRHS, LDA, INFO;
99      double *aux_A, *aux_L;
100
```

```
101     // Lapack variables
102     char UPLO = 'L', TRANS = 'N', DIAG = 'U';   // It is
           unitdiagonal
103     N = size_bl_start; NRHS = N; LDA = N; INFO = 0;
104
105     next_in_row = 1;
106     aux_A = malloc(size_bl_start * size_bl_start * sizeof(double
           ));
107     aux_L = malloc(size_bl_start * size_bl_start * sizeof(double
           ));
108
109     for (i=1; i<levels_end_i[bl_start_lvl-1]-levels_start_i[
           bl_start_lvl-1]+1; i++) // All blocks of this size except
            1st
110     {
111         if(i==1)
112             for (j=0; j<size_bl_start; j++)
113                 for (k=j+1; k<size_bl_start; k++)
114                     aux_L[ j*size_bl_start + k ] = (*LU)[
                           first_block_indexes[ j*size_bl_start + k
                           ] ];
115         next_row_start = first_block_start_i + levels_sizes[0] *
                size_bl_start * next_in_row;
116         if(block_indexes[levels_start_i[bl_start_lvl-1]+i] ==
               next_row_start)
117         {
118             next_in_row ++;
119             GetIndexes(&trsm_bl_indexes, next_row_start,
                   size_bl_start, levels_sizes[0]);
120             printf("\t\t> TRSM number %d, of block starting in
                   index %d\n", next_in_row-1, trsm_bl_indexes[0]);
121             PrintVector(trsm_bl_indexes, size_bl_start*
                   size_bl_start);
122             for (j=0; j<size_bl_start; j++)
```

```
123                 for (k=0; k<size_bl_start; k++)
124                     aux_A[ j*size_bl_start + k ] = A[
                            trsm_bl_indexes[ j*size_bl_start + k ]];
125             dtrtrs_(&UPLO, &TRANS, &DIAG, &N, &NRHS, aux_L, &LDA
                    , aux_A, &LDA, &INFO);
126             for (j=0; j<size_bl_start*size_bl_start; j++) (*LU)[
                    trsm_bl_indexes[j]] = aux_A[j];
127             free(trsm_bl_indexes);
128         }
129     }
130 
131     printf("\t... end of TRSM (row)\n" );
132 
133     *TRSM_performed = next_in_row - 1;
134     free(aux_A); free(aux_L);
135 }
136 
137 void TRSM_col( int first_block_start_i, int size_bl_start, int
       bl_start_lvl, int *first_block_indexes, int n_levels, int *
       levels_sizes, int *block_indexes, double *A, double **LU, int
        *levels_start_i, int *levels_end_i, int *TRSM_performed )
138 {
139     printf("\tTRSM(s) (col) from block starting in index %d...\n
           ", first_block_indexes[0] );
140     int i, j, k, next_in_col, next_col_start, *trsm_bl_indexes,
           N, NRHS, LDA, INFO;
141     double *aux_A, *aux_U;
142 
143     // Lapack variables
144     char UPLO = 'L', TRANS = 'N', DIAG = 'N';
145     N = size_bl_start; NRHS = N; LDA = N; INFO = 0;
146 
147     next_in_col = 1;
```

```
148     aux_A = malloc(size_bl_start * size_bl_start * sizeof(double
           ));
149     aux_U = malloc(size_bl_start * size_bl_start * sizeof(double
           ));
150
151     for (i=1; i<levels_end_i[bl_start_lvl-1]-levels_start_i[
           bl_start_lvl-1]+1; i++) // All blocks of this size except
            1st
152     {
153         if(i==1)
154             for (j=0; j<size_bl_start; j++)
155                 for (k=j; k<size_bl_start; k++)
156                     aux_U[ j * size_bl_start + k ] = (*LU)[
                           first_block_indexes[ k * size_bl_start +
                           j ]];
157         next_col_start = first_block_start_i + size_bl_start *
               next_in_col;
158         if(block_indexes[levels_start_i[bl_start_lvl-1]+i] ==
               next_col_start)
159         {
160             next_in_col ++;
161             GetIndexes(&trsm_bl_indexes, next_col_start,
                   size_bl_start, levels_sizes[0]);
162             printf("\t\t> TRSM number %d, of block starting in
                   index %d\n", next_in_col-1, trsm_bl_indexes[0]);
163             PrintVector(trsm_bl_indexes, size_bl_start*
                   size_bl_start);
164         for (j=0; j<size_bl_start; j++)
165                 for (k=0; k<size_bl_start; k++) // Store U
                       matrix fragment (transposed)
166                     aux_A[ j * size_bl_start + k ] = A[
                           trsm_bl_indexes[ k * size_bl_start + j
                           ]];
```

```
167             PrintVector(first_block_indexes, size_bl_start*
                    size_bl_start);
168             dtrtrs_(&UPLO, &TRANS, &DIAG, &N, &NRHS, aux_U, &LDA
                    , aux_A, &LDA, &INFO);
169             for (k=0; k<size_bl_start; k++) // Store res in LU (
                    transpose way)
170                 for (j=0; j<size_bl_start; j++)
171                     (*LU)[trsm_bl_indexes[ k * size_bl_start + j
                            ]] = aux_A[ j * size_bl_start + k ];
172             free(trsm_bl_indexes);
173         }
174     }
175     printf("\t... end of TRSM (col)\n" );
176     *TRSM_performed = next_in_col - 1;
177     free(aux_A); free(aux_U);
178 }
179 
180 void Update( int first_block_start_i, int size_bl_start, int
       bl_start_lvl, int n_levels, int *levels_sizes, int *
       block_indexes, double **A, double *LU, int *levels_start_i,
       int *levels_end_i, int to_upd )
181 {
182     printf("\tUPD from block starting in index %d...\n",
           first_block_start_i);
183     int bl, *upd_bl_indexes, diag_start, diag_start_row,
           diag_start_col, i, j, *U_idx, *L_idx, N, LDA;
184     double *aux_U, *aux_L, *aux_X;
185 
186     // VARS for UPD
187     N = size_bl_start;
188     LDA = N;
189     char TRANS = 'N';
190     double ALPHA = 1.0, BETA = 0.0;
191 
```

```
192         aux_L = malloc(size_bl_start * size_bl_start * sizeof(double
               ));
193         aux_U = malloc(size_bl_start * size_bl_start * sizeof(double
               ));
194         aux_X = malloc(size_bl_start * size_bl_start * sizeof(double
               ));
195 
196         // UPD from the first smaller block to same-sized blocks in
               the same column
197         if(to_upd > 0)
198         {
199 
200             for(bl=1; bl<=to_upd; bl++)
201             {
202                 i = first_block_start_i + levels_sizes[0] *
                       size_bl_start * bl;
203                 GetIndexes(&U_idx, i, size_bl_start, levels_sizes
                       [0]);
204                 i = first_block_start_i + size_bl_start * bl;
205                 GetIndexes(&L_idx, i, size_bl_start, levels_sizes
                       [0]);
206                 for (j=0; j<size_bl_start*size_bl_start; j++)
207                 {
208                     aux_U[j] = LU[ U_idx[j] ];
209                     aux_L[j] = LU[ L_idx[j] ];
210                 }
211                 diag_start = first_block_start_i + bl * levels_sizes
                       [0] * size_bl_start + bl * size_bl_start;
212                 GetIndexes(&upd_bl_indexes, diag_start,
                       size_bl_start, levels_sizes[0]);
213                 printf("\t\t> Updating block starting in index %d\n
                       ", upd_bl_indexes[0]);
214                 PrintVector(upd_bl_indexes, size_bl_start*
                       size_bl_start);
```

```
215                 dgemm_(&TRANS, &TRANS, &N, &N, &N, &ALPHA, aux_L, &
                       LDA, aux_U, &LDA, &BETA, aux_X, &N);
216                 for (i=0; i<size_bl_start*size_bl_start; i++) (*A)[
                       upd_bl_indexes[i]] = (*A)[upd_bl_indexes[i]] -
                       aux_X[i];
217                 for(j=1; j<=to_upd-bl; j++) // Blocks on the right
                       side of diagonal block (in the row)
218                 {
219                     printf("diag_start = %d\n", diag_start);
220                     printf("U_idx = "); PrintVector(U_idx,
                           size_bl_start*size_bl_start);
221                     printf("L_idx = "); PrintVector(L_idx,
                           size_bl_start*size_bl_start);
222                     i = first_block_start_i + levels_sizes[0] *
                           size_bl_start * (j+1);
223                     GetIndexes(&U_idx, i, size_bl_start,
                           levels_sizes[0]);
224                     for (i=0; i<size_bl_start*size_bl_start; i++)
                           aux_U[i] = LU[ U_idx[i] ];
225                     diag_start_row = diag_start + size_bl_start *
                           levels_sizes[0] * j;
226                     GetIndexes(&upd_bl_indexes, diag_start_row,
                           size_bl_start, levels_sizes[0]);
227                     printf("\t\t> Updating block starting in index %
                           d\n", diag_start_row);
228                     PrintVector(upd_bl_indexes, size_bl_start*
                           size_bl_start);
229                     dgemm_(&TRANS, &TRANS, &N, &N, &N, &ALPHA, aux_L
                           , &LDA, aux_U, &LDA, &BETA, aux_X, &N);
230                     for (i=0; i<size_bl_start*size_bl_start; i++) (*
                           A)[upd_bl_indexes[i]] = (*A)[upd_bl_indexes[i
                           ]] - aux_X[i];
231                 }
```

```
232             i = first_block_start_i + levels_sizes[0] * size_bl_start * bl;
233             GetIndexes(&U_idx, i, size_bl_start, levels_sizes[0]);
234             for (j=0; j<size_bl_start*size_bl_start; j++) aux_U[j] = LU[ U_idx[j] ];
235             for(j=1; j<=to_upd-bl; j++) // Blocks under diagonal block (in the column)
236             {
237                 printf("diag_start = %d\n", diag_start);
238                 printf("U_idx = "); PrintVector(U_idx, size_bl_start*size_bl_start);
239                 printf("L_idx = "); PrintVector(L_idx, size_bl_start*size_bl_start);
240                 i = first_block_start_i + size_bl_start * (j+1);
241                 GetIndexes(&L_idx, i, size_bl_start, levels_sizes[0]);
242                 for (i=0; i<size_bl_start*size_bl_start; i++) aux_L[i] = LU[ L_idx[i] ];
243                 diag_start_col = diag_start + size_bl_start * j;
244                 GetIndexes(&upd_bl_indexes, diag_start_col, size_bl_start, levels_sizes[0]);
245                 printf("\t\t> Updating block starting in index %d\n", diag_start_col);
246                 PrintVector(upd_bl_indexes, size_bl_start*size_bl_start);
247                 dgemm_(&TRANS, &TRANS, &N, &N, &N, &ALPHA, aux_L, &LDA, aux_U, &LDA, &BETA, aux_X, &N);
248                 for (i=0; i<size_bl_start*size_bl_start; i++) (*A)[upd_bl_indexes[i]] = (*A)[upd_bl_indexes[i]] - aux_X[i];
249             }
250         }
```

```
251            free(aux_X); free(U_idx); free(L_idx); free(
                   upd_bl_indexes);
252        }
253        printf("\t... end of UPD\n");
254        free(aux_U); free(aux_L);
255    }
256
257    void CheckLU(double *LU, int size, int *BIG_IPIV)
258    {
259        int i,j;
260        char SIDE = 'L', UPLO = 'L', TRANSA = 'N', DIAG = 'U';
261        double ALPHA = 1.0, *U = malloc(size*size*sizeof(double));
262        for(i=0; i<size; i++)
263            for(j=0;j<size;j++)
264            {
265                if(j>i) U[i*size+j] = 0.0;
266                else U[i*size+j] = LU[i*size+j];
267            }
268        dtrmm_(&SIDE, &UPLO, &TRANSA, &DIAG, &size, &size, &ALPHA,
              LU, &size, U, &size);
269        ReversePivot(&U, BIG_IPIV, size, size, 0);
270        PrintMatrix(U, size*size);
271        free(U);
272    }
273
274    int main (int argc, char *argv[])
275    {
276
277        // ----------------------- CHECK USAGE
278        if (argc != 2)
279        {
280            printf ("Usage: Sequential_LU <FileOfMatrix>\n"); return
                    0;
281        }
```

```
282
283     // ----------------------- VARIABLES DECLATIONS
284     ...
285
286     // ----------------------- GET DATA FROM INPUT
287     file_name = argv[1];
288
289     // ----------------------- READ MATRIX DATA FROM FILE
290     printf("\nGiven file containing matrix information: %s\n\n",
            file_name);
291     printf("Obtaining matrix information... ");
292     ObtainMatrixData(file_name, &n_levels, &levels_sizes, &
           n_blocks, &block_indexes, &A);
293     printf("DONE\n\n");
294
295     // ----------------------- GET LEVELS INDEXES
296     printf("Obtaining levels indexes... ");
297     GetBlocksIndexes(&levels_start_i, &levels_end_i, n_levels,
           n_blocks, block_indexes);
298     printf("DONE\n\n");
299
300     // ----------------------- PERFORM LU
301     printf("Computing L and U matrices... \n");
302
303     // Initialize L, U matrices to zeros
304     LU = malloc(levels_sizes[0] * levels_sizes[0] * sizeof(
           double));
305     aux_A = malloc(levels_sizes[n_levels-1]*levels_sizes[
           n_levels-1] * sizeof(double));
306
307     // Vars for LU
308     INFO = 0;
309     N = levels_sizes[n_levels-1];
310     LDA = levels_sizes[n_levels-1];
```

```
311     int IPIV[LDA]; int BIG_IPIV[levels_sizes[0]];
312
313     // LU of the first smallest block
314     first_block_start_i = 0;
315     n_blocks_level = 1;
316     first_block_indexes = malloc(n_blocks_level * sizeof(int));
317     GetIndexes(&first_block_indexes, first_block_start_i,
           levels_sizes[n_levels-1], levels_sizes[0]);
318     printf("\n\tLU of block starting in index %d...\n",
           first_block_indexes[0]);
319     for (i=0; i<levels_sizes[n_levels-1]*levels_sizes[n_levels
           -1]; i++) aux_A[i] = A[first_block_indexes[i]];
320     dgetrf_(&N, &N, aux_A, &LDA, IPIV, &INFO);
321     PrintVector(IPIV, LDA);
322     ApplyPivot(&A, IPIV, LDA, levels_sizes[0], 0);
323     for (i=0; i< levels_sizes[0]; i++) BIG_IPIV[i] = IPIV[i];
324     for (i=0; i<levels_sizes[n_levels-1]*levels_sizes[n_levels
           -1]; i++) LU[first_block_indexes[i]] = aux_A[i];
325     printf("\t... end of LU\n" );
326
327     // Vars for TRSM (row)
328     INFO = 0;
329     N = levels_sizes[n_levels-1];
330     LDA = N;
331     NRHS = N;
332
333     // TRSM from the first smaller block to same-sized blocks in
            the same row
334     TRSM_row( first_block_start_i, levels_sizes[n_levels-1],
           n_levels, first_block_indexes, n_levels, levels_sizes,
           block_indexes, A, &LU, levels_start_i, levels_end_i, &
           TRSMs_in_row );
335
```

```
336     // TRSM from the first smaller block to same-sized blocks in
            the same column
337     TRSM_col( first_block_start_i, levels_sizes[n_levels-1],
           n_levels, first_block_indexes, n_levels, levels_sizes,
           block_indexes, A, &LU, levels_start_i, levels_end_i, &
           TRSMs_in_col );
338
339     // UPD from the first smaller block to same-sized blocks in
           the diagonal
340     if ((TRSMs_in_row) < (TRSMs_in_col)) to_upd = TRSMs_in_row;
341     else to_upd = TRSMs_in_col;
342     Update( first_block_start_i, levels_sizes[n_levels-1],
           n_levels, n_levels, levels_sizes, block_indexes, &A, LU,
           levels_start_i, levels_end_i, to_upd );
343
344     // REMARK: diagonal is made of smallest level blocks
345     // Move along diagonal blocks and execute: LU + TRSM(s) +
           UPD(s)
346     for(i_diag=1; i_diag<levels_sizes[0]/levels_sizes[n_levels
           -1]; i_diag++)
347     {
348         diag_start_i = i_diag * levels_sizes[0] * levels_sizes[
               n_levels-1] + i_diag * levels_sizes[n_levels-1];
349         GetIndexes(&diag_bl_indexes, diag_start_i, levels_sizes[
               n_levels-1], levels_sizes[0]);
350
351         // LU of diagonal block
352         printf("\n\tLU of block starting in index %d...\n",
               diag_start_i);
353         for (i=0; i<levels_sizes[n_levels-1]*levels_sizes[
               n_levels-1]; i++) aux_A[i] = A[diag_bl_indexes[i]];
354         dgetrf_(&N, &N, aux_A, &LDA, IPIV, &INFO);
355         j = diag_bl_indexes[0] % levels_sizes[0];
```

```
356            for (i=0; i<LDA; i++) { IPIV[i] = IPIV[i] + j; BIG_IPIV[
                  i + j] = IPIV[i]; }
357            PrintVector(IPIV, LDA);
358            ApplyPivot(&A, IPIV, LDA, levels_sizes[0], j);
359            ApplyPivot(&LU, IPIV, LDA, levels_sizes[0], j);
360            for (i=0; i<levels_sizes[n_levels-1]*levels_sizes[
                  n_levels-1]; i++) LU[diag_bl_indexes[i]] = aux_A[i];
361            printf("\t... end of LU\n");
362
363            // TRSM of next blocks of same level in same row that
                  diagonal block
364            TRSM_row( diag_start_i, levels_sizes[n_levels-1],
                  n_levels, diag_bl_indexes, n_levels, levels_sizes,
                  block_indexes, A, &LU, levels_start_i, levels_end_i,
                  &TRSMs_in_row );
365
366            // TRSM of next blocks of same level in same column that
                   diagonal block
367            TRSM_col( diag_start_i, levels_sizes[n_levels-1],
                  n_levels, diag_bl_indexes, n_levels, levels_sizes,
                  block_indexes, A, &LU, levels_start_i, levels_end_i,
                  &TRSMs_in_col );
368
369            // UPD of next diagonal blocks of same level (as much as
                   TRSMs have been performed)
370            // If no TRSM's have been performed, then evaluate if
                  there is a bigger block to perform TRSM(s) and UPD(s)
371            if ((TRSMs_in_row) < (TRSMs_in_col)) to_upd =
                  TRSMs_in_row;
372            else to_upd = TRSMs_in_col;
373            if(to_upd > 0)
374            {
375                Update( diag_start_i, levels_sizes[n_levels-1],
                      n_levels-1, n_levels, levels_sizes, block_indexes
```

```
                , &A, LU, levels_start_i, levels_end_i, to_upd );
376         }
377         else
378         {
379             prev = 1;
380             keep_prev = 1;
381             previous_lvl_bl_start_i = 2;
382             while (keep_prev && (previous_lvl_bl_start_i > 0))
383             {
384                 previous_lvl_bl_start_i = diag_bl_indexes[
                    levels_sizes[n_levels-1]*levels_sizes[
                    n_levels-1]-1] - ( (levels_sizes[n_levels-1-
                    prev] - 1) *levels_sizes[0] + levels_sizes[
                    n_levels-1-prev] - 1);
385                 GetIndexes(&previous_lvl_bl_indexes,
                    previous_lvl_bl_start_i, levels_sizes[
                    n_levels-1-prev], levels_sizes[0]);
386                 printf("\tNo TRSM(s) have been performed --> Try
                     TRSM(s) of bigger levels blocks.\n\tPrev
                    level block start i: %d (size = %dx%d)\n",
                    previous_lvl_bl_start_i, levels_sizes[
                    n_levels-1-prev], levels_sizes[n_levels-1-
                    prev]);
387
388                 // TRSM of next blocks of same level in same row
                     that bigger block
389                 TRSM_row( previous_lvl_bl_start_i, levels_sizes[
                    n_levels-1-prev], n_levels-prev,
                    previous_lvl_bl_indexes, n_levels,
                    levels_sizes, block_indexes, A, &LU,
                    levels_start_i, levels_end_i, &TRSMs_in_row )
                    ;
390
```

```
391                 // TRSM of next blocks of same level in same
                       column that bigger block
392                 TRSM_col( previous_lvl_bl_start_i, levels_sizes[
                       n_levels-1-prev], n_levels-prev,
                       previous_lvl_bl_indexes, n_levels,
                       levels_sizes, block_indexes, A, &LU,
                       levels_start_i, levels_end_i, &TRSMs_in_col )
                       ;
393
394                 // UPD of next diagonal blocks of same level (as
                        much as TRSMs have been performed)
395                 if ((TRSMs_in_row) < (TRSMs_in_col)) to_upd =
                       TRSMs_in_row;
396                 else to_upd = TRSMs_in_col;
397                 if(to_upd > 0)
398                 {
399                     keep_prev = 0;
400                     Update( previous_lvl_bl_start_i,
                           levels_sizes[n_levels-1-prev], n_levels-
                           prev-1, n_levels, levels_sizes,
                           block_indexes, &A, LU, levels_start_i,
                           levels_end_i, to_upd );
401                 }
402                 else prev++;
403                 free(previous_lvl_bl_indexes);
404             }
405         }
406         free(diag_bl_indexes);
407     }
408
409     printf("\nDONE\n\n");
410
411     // Print solution (L and U matrices have been stored in the
           same matrix)
```

```
412        printf("Computed matrix LU is: \n\n");
413        PrintMatrix(LU, levels_sizes[0] * levels_sizes[0]);
414        printf("\nL * U =\n\n");
415        CheckLU(LU, levels_sizes[0], BIG_IPIV);
416
417        // ----------------------- FREE MEMORY
418        printf("\nFree memory... ");
419        free(A); free(LU); free(aux_A);
420        free(levels_sizes); free(block_indexes);
421        free(levels_start_i); free(levels_end_i);
422        free(first_block_indexes);
423        printf("DONE\n\n");
424    }
```

A.2. Matrices para las pruebas

Cada uno de los ficheros que contiene matrices ejemplo para probar mi implementación se compone de cinco líneas, que representan, en este orden:

- Número total de niveles que conforman la jerarquía de la matriz (siempre se considerará el nivel 0 como la matriz completa sin particionar).
- Tamaño de cada nivel. Dado que se asumen bloques cuadrados, el número que se almacena es la raíz del total de elementos, es decir, si el bloque contiene 64 elementos, su tamaño almacenado en el fichero será 8.
- Total de bloques que conforman la matriz (sea cual sea su tamaño).
- Índices de inicio de cada uno de los bloques de cada nivel, ordenados por niveles desde el que contiene más elementos (nivel 0) hasta el que contiene menos.
- Valores de la matriz, por columnas.

A.2.1. *matrix_1.txt*

Esta matriz, la más sencilla de las tres que se han utilizado para hacer pruebas, contiene un total de 64 elementos, distribuidos en tres niveles de tamaños 8×8, 4×4, 2×2, respectivamente. La forma de la matriz (incluyendo particionado) es la que se muestra en la Figura A.1.

9	1	1	1	1	1	1	10
1	10	1	1	1	1	1	1
1	1	9	5	1	1	1	1
1	-1	1	9	1	1	1	1
1	7	1	1	9	1	1	1
1	1	1	1	1	8	1	1
1	1	1	1	1	1	9	2
1	1	1	1	1	1	1	9

Figura A.1: Visualización de la matriz matrix_1.txt.

```
3

8,4,2

13

0,0,4,32,36,0,2,16,18,36,38,52,54

9,1,1,1,1,1,1,1,1,10,1,-1,7,1,1,1,1,1,9,1,1,1,1,1,1,1,5,9,
1,1,-1,1,-1,1,1,1,9,1,1,1,1,1,1,-1,1,8,1,1,1,1,1,1,1,1,9,1,
10,1,1,1,1,1,2,9
```

A.2.2. *matrix_2.txt*

Esta matriz contiene un total de 144 elementos, distribuidos en tres niveles de tamaños 12×12, 6×6, 3×3, respectivamente. La forma de la matriz (incluyendo particionado)

es la que se muestra en la Figura A.2.

3	23	92	81	54	42	42	35	93	18	56	51
43	38	81	99	89	99	61	89	78	83	78	44
19	9	75	3	90	95	76	46	5	77	32	100
73	65	82	54	63	68	59	42	38	94	18	82
38	19	39	9	14	99	56	22	71	11	34	49
85	5	62	81	22	77	59	13	73	19	22	90
74	73	58	99	19	34	52	31	23	10	52	14
58	35	54	7	5	67	9	73	27	49	91	40
18	67	28	94	11	25	72	79	68	20	63	93
96	39	25	2	62	30	100	70	48	90	11	92
27	63	46	69	94	69	36	1	63	10	40	72
93	3	23	79	36	53	98	85	24	5	6	62

Figura A.2: Visualización de la matriz matrix_2.txt.

```
3

12,6,3

13

0,0,6,72,78,0,3,36,39,78,81,114,117

3,43,19,73,38,85,74,58,18,96,27,93,23,38,9,65,19,5,73,35,67,
39,63,3,92,81,75,82,39,62,58,54,28,25,46,23,81,99,3,54,9,81,
99,7,94,2,69,79,54,89,90,63,14,22,19,5,11,62,94,36,42,99,95,
68,99,77,34,67,25,30,69,53,42,61,76,59,56,59,52,9,72,100,36,
98,35,89,46,42,22,13,31,73,79,70,1,85,93,78,5,38,71,73,23,27,
68,48,63,24,18,83,77,94,11,19,10,49,20,90,10,5,56,78,32,18,34,
22,52,91,63,11,40,6,51,44,100,82,49,90,14,40,93,92,72,62
```

A.2.3. $matrix_3.txt$

Esta matriz contiene los mismos elementos que la matriz anterior, pero distribuidos con un particionado distinto, en tres niveles de tamaños 12×12, 4×4, 2×2, respectivamente. La forma de la matriz (incluyendo particionado) es la que se muestra en la Figura A.3.

3	23	92	81	54	42	42	35	93	18	56	51
43	38	81	99	89	99	61	89	78	83	78	44
19	9	75	3	90	95	76	46	5	77	32	100
73	65	82	54	63	68	59	42	38	94	18	82
38	19	39	9	14	99	56	22	71	11	34	49
85	5	62	81	22	77	59	13	73	19	22	90
74	73	58	99	19	34	52	31	23	10	52	14
58	35	54	7	5	67	9	73	27	49	91	40
18	67	28	94	11	25	72	79	68	20	63	93
96	39	25	2	62	30	100	70	48	90	11	92
27	63	46	69	94	69	36	1	63	10	40	72
93	3	23	79	36	53	98	85	24	5	6	62

Figura A.3: Visualización de la matriz matrix_3.txt.

```
3

12,4,2

22

0,0,4,8,48,52,56,96,100,104,0,2,24,26,52,54,76,78,104,106,128,130

3,43,19,73,38,85,74,58,18,96,27,93,23,38,9,65,19,5,73,35,67,39,
63,3,92,81,75,82,39,62,58,54,28,25,46,23,81,99,3,54,9,81,99,7,94,
2,69,79,54,89,90,63,14,22,19,5,11,62,94,36,42,99,95,68,99,77,34,
67,25,30,69,53,42,61,76,59,56,59,52,9,72,100,36,98,35,89,46,42,
22,13,31,73,79,70,1,85,93,78,5,38,71,73,23,27,68,48,63,24,18,83,
77,94,11,19,10,49,20,90,10,5,56,78,32,18,34,22,52,91,63,11,40,6,
51,44,100,82,49,90,14,40,93,92,72,62
```