UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

# Diseño e implementación de software para tratamiento de datos masivos

Trabajo Fin de Grado

Grado en Ingenieria Informática

*Autor:* Eusebio Soriano Benegas

*Tutor:* Antonio Manuel Vidal Maciá

*Cotutor:* Francisco José Martínez Zaldívar

Curso 2017-2018

# Resumen

Actualmente la generación y el tratamiento de datos masivos se ha convertido en un desafío importante en el campo de la Informática. La cantidad de datos generados en muchos procesos informáticos supera la capacidad del *software* convencional para procesarlos en un tiempo razonable. En muchas aplicaciones de Ingeniería clásicas (Astrofísica, Modelado Electromagnético, Simulación,...) es necesario resolver problemas computacionales en los que los datos estructurados, debido a su gran tamaño, deben almacenarse en memoria secundaria (disco,...). La gestión de las transferencias de datos condiciona estos algoritmos que deben diseñarse cuidadosamente para evitar tiempos de ejecución excesivos. El objetivo del proyecto es el diseño e implementación de algoritmos para el tratamiento de datos almacenados fuera de la RAM, utilizando un computador multicore de última generación. A este tipo de algoritmos se les denomina *out-of-core*. En particular, en este proyecto se realiza un estudio de la resolución de sistemas de ecuaciones para matrices simétricas definidas positivas mediante la descomposición de Cholesky. Concretamente, se analiza el caso en el que el tamaño de la matriz es tan grande que no cabe en la memoria de acceso aleatorio (RAM). Para abordar este problema, se han implementado dos algoritmos *out-of-core* basados en el algoritmo por bloques clásico. Para las operaciones matriciales se ha utilizado la librería MKL de intel.

**Palabras clave:** Datos en memoria masiva, Algoritmos *out-of-core*, factorización de choesky, Sistemas de ecuaciones lineales

# Abstract

Generation and management of big data has currently become one of the most important challenges in the field of computer engineering. The huge amount of data generated by many computing procedures makes conventional softwares struggle when trying to compute this data in a reasonable time. The structured data must be stored in hard drives -due to its gigantic size – in some of the classic engineering applications such as Astophysics, Electromagnetic Modelling, Simulation. Therefore it is an imperative to solve this computational disadvantage. The data transfer management affects the algorithms since they must be cautiously designed in order to avoid

excessive execution times. The main goal of this project is both the design and implementation of algorithms using a state-of-art multicore computer in the treatment of stored data outside of the RAM. This sort of techniques are called out-of-core algorithms The key performance in this project involves an essay about solving equation systems for symmetric positive-definite matrix using the Cholesky factorization. Being more precise, the exercise analyses those cases in which the matrix size does not fit within the Random Access Memory (RAM). The implementation of two out-of-core algorithms based on the classic block algorithm has been the chosen approach of the problem. The intel's MKL library has been used for the matrix operations.

**Key words:** Masive data in memory, Out-of-core algorithms, Cholesky factorization, Lineal ecuations systems

# índice general

# Índice de figuras

# Índice de tablas

# Capítulo 1

# Introducción

## 1.1 Motivación

Actualmente la generación y el tratamiento de datos masivos se ha convertido en un desafío importante en el campo de la Informática. La cantidad de datos generados en muchos procesos informáticos supera la capacidad del *software* convencional para procesarlos en un tiempo razonable. En muchas aplicaciones de Ingeniería clásicas (Astrofísica, Modelado Electromagnético, Simulación,...) es necesario resolver problemas computacionales en los que los datos estructurados, debido a su gran tamaño, deben almacenarse en memoria secundaria (disco,...). La gestión de las transferencias de datos condiciona estos algoritmos que deben diseñarse cuidadosamente para evitar tiempos de ejecución excesivos.
Un problema típico de esta clase es la resolución de sistemas de ecuaciones lineales con matrices de gran dimensión. El problema de resolver un sistema lineal es fundamental en la computación científica. En campos como los gráficos por ordenador, la Física, la Ingeniería, la Medicina y la Química, el desarrollo de librerías y algoritmos capaces de resolver estos problemas de forma eficiente y precisa, se ha convertido en un aspecto crítico.

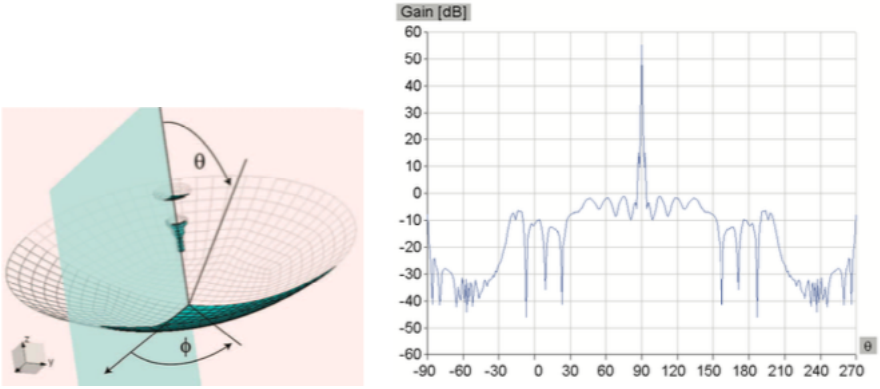



**Figura 1.1:** Modelo 3D de una antena reflectora Cassegrain (izq.). Patrones de radiación para un antena reflectora de diámetro 240 $\lambda$ (2824 mm) calculados mendiante la factorización LU con un algoritmo *out-of-core*

En la figura 1.1 vemos un ejemplo de uno de estos problemas, en este caso se requería resolver el sistema de ecuaciones generado a partir de una estructura electromagnética grande mediante el uso del método de los momentos y con él calcular los patrones de radiación de la antena. En el caso del problema resuelto en la figura 1.1, para una antena de 2824 mm de diámetro, se generaban nada menos que 151 898 coeficientes [3], lo cual se traduce en una matriz de aproximadamente 171,90 GB. Como uno se puede imaginar, el gran tamaño de la matriz representa un obstáculo a la hora de abordar el problema; para salvarlo una solución es el empleo de algoritmos *out-of-core*.

Los algoritmos *out-of-core* se basan en particionar los datos, en este caso matrices, en bloques de un tamaño que sí que tenga cabida en la memoria principal. El implementar esta técnica no es un tarea trivial y a menudo requiere de la adaptación de los algoritmos originales a este paradigma. Además, a la optimización de las operaciones intrínsecas de la resolución de sistemas lineales, hay que añadirle un nuevo factor más a optimizar, la entrada salida (I/O).

En el presente proyecto, se pretende adaptar el algoritmo para la descomposición de Cholesky al paradigma *out-of-core* para, de esta manera, ser capaces de afrontar problemas como el presentado en la figura 1.1. La factorización de Cholesky es el método para factorizar matrices definidas positivas. Entre sus usos destacan aplicaciones de ingeniería con propiedades

de simetría, como por ejemplo la simulación de sistemas con múltiples variables correlacionadas cuando se utiliza el método de Montecarlo para su resolución[4]. En el ejemplo de la figura 1.2, la factorización de Cholesky es utilizada a la hora de resolver los sistemas de ecuaciones generados por la información procedente de las distintas fuentes que se utilizan para la reconstrucción 3D[5].

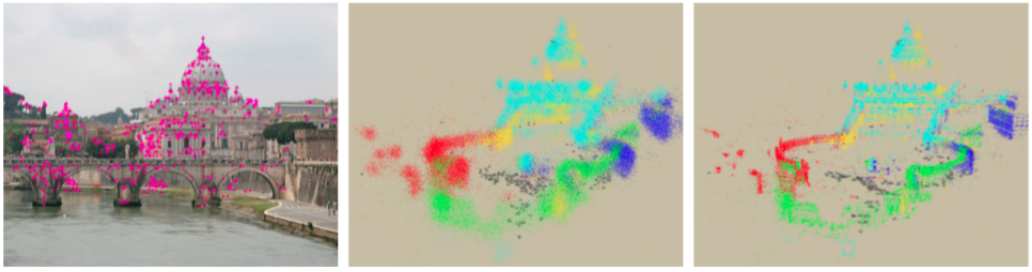


**Figura 1.2:** Reconstrucción 3D de la basílica de San Pedro

## 1.2 Objetivos

El principal objetivo de este proyecto es diseñar e implementar un algoritmo out-of-core capaz de resolver sistemas lineales $Ax = b$ mediante el uso de la factorización de Cholesky, suponiendo que la matriz $A$ es simétrica definida positiva (MSDP), utilizando un computador de tipo multicore con gran capacidad de almacenamiento masivo y librerías de Altas Prestaciones que permitan aprovechar el paralelismo de la máquina.

Además de la implementación, se realizarán una serie de pruebas con diferentes tamaños, $N \times N$, de matriz (hasta $N = 1\,000\,000$) para comprobar si el problema se resuelve en un tiempo razonable y con una solución aceptable (reducir al máximo el error relativo). También se realizará un estudio pormenorizado de los tiempos en ambos algoritmos para comprobar en qué se consume la mayoría del tiempo y encontrar así posibles cuellos de botella.

## 1.3 Estructura del proyecto

Este documento se ha organizado en 5 capítulos:

1. **Introducción**: En este capítulo se realiza una breve presentación del proyecto. En primer lugar se encuentra la motivación, donde se introduce al lector en la importancía de los algoritmos *out-of-core* para resolver problemas en los campos de la computación científica e ingeniería. En segundo lugar se plantea brevemente los objetivos del trabajo. Por último se presenta la estructura del proyecto, donde se exponen los diferentes capítulos que componen la memoría del proyecto

2. **La resolución de sistemas de ecuaciones en matrices simétricas definidas positivas**: Aquí se expone de forma más teórica el problema tratado en el proyecto. Haciendo especial hincapié en las MSDPs y la descomposición de Cholesky, sobre todo en las propiedades que permiten implementar un algoritmo por bloques, así como su coste temporal.

3. **Algoritmo de Cholesky *out-of-core***: En este tercer capítulo, se analizan, por una parte, la estructura de datos diseñada para almacenar la matriz y las rutinas implementadas para su lectura y escritura. Por la otra, la estructura y lógica de los algoritmos por bloques que realizarán la descomposición.

4. **Resultados experimentales**: El cuarto capítulo está dedicado a mostrar y analizar los resultados de las distintas pruebas realizadas. Para que el experimento sea reproducible por el lector también se añaden los detalles del *hardware* de la máquina, como también del *software* utilizado.

5. **Conclusiones y líneas abiertas**: Finalmente, en este último capítulo se realiza una síntesis de los resultados obtenidos en las pruebas y, además, una valoración general del proyecto. Adicionalmente, se plantean otras líneas de investigación como por ejemplo, la eficiencia energética, el uso de GPU, etc.

# Capítulo 2

# La resolución de sistemas de ecuaciones de matrices simétricas definidas positivas

El caso concreto de los sistemas simétricos definidos positivos constituye una de las clases más importantes de poblemas $Ax = b$[6]. Una MSDP tiene una diagonal que es lo suficientemente "pesada" como para evitar la necesidad de pivotar. Es por ello que este tipo característico de matrices tienen una factorización especial llamada factorización de Cholesky, que explota tanto la simetría como el hecho de que sea definida positiva.

## 2.1 La factorización de Cholesky $A = GG^T$

La factorización de Cholesky se obtiene fácilmente a partir de la descomposición $LDL^T$ de una matriz simétrica $A$, si ésta es definida positiva.

$$A = LDL^T = LD^{1/2}D^{1/2}L^T = (LD^{1/2})(LD^{1/2})^T = GG^T \tag{2.1}$$

En palabras esto se traduce en que dada una MSDP, cuyos valores pertenecen al conjunto $\mathbb{R}$, se puede asegurar que existe una matriz triangular inferior única $G \in \mathbb{R}^{n\times n}$ tal que $A = GG^T$. La primera igualdad de la

ecuación($A = LDL^T$) viene dada por el hecho de que la matriz es simétrica y es no singular [7]. Esta segunda propiedad se cumple en el caso de las matrices definidas positivas debido a que las submatrices principales de estas son también definidas positivas, siendo las submatrices por tanto no singulares. Esto incluye a los valores de su diagonal principal [8]. La segunda igualdad se aprovecha del hecho de que la diagonal $D$ es definida positiva y, por consiguiente, que sus elementos ($d_k$) son positivos, esto permite descomponer $D$ en el producto de las raíces cuadradas de la misma diagonal ($D^{1/2}D^{1/2}$). Finalmente, se asocia cada $D$ a su correspondiente $L$ y como resultado tenemos el producto buscado: $A = GG^T$.

La matriz $G$, conocida como el *factor de Cholesky*, es una matriz triangular inferior con elementos positivos en la diagonal. La unicidad de la matriz viene dada por la unicidad de la factorización $LDL^T$.

Una vez demostrada 2.1, se podría implementar cada paso directamente para conseguir la deseada factorización. Sin embargo, es más efectivo desarrollar un proceso que trabaje directamente sobre la igualdad $A = GG^T$:

**Algoritmo 2.1.1** (**Algoritmo de Cholesky**) donde $A \in \mathbb{R}^{n\times n}$ es simétrica y definida positiva. El factor de Cholesky $G$ sobrescribe la triangular inferior de $A$

```
1: for k = 1:n do
2:     A_kk = √A_kk                                  ▷ (1)
3:     for i = k + 1:n do
4:         A_ik = A_ik/A_kk                          ▷ (2)
5:         for j = k + 1:i do
6:             A_ij = A_ij − A_ik A_jk               ▷ (3)
7:         end for
8:     end for
9: end for
```

El algoritmo 2.1.1 se divide básicamente en tres pasos. En primer lugar (1), mediante la multiplicación de la primera fila de $G$ obtenemos lo siguiente: $a_{11} = {g_{11}}^2 \rightarrow g_{11} = \sqrt{a_{11}}$. Una vez se tiene el primer valor de $G$, es posible utilizarlo para obtener los demás elementos de la misma columna (2): $g(1\!:\!n, 1) = a(1\!:\!n, 1)/g_{11}$. Finalmente, a la matriz sobrante se realiza un ajuste para la siguiente iteración: $a(2\!:\!n, 2\!:\!n) = a(2\!:\!n, 2\!:\!n) -$

$g(2:n,1)g(2:n,1)^t$. Este proceso se repite hasta que ya solo quede un elemento de la matriz por obtener. El algoritmo tiene un coste de $n^3/3$ flops.

## 2.2 Cholesky por bloques

Los algoritmos más eficientes para la obtención de la descomposición de Cholesky son aquellos en los que la matriz se estructura por bloques. La organización por bloques del algoritmo permite evitar operaciones escalares convirtiéndolas en operaciones por bloques que permiten obtener el máximo rendimiento de librerías de altas prestaciones como BLAS o LAPACK. A continuación se mostrará un algoritmo por bloques que posteriormente se utilizará como base para el desarrollo de los algoritmos out-of-core. Pero antes de entrar en profundidad en el algoritmo se deben explicar ciertos detalles de la estructura de la matriz y su partición en bloques:

$$\begin{bmatrix} A_{11} & \cdots & A_{1N} \\ \vdots & \ddots & \vdots \\ A_{N1} & \cdots & A_{NN} \end{bmatrix} = \begin{bmatrix} G_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ G_{N1} & \cdots & G_{NN} \end{bmatrix} \begin{bmatrix} G_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ G_{N1} & \cdots & G_{NN} \end{bmatrix}^T \tag{2.2}$$

La matriz $A \in \mathbb{R}$ se particiona en $N \times N$ submatrices $A_{ij} \in \mathbb{R}^{\mathrm{r}\times\mathrm{r}}$, donde $N$ es tal que $n = Nr$. Si se equipara la triangular inferior se puede deducir la siguiente igualdad:

$$A_{ij} = \sum_{k=1}^{j} G_{ik} {G_{jk}}^T$$

De ahí se define una nueva matriz $S$ tal que:

$$S = A_{ij} - \sum_{k=1}^{j-1} G_{ik} G_{jk}^T$$

Cuando $i = j$, es decir cuando el bloque se encuentra en la diagonal, la submatriz $G_{jj}$ se determina realizando su factorización de Cholesky ($S = G_{ij}G_{ij}^T$). Finalmente, para el resto de los bloques de la columna ($i > j$), se resuelve el sistema de ecuaciones múltiple $G_{ij}G_{jj}^T = S$ donde $G_{ij}$ son las incógnitas del sistema. Este proceso se realizará para cada columna de la matriz.

**Algoritmo 2.2.1** (**Algoritmo por bloques Cholesky**) dada una matriz $A \in \mathbb{R}^{n\times n}$ particionada en $N$ bloques tal que $n = Nr$, se pretende calcular el factor de Cholesky $G \in \mathbb{R}$. La parte triangular inferior de $A$ es sobrescrita por la parte triangular inferior de G

1: **for** $j = 1:N$ **do**
2:  **for** $i = j:N$ **do**
3:  $S = A_{ij} - \sum_{k=1}^{j-1} G_{ik}G_{jk}^T$
4:  **if** $i = j$ **then**
5:  Calcular factorización de Cholesky $S = G_{jj}G_{jj}^T$
6:  **else**
7:  Resolver $G_{ij}G_{jj}^T = S$ para $G_{ij}$
8:  **end if**
9:  $A_{ij} = G_{ij}$
10:  **end for**
11: **end for**

El proceso tiene un coste de $n^3/3$ como la anterior implementación de la factorización (2.1.1). Aunque el algoritmo está diseñado para el caso en el que $r$ divide a $n$, este puede ser modificado para el caso contrario. De hecho, el método implementado para la realización de este proyecto si que cubre este caso.

# Capítulo 3

# Algoritmo de Cholesky *out-of-core*

El algoritmo clásico de Cholesky por bloques presentado en el anterior capítulo puede ser implementado directamente de forma fácil en cualquier lenguaje de programación. Sin embargo, teniendo en cuenta las dependencias existentes entre las operaciones del proceso, este puede ser modificado para que, por ejemplo, acceda a los bloques de forma eficiente reduciendo al máximo el número de lecturas y escrituras, ya que no hay que olvidarse de la importancía de las operaciones de I/O en los algoritmos por bloques. Además del acceso a los datos, el hecho de que el algoritmo sea flexible a la hora de su implementación también permite crear esquemas más paralelizables que permitan realizar las operaciones matriciales de forma más eficiente.

Es este cápitulo se exponenen dos algoritmos out-of-core basados en la versión por bloques del algoritmo de Cholesky, dando especial importancía a la orientación en la que se realizan las operaciones. El primero, tiene una orientación por filas, mientras que el segundo realiza las operaciones por columna. Ambos tienen sus ventajas y desventajas que serán analizadas en las proximas secciones. Por otro lado, el almacenamiento de la matriz es un factor a tener en cuenta puesto que cómo se diseñe la estructura de datos puede afectar también al rendimiento general del proceso. Para la estructura se han planteado 2 opciones diferentes.

## 3.1 Estructura de datos

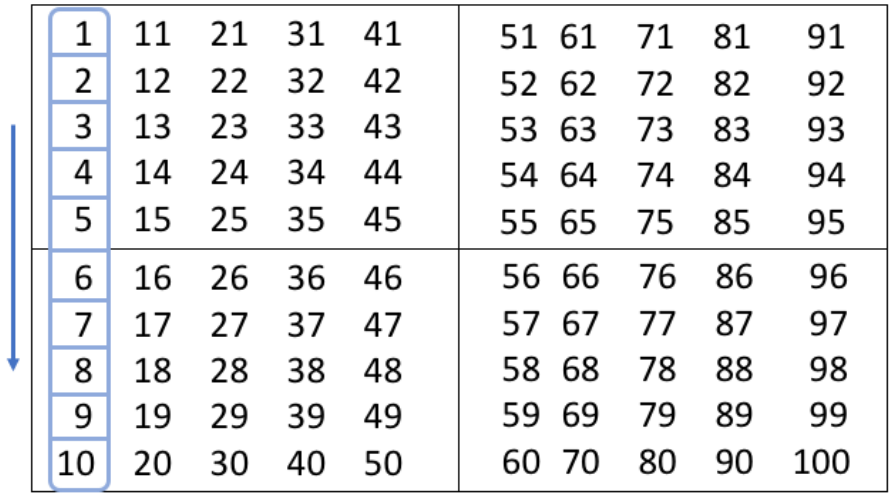
Uno de los aspectos más importante a la hora de diseñar una estructura de datos en la que almacenar una matriz por bloques, es la facilidad con la que posteriormente se accederá a estos, ya que el coste que puede tener tanto el posicionamiento del puntero como la posterior lectura de los bytes, puede lastrar de forma importante el rendimiento general del algoritmo cuando la matriz es de dimensiones grandes.

Por otro lado, en este proyecto también se ha buscado que la generación de la estructura no suponga un coste elevado y que la lógica utilizada para la selección del bloque no sea compleja y sea fácil de implementar.

### 3.1.1 Matriz por columnas

El primer método de distribución planteado es el conocido como *Column-major Order*. Esta distribución puede ser considerada como un estándar a la hora de representar matrices y vectores en la computación científica. Librerías como la MKL de Intel, OpenGL y OpenGL ES lo utilizan como su distribución por defecto. También en algunos lenguajes como Fortran y MATLAB.

Básicamente, este método se basa en almacenar cada elemento de las columnas de una matriz contiguamente en memoria. Esto quiere decir que la matriz es almacenada como si fuera un vector de longitud $M \times N$. Pudiéndose aprovechar así de las instrucciones SIMD (*Single Instruction, Multiple Data*).



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 11 | 21 | 31 | 41 | 51 | 61 | 71 | 81 | 91 |
| 2 | 12 | 22 | 32 | 42 | 52 | 62 | 72 | 82 | 92 |
| 3 | 13 | 23 | 33 | 43 | 53 | 63 | 73 | 83 | 93 |
| 4 | 14 | 24 | 34 | 44 | 54 | 64 | 74 | 84 | 94 |
| 5 | 15 | 25 | 35 | 45 | 55 | 65 | 75 | 85 | 95 |
| 6 | 16 | 26 | 36 | 46 | 56 | 66 | 76 | 86 | 96 |
| 7 | 17 | 27 | 37 | 47 | 57 | 67 | 77 | 87 | 97 |
| 8 | 18 | 28 | 38 | 48 | 58 | 68 | 78 | 88 | 98 |
| 9 | 19 | 29 | 39 | 49 | 59 | 69 | 79 | 89 | 99 |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

**Figura 3.1:** Distribución por columnas, los elementos dentro de la figura azul son contiguos en memoria.

Si nos centramos en la aplicación de esta distribución en algoritmos por bloques, no es díficil ver cual va ser su principal desventaja, no es posible leer un bloque de una sola vez. Esto obliga a iterar sobre las columnas del bloque realizando un posicionamiento y lectura por cada una. Creando, por consiguiente, una sobrecarga en el proceso que se debe evitar.

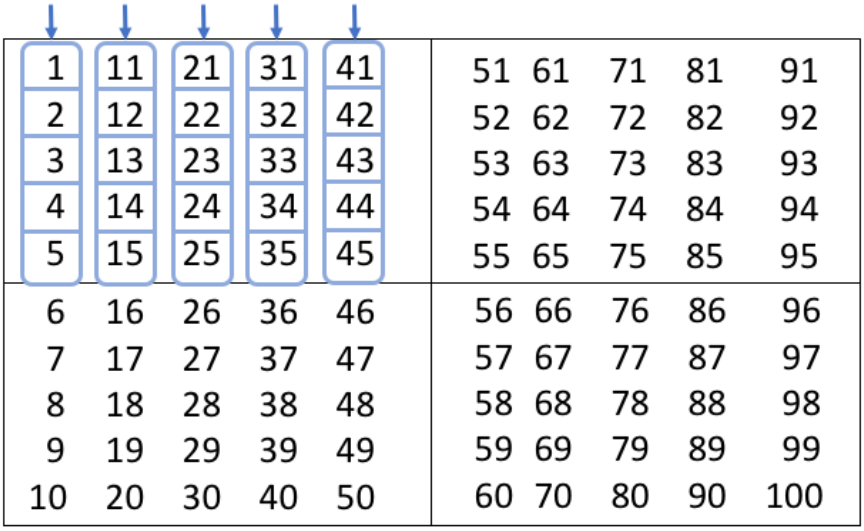



**Figura 3.2:** Para una matriz $10 \times 10$ partida en bloques de $5 \times 5$, para obtener el primer bloque es necesario realizar 5 lecturas.

A la hora de implementar la lectura de bloques como la representada en la figura 3.2, es necesario en primer lugar calcular aritméticamente los saltos necesarios para colocar el cursor al principio de cada columna. La formula sería la siguiente:

$$\text{offset} = (i + nbc \times tb) * N + (j + nbf \times tb)$$

donde $i$ es el índice de la columna, $nbc$ el índice de la columna a nivel de bloque, $tb$ el tamaño de bloque, $N$ el número de filas, $j$ el índice de la fila y $nbf$ el índice de la fila a nivel de bloque. La figura 3.3 representa como sería la subrutina implementada en el lenguaje C.

```
1  void readBlock1(double *block, int N, int nbf, int nbc, int tb
       , FILE *file)
2  {
3      int i;
4      double *pointerBlock;
5
6      for(i = 0; i < tb; i++)
7      {
8          pointerBlock = (block + tb * i);
9          long offset = (i + (long)nbc * tb) * N + (0 + nbf * tb
               );
10         fseek(file, 2 * sizeof(int) + offset * sizeof(double)
               ,SEEK_SET);
11         fread(pointerBlock, sizeof(double), tb, file);
12     }
13 }
```

**Figura 3.3:** Subrutina en C capaz de leer un bloque para la distribución *Column-major Order*

En el caso de la escritura sería equivalente a la lectura solo que se cambiaría la función encargada de leer los *bytes* por una que los escriba en disco.

En definitiva, la distribución por columnas presenta la ventaja de ser una opción muy extendida en el ámbito científico y por tanto compatible con muchas otras aplicaciones. Por el contrario, el hecho de que no se puedan leer los bloques con una sola lectura en memoria provoca, en el algoritmo de Cholesky, una sobrecarga importante ya que en su ejecución se producen numerosas lecturas y escrituras de bloques.

### 3.1.2 Matriz por columnas de bloques

Esta segunda alternativa nace con el objetivo de subsanar el principal incoveniente de la estructura planteada en el apartado 3.1.1. Partiendo de la misma base que la anterior, en este caso los elementos contiguos en memoria no son los de las columnas de la matriz grande, sino aquellos que componen las columnas de los bloques en los que se haya particionado la matriz. En la figura 3.4 se puede ver como quedaría un bloque en la nueva distribución. Igual que en el apartado anterior, los elementos en las columnas son contiguos en memoria.

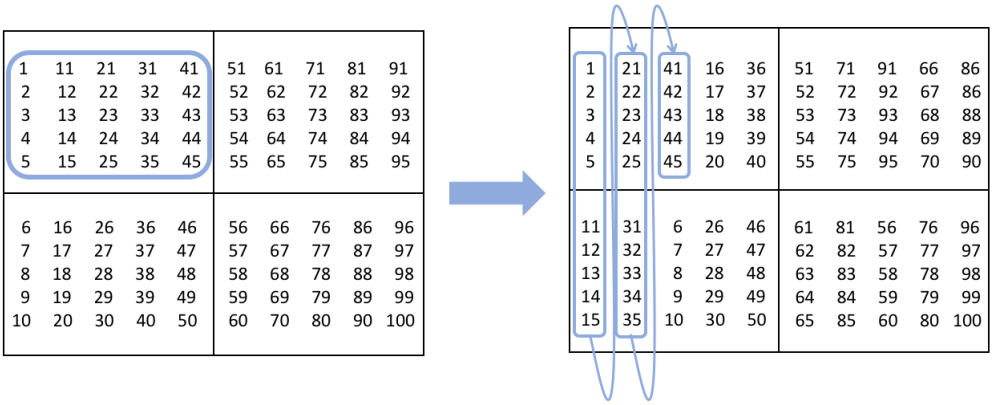



**Figura 3.4:** Transición de matriz por columnas a columnas de bloques

Con esta nueva distribución es posible leer cualquier bloque de una sola lectura. Además, puesto que los bloques en sí son contiguos, es posible leer más de un bloque en una sola vez, característica que será explotada en los algoritmos planteados en este proyecto. En la figura 3.4, se muestra como querdaría un bloque en memoria con la nueva distribución. A la izquierda se muestra el formato estándar, donde cada columna del bloque está separada por tantos elementos como filas tenga la matriz. A la derecha se muestra la nueva distribución donde todos los elementos del bloque estan contiguos en memoria.

Para calcular la posición de los bloques en memoria la fórmula no difiere mucho de la presentada en el anterior apartado:

$$\text{offset} = nbf \times tb^2 + nbc \times NB \times tb^2$$

donde $nbf$ es el índice de la fila a nivel de bloque, $tb$ el tamaño de bloque, $nbc$ es el índice de la columna a nivel de bloque y $NB$ es el número de bloques. En la figura 3.5 se muestra como sería una subrutina para leer los bloques para esta estructura.

```
1 void readBlock2(double *block, int NB, int nbf, int nbc, int
      tb, int num_blocks, FILE *matrix)
2 {
3     long foffset, coffset;
4     foffset = (long)nbf * tb * tb;
5     coffset = (long)nbc * NB * tb * tb;
6     fseek(matrix, (foffset + coffset) * sizeof(double) + 2 *
          sizeof(int), SEEK_SET);
7     fread(block, sizeof(double), (long)tb * tb * num_blocks,
          matrix);
8 }
```

**Figura 3.5:** Subrutina en C para leer uno o más bloques para la distribución columnas de bloques

De la misma forma que en 3.1.1, la subrutina para escribir en disco los bloques de la matriz es equivalente al algoritmo 3.5 pero sustituyendo la función de lectura por su equivalente de escritura.

La principal ventaja de este modelo frente al anterior es, como hemos dicho anteriormente, el hecho de poder leer los bloques sin tener que realizar más de un posicionamiento, esta característica nos permite reducir las lecturas y escrituras al mínimo. Por otro lado, la principal desventaja es que, al no ser un modelo estándar, para utilizarlo es problable que se necesite de un paso intermedio para transformar la matriz. En ambos casos, se reservan 8 bytes al principio del fichero para indicar el tamaño de la matriz y el tamaño de bloque.

## 3.2 Estructura de los algoritmos

Tomando como base el algoritmo 2.2.1 planteado en el capítulo 2, para este proyecto se han implementado dos variantes. Ambas tiene el objetivo de aprovechar al máximo las ventajas del modelo de datos diseñado en el apartado 3.2.2. Además, a sabiendas de la importancía de la concurrencia en la computación científica, se ha tratado de diseñar un esquema que permita una paralelización "económica" tanto de coste de implementación como computacional.

En los dos métodos se utiliza la librería matemática de intel, MKL, para las operaciones con matrices.

### 3.2.1 Algoritmo por filas

La denominación de algoritmo por filas le viene dada por el hecho de que las operaciones se realizan fila por fila. Empezando por la factorización de Cholesky del bloque en la diagonal principal, hasta la resolución del sistema lineal múltiple para el resto de bloques de la columna. El algoritmo, por tanto, se podría dividir en dos fases: la factorización de Cholesky y la resolución de los sistemas múltiples. Además hay una terceda fase que se realiza de forma implícita en la dos anteriores, la actualización de la matriz.

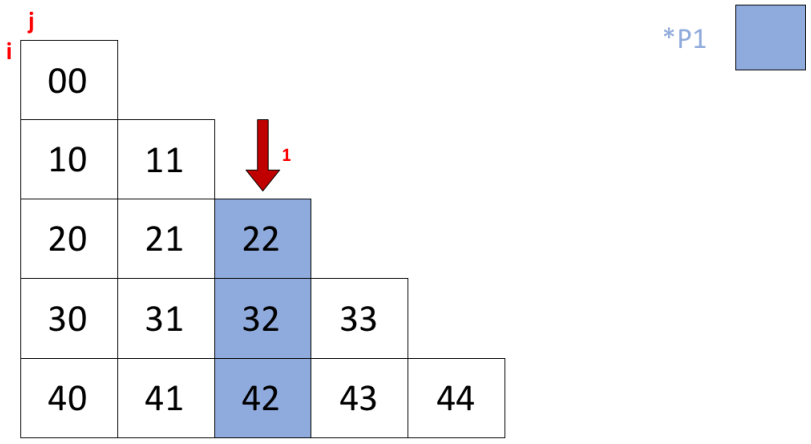



**Figura 3.6:** Primera fase: Lectura de los bloques de la columna

En primer lugar se procede a la lectura de los bloques que van a ser computados, en el caso de la figura 3.6 serían desde $A_{22}$ hasta $A_{42}$. Cabe puntualizar que los bloques no son escritos directamente desde el primer registro de la memoria que ha sido reservada, sino que se desplaza el puntero $j$ bloques de tal forma que quede espacio para los situados en la fila $j$, $A(j, 0{:}j-1)$, que serán utilizados tanto en la fase uno como posteriormente en la fase dos.

De este modo se evita el tener que reservar más memoria de la extrictamente necesaria.

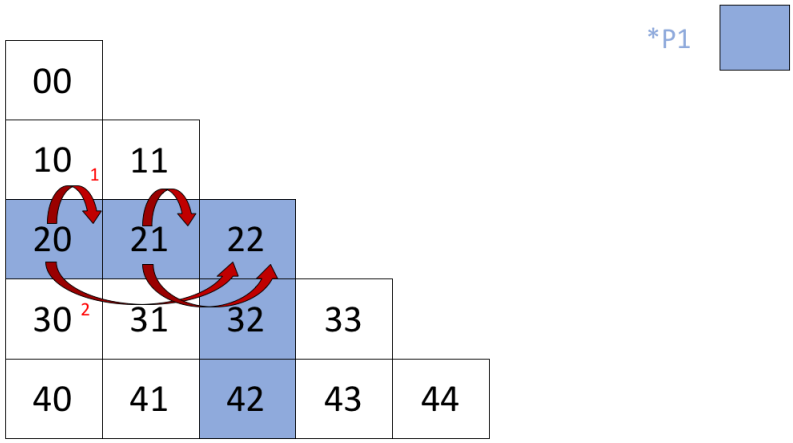



**Figura 3.7:** Primera fase: Computo de $S = A_{jj} - \sum_{k=0}^{j-1} G_{jk} {G_{jk}}^T$

Una vez realizada la lectura de la columna $j$, se procede a realizar la computación del bloque auxiliar, $S$. En el ejemplo de la figura 3.7, la operación sería la siguiente: $S = A_{22} - G_{20}{G_{20}}^T - G_{21}{G_{21}}^T$. Primero se lee el bloque $G_{20}$, a continuación se realiza la multiplicación, y finalmente, se realiza la resta sobre $A_{22}$. Esto se repetirá para cada bloque de la fila hasta $j-1$. A la hora de almacenar el resultado se sobreescribe en $A_{22}$.

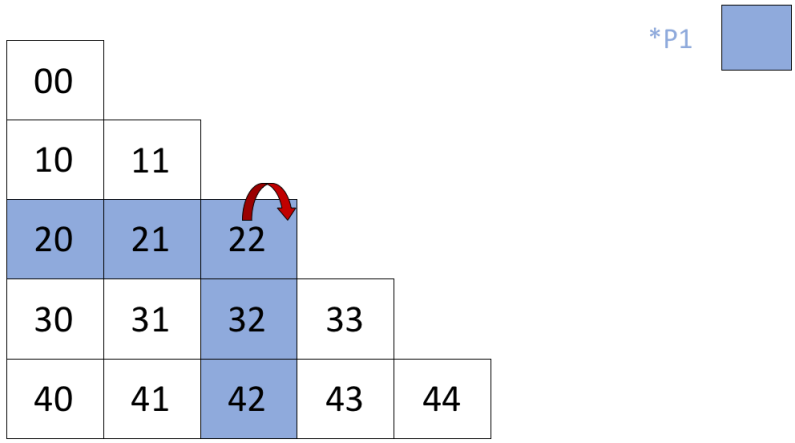



**Figura 3.8:** Primera fase: factorización de Cholesky $S = G_{jj}Gjj^T$

Finalmente, para terminar la fase uno, se realiza la factorización de Cholesky del bloque $S$. De la misma forma que en el paso anterior, el resultado se guarda sobre el bloque $S$ y posteriormente se escribe en memoria.

El método sigue con la fase dos, en la se obtendrán el resto de bloques $G$ de la columna. Puesto que las particiones de la matriz substraidas en la fase anterior van a ser utilizadas en esta, es necesario crea un puntero nuevo para la lectura de los bloques que van a ser utilizados en el computo de $S$ en esta última parte del algoritmo.

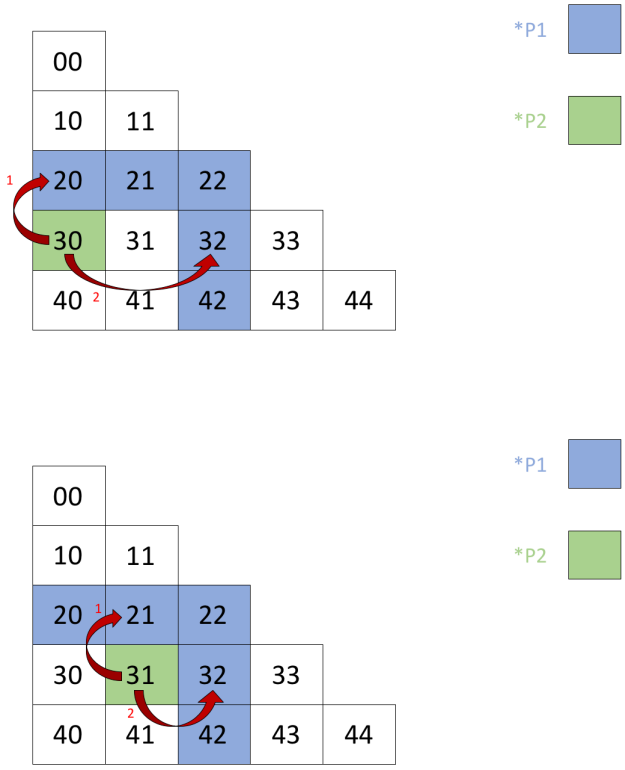



**Figura 3.9:** Segunda fase: Computo de $S = A_{ij} - \sum_{k=0}^{j-1} G_{ik} {G_{jk}}^T$

De modo equivalente a la fase uno, se realiza el cómputo del bloque $S$ antes de resolver los sitemas lineales múltiples. El primer paso es almacenar el bloque pivote en el nuevo puntero $^*P2$, que en el caso de la figura 3.9 corresponde a $G_{30}$ en la primera iteración y $G_{31}$ en la segunda. Una vez leído el bloque pivote, se realiza el producto de este con el bloque situado en la fila $j$, transpuesto (${G_{30}G_{20}}^T$ y ${G_{31}G_{21}}^T$). Finalmente, el resultado del producto es restado al bloque de la columna de la cual se está calculando el factor, $G_{32}$ en el caso de la figura de ejemplo.

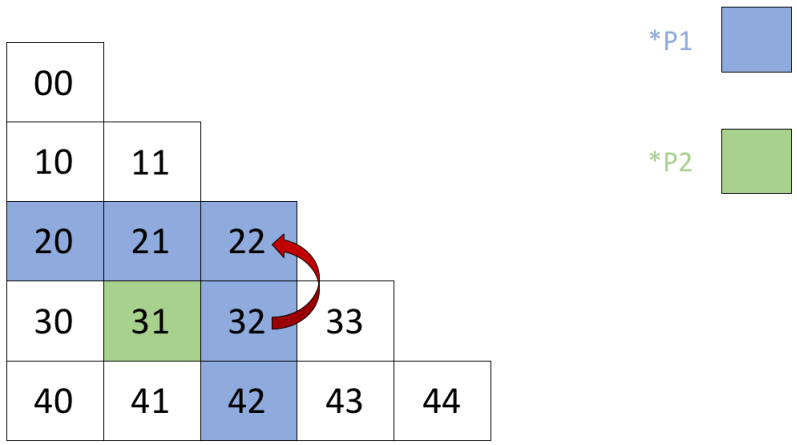



**Figura 3.10:** Segunda fase: Resolución del sistema $G_{ij}G_{jj}^{T} =$ para $G_{ij}$

Por último, se resuelve el sistema lineal múltiple con el bloque que hemos factorizado anteriormente en la fase uno y el bloque $S$. El resultado del proceso se sobreescribe en el puntero y luego en el fichero de entrada. Este proceso se repetirá para todas las filas restantes y ambás fases se repetiran para todas las columnas.

**Algoritmo 3.2.1** (**Algoritmo por filas**) Dado un puntero a un fichero donde está almacenada la matriz, el número de bloques ($N$) y el tamaño de bloque ($\#b$). Este algoritmo calcula el factor de Cholesky $G$ por bloques. El resultado se sobreescribe en el mismo fichero de entrada.

1: Reservar $(N \times \#b^2) \times sizeof(double)$ *bytes* de memoria para $^*P1$
2: Reservar $\#b^2 \times sizeof(double)$ *bytes* de memoria para $^*P2$
3: **for** ( $j = 0$; $j < N$; $j$++ ) **do**
4: Lectura bloques $A[j\!:\!N-1, j]$ ▷ Se almacena en $^*P1$
5: **for** ( $k = 0; k < j; k$++ ) **do**
6: Lectura bloque $G_{jk}$ ▷ Se almacena en $^*P1$
7: $A_{jj} = A_{jj} - G_{jk}G_{jk}^T$
8: **end for**
9: Calcular factorización de Cholesky $A_{jj} = G_{jj}G_{jj}^T$
10:
11: **for** ( $i = j+1$; $i < N$; $i$++ ) **do**
12: **for** ( $k = 0$; $k < j$; $k$++ ) **do**
13: Lectura bloque $G_{ik}$ ▷ Se almacena en $^*P2$
14: $A_{ij} = A_{ij} - G_{ik}G_{jk}^T$
15: **end for**
16: Resolver $G_{ij}G_{jj}^T = A_{ij}$ para $G_{ij}$
17: **end for**
18: Escribir en disco $G[j\!:\!N-1, j]$
19: **end for**

Si se alnaliza el coste teórico del algoritmo, se aprecia que no varía mucho del original presentado en el capítulo anterior 2.2.1. Hay dos bucles *for* principales que recorren las columnas y las filas respectivamente. A cada iteración hay que sumarle el coste de las operaciones matriciales que componen el algoritmo. Esto se traduce en un coste de $n^3/3$ flops. A esto habría que sumarle también el coste de las lecturas y escrituras que tambíen resultan determinantes en coste global del algoritmo.

En terminos de rendimiento, el algoritmo consume una memoria aproximada de $NB \times \#b^2 + \#b^2$, donde $NB$ es el número de bloques y $\#b$ el tamaño de bloque. Por otro lado, tomando como ejemplo la matriz utilizada como apoyo para describir el algoritmo, se aprecia que en el transcurso total del método se realizan 25 lecturas, muchas de ellas repetidas. Esto es debido a

la naturaleza por columnas de la estructura de datos utilizada. Este punto, claramente negativo, se intenta subsanar en el próximo algoritmo.

### 3.2.2 Algoritmo por columnas

Del mismo modo que el algoritmo expuesto en el anterior apartado, el algoritmo por columnas puede dividirse en dos fases: El cálculo de las $S$ y el cálculo de los bloques de $G$.
Con el objetivo de aprovechar la estructura de datos presentada en 3.1.2, las operaciones pasan de realizarse fila a fila, a realizarse columna a columna. Aplicando cada paso a todos los bloques a calcular. De este modo, en la primera fase se computan todas las $S$ de la columna para después, en la segunda fase, calcular cada bloque de $G$.

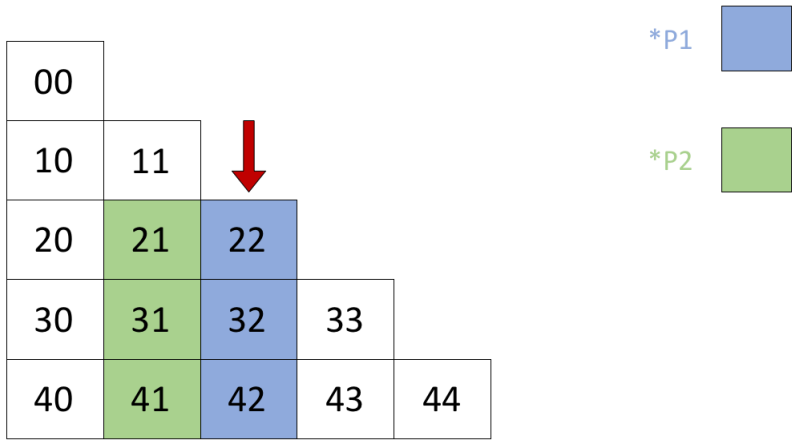



**Figura 3.11:** Primera fase: Lectura de la columna a computar

Al igual que en el algoritmo por filas, el primer paso es leer la columna a computar. Cabe destacar que en el caso de este algoritmo, al final de cada iteración, se realiza una copia de la columan calculada en $^*P1$ a $^*P2$. De este modo, se elimina una lectura más en cada iteración. Reduciendo aún más si cabe el número de lecturas con respecto al algoritmo anterior.

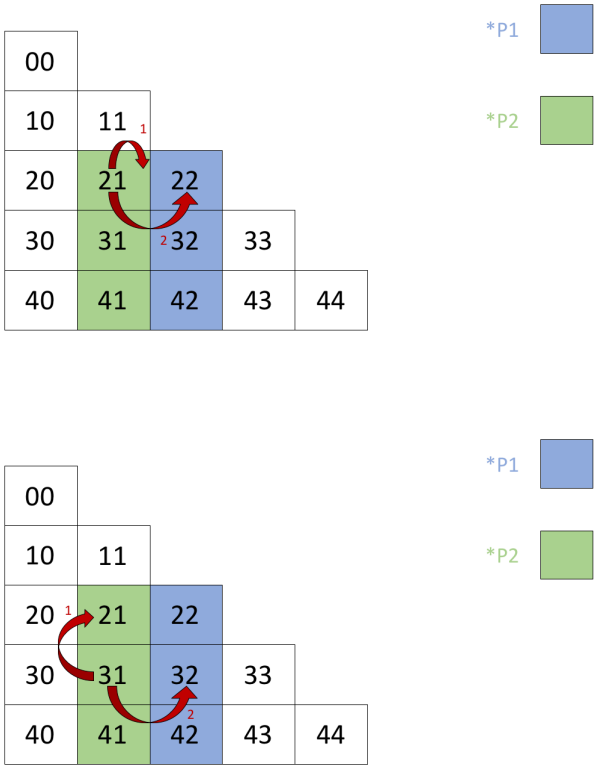



**Figura 3.12:** Primera fase: Computo de todas las $S$

Una vez leída la columna, se procede a realizar el computo de las $S$s. Para ello se itera sobre las columnas desde $j-1$ hasta 0, calculando, para cada bloque, los productos y restas parciales que finalmente darán como resultado los bloques $S$ buscados. Puesto que todos los elementos necesarios para realizar está fase se encuentran en memoria y no existen dependencias entre las operaciones, estas podrían realizarse en paralelo incrementando de esta forma el rendimiento general de la apliación, en el caso, claro, de no contar con librerías que se aprovechen de forma eficiente de la concurrencia.

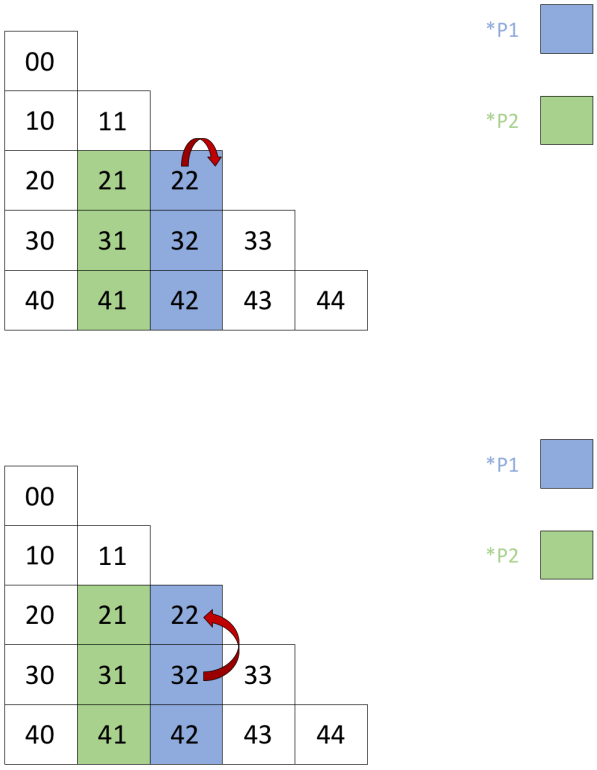



**Figura 3.13:** Segunda fase: Factorización Cholesky y resolución de los sistemas lineales múltiples

Finalmente, se procede a calcular los bloques de $G$ de la columna leída. Análogamente al anterior algoritmo, en primer lugar calculamos la factorización de Cholesky del primer bloque de la columna. En el caso de la figura 3.13, se trata del bloque $G_{22}$. Calculado el factor, se sigue con la resolución de los sistemas múltiples, que en el caso de la figura sería ${G_{32}G_{22}}^T = S_{32}$ para $G_{32}$. Como salta a la vista el orden de las operaciones es importante en esta fase, es por eso que la paralelización no puede ser total, pero si que puede ser aprovechada en la resolución de los sistemas.

**Algoritmo 3.2.2** (**Algoritmo por columnas**) dado un puntero a un fichero que contiene la matriz estructurada por columnas de bloque, dado el número de bloques ($N$) y el tamaño de bloque ($\not{b}$). Este algoritmo calcula la factorización de Cholesky de la matriz dada. El resultado se guarda sobreescribiendo sobre el fichero de entrada

1: Reservar $(N \times \not{b}^2) \times sizeof(double)$ *bytes* de memoria para $^*P1$
2: Reservar $(N \times \not{b}^2) \times sizeof(double)$ *bytes* de memoria para $^*P2$
3: **for** ( $j = 0$; $j < N$; $j$++ ) **do**
4: Leer bloques $A[j:N-1, j]$ ▷ Se almacena en $^*P1$
5: $primero = 1$
6: **for** ( $k = j-1$; $k >= 0$; $k$++ ) **do**
7: **if** *primero* **then**
8: Leer bloques $G[j:N-1, k]$ ▷ Se almacena en $^*P2$
9: **end if**
10: $A_{jj} = A_{jj} - G_{jk}G_{jk}^T$
11: **for** ( $i = j+1$; $i < N$; $i$++ ) **do**
12: $A_{ij} = Aij - G_{ik}G_{jk}^T$
13: **end for**
14: $primero = 0$
15: **end for**
16: Calcular factorización de Cholesky $A_{jj} = G_{jj}G_{jj}^T$
17: **for** ($i = j+1$; $i < N$; $i$++) **do**
18: Resolver $G_{ij}G_{jj}^T = A_{ij}$ para $G_{ij}$
19: **end for**
20: Copiar puntero $^*P1$ en $^*P2$
21: Escribir en disco $G[j:N-1, j]$
22: **end for**

Comparando este método con el anterior, cogiendo como ejemplo el escenario de las figuras ilustrativas de este capítulo, en primer lugar se aprecia una reducción grande del número de lecturas pasando de 25 en el algoritmo por filas a 11 en el algoritmo por columnas, una reducción del 56 %, que en tamaños mayores es más notoria. Por otro lado, no hay niguna mejora aparente en cuanto a las operaciones matemáticas, ya que solo se ha cambiado el orden en el que se calculan, por tanto podemos asumir que el coste teórico en flops es equivalente al anterior. En lo que concierne al consumo de memoria, en esta segunda alternativa el consumo asciende a casi el doble,

$2 \times (NB - 1) \times tb^2$, lo cual puede representar un gran incoveniente en el caso de que el tamaño de bloque afecte al rendimiento general del algoritmo.

# Capítulo 4

# Resultados experimentales

En este capítulo se exponen y analizan los resultados obtenidos de las pruebas realizadas con los dos algoritmos, comparandolos entre ellos y finalmente con la función de la librería MKL, *dpotf*, que computa la factorización de Cholesky. Además, también se comparan las dos estructuras de datos para ver si, efectivamente, existe una diferencia significativa entre ellas. Por otro lado, antes de analizar los resultados, describiremos tanto el *hardware* utilizado como el *software* con el que hemos desarrollado el proyecto.

## 4.1 *Hardware* y *software* utilizado

El *hardware* en el que se ha ejecutado el algoritmo desarrollado tiene las siguientes especificaciones:

| Componente | Descripción |
|---|---|
| SO | Ubuntu 16.04.2 LTS x86_64 |
| CPU | 2 × Intel Xeon E5-2697 v3 2,60 GHz |
| # Procesadores | 2 |
| # Núcleos por procesador | 14 |
| RAM | 16 × 8 GB 2,13 GHz |

**Tabla 4.1:** Especificaciones del *cluster* usado para los experimentos

El componente más destacado sería sin duda la CPU de Intel, lanzada en 2014, cuenta con 14 núcleos a una frecuencia de 2,60 GHz que pueden llegar a 3,60 GHz si está activada la tecnología Turbo Boost. Además, cuenta con 35 MB de meoria caché que funciona con la arquitectura *SmartCache* que permite a todos los núcleos compartir dinámicamente el acceso a esta memoria de alto nivel. En cuanto a la memoria principal, la máquina cuenta con un total de 128 GB capaz de almacenar una matriz de hasta $131\,000 \times 131\,000$ *doubles*.

Si nos centramos en el *software*, todo del proyecto ha sido desarrollado usando el lenguaje de programación C. La elección de este lenguaje se debe, principalmente, a su velocidad y a la flexibilidad que proporciona a la hora de manejar tanto punteros como la memoria reservada para ellos. Esto permite una mejor optimización y por tanto un mejor rendimiento del proceso. Adicionalmente, C cuenta con librerias matemáticas potentes como por ejemplo la MKL de Intel, usada en este proyecto. Esta libreria no solo cuenta con funciones de alto nivel como la factorización de Cholesky y la resolución de sistemas lineales múltiples, sino que también viene con una paralelización por defecto que permite aprovechar al máximo la CPU *multicore* sin necesidad de añadir más complejidad al código base.
Finalmente, el elemento más importante del software utilizado para este trabajo es el compilador de Intel *icc*, no solo porque es necesario para poder utilizar la MKL, sino que también por todas las opciones de optimización que trae consigo.
Para compilar el programa se ha utilizado la siguiente sentencia:

```
1  $ icc -c -O3 -xHost ./src/ctimer.c ./src/blockCholeskyMKL.
       c ./src/utils.c -I./include
2  $ icc -o blockCholesky ./src/main.c ctimer.o
       blockCholeskyMKL.o utils.o -I./include -mkl -
       D_FILE_OFFSET_BITS=64
```

En estos dos comandos cabe destacar algunos elementos:

- **03**: Esta opción realiza optimizaciones en los bucles, optimizaciones interprocedurales dentro del ficheo y incorporación de funciones intrínsecas del procesador. Estas operaciones afectan sobre todo a los bucles y a los accesos a memoria.

- **xHost**: Esta opción permite al compilador incorporar optimizaciones especifica de cada procesador. En el caso concreto de *xHost*, el compilador en primer lugar determina cual es el procesador disponible para luego realizar las optimizaciones.
- **mkl**: Esta es el *switch* que permite la utilización de las funciones de la librería matemática de Intel.

## 4.2 Resultados

En primer lugar realizamos un estudio de las estructuras de datos utilizadas como entrada en ambos algoritmos, con ello se intenta comprobar si efectivamente existe una mejora notable en utilizar la estructura por columnas de bloques en contraposición a la estructura *Column-major*. Para ello primero calculamos cuanto se tarda en leer un bloque y finalmente cuanto se tarda en leer varios.

En el caso de la lectura de una sola porción de matriz, aunque la estructura por columnas de bloques es más rápida, la diferencia entre las dos opciones es casi inexistente. La **figura 4.1** refleja de forma clara este fenómeno.

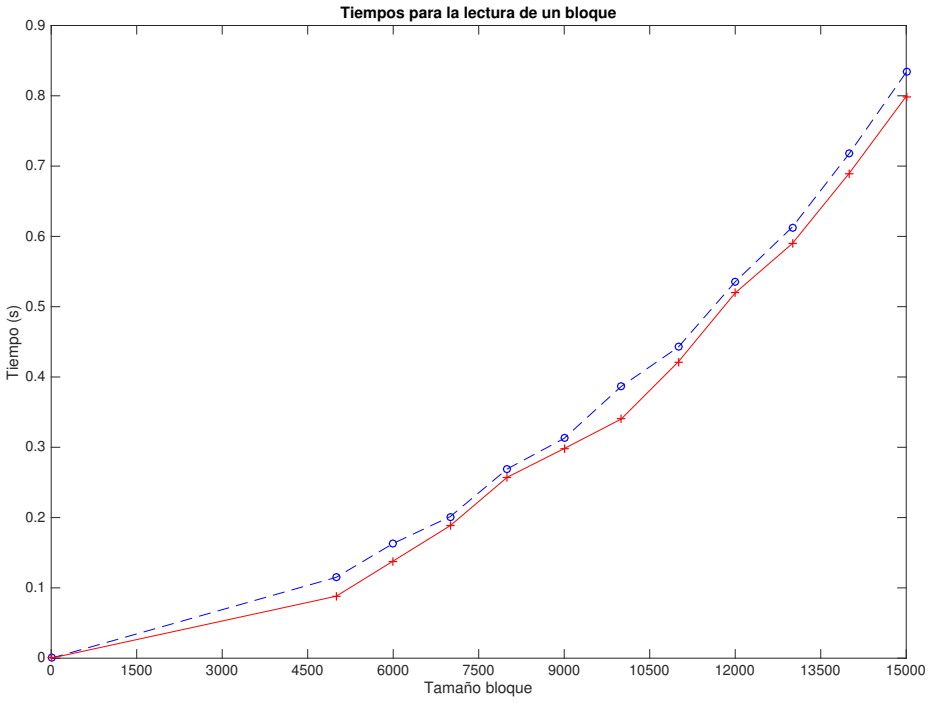



**Figura 4.1:** Tiempos para la lectura de un bloque.

En cambio, en el caso de leer varios bloques es donde la estructura diseñada para este proyecto supera ampliamente a la estrutura estándar. En la **tabla 4.2** son reflejados los resultados obtenidos en las pruebas realizadas.

| Tamaño bloque | T.Bloques($s$) | T.normal($s$) |
|---|---|---|
| 5000 | 0,764 | 0,958 |
| 6000 | 0,851 | 1,100 |
| 7000 | 0,808 | 1,086 |
| 8000 | 0,908 | 1,184 |
| 9000 | 0,861 | 1,180 |
| 10000 | 1,081 | 1,499 |
| 11000 | 0,814 | 1,348 |
| 12000 | 1,020 | 1,586 |
| 13000 | 0,591 | 1,242 |
| 14000 | 0,683 | 1,388 |
| 15000 | 0,768 | 1,682 |

**Tabla 4.2:** Tabla con los tiempos medidos para la lectura de varios bloques seguidos

Si obtenemos el tiempo promedio y calculamos el *Speedup*:

$$S_{execution} = \frac{T_{normal}}{T_{Bloques}} = \frac{1,29}{0,832} = 1,55$$

Esto se traduce en una mejora media de un 55 % en velocidad de lectura.En el caso de los tamaños de bloque más grandes (13000, 14000, 15000), el *Speedup* sobrepasa el 2.

Una vez mostrados los resulados de los experimentos sobre las estructuras de datos, procedemos a mostrar los tiempos obtenidos en las pruebas realizadas sobre los algoritmos descritos en el **capítulo 3.2**. Primero, empezamos mostrando los tiempos para el algoritmo por filas. En la **tabla 4.3** están expuestos los resultados, desglosados por el tiempo total requerido para descomponer la matriz y resolver el sistema (T.Total), el tiempo consumido en la resolusción del sistema(T.Sistema), el tiempo consumido por la descomposición de Cholesky (T.Cholesky) y finalmente el error relativo, calculado de la siguiente forma: $\frac{||Ax-b||_2}{||A||_f}$.

| N | T.Total | T.Sistema | T.Cholesky | Error relativo |
|---|---|---|---|---|
| 50 000 | 100,42 | 12,58 | 87,83 | 1,6E-14 |
| 100 000 | 651,53 | 58,19 | 593,33 | 7,2E-12 |
| 150 000 | 1986,11 | 136,99 | 1849,11 | 2,9E-11 |
| 200 000 | 4953,65 | 414,28 | 4539,37 | 4,0E-14 |
| 250 000 | 10394,57 | 776,04 | 9618,53 | 4,3E-11 |
| 300 000 | 18157,18 | 1222,49 | 16934,69 | 6,7E-15 |
| 400 000 | 41723,52 | 1960,58 | 39762,94 | 2,9E-11 |
| 500 000 | 77895,1 | 2838,79 | 75056,32 | 2,9E-11 |
| 600 000 | 132090,34 | 3551,69 | 128538,18 | 1,7E-11 |
| 800 000 | 350823,57 | 8384,39 | 342439,18 | 1,2E-10 |
| 1 000 000 | 603624,56 | 9062,59 | 594561,97 | 5,8E-11 |

**Tabla 4.3:** Tabla con los tiempos desglosados de la resolución del sistema($s$) y el error relativo

En un primer vistazo, vemos como a la hora de resolver el sistema lineal de ecuaciones: $Ax = b$. La mayoría del tiempo computacional se invierte en la factorización de la matriz, utilizando el algoritmo de *Cholesky*. En el caso de la matriz $A_{NxN}$ para $N = 1000000$, el proceso total tarda unos $603624s$, lo cual se traduce en aproximadamente 7 días, de ese tiempo *Cholesky* ocupa un 98,49 % de ahí la importancia de la optimización de este proceso. También cabe destacar el bajo error relativo obtenido, fruto de la estabilidad numérica del método[8].

Identificado el peso computacional que tiene la factorización de Cholesky, vamos a centrarnos ahora en estudiar las operaciones internas del algoritmo en base a los tiempos obtenidos en las pruebas expuestos en la **tabla 4.4**. Las mediciones se han obtenido de dividir las operaciones por: operaciones matriciales (multiplicaciones, Cholesky y resolución de sistemas) y operaciones de entrada/salida.

| N | T.Cholesky | T.Operaciones matriciales | T.I/O |
|---|---|---|---|
| 50000 | 87,83 | 68,51 | 19,32 |
| 100000 | 593,33 | 486,58 | 106.75 |
| 150000 | 1849,11 | 1553,08 | 296,03 |
| 200000 | 4539,37 | 3710.72 | 828,65 |
| 250000 | 9618,53 | 7099,74 | 2518,78 |
| 300000 | 16934,69 | 12473,37 | 4461,32 |
| 400000 | 39762,94 | 29154,3 | 10608,64 |
| 500000 | 75056,32 | 53955,18 | 21101,14 |
| 600000 | 128538,18 | 91951,01 | 36587,64 |
| 800000 | 342439,18 | 224512,97 | 117926,21 |
| 1000000 | 594561,97 | 438499,11 | 156062,86 |

**Tabla 4.4:** Tiempos de la factorización de Cholesky ($s$)

En base a estos resultados, vemos como las operaciones matriciales ocupan la mayoria del tiempo computacional del algoritmo con un 75 % de media frente al 25 % de la lectura y escritura de los bloques.

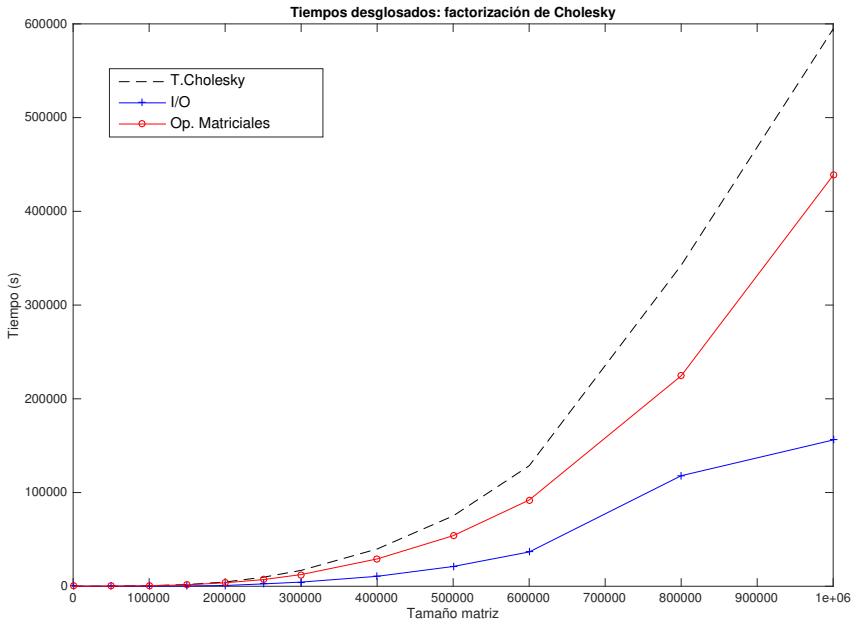



**Figura 4.2:** Gráfico de lineas representando los tiempos de la tabla 4.4

Optimizar las operaciones matriciales no es trivial, ya que con la MKL aprovechamos completamente la capacidad de la CPU gracias a que los métodos de esta librería están paralelizados por defecto. Una posible mejora sería mover ciertas operaciones del algoritmo a la GPU, esto será abordado en las conclusiones. Por otro lado, está la mejora de las operaciones de entrada/salida que pese a que no son dominantes en el peso computacional del proceso, siguien siendo un parte importante de los algoritmos *out-of-core*.

El método por columnas expuesto en el **subsección 3.2.2**, nace con el objeto de mejorar ese 25 %. A continuación, vamos a mostrar los tiempos obtenidos en las pruebas con este algoritmo y comprobar si, verdaderamente, se ha mejorado el tiempo en ese apartado:

| N | T.I/O bloques | T.I/O filas |
|---:|---:|---:|
| 50000 | 14,67 | 19,32 |
| 100000 | 92,34 | 106,75 |
| 150000 | 247,62 | 296,03 |
| 200000 | 1015,24 | 828,65 |
| 250000 | 2270,92 | 2518,78 |
| 300000 | 4588,37 | 4461,32 |
| 400000 | 9855,64 | 10608,64 |
| 500000 | 17511,13 | 21101,14 |
| 600000 | 27084,93 | 36587,64 |
| 800000 | 52447,81 | 117926,21 |
| 1000000 | 86097,70 | 156062,86 |

**Tabla 4.5:** Comparativa entre los tiempos entre el algoritmo por bloques y el algoritmo por filas para la entrada/salida

En general, vemos que los cambios aplicados en la estructura del algoritmo sí que se han traducido en una mejora en los tiempos de lectura y escritura. Concretamente ha mejora una media de un 26 % respecto al algoritmo por filas. En la **figura 4.3** es posible ver claramente la diferencia entre los dos algoritmos.

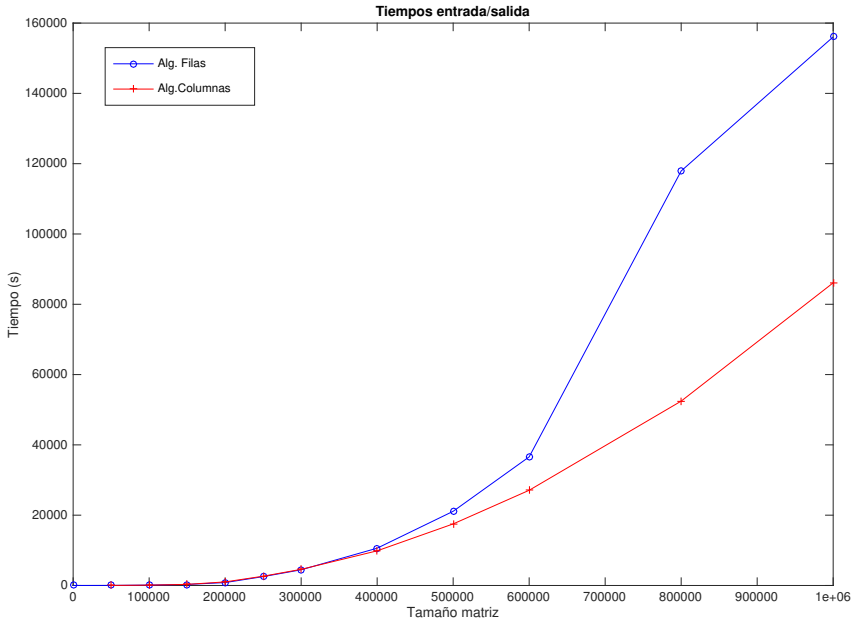



**Figura 4.3:** Tiempos de entrada/salida

Finalmente, para terminar de exponer los resultados, mostramos una gráfica donde se comparan los tiempos de factorizar una matriz simétrica definida positiva utilizando los dos algoritmos desarrollados para este proyecto y además el método proporcionado por la MKL para realizar la factorización.

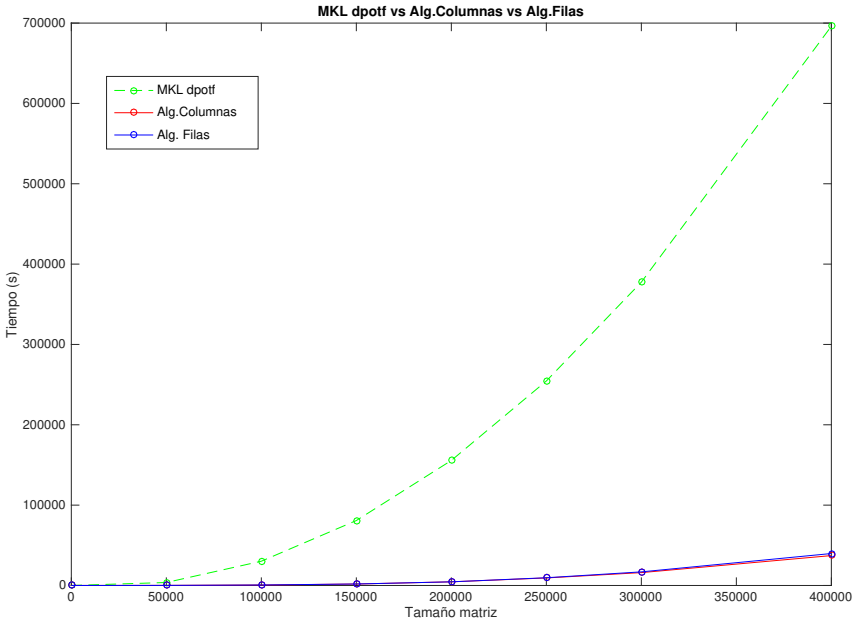



**Figura 4.4:** Tiempos para factorizar una matriz con el algoritmo por filas, por columnas y la librería de MKL, *dpotf2*

Pese al optimizado de la subrutina *dpotf2*, vemos como en tamaños grandes de matrices el tiempo se dispara llegando casi a los $700\,000s$ para el caso de $N = 400000$, mientras que en el caso de los algoritmos *out-of-core* el tiempo de computación no llega ni a los $50\,000s$. A esto habría que añadirle, además, el coste de memoría que tendría almacenar una matriz de $400000 \times 400000$ en la memoría principal.

# Capítulo 5

# Conclusiones y líneas abiertas

## 5.1 Conclusiones

La aparición de fenómenos como las redes sociales, la digitalización de muchos servicios (banca, comercio, etc) y la aparición del denominado "internet de las cosas", donde cada vez más dispositivos están conectados a la red compartiendo información, ha contribuido en el crecimiento exponencial de datos generados en este siglo que se estima que creceran hasta los 163 *zettabytes* para el 2025 [10]. Esto junto con el abaratamiento del almacenamiento de la información, ha hecho que las empresas y otras organizaciones cada vez le den más valor a los datos hasta el punto de que algunos lo consideran el nuevo petroleo [11].


**Figura 5.1:** Caracterización de la cita *"Data is the new oil"*

Es por ello que el desarrollo de técnicas capaces de tratar adecuadamente y eficientemente estos datos, se ha convertido en un desafío importante en el campo de la informática. Ya no solo a la hora de de realizar operaciones complejas en un tiempo razonable, sino tambíen a la hora de realizar operaciones básicas, como lo son la lectura y escritura, que en los algoritmos *out-of-core* se convierten en una parte importante a la hora de buscar la eficiencia.

En el presente proyecto se ha abordado la problemática de tratar cantidades masivas de datos con la implementación de un algoritmo *out-of-core* capáz de factorizar una matriz del orden de *terabytes* mediante el algoritmo de Cholesky. Con los resultados expuestos en el capítulo anterior, podemos concluir que las operaciones complejas, en este caso operaciones matriciales, representan la mayor parte del tiempo computacional. Además, las operaciones de entrada/salida, insignificantes en algoritmos para pequeñas cantidades de datos, se convierten en un factor a tener en cuenta, llegando a representar un 25 % del peso computacional en el caso del algoritmo expuesto en este proyecto.

En cuanto a la optimización de estos tiempos, cabe destacar los buenos resultados obtenidos gracias al diseño tanto de la estructura de datos como de los dos algoritmos *out-of-core*, que no solo superan en rendimiento

a la rutina *dpotf2* de MKL, sino que el algoritmo por columnas es un 26 % más rápido que su homónimo por filas, lo cual demuestra la importancía de desarrollar algoritmos que se aprovechen de las carasterísticas específicas de cada estructura de datos. Finalmente, es necesario indicar que las soluciones expuestas en este proyecto han afectado sobre todo en las operaciones de entrada/salida, lamentablemente mejorar los tiempos proporcionados por la librería MKL para las operaciones matriciales ha quedado fuera del alcance de este proyecto.

## 5.2 Líneas abiertas

Pese a que los resultados obtenidos han sido positivos, no hay que olvidar que la mayor parte del tiempo es consumido por las operaciones matriciales. Si bien con la solución actual la CPU es aprovechada al máximo, el estado del arte de la computación científica actual es el uso de GPU a la hora de realizar cálculos matemáticos. Este hardware, especialmente diseñado para el procesamiento de gráficos, tiene la principal ventaja de realizar operaciones matriciales de forma muy eficiente gracias a utilizar pequeños procesadores que pueden encargarse de miles de operaciones simultáneamente. Por contra, la limitada memoria interna de la que disponen estos dispositivos supone un desafío importante para el desarrollador, que tiene que transportar eficientemente los datos de la memoria RAM a la GPU para que esto no lastre el rendimiento general. El uso de GPU sería especialmente útil para el algoritmo por columnas, **algoritmo 3.2.2**, concretamente en la fase donde se realizan las múltiplicaciones y las restas parciales a la columna de la cual se está calculando el factor de Cholesky.

Finalmente, otra aspecto muy importante que no se ha tenido en cuenta en este proyecto y sería otra buena línea de investigación es el de la eficiencia energética. En un mundo cada vez más concienciado en los problemas que el uso desmedido de energía causa al planeta, ya no vale con crear la máquina con más FLOP por segundo sino también aquella que pueda generar más FLOPS sin un consumo desproporcionado de vatios. Una prueba de ello es la iniciativa del famoso ranking de supercomputadores, TOP500, para valorar aquellos supercomputadores que alcancen un mayor número de FLOPS por vatio con la creación de un nuevo ranking llamado *The GREEN 500*.

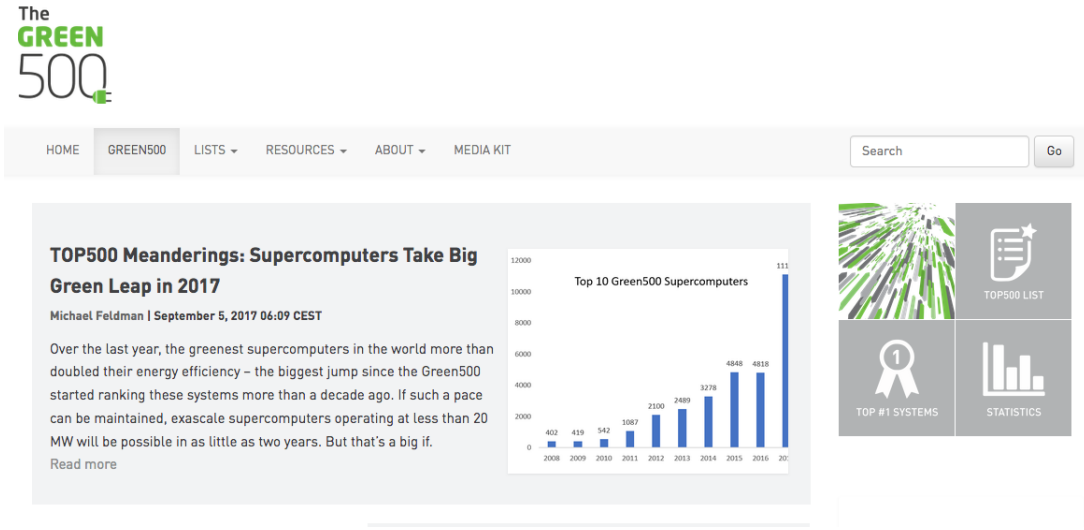



**Figura 5.2:** Imagen de la página web the GREEN 500

# Bibliografía


[1] Gene H. Golub y Charler F. Van Load. *Matrix computations 4th Edition*. Cuarta. 2013.

[2] Sergio Rivas-Gómez, Domingo Giménez y Javier Cuenca. *A wide comparasion between different linear algera libraries using the weel-known LU factorization problem*. Articulo no publicado.

[3] Dusan P. Zoric, Dragan I. Olcan y Branko M. Kolundzija. *Solvinf Electrically Large EM Problems by Using Out-of-Core Solver Accelerated with Multiple Graphical Processing Unit*. Articulo no publicado (vid. pág. 2).

[4] Reuven Y. Rubinstein y Dirk P. Kroese. *Simulation and the Monte Carlo Method*. Tercera. 2017 (vid. pág. 3).

[5] Kai Ni, Drew Steedly y Frank Dellaert. "Out-of-core Bundle Adjustment for Large-Scale 3D Reconstruction". En: *Computer Vision* (oct. de 2007). ISSN: 1550-5499 (vid. pág. 3).

[6] Gene H. Golub y Charler F. Van Load. "Matrix computations 4th Edition". En: Cuarta. 2013, pág. 159 (vid. pág. 5).

[7] Gene H. Golub y Charler F. Van Load. "Matrix computations 4th Edition". En: Cuarta. 2013, pág. 157 (vid. pág. 6).

[8] Gene H. Golub y Charler F. Van Load. "Matrix computations 4th Edition". En: Cuarta. 2013, pág. 160 (vid. págs. 6, 32).

[9] Yang-wang. *Step by Step Performance Optimization with Intel® C++ Compiler*. URL: `https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler`.

[10] David Reinsel, Jonh Gantz y Jonh Rydning. "Data Age 2025: The Evolution of Data to Life-Critical". En: (abr. de 2017) (vid. pág. 37).

[11] The economist. *The world's most valuable resource is no longer oil, but data*. URL: `https://www.economist.com/news/leaders/21721656-data-economy-demands-new-approach-antitrust-rules-worlds-most-valuable-resource` (vid. pág. 37).