UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Trabajo final de master

Diseño e implementación de software paralelo para tratamiento de datos en memoria masiva

Master universitario en computación paralela y distribuida

Autor: Pablo San Juan Sebastián

Tutor: Antonio Vidal Macia

Septiembre 2014

Agradecimientos

Para comenzar, quiero agradecer a mi tutor de proyecto su dedicación y el apoyo que me ha brindado durante la realización de este trabajo, así como el haberme encauzado de nuevo las veces que me he atascado.

A continuación, a mis padres por estar siempre ahí, apoyándome cuando lo he necesitado y ofreciéndome su apoya aun cuando no lo necesitara.

Por último, pero no menos importante, a mis amigos por animarme cuando me vi desbordado y hacerme seguir adelante.

Resumen

El presente trabajo se encarga de resolver la descomposición de Cholesky de matrices de gran tamaño. Dichas matrices de gran tamaño plantean la problemática de que no es posible almacenarlas en memoria para el cálculo de la descomposición y por tanto hay que fragmentar la matriz y los procesos que se realizan durante la descomposición para que sus operandos quepan en la memoria. A este tipo de técnicas se las conoce como técnicas out-of-core debido a que la lectura y escritura de memoria masiva forma parte del propio algoritmo y se lleva a cabo durante la ejecución del mismo. En este tipo de algoritmos se debe minimizar la cantidad de operaciones de entrada salida para evitar penalizar el rendimiento del algoritmo.

Como se desprende del párrafo anterior, este trabajo se encarga de estudiar la descomposición de Cholesky para llevar a cabo una implementación adecuada de un algoritmo out-of-core capaz de descomponer matrices de gran tamaño que excedan la memoria de la máquina en la que se ejecute el algoritmo. Para ello se han estudiado las dependencias de cada uno de los pasos del algoritmo para decidir que partes de la matriz deberán mantenerse en memoria principal y cuales deberán volcarse a memoria masiva en cada momento para conseguir minimizar las transacciones entre ambas memorias.

El trabajo también incluye una importante labor de implementación en la que por un lado se optimiza la ejecución de todos los cálculos necesarios para llevar a cabo la descomposición utilizando librerías de computación de altas prestaciones, mientras por otro lado se utilizan estructuras de datos eficientes para minimizar el espacio necesario para almacenar el problema y la solución, basándose para ello en la estructura de la matriz. También se aprovechan las arquitecturas multinúcleo presentes en la mayoría de equipos destinados a computación científica en la actualidad, con lo que se utilizará la potencia ofrecida por dichas arquitecturas para reducir el tiempo de ejecución.

Índice

Índice de Figuras

Índice de tablas

1. Introducción

Hoy en día debido a los numerosos avances técnicos, tanto en simulación por computador como en equipos de medida, se han producido problemas que manejan un enorme volumen de datos que supera con creces la capacidad de memoria principal de la gran mayoría de computadores. En muchas aplicaciones de ingeniería es necesario resolver dichos problemas de gran tamaño, y cuando superan el tamaño del computador o conjunto de computadores de los que dispone el equipo interesado en la resolución del problema, es necesario recurrir a la memoria masiva o almacenamiento secundario.

En algunos casos, se puede explotar la estructura de la matriz para reducir la memoria necesaria para tratarla y conseguir que entre en memoria principal utilizando técnicas y algoritmos de matrices dispersas. No obstante si los problemas de gran magnitud son densos, no hay forma de almacenarlos en memoria y se vuelve necesario el almacenamiento en memoria masiva y con ello el desarrollo de algoritmos capaces de trabajar con los problemas en este tipo de memoria de la manera más eficiente posible.

Los algoritmos que utilizan memoria masiva son conocidos comúnmente como algoritmos out-of-core y será sobre estos algoritmos sobre los que tratará este trabajo. Como el abordar todo tipo de algoritmos out-of-core es un proyecto de demasiada magnitud, este trabajo se centrara en las descomposiciones matriciales. Esto es debido a que un gran número de aplicaciones de computación utilizan este tipo de descomposiciones en la resolución de sus problemas de gran magnitud. Como estas descomposiciones son a su vez numerosas, este trabajo se centrará más concretamente en la descomposición de Cholesky out-of-core.

1.1. Máquina de trabajo

El trabajo está enfocado a utilizar máquinas de última generación como son procesadores multi-core. Para ello el trabajo se desarrollará sobre la máquina knights.dsic.upv.es y tratará de aprovechar al máximo las capacidades que nos ofrece la máquina. Dicha máquina posee las siguientes características técnicas:

- 2 Procesadores *Intel® Xeon® Processor E5-2697 v2* con 12 núcleos físicos y 24 lógicos cada uno sumando 48 núcleos lógicos en total.
- 128 GB de memoria RAM
- 700GB de espacio en disco disponible

Además posee a nivel software:

- Compilador de Intel icc versión 14.0.1
- Librería matemática Intel mkl
- Librería de paralelismo OpenMP

1.2. Objetivos

Apoyándonos en lo expuesto anteriormente, vamos a plantear los objetivos que pretende cumplir este proyecto.

- Lograr realizar la descomposición de Cholesky de una matriz de n x n donde n será el tamaño máximo que podamos almacenar en memoria secundaria, que será mucho mayor que el tamaño máximo almacenable en memoria principal y hará necesario la utilización de algoritmos out-of-core. Dicho tamaño corresponderá en la máquina de trabajo a 700GB frente a los 128GB de memoria principal. El valor exacto de n dependerá de la estructura de datos que se utilice para almacenar la matriz y la eficiencia de la misma.
- Llevar a cabo dicha descomposición en la máquina propuesta y en el menor tiempo posible. Esto implica aprovechar tanto las capacidades hardware como software de las que se dispone en la máquina de trabajo.
- Aprovechar las capacidades paralelas de los procesadores multinúcleo. Para alcanzar este objetivo se deberán utilizar los 48 núcleos de los que dispone la máquina de forma concurrente para aumentar al máximo el rendimiento del algoritmo y apoyar así la reducción de tiempo exigida en el objetivo anterior.

2. Estado del arte

Antes de abordar el problema se debe conocer los trabajos relacionados con la descomposición de Cholesky out-of-core que se hayan realizado hasta el momento y profundizar también en los avances en algoritmos out-of-core aunque se hayan utilizado para resolver otros problemas.

Para comenzar analizamos un poster dedicado a la factorización de Cholesky en GPGPUs utilizando algoritmos out-of-core [1], en este poster podemos ver como utilizan un enfoque out-of-core para resolver matrices utilizando GPGPUs que disponen de mucha menos memoria que la memoria principal de las máquinas dedicadas a computación. Este trabajo nos da una idea de cómo abordar la descomposición de Cholesky utilizando un enfoque out-of-core, no obstante no nos sirve para comparar resultados ni como ejemplo puesto que nuestro objetivo son matrices mucho más grandes (700GB frente a los 10GB que nombra en el poster) e implica transferencias de datos entre memoria principal y memoria masiva, cuando aquí se realizan entre memoria principal y GPGPU. No obstante, este trabajo nos podría ser útil si decidiéramos utilizar GPGPUs para mejorar el rendimiento de nuestro algoritmo.

A continuación profundizamos en la algorítmica out-of-core a través de un artículo dedicado a la factorización LU out-of-core [2] . Este artículo analiza diversas maneras de atacar el problema de la factorización LU para obtener un algoritmo out-of-core paralelo eficiente. Nos permite conocer los detalles algorítmicos y profundizar en las distintas técnicas para mejorar el rendimiento de las operaciones de entrada/salida. No obstante, este algoritmo está enfocado a la paralelización en máquinas en memoria distribuida mientras en nuestro trabajo utilizaremos un sistema de memoria compartida con lo que no podremos aprovechar los particionamientos del problema y los modelos de entrada/salida paralela que propone el artículo. Además, debido a la fecha de publicación del articulo (1996) las características de las máquinas utilizadas son bastante humildes y con ello los tamaños tratados son demasiado bajos para que podamos utilizarlos como referencia.

El siguiente artículo [3] nos es de gran utilidad ya que nos ofrece gran cantidad de información y resultados de diversas librerías de computación de altas prestaciones. En este artículo, aunque también tratan la LU como en el caso anterior, utilizan tecnologías y máquinas más modernas con arquitectura de memoria compartida como la máquina en la que se desarrollará el trabajo. Para comenzar nos ofrece un análisis de la factorización LU con el objetivo de mejorar su rendimiento paralelizándola en memoria compartida, para a continuación comparar el rendimiento entre las diversas versiones analizadas y la utilización de librerías específicas de computación de altas prestaciones.

A continuación nos ofrece el enfoque out-of-core que será el más interesante para nosotros, en esta sección nos ofrece información acerca de distinto s métodos de almacenamiento y acceso a los datos que se encuentran en memoria masiva. Con respecto a los resultados obtenidos, primero debemos notar que la máquina que se utiliza en dicho artículo tiene 32 GB de RAM y 24 núcleos con lo que es ligeramente menos potente que nuestra máquina de trabajo, los resultados obtenidos y que nos servirán de guía es un tiempo de algo menos de 1 hora para factorizar una matriz de 50000x50000.

Una vez analizados los diversos trabajos relacionados con la problemática de los algoritmos out-of-core, vemos que la mayoría de trabajos que se han enfocado a resolver descomposiciones de matrices de gran tamaño se han centrado en la factorización LU. Estos trabajos nos sirven de guía puesto que la descomposición de Cholesky es un caso particular de la factorización LU, no obstante podemos obtener mejores resultados si nos centramos en la descomposición de Cholesky explotando la estructura simétrica de la matriz y el factor resultante. En ese caso, llevaremos a cabo un trabajo algorítmico más específico y centrado en el problema que los estudiados en la bibliografía, pero gracias al cual podremos optimizar el algoritmo y mejorar el rendimiento.

De los trabajos estudiados también se desprende que los mejores rendimientos se obtienen al utilizar librerías de computación de altas prestaciones como son el BLAS [4] y el LAPACK [5]. Como nuestra máquina de trabajo dispone de la librería matemática Intel mkl [6] que incluye implementaciones de estas librerías y partiendo de los resultados observados en los trabajos relacionados, utilizaremos esta librería para mejorar el rendimiento de nuestro algoritmo delegando los procesos de cálculo a las funciones de la librería, las cuales están optimizadas para procesadores Intel como ventaja añadida.

3. Algoritmos y procedimientos desarrollados

En este apartado vamos a hablar de los algoritmos tenidos en cuenta y evaluados durante el desarrollo del trabajo. Para comenzar daremos una pequeña explicación acerca de la descomposición de Cholesky, para continuar analizando las distintas aproximaciones de la descomposición hasta llegar a nuestro algoritmo out-of-core.

3.1. Descomposición de Cholesky

Dada una matriz A simétrica y definida positiva, existe una matriz única triangular inferior G con entradas positivas en su diagonal, denominada factor de Cholesky, que cumple:

$$A = G \cdot G^T$$

A continuación se muestra un algoritmo [7] que sobreimprime la matriz A con su factor de Cholesky, de forma secuencial, con un coste de $n^3/3 \ flops$:

```
for j=1:n
        If j > 1
                A(j:n,j) = A(j:n,j) - A(j:n,1:j-1) A(j,1:j-1)^t
        end
        A(j:n,j) = A(j:n,j) /√A(j:j,j)
end
```

No obstante, como lo que se persigue es un algoritmo out-of-core nos interesa una versión a bloques, no por el mejor rendimiento que suelen obtener las versiones a bloques sino por tomar el bloque como la unidad de transferencia entre el disco y la memoria.

En las secciones siguientes haremos un pequeño análisis a nivel algorítmico con 2 versiones a bloques de la descomposición de Cholesky para elegir que algoritmo llevaremos a la implementación en el caso out-of-core. En estos análisis nos centraremos en evaluar las lecturas y escrituras para minimizar el coste debido a ellas.

3.2. Versión 1

Esta aproximación procede de la web de netlib.org [8] y en ella se esboza el funcionamiento del algoritmo de Cholesky a bloques que está implementado en el LAPACK. La Figura 3.1 muestra un paso del algoritmo, siendo el amarillo la parte ya calculada de la descomposición. El algoritmo es recursivo y en cada paso se vuelve a dividir en bloques la matriz restante para repetir el proceso.

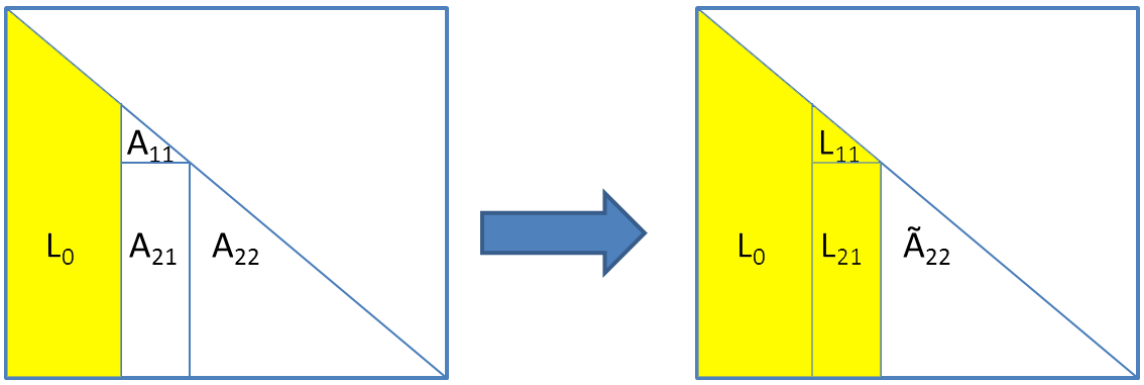

Figura 3.1 Iteración de Cholesky versión 1

A continuación se muestran las operaciones (llamadas al LAPACK) que realiza el algoritmo en cada paso:

- `DPOTF2`: Calcular la descomposición de Cholesky del bloque diagonal A_{11}.
  - $A_{11} \rightarrow L_{11}L_{11}^T$
- `DTRSM`: Calcular el bloque columna L_{21}, utilizando una resolución de sistemas triangulares con múltiples lados derechos.
  - $L_{21} \leftarrow A_{21}(L_{11}^T)^{-1}$
- `DSYRK`: Actualizar el resto de la matriz.

$$\tilde{A}_{22} \leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T$$

Como se puede ver en la figura y las operaciones, en cada iteración necesitará leer A_{11}, A_{21} y A_{22}, a la vez que escribir L_{11}, L_{21} y A_{22}. Por tanto, en cada iteración, vuelve a leer y a escribir la totalidad de la matriz restante.

Se debe tener en cuenta, que en memoria masiva, las escrituras son considerablemente más lentas que las lecturas, por tanto debemos minimizar el número de escrituras. Debido al alto coste de escrituras de esta versión unido a su naturaleza recursiva, que provoca que el tamaño de bloque (y con ello el tamaño ocupado en memoria) varíe en cada iteración, se optó por descartar esta versión en favor de la versión 2 que veremos a continuación.

3.3. Versión 2

Esta versión fue obtenida del algoritmo 4.2.3 *Cholesky block dot product versión* [9]*;* a continuación se muestra el código del algoritmo:

Entrada: $A \in \mathbb{R}^{nxn}$ *estructurada en una matrix a bloques de tamaño* NxN
Salida: $G \in \mathbb{R}^{nxn}$ $sobreescrita\ en\ A$
for j=1:N
    for i=j:N
        $S = A_{ij} - \sum_{k=1}^{j-1} G_{ik} G_{jk}^T$
        If i = j
            Calcular descomposición de Cholesky $S = G_{jj} G_{jj}^T$
        else
            Resolver $G_{ij} G_{jj}^T = S$ *(obtener* G_{ij}*)*
        end
        Sobreescribir A_{ij} *con* G_{ij} ***
    end
end

**Nótese que debido a la sobreescritura todos los bloques G pueden ser sustituidos por A directamente y eliminar la línea de la sobreescritura.*

Añadiendo a este algoritmo las lecturas y escrituras de los bloques en disco obtenemos un primer esbozo de nuestro algoritmo out-of-core. Este algoritmo divide la matriz en N bloques de tamaño tB, siendo todas las matrices que se ven en el algoritmo bloques de tB * tB. Los únicos lugares donde se escribe son ambos casos del *if*, donde escribe un bloque. Mientras el lugar donde se realizan las lecturas es el cálculo de S. Como se desprende del algoritmo, el número de lecturas es muy inferior a la versión 1 puesto que en esta versión solo se escribe una vez cada bloque. Y el número de lecturas es también inferior aunque no sea apreciable a simple vista en el algoritmo.

Entrada: $A \in \mathbb{R}^{nxn}$ *estructurada en una matrix a bloques de tamaño* NxN
Salida: $G \in \mathbb{R}^{nxn}$ *sobreescrita en* A

```
for j=1:N
    for i=j:N
        for k=1:j-1
            leer A_ik
            leer A_jk
            S = S + A_ik A_jk^T
        end
        Leer A_ij
        S = A_ij − S

        If i = j
            Calcular descomposición de Closesky S = A_jj A_jj^T
            Escribir A_jj
        else
            Resolver A_ij A_jj^T = S
            Escribir A_ij
        end
    end
```

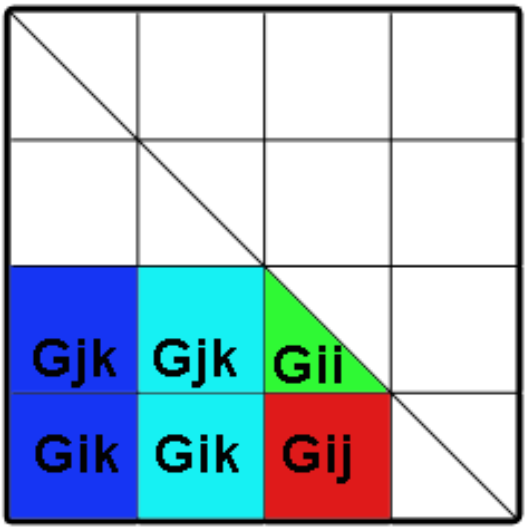

Figura 3.2 Representación gráfica algoritmo out-of-core

En la Figura 3.2 se muestran los bloques que son necesarios para el cálculo de un bloque de la descomposición. Para el cálculo del bloque rojo (G_{ij}) dispondremos en memoria del bloque verde (G_{ii}), necesitaremos leer el bloque (A_{ij}), también los bloques azules (G_{ik} ,G_{jk}) que serán necesarios para el cálculo de S y finalmente se escribirá el bloque rojo. Cada pareja de bloques azules son los operandos de una de las multiplicaciones necesarias para calcular S.

3.4. Análisis teórico

En este apartado vamos a hacer un pequeño análisis del coste teórico de la versión 2 del algoritmo, que como hemos justificado anteriormente, será la que se llevará a la fase de implementación debido a su menor número de escrituras.

Hemos extraído de [9] el coste del algoritmo completo que es $O(\frac{n^3}{3})$ y vamos a formalizar el coste de cada una de las operaciones que forman parte de nuestro algoritmo, y lo que es más importante, cuantas lecturas y escrituras necesitará nuestro algoritmo en función de n.

	DGEMM	DPFTRF	DTRSM	RESTAS
Número de operaciones	$\frac{nB^3 - nB}{6}$	nB	$nB\frac{(nB+1)}{2} - nB$	$nB\frac{(nB+1)}{2}$
Coste operación	$2tB^3$	$\frac{tB^3}{3}$	tB^3	tB^2
Coste	$\frac{n^3 - n\,tB^2}{3}$	$\frac{n\,tB^2}{3}$	$\frac{n^2tB - n\,tB^2}{2}$	n^2

Tabla I Costes de operaciones de la versión 2

En la Tabla I se muestran el número de veces que se realiza cada operación en función de nB junto con el coste de cada operación en función de tB para finalmente mostrar el coste de las operaciones en función de n.

El número de escrituras es sencillo de calcular ya que cada bloque se escribe una única vez, con lo cual, su número corresponderá al número de bloques:

$$Escrituras = nB\frac{(nB+1)}{2} = \frac{nB^2 + nB}{2}$$

$$tEscrituras = nEscrituras \cdot (tEscritura\ bloque\ tB\ x\ tB)$$

El número de lecturas es un poco más complejo de calcular ya que se debe tener en cuenta las lecturas correspondientes a cada bloque más las lecturas necesarias para las multiplicaciones:

$$lecturasBloques = Escrituras = \frac{nB^2 + nB}{2}$$

$$lecturasMultiplicación = \frac{nB^3 - nB}{6} \cdot 2$$

$$tLecturas = lecturasBloques + lecturasMultiplicacion \cdot (t\ bloque\ tB\ x\ tB)$$

Finalmente el tiempo teórico total del algoritmo out-of-core será:

$$tOOC = \frac{n^3}{3} \cdot tflop + tEscrituras + tLecturas$$

4. Análisis experimental

En este apartado nos vamos a centrar en el proceso de implementación del trabajo y en los resultados experimentales de nuestro algoritmo. Para comenzar analizaremos diversas implementaciones y experimentos previos a la implementación del algoritmo out-of-core que se utilizaron para profundizar en el conocimiento del problema y en las posibilidades que nos brindaba la máquina de trabajo. A continuación se mostrarán los detalles de implementación tenidos en cuenta durante el desarrollo del algoritmo out-of-core para optimizar el algoritmo, para finalmente exponer los resultados obtenidos con nuestro algoritmo.

4.1. Implementaciones previas

Inicialmente se decidió implementar varias versiones del algoritmo:

- Implementación A: una versión secuencial del algoritmo que se apoya en las primitivas del BLAS y del LAPACK ofrecidos por la mkl.
- Implementación B: una versión paralelizada con openMP [10] en la que los distintos *dtrsm* de cada columna se realizan en paralelo.
- Implementación C: una llamada directa a la función *dpotrf* del LAPACK; en esta implementación se delega totalmente los resultados al LAPACK.

Las tres versiones se probaron utilizando tanto la mkl secuencial como la mkl paralela para evaluar el impacto de utilizar varios núcleos en el rendimiento de la mkl. En el ANEXO I se encuentra un fichero de código fuente que contiene estas tres versiones. En la Figura 4.1 se muestran los tiempos de ejecución de las tres versiones con la mkl en modo secuencial, mientras que en la Figura 4.2 se muestran los tiempos de las mismas con la mkl en modo paralelo. A continuación se muestran los datos que producen ambas tablas:

	MKL secuencial			**MKL paralela**		
	A	B	C	A	B	C
1024	0,0254925	0,185056	0,0211567	0,0951042	0,0749674	0,00690463
2048	0,16825	0,248346	0,240418	0,077195	0,078384	0,0200273
4096	1,2771	0,537786	1,04983	0,423445	0,312935	0,094392
8192	10,299	1,99194	8,37566	2,74025	1,19022	0,793514
16384	88,5779	9,55975	73,1797	22,2464	8,50319	6,54892

Tabla II Tiempos de ejecución de versiones de cholesky

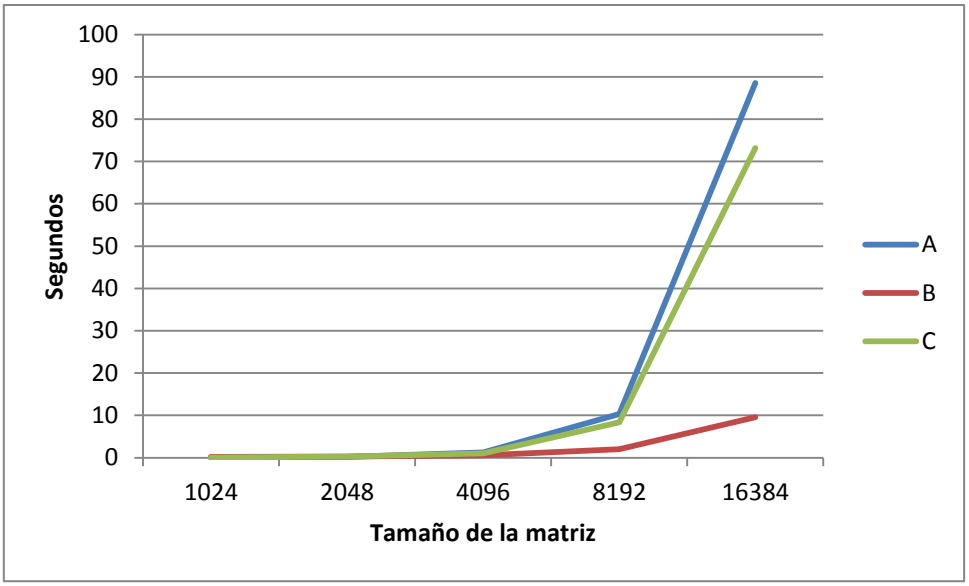

Figura 4.1 Versiones de cholesky con mkl secuencial

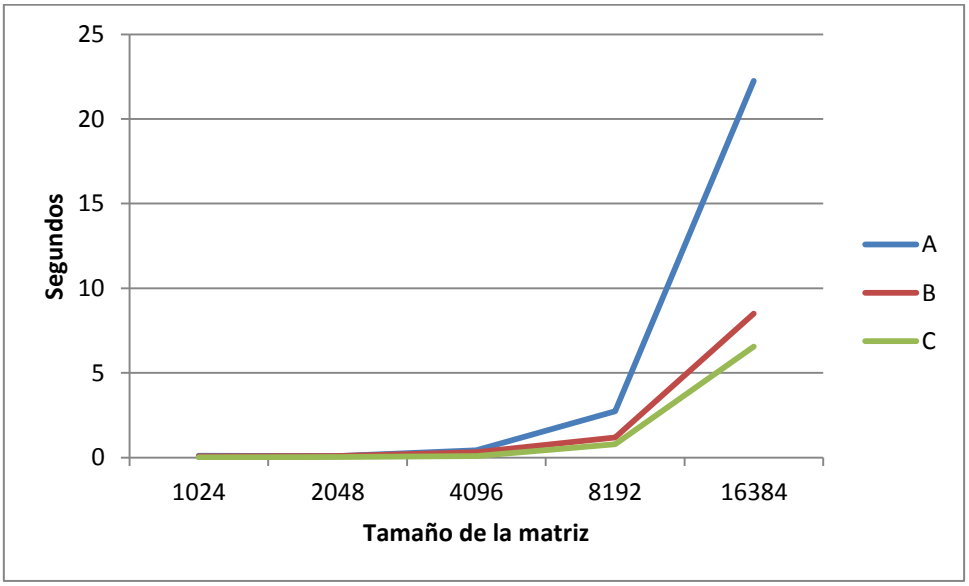

Figura 4.2 Versiones de cholesky con mkl paralela

Los resultados son los esperados: en el primer caso la versión paralela es la más rápida de todas pues aprovecha mejor la potencia multinúcleo ofrecida por la máquina de trabajo. Mientras en el segundo la versión que solo llama a la librería mkl es la más rápida pues la librería esta optimizada a bajo nivel para procesadores Intel como el que equipa nuestra máquina. No obstante, en el segundo caso todas las versiones mejoran frente a las versiones con mkl secuencial debido a que la mkl paralela aprovecha también el potencial multinúcleo. Estos resultados nos llevan a delegar en la mkl cuando vayamos a implementar el algoritmo out-of-core en lugar de intentar paralelizar el algoritmo de forma independiente puesto que los resultados han hablado a su favor.

4.2. Detalles de implementación del algoritmo out-of-core

Una vez decidido el algoritmo a utilizar y probadas las capacidades de la máquina se procedió a la implementación del algoritmo. Para ello primero se estableció el modo de almacenamiento que sería utilizado para almacenar los problemas y sus posteriores soluciones en disco, después se codificaron unas aplicaciones auxiliares: la primera capaz de generar un problema válido en el sistema de almacenamiento adecuado con un algoritmo out-of-core (para poder generar problemas suficientemente grandes) y la segunda una capaz de leer un resultado en el formato adecuado y comprobar si es correcto (aplicación necesaria durante el desarrollo del algoritmo de Cholesky out-of-core). Finalmente se implementó el algoritmo teniendo en cuenta ciertas consideraciones para reducir al mínimo el tiempo de ejecución manteniendo un coste de memoria constante.

4.2.1. Sistema de almacenamiento

Para ocupar el mínimo espacio posible en disco se analizó tanto el problema como la solución, con lo que se decidió eliminar toda información sobrante y guardar la matriz en formato binario.

Para comenzar la matriz tendrá un tamaño n y se dividirá en nB bloques de tamaño tB, al ser la matriz problema una matriz simétrica y definida positiva, solo se guardarán en disco los bloques correspondientes a la parte triangular inferior de la matriz. Los bloques se almacenaran contiguos siguiendo un orden por columnas (de arriba abajo y de derecha a izquierda), a su vez cada bloque será un volcado a disco de la representación en memoria que será un array unidimensional tamaño tB*tB. En la Figura 4.5 se muestra un esquema de como quedarían guardados los elementos de la matriz en el fichero.

Por otro lado, los bloques correspondientes a la diagonal son a su vez simétricos. Para minimizar el espacio ocupado por estos bloques se utilizará el formato *Rectangular Full Packed Storage* de la mkl [11]. Este formato permite almacenar las matrices triangulares, simétricas o hermitianas de forma compacta ocupando el mínimo espacio posible, mientras al estar almacenada en un formato completo mantiene la eficiencia de las operaciones que se realizan sobre la matriz al utilizar kernerls de BLAS3. La figura Figura 4.3 muestra un ejemplo de cómo quedaría almacenada una matriz triangular inferior con N impar en formato RPF comparado con el formato completo.

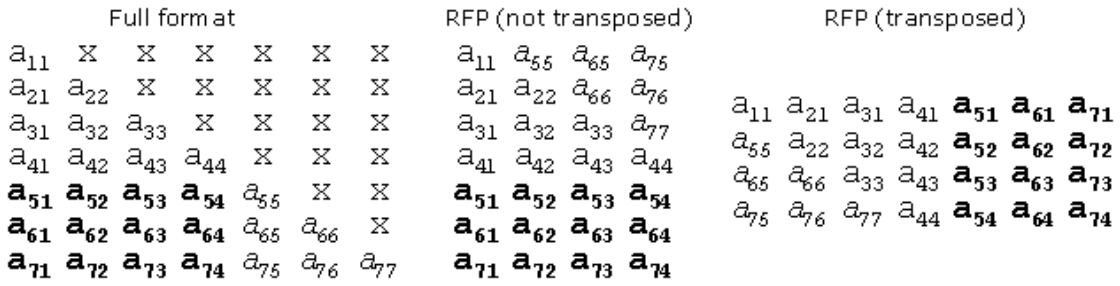

Figura 4.3 Formato RFP de la mkl para matrices triangulares

Debido a que el fichero solo contendrá datos compactos en formato binario sin ningún modificador deberemos guardar en las 2 primeras posiciones del fichero el número de elementos n y el tamaño del bloque tB. Para acceder a cada bloque de la parte triangular estrictamente inferior en la memoria para leerlo o escribirlo se deberá calcular su posición en el fichero utilizando la siguiente fórmula:

$$pos = \left(i + j * nB - \left(j * \frac{j+1}{2}\right)\right) * bytesBloque - (j+1) * bytesUpper +$$

$$2 * (tamaño\ de\ un\ numero\ en\ doble\ precision)$$

Dónde:

$$bytesBloque = tB * tB * (tamaño\ de\ un\ numero\ en\ doble\ precision)$$

$$bytesUpper = tB * \frac{tB-1}{2} * (tamaño\ de\ un\ numero\ en\ doble\ precision)$$

$$i = fila\ del\ boque \quad j = columna\ del\ bloque$$

El primer fragmento entre paréntesis nos permite calcular el número de bloque que nos corresponde dentro del fichero; para ello se calcula el número de bloque que correspondería en caso de estar almacenados todos los bloques y se le restan los bloques que no se han almacenado en función de la columna.

La segunda parte de la ecuación (tras el signo menos) se encarga de restar el espacio de diferencia entre un formato compacto y el formato RFP en función de la columna puesto que solo hay un bloque diagonal por columna.

Finalmente la parte tras el signo más corresponde a lo que ocupan los 2 primeros números del fichero que contienen n y tB.

Para los bloques de la diagonal la fórmula es ligeramente diferente, pues no hay que restar el bloque diagonal de su columna, quedando la formula como sigue:

$$pos = \left(i + j * nB - \left(j * \frac{j+1}{2}\right)\right) * bytesBloque - (j+1) * bytesUpper +$$

$$2 * (tamaño\ de\ un\ numero\ en\ doble\ precision)$$

A continuación se muestra un ejemplo de cómo quedaría una matriz de tamaño 6x6 y tamaño de bloque 3 almacenada en fichero siguiendo el esquema. En la Figura 4.4 se muestra la matriz con sus 36 elementos, mientras que en la Figura 4.5 se muestra como quedarían almacenados dichos elementos en el fichero.

a0	a6	a12	a18	a24	a30
a1	a7	a13	a19	a25	a31
a2	a8	a14	a20	a26	a32
a3	a9	a15	a21	a27	a33
a4	a10	a16	a22	a28	a34
a5	a11	a17	a23	a29	a35

Figura 4.4 Matriz n=6 tB=3

n	tB	a0	a1	a2	a14	a7	a8	a3	a4	a5	a9	a10	a11	a15	a16	a17	a21	a22	a23	a35	a28	a29	eof

Figura 4.5 Matriz n=6 tB=3 almacenada en fichero

4.2.2. Aplicaciones auxiliares

- **Generador de problema out-of-core**

Para generar el problema, es decir, una matriz simétrica y definida positiva, se generaba una matriz aleatoria triangular inferior con elementos entre 0 y 100 en la triangular inferior estricta y 1000 en la diagonal, para después multiplicarla por su traspuesta y así obtener la matriz simétrica y definida positiva. Una vez generada, se guardaba la matriz por bloques en el formato expuesto anteriormente.

Al aumentar el tamaño más allá de las capacidades de la máquina fue necesario el desarrollo de una versión out-of-core para la creación de esta matriz. En el ANEXO II se encuentra el código del generador out-of-core, que consiste en un desplegado de la multiplicación matricial con un esquema muy similar al utilizado en nuestro algoritmo de Cholesky out-of-core. En esta ocasión se generaran las matrices aleatorias por bloques con dos casos distintos: para los bloques de la diagonal principal se utilizará el método que se utilizaba para generar la matriz en el generador completo; mientras para los bloques de la triangular inferior se generará una única matriz cuadrada con números aleatorios entre 0 y 100 y se utilizará esta matriz como operando para todas las operaciones que necesiten un bloque de la triangular inferior. Con este atajo, mantenemos que la matriz resultante sea simétrica y definida positiva pero evitamos el coste de tener que generar todos los bloques aleatorios correspondientes a la triangular inferior. Debe tenerse en cuenta que en las aplicaciones que requieren el uso de algoritmos out-of.core, la matriz simétrica, definida positiva, a procesar suele ser un dato de partida, ya almacenada en un fichero, por lo que la técnica utilizada no resta generalidad al algoritmo implementado.

- **Comprobador de corrección del resultado**

  En el desarrollo de cualquier algoritmo siempre es necesario comprobar el resultado ofrecido por el algoritmo para comprobar si la implementación es correcta. Para ello se realizó un código que carga la matriz problema(A) desde un fichero y el factor de cholesky(L) desde otro (ambos en el formato de almacenamiento propuesto en 4.2.1) el cual se encuentra en el ANEXO III.

  Una vez cargadas ambas matrices se multiplica el factor de Cholesky por su traspuesta para calcular una matriz B que debería ser igual a la matriz A original; esto se comprueba restando ambas matrices, tratando el resultado de la resta como un vector y calculando la 2-norma del vector cuyo resultado debe ser 0:

$$B = L * L^T$$
$$B = B - A$$
$$\|B\| == 0\ ?$$

  Esta operación es matemáticamente equivalente a calcular la norma de Frobenius de la diferencia entre ambas matrices:

$$\|A - LL^T\|_F$$

4.2.3. Implementación del algoritmo

Teniendo en cuenta las implementaciones previas, y el objetivo out-of-core del problema se decidió implementar el algoritmo utilizando la mkl paralela y delegando la paralelización sobre la mkl ya que el límite vendría dado por la cantidad de datos que podemos almacenar en memoria en un momento determinado. La implementación de dicho algoritmo que vamos a comentar a continuación se encuentra en el ANEXO IV.

El algoritmo presentado en la sección 3.3 tiene un coste de memoria de 4 veces el tamaño de un bloque como máximo. A continuación se encuentra un esbozo del algoritmo con las primitivas de la mkl y los puntos donde se libera memoria:

Entrada: $A \in \mathbb{R}^{nxn}$ *estructurada en una matrix a bloques de tamaño* NxN
Salida: $G \in \mathbb{R}^{nxn}$ *sobreescrita en* A

```
for j=1:nB
    for i=j:nB
        If i = j
            S = 0
            for k=1:j-1
                leer A_ik
                dgemm (S = S + A_ik A_ik^T)
                liberar A_ik
            end

            leer A_ii
            daxpy (S = A_ii − S)

            dpftrf (S = A_ii A_ii^T)
            escribir A_ii
            liberar S

        else
            S = 0
            for k=1:j-1
                leer A_ik
                leer A_jk
                dgemm( S = S + A_ik A_jk^T)
                liberar A_ik
                liberar A_jk
            end

            leer A_ij
            daxpy (S = A_ij − S)

            dtrsm (A_ij A_jj^T = S)
            escribir A_ij
            liberar A_ij
        end
    end
    liberar A_ii
end
```

En este algoritmo es fácilmente observable el límite de memoria de los 4 bloques, que se da en el cálculo de los bloques estrictamente triangulares inferiores (la sección del código dentro del *else* donde i ≠ j), 3 bloques corresponden a los operandos del dgemm (S , A_{ik} y A_{jk}) y el cuarto bloque es el que almacena el resultado de la descomposición de Cholesky del bloque diagonal de la columna actual(A_{ii}). Este último bloque se mantiene en memoria durante todas las iteraciones correspondientes a su columna porque es necesario para los *dtrsm*, y por tanto deberíamos leerla igualmente todas las veces aumentando el número de lecturas (una de las cosas que tratamos de disminuir).

También se aprecia que el cálculo de S se ha diferenciado entre los casos de la diagonal y los bloques inferiores. Esto es debido a que en la diagonal, A_{ik} y A_{jk} son el mismo bloque, y por tanto podemos ahorrarnos la mitad de las lecturas diferenciando este caso.

Otro detalle de implementación que no se ha representado en este algoritmo pero es fácilmente deducible es que en la primera iteración del bucle (primera columna de bloques) no se calcula S, por tanto en el código implementado se ha desplegado esta primera iteración fuera del bucle para eliminar todas las operaciones relacionadas con el cálculo de S.

Para finalizar, siguiendo las recomendaciones de la guía de la mkl, se utilizó el mkl_malloc con alineamiento de 64 bits en lugar del malloc estándar de C para mejorar el rendimiento. Resultados experimentales realizados demostraron que las operaciones de la mkl eran más rápidas cuando se alineaba la memoria tal y como recomendaban.

4.3. Resultados experimentales

Antes de presentar los resultados de la implementación del algoritmo out-of-core cabe destacar la configuración de ejecución utilizada. Se decidió dejar la variable de entorno MKL_DYNAMIC a falso para así controlar nosotros directamente el número de hilos de ejecución. Si se deja en el valor por defecto, que es verdadero, esto permite que la mkl controle el número de hilos que se lanzan en cada momento. A su vez utilizamos la mkl con 48 cores.

La primera prueba consiste en comparar nuestro algoritmo out-of-core para tamaños que quepan en la memoria con una programa que lee la matriz de fichero y ejecuta el *dpotrf* de la mkl sobre la matriz problema completa. En la Figura 4.6 se muestra la comparación de ambas siguiendo el aumento exponencial del tamaño como en las implementaciones previas.

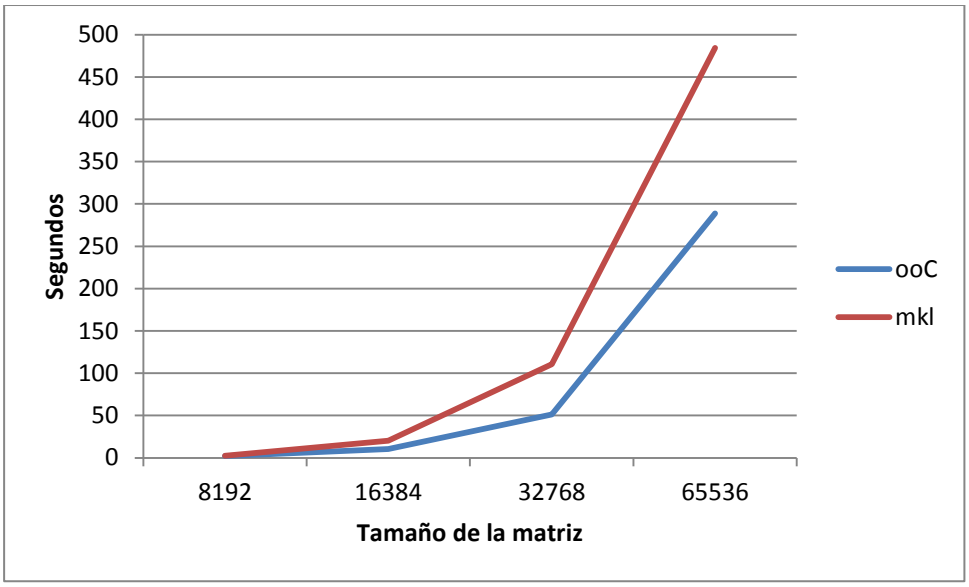

Figura 4.6 Out of core vs Mkl

En todas estas ejecuciones se ha utilizado el mismo tamaño de matriz que tamaño de bloque, pues todavía caben completas en memoria. El motivo de que la versión out-of-core sea más rápida que la versión mkl es que opera más eficientemente con la matriz al realizar las operaciones necesarias con los bloques que lee de fichero, mientras que la versión mkl primero lee la matriz del fichero y reconstruye una matriz completa densa para después realizar la descomposición de Cholesky de dicha matriz. Además la versión out-of-core escribe la matriz durante la realización de la descomposición y la versión de la mkl no escribe la matriz, con lo que en realidad todavía sería mayor su tiempo si le sumáramos el de las escrituras.

Siguiendo la escala exponencial el siguiente tamaño a tener en cuenta es 131072, este tamaño queda ya fuera de la anterior comparativa puesto que no cabe en la memoria, debido a que la versión de la mkl utiliza un almacenamiento completo de la matriz. Por tanto en la Figura 4.7 utilizamos ya los primeros tamaños que son realmente out-of-core partiendo del tamaño más grande analizado en la Figura 4.6 y utilizando bloques de 65536. En la figura llegaremos a alcanzar el máximo tamaño ejecutable en esta escala que es 262144, ya que el siguiente tamaño excederá ya los 700GB de los que disponemos.

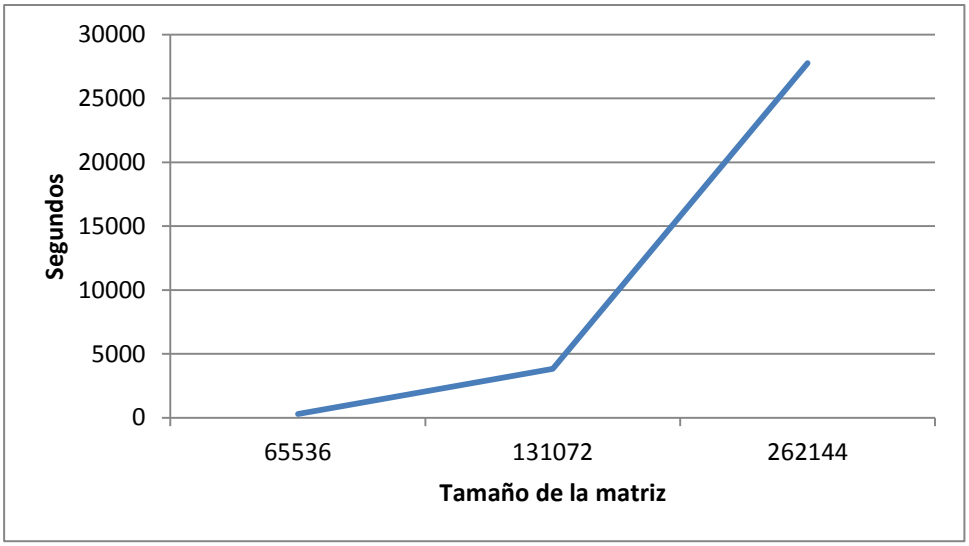

Figura 4.7 Tiempos out-of-core (escala exponencial)

Como se ha comentado anteriormente, el máximo tamaño analizable experimentalmente con la máquina utilizada corresponde a una matriz de 262144x262144, Podemos intentar predecir los tiempos que obtendríamos suponiendo que se dispusiera de una memoria en disco capaz de almacenar matrices más grandes y siguiendo dicha exponencial. Los siguientes tamaños representados en la Figura 4.8 como líneas discontinuas se han calculado extrapolando la relación entre la matriz de 131072 y la de 262144 que es de 7,24.

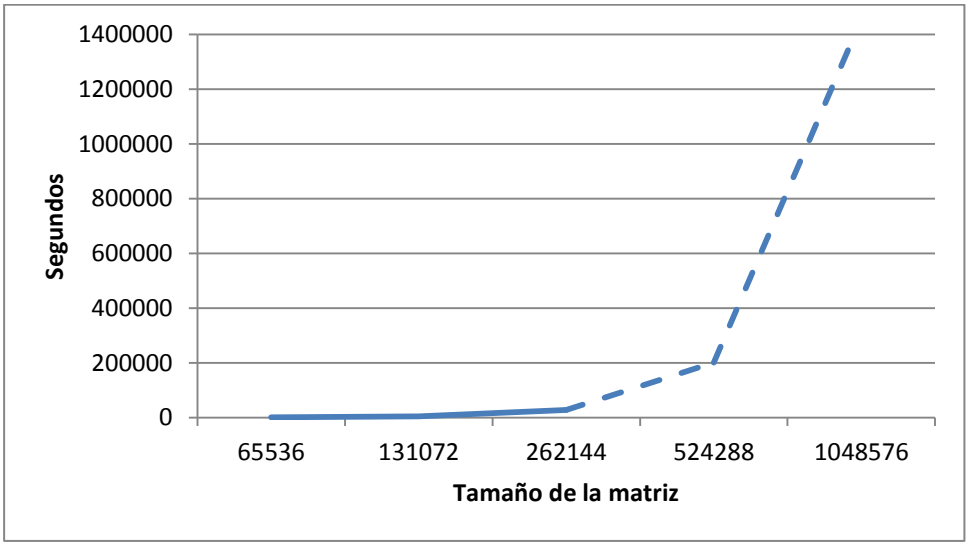

Figura 4.8 Extrapolación de tiempos out-of-core (escala exponencial)

Dicha relación es incluso mejor que la relación teórica que es 8, la cual se obtiene a partir del coste teórico de Cholesky que es de $O(\frac{n^3}{3})$, por tanto al aumentar n en un factor 2n, el coste en flops y con ello el tiempo, aumentan en 2^3. En la Figura 4.9 se muestran los flops teóricos de una descomposición de Cholesky, como se puede observar, la función representada es similar a la representada en la Figura 4.8.

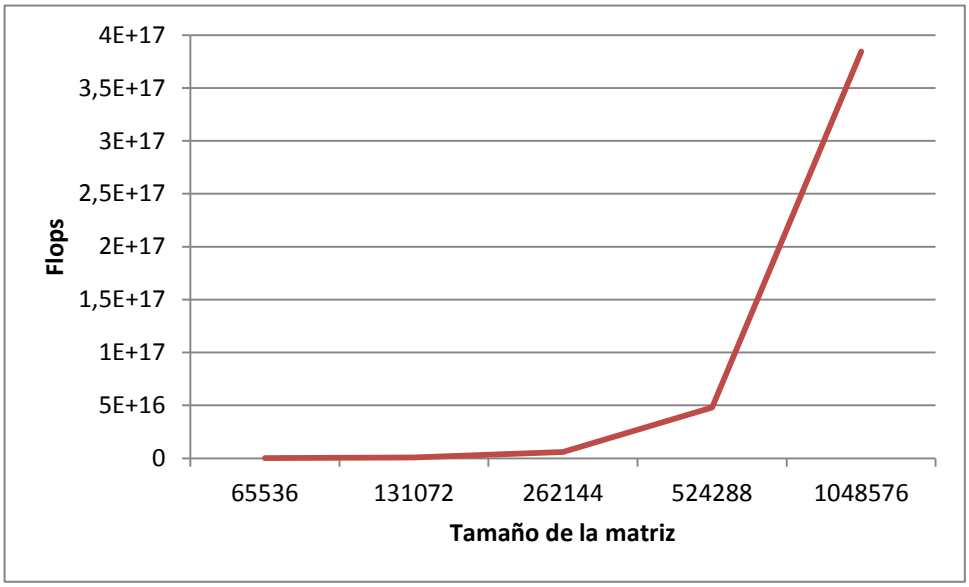

Figura 4.9 Flops Cholesky

Como hemos razonado anteriormente, con la escala actual aún nos sobra espacio en disco y el siguiente tamaño dentro de la escala ya no cabe dentro del disco. Por tanto, se harán algunas pruebas de rendimiento en una escala lineal para llegar a realizar la descomposición con el tamaño máximo. En la Figura 4.10 se muestran los tiempos del algoritmo desde 131072 hasta 393216 en bloques de 65536 y aumentando el tamaño en unidades de un bloque.

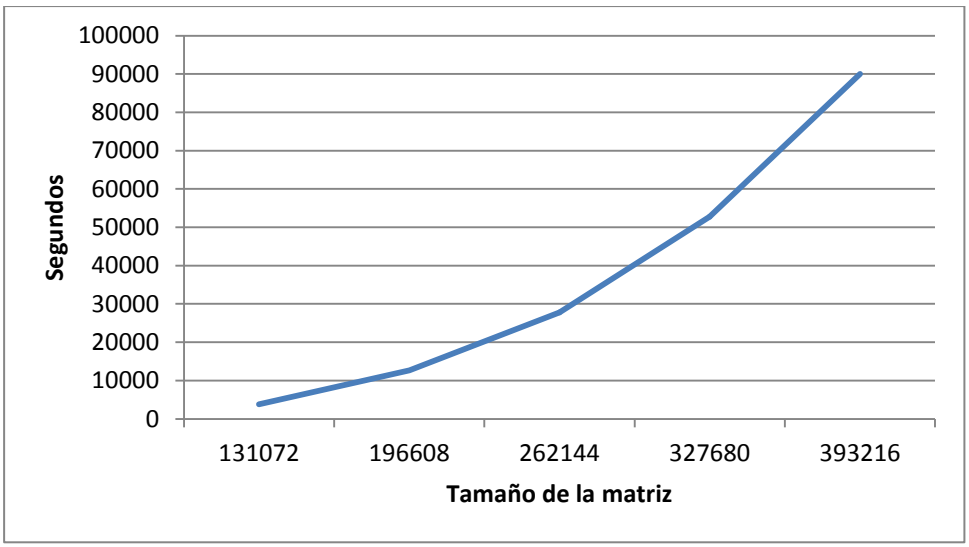

Figura 4.10 Tiempos out-of-core (escala lineal)

Para llevar a cabo un estudio más profundo del algoritmo se ha desglosado el tiempo total de la ejecución en las distintas operaciones que realiza el algoritmo. En la Tabla III se muestra el desglose de las operaciones y se aprecia como la multiplicación matricial es el proceso que más tiempo consume, seguida de las lecturas.

	Lecturas	Escrituras	DGEMM	DPFTRF	DTRSM	Cambio formato	RESTAS
131072	715,791	113,556	1361,94	460,659	696,178	480,037	0,0356488
196608	3261,030	340,884	4980,90	697,763	2559,260	840,040	0,1139640
262144	7696,930	597,422	13314,40	936,789	4260,010	909,191	0,1731920
393216	28343,80	1574,210	46063,8	1410,180	11068,50	1396,86	0,4129800

Tabla III Desglose de operaciones del out-of-core-cholesky

En las Tabla IV y Tabla V se muestran los tiempos de ejecución experimentales así como la memoria que sería necesaria para ejecutar el problema directamente sin recurrir a técnicas out-of-core, que a su vez coincide con el espacio ocupado en disco.

	131072	262144	524288	1048576
Tiempo	63,88 min	7,71 horas	2,32 días*	16,84 días*
Tamaño	64 GB	256 GB	1 TB	4 TB

Tabla IV Tiempo y tamaño de los problemas out-of-core (escala exponencial)

** Estas medidas son extrapolaciones teóricas*

	131072	196608	262144	327680	393216
Tiempo	63,88 min	3,52 horas	7,71 horas	14,6 horas	25 horas
Tamaño	64 GB	144 GB	256 GB	400 GB	576 GB

Tabla V Tiempo y tamaño de los problemas out-of-core (escala lineal)

5. Conclusiones y trabajo futuro

5.1. Conclusiones

La principal conclusión es que se ha alcanzado el objetivo del proyecto, el cual consistía en la resolución de una descomposición de Cholesky de una matriz simétrica y positiva definida tan grande como fuera posible. Experimentalmente alcanzamos el tamaño de 393216 x 393215 que como se ha expuesto anteriormente ocupa 576 GB, no obstante hemos extrapolado teóricamente mucho más allá del límite de espacio en disco con unos tiempos razonables. Como el algoritmo ha sido diseñado con un coste de memoria constante, simplemente necesitaremos más tiempo cuanto más grandes sean los problemas a tratar, pero **siempre podremos resolverlos**. Además este algoritmo también responde bien a la escalabilidad vertical, si se consiguiera una máquina con más memoria, solo habría que aumentar el tamaño del bloque para así aumentar el rendimiento del algoritmo.

Además se ha **aprovechado el hardware** debido a que se ha delegado la mayoría del cómputo a la librería mkl, la cual esta optimizada a bajo nivel para trabajar con procesadores Intel como el que posee la máquina de trabajo.

Por otro lado, aunque no se ha realizado una paralelización manual y se ha delegado en la mkl, sí que se ha aprovechado el potencial multinúcleo de la máquina ya que la mkl en su versión paralela y con la configuración adecuada **utiliza todos los núcleos de la máquina**.

5.2. Trabajo futuro

Lo primero a tener en cuenta para trabajos futuros sobre este algoritmo es intentar mejorar el rendimiento del algoritmo propiamente dicho, para ello la única opción es enmascarar las operaciones de entrada/salida con la computación con lo que se conseguiría aparentemente eliminar los costes de lectura/escritura. Aunque el coste seguiría siendo el mismo y las operaciones seguirían realizándose habríamos eliminado en un caso perfecto todo el tiempo de las lecturas y escrituras del tiempo total de ejecución. Para ello necesitaríamos dejar un hilo dedicado a la entrada y salida mientras los 47 restantes se encargan de la computación, el enmascaramiento distingue dos casos bien diferenciados:

- **Enmascaramiento de escrituras**: Si se observa con detenimiento el algoritmo se aprecia que cuanto un bloque es escrito no vuelve a ser utilizado hasta pasadas varias operaciones excepto el bloque triangular, el cual como se conserva en memoria tampoco nos causa ningún problema. Por tanto deberíamos utilizar un hilo dedicado para las escrituras y a la vez que se escribe un resultado que se continúe con la ejecución. El único caso que daría problemas y habría que contemplar de manera especial seria la escritura del último bloque de la penúltima columna que será accedido inmediatamente después de terminar la ejecución de sus operaciones.

- **Enmascaramiento de lecturas**: El enmascaramiento de lecturas es un poco más complejo, este puede llevarse a cabo en el bucle del cálculo de S pero a cambio de dicho enmascaramiento deberemos sacrificar memoria. En cada iteración de dicho bucle se leen 2 bloques y se comienza la multiplicación, por tanto podemos leer los siguientes dos bloques mientras se realiza la multiplicación. Para enmascarar las lecturas necesitaremos 2 bloques de memoria extra subiendo el máximo de memoria necesaria de 4 bloques a 6 bloques, además este método solo comenzará a ofrecer beneficios cuando el número de bloques (nB) sea superior a 4.

El siguiente paso futuro sería adaptar el algoritmo para trabajar con aceleradores de cálculo y GPUs, puesto que actualmente son unas arquitecturas en alza para la computación de altas prestaciones. Se debería estudiar bien las arquitecturas para aprovechar su rendimiento al máximo, pero teniendo en cuenta que se trata de un enfoque out-of-core la memoria disponible en dichos dispositivos será un gran limitante.

Finalmente la parte más importante que debería desarrollarse en un futuro es un algoritmo apto para trabajar en memoria distribuida, es decir, en clústeres de

computadores, ya que esto nos permitiría escalar el problema horizontalmente, lo que suele conllevar un coste menor que un escalado vertical de la máquina de trabajo. Además sería interesante desarrollar esta versión de memoria distribuida apta para clústeres heterogéneos con lo que no solo permitiría aprovechar equipos de diferentes características sino también aprovechar nuevas tecnologías como son los aceleradores de cálculo o las gpgpus comentados en el apartado anterior.

En la mayoría de ocasiones las descomposiciones matriciales no son el fin en sí mismas, sino que son un paso en el proceso para resolver sistemas de ecuaciones. Sería muy interesante desarrollar métodos de resolución de sistemas con el mismo enfoque y sistema de almacenamiento que el algoritmo desarrollado en el trabajo, para así poder llevar a cabo toda la resolución del sistema lo más rápido posible y aprovechando al máximo las capacidades del hardware.

6. Referencias

[1] E. e. a. D'Azvedo, « Out-of-core algorithms for dense matrix multiplication on GPGPU».

[2] J. J. DONGARRA, S. HAMMARLING y D. W. WALKER, «Key concepts for parallel out-of-core LU factorization.,» *Parallel Computing*, vol. 23, nº 1, pp. 49-70, 1997.

[3] S. Rivas-Gomez, D. Giménez y J. Cuenca, «A wide comparison between different linear algebra libraries using the well-known LU factorization problem».

[4] «BLAS netlib,» [En línea]. Available: http://www.netlib.org/blas/. [Último acceso: Septiembre 2014].

[5] «LAPACK (Linear Algebra PACKage) netlib,» [En línea]. Available: http://www.netlib.org/lapack/. [Último acceso: septiembre 2014].

[6] «Documentacion MKL,» [En línea]. Available: https://software.intel.com/en-us/articles/intel-math-kernel-library-documentation. [Último acceso: Septiembre 2014].

[7] G. H. Golub y C. F. Van Loan, «Algorithm 4.2.1 Cholesky gaxpy version,» de *Matrix Computation (3rd edition)*, JHU Press, 2012, pp. 145-146.

[8] S. Ostrouchov, «Netlib.org,» 28 Abril 1995. [En línea]. Available: http://www.netlib.org/utk/papers/factor/node9.html. [Último acceso: Septiembre 2014].

[9] G. H. Golub y C. F. Van Loan, «Algorithm 4.2.3 Cholesky block dot product version,» de *Matrix Computation (3rd edition)*, JHU Press, 2012, pp. 145-146.

[10] OpenMP Architecture Review Board, «OpenMP Application Interface V4,» Julio 2013. [En línea]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf. [Último acceso: Septiembre 2014].

[11] «Matrix Arguments MKL,» [En línea]. Available: https://software.intel.com/sites/products/documentation/hpc/mkl/mklman/GUID-7B11079E-CB74-4A5F-AEEA-D6C9B7181C42.htm#GUID-7B11079E-CB74-4A5F-AEEA-D6C9B7181C42. [Último acceso: septiembre 2014].

[12] «MKL User's Guide,» [En línea]. Available: https://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mkl_userguide_lnx/. [Último acceso: Septiembre 2014].

[13] «MKL Reference Manual,» [En línea]. Available: https://software.intel.com/sites/products/documentation/hpc/mkl/mklman/. [Último acceso: Septiembre 2014].

7. Anexos

ANEXO I

En este anexo se muestra el código de las implementaciones previas al desarrollo del algoritmo out-of-core. El código fuente completo se encuentra en el archivo adjunto *choleskyPar.c:*

```
int main(int argc, const char* argv[])
{
	int i, j;
	int TB;
	unsigned long n;
	double t;

	if(argc < 3)
	{
		printf("Uso: cholesky tam tam_bloque");
		exit(1);
	}
	else
	{
		n = atoi(argv[1]);
		TB = atoi(argv[2]);
	}

	/******** reserva de memoria para las matrices y creación de A*******/
	double * A = (double *) malloc( n*n*sizeof(double) );
	if(A == NULL){
		printf("Error en la reserva de memoria de A\n");
		printf("Tamaño: %d", n*n*sizeof(double));
		//exit(0);
	}
	double * Ac = (double *) malloc( n*n*sizeof(double) );
	if(Ac == NULL){
		printf("Error en la reserva de memoria de Ac\n");
		//exit(0);
	}
	double * Gb =  (double*)calloc(n * n,sizeof(double));
	double * Gbp =  (double*)calloc(n * n,sizeof(double));
	double * Glap =  (double*)calloc(n * n,sizeof(double));
	srand(time(0));

	printf("Creating matrix...\n");
	createCholMatrix(A,n);
	//printMatrix(A,n);
	cblas_dcopy(n*n,A,1,Ac,1);
	/******** computo cholesky secuencial **********/
  /*printf("Procesing...\n");
```

```
   t = blockCholBlas(A,n,TB);
        printf("Cholesky blas block time: %g \n",t);
            extraeTriInf(A,Gb,n,n,n);
            cblas_dcopy(n*n,Ac,1,A,1);
        t = blockCholBlasPar(A,n,TB);
        printf("Cholesky blas block par time: %g \n",t);
            extraeTriInf(A,Gbp,n,n,n);
            cblas_dcopy(n*n,Ac,1,A,1);*/
        t = lapackChol(A,n);
        printf("Cholesky lapack time: %g \n",t);
            //extraeTriInf(A,Glap,n,n,n);
            //cblas_dcopy(n*n,Ac,1,A,1);
        //t = lapackCholUnb(A,n);
     // printf("Cholesky lapack unblocked time: %g \n",t);

        //printf("Norma A-LL' blockBlas  = %.3e\n",checkNorm(Ac,Gb,n));
        //printf("Norma A-LL' blockBlasPar  = %.3e\n",checkNorm(Ac,Gbp,n));
        //printf("Norma A-LL' lapack  = %.3e\n",checkNorm(Ac,A,n));
        //printf("Norma A-LL' lapack unblocked  = %.3e\n",checkNorm(Ac,A,n));

        /*if (argc==4)
        {
            printMatrix(Gb,n);
            printf("\n----------------------------\n");
            printMatrix(A,n);
        }*/

        return 0;
}


//Auxiliar function to create a positive definite matrix
void createCholMatrix(double* A, long n)
{
        int i,j;
        double* G =  (double*)calloc(n * n,sizeof(double));


        for(i = 0; i < n; i++)
            for(j = 0; j < n; j++)
                G[j+i*n] = rand() % 100 + 1;

        cblas_dgemm(CblasColMajor,CblasTrans,CblasNoTrans,n,n,n,1,G,n,G,n,0,A,n);

        free(G);
}
// Extrae la triangular inferior de G, calcula GG', resta A-GG' y devuelve la 2norma de la matriz
resultante.
double checkNorm(double* A, double* G, long n)
{
```

```
	double* L =  (double*)calloc(n * n,sizeof(double));
	double* M =  (double*)calloc(n * n,sizeof(double));
	int i,j, norm;

	extraeTriInf(G,L,n,n,n);

	cblas_dgemm (CblasColMajor, CblasNoTrans, CblasTrans, n, n, n, 1, L, n, L, n, 0, M, n);

	for(i=0;i<n;i++)
		for(j=0;j<n;j++)
			M(i,j)=M(i,j)-A(i,j);

	norm = cblas_dnrm2(n*n,M,1);
	//printMatrix(M,n);

	free(L);free(M);

	return norm;

}

void printMatrix(double* M, long n)
{
   int i,j;

	for(i = 0; i < n; i++)
	{
		for(j = 0; j < n; j++)
			printf("%g ",M(i,j));
		printf("\n");
	}
}

void restaMatriz(double* M1,double* M2, double* R,long n, long nMem1, long nMem2)
{
	int i,j;
	for(i = 0; i < n; i++)
		for(j = 0; j < n; j++)
			R[i+j*n] = M1[i+j*nMem1] - M2[i+j*nMem2];

}



void extraeTriInf(double* M, double* L,long n,int ldm, int ldl)
{
	int i,j;

	for(i=0;i<n;i++)
	   {
```

```
                for(j=i+1;j<n;j++)
                        L[i+j*ldl]=0.0;
                for(j=0;j<i+1;j++)
                        L[i+j*ldl]=M[i+j*ldm];
        }


}
/*_____________________________________________________________________*/

double blockCholBlas(double* A,MKL_INT n,int TB)
{
   int i, j, k,x,y;
   int nB, info;
   double ini, fin;
   double * S,* SUM,* Aij;
        const char * L = "L";

     nB = n/TB;

     ini = dsecnd();

        for(j = 0; j < nB; j++)
            for(i = j; i < nB; i++)
            {
                SUM = (double*)calloc(TB*TB,sizeof(double));
                S = (double*)calloc(TB*TB,sizeof(double));
                for(k = 0; k < j;k++)
                        cblas_dgemm(CblasColMajor,CblasNoTrans,CblasTrans,TB,TB,TB,1,&A[i*TB+k
                *TB*n],n,&A[j*TB +k*TB*n],n,1,SUM,TB);

                restaMatriz(&A[i*TB+j*TB*n],SUM,S,TB, n,TB);

                if( i == j){
                        dpotf2(L,&TB,S,&TB,&info);
                        if (info != 0)
                                printf("Error en lapack: %d\n",info);

                        extraeTriInf(S,&A[j*TB+j*TB*n],TB,TB,n);
                }
                else
                {
                        cblas_dtrsm(CblasColMajor,CblasRight,CblasLower,CblasTrans,CblasNonUnit,T
                B,TB,1,&A[j*TB+j*TB*n],n,S,TB);

                        //copia S en Gij
                        Aij = &A[i*TB+j*TB*n];
                        for(x = 0; x < TB; x++)
                                for(y = 0; y < TB; y++)
                                        Aij[x+y*n] = S[x+y*TB];
```

```
                }
                free(S);
                free(SUM);
            }
    fin = dsecnd();

  return fin-ini;
}

double blockCholBlasPar(double* A,MKL_INT n,int TB)
{
  int i, j, k,x,y;
  int nB, info;
  double ini, fin;
  double * S,* SUM,* Aij;
        const char * L = "L";

    nB = n/TB;

    ini = dsecnd();


        for(j = 0; j < nB; j++)
        {
                SUM = (double*)calloc(TB*TB,sizeof(double));
                S = (double*)calloc(TB*TB,sizeof(double));
                for(k = 0; k < j;k++)
                        cblas_dgemm(CblasColMajor,CblasNoTrans,CblasTrans, TB,TB,TB, 1,
&A[j*TB+k*TB*n],n, &A[j*TB +k*TB*n],n,1,SUM,TB);

                restaMatriz(&A[j*TB+j*TB*n],SUM,S,TB, n,TB);

                dpotf2(L,&TB,S,&TB,&info);
                if (info != 0)
                        printf("Error en lapack: %d\n",info);

                extraeTriInf(S,&A[j*TB+j*TB*n],TB,TB,n);


                free(S);
                free(SUM);
                #pragma omp parallel for schedule(dynamic) private(S,SUM,i,k,x,y,Aij)
shared(j,A,nB,TB,n) default(none)
            for(i = j+1; i < nB; i++)
            {
                SUM = (double*)calloc(TB*TB,sizeof(double));
                S = (double*)calloc(TB*TB,sizeof(double));
                for(k = 0; k < j;k++)
```

```
                        cblas_dgemm(CblasColMajor,CblasNoTrans,CblasTrans,TB,TB,TB,1
,&A[i*TB+k*TB*n],n,&A[j*TB +k*TB*n],n,1,SUM,TB);

                restaMatriz(&A[i*TB+j*TB*n],SUM,S,TB, n,TB);

                cblas_dtrsm(CblasColMajor,CblasRight,CblasLower,CblasTrans ,CblasNonUnit, TB,TB,1 ,
&A[j*TB+j*TB*n],n,S,TB);

                //copia S en Gij
                Aij = &A[i*TB+j*TB*n];
                for(x = 0; x < TB; x++)
                        for(y = 0; y < TB; y++)
                                Aij[x+y*n] = S[x+y*TB]
                free(S);
                free(SUM);
            }
        }
    fin = dsecnd();

  return fin-ini;
}
/*_____________________________________________________________________________*/
double lapackChol(double* A, long n)
{
        double ini,fin;
        const char * L = "L";
        int info;
        ini = dsecnd();

        //mkl_mic_enable();
        //mkl_mic_set_workdivision(MKL_TARGET_MIC,0,0.1);
        //mkl_mic_set_workdivision(MKL_TARGET_MIC,1,0.0);
        info = LAPACKE_dpotrf(LAPACK_COL_MAJOR,'L',n,A,n);
        fin = dsecnd();

        if (info != 0)
                printf("Error en lapack: %d\n",info);

  return fin-ini;
}

double lapackCholUnb(double* A, int n)
{
        double ini,fin;
        const char * L = "L";
        int info;
        ini = dsecnd();

        dpotf2(L,&n,A,&n,&info);
```

```
        fin = dsecnd();

        if (info != 0)
                printf("Error en lapack unblocked: %d\n",info);

   return fin-ini;
}
```

ANEXO II

En el presente anexo se muestra el código del generador de matrices simétricas y definidas positivas out-of-core. El código fuente completo se encuentra en el archivo adjunto *generadorProblema.c:*

```
int main(int argc, const char* argv[])
{
        unsigned long n, sbTam,nB,i,j,k,x,y;
        unsigned int count = 0;
        double ini,fin;
        const char* fname;
        double * B, * Bc, * Bt, * Aij;
        FILE* f;

        if(argc != 4)
        {
                printf("Uso: generadorP tamaño_matriz tamaño_superbloque nomFich");
                exit(1);
        }
        else
        {
                n = atol(argv[1]);
                sbTam = atol(argv[2]);
                nB = n/ sbTam;
                printf("Tam = %d sbTam= %d nB= %d\n",n,sbTam,nB);
                fname = argv[3];
        }
        srand(time(0));
        printf("Creating matrix...\n");


        //printMatrix(A,n);
        /****** Escritura por bloques en fichero *******/

ini = dsecnd();
        f = fopen(fname,"wb");
        fwrite(&n,sizeof(unsigned long),1,f);
        fwrite(&sbTam,sizeof(unsigned long),1,f);
```

```
Bc = (double *) calloc( sbTam*sbTam,sizeof(double));

for(y = 0; y < sbTam; y++)
        for(x = 0; x < sbTam; x++)
                Bc[x+y*sbTam] = rand() % 100 + 1;

for(j = 0; j < nB; j++)
{

        for(i = j; i < nB; i++ )
        {

                Aij = (double *) calloc( sbTam*sbTam,sizeof(double) );

                if (i == j)
                {
                        Bt = (double *) calloc( sbTam*sbTam,sizeof(double));
                        for(y = 0; y < sbTam; y++)
                        {
                                Bt[y+y*sbTam] = 1000;
                                for(x = y+1; x < sbTam; x++)
                                        Bt[x+y*sbTam] = rand() % 100 + 1;
                        }
                }

                for(k=0; k < j; k++)
                {

                        cblas_dgemm(CblasColMajor,CblasNoTrans,CblasTrans,sbTam,
                        sbTam, sbTam,1,Bc,sbTam,Bc,sbTam,1,Aij,sbTam);
                }




                if(i == j)
                {
                        cblas_dgemm(CblasColMajor,CblasNoTrans,CblasTrans,sbTam,
                        sbTam,sbTam,1,Bt,sbTam,Bt,sbTam,1,Aij,sbTam);
                        B = (double *) malloc( sbTam*(sbTam+1)/2*sizeof(double) );
                        LAPACKE_dtrttf(LAPACK_COL_MAJOR,'N','L',sbTam,Aij,sbTam,B);
                        fwrite(B,sizeof(double),sbTam*(sbTam+1)/2,f);
                        free(B);
                }
                else
                {
                        cblas_dgemm(CblasColMajor,CblasNoTrans,CblasTrans,sbTam ,
                        sbTam,sbTam,1,Bc,sbTam,Bt,sbTam,1,Aij,sbTam);
                        fwrite(Aij,sizeof(double),sbTam*sbTam,f);
```

```
                        }

                        free(Aij);
                }
                free(Bt);
        }
        fclose(f);

        fin = dsecnd();
        printf("Creation time: %g \n",fin-ini);

        return 0;
}

//Auxiliar function to create a positive definite matrix
void createCholMatrix(double* A, unsigned long n)
{
        long i,j;
        double* G =  (double*)calloc(n * n,sizeof(double));

        for(j = 0; j < n; j++)
        {
                G[j+j*n] = 1000;
                for(i = j+1; i < n; i++)
                        G[i+j*n] = rand() % 100 + 1;
        }

        cblas_dgemm(CblasColMajor,CblasTrans,CblasNoTrans,n,n,n,1,G,n,G,n,0,A,n);
        //cblas_dtrmm(CblasColMajor,CblasRight,CblasLower,CblasTrans,CblasNonUnit,n,n,1,G,n,G,n);
        free(G);
}
```

ANEXO III

En el presente anexo se muestra el código encargado de comprobar si el resultado de la descomposición es correcto. El código fuente completo se encuentra en el archivo adjunto *normCheck.c:*

```
int main(int argc, const char* argv[])
{
        int i, j,x,y;
        int TB;
        unsigned long n,sbTam,nB;
        FILE* f;
        double ini,fin,t;
        const char* fname,*fnameChol;
        double* B,*Aij, *L, *Lc;
```

```
	if(argc != 3)
	{
		printf("Uso: cholesky problem_file cholesky_file");
		exit(1);
	}
	else
	{
		fname = argv[1];
		fnameChol = argv[2];
	}

	/******** reserva de memoria para las matrices y creación de A*******/
	double * A;


	printf("Reading problem...\n");
ini = dsecnd();
	f = fopen(fname,"rb+");
	fread(&n,sizeof(unsigned long),1,f);
	fread(&sbTam,sizeof(unsigned long),1,f);
	A = (double *) calloc( n*n,sizeof(double) );
	if(A == NULL){
		printf("Error en la reserva de memoria de A\n");
	}
	nB = n/sbTam;
	for(j = 0; j < nB; j++)
		for(i = j; i < nB; i++ )
		{



			if(i == j)
			{
				B = (double *) malloc( sbTam*(sbTam+1)/2*sizeof(double) );
				fread(B,sizeof(double),sbTam*(sbTam+1)/2,f);
				LAPACKE_dtfttr(LAPACK_COL_MAJOR,'N','L',
				sbTam,B,&A[i*sbTam+j*sbTam*n],n);

				if(i == 0)
				{
					L = (double *) malloc( sbTam*sbTam*sizeof(double));
					LAPACKE_dtfttr(LAPACK_COL_MAJOR,'N','L'
					,sbTam,B,L,sbTam);
					for(y = 0; y < sbTam; y++)
						for(x = y+1; x < sbTam; x++)
							L[y+x*sbTam]=L[x+y*sbTam];
				}
```

```
                        free(B);

                }else
                {
                        B = (double *) malloc( sbTam*sbTam*sizeof(double));
                        fread(B,sizeof(double),sbTam*sbTam,f);
                        Aij = &A[i*sbTam+j*sbTam*n];
                        for(x = 0; x < sbTam; x++)
                                for(y = 0; y < sbTam; y++)
                                        Aij[x+y*n] = B[x+y*sbTam] ;
                        free(B);
                }
        }

        //Reconstrucción de matriz completa
        for(j = 0; j < n; j++)
                for(i = j+1; i < n; i++)
                        A[j+i*n]=A[i+j*n];

        fclose(f);

        fin = dsecnd();
        printf("Read time: %g \n",fin-ini);

        double * Ac;


        printf("Reading cholesky...\n");
ini = dsecnd();
        f = fopen(fnameChol,"rb+");
        fread(&n,sizeof(unsigned long),1,f);
        fread(&sbTam,sizeof(unsigned long),1,f);
        Ac = (double *) calloc( n*n,sizeof(double) );
        if(Ac == NULL){
                printf("Error en la reserva de memoria de A\n");
        }
        nB = n/sbTam;
        for(j = 0; j < nB; j++)
                for(i = j; i < nB; i++ )
                {
                        if(i == j)
                        {
                                B = (double *) malloc( sbTam*(sbTam+1)/2*sizeof(double) );
                                fread(B,sizeof(double),sbTam*(sbTam+1)/2,f);
                                LAPACKE_dtfttr(LAPACK_COL_MAJOR,'N','L',
                                sbTam,B,&Ac[i*sbTam+j*sbTam*n],n);

                                if(i == 0)
                                {
                                        Lc = (double *) malloc( sbTam*sbTam*sizeof(double));
```

```
                                        LAPACKE_dtfttr(LAPACK_COL_MAJOR,'N','L',
                                        sbTam,B,Lc,sbTam);
                                        for(y = 0; y < sbTam; y++)
                                                for(x = y+1; x < sbTam; x++)
                                                        L[y+x*sbTam]=L[x+y*sbTam];
                                }
                                free(B);


                        }else
                        {
                                B = (double *) malloc( sbTam*sbTam*sizeof(double));
                                fread(B,sizeof(double),sbTam*sbTam,f);
                                Aij = &Ac[i*sbTam+j*sbTam*n];
                                for(x = 0; x < sbTam; x++)
                                        for(y = 0; y < sbTam; y++)
                                                Aij[x+y*n] = B[x+y*sbTam] ;
                                free(B);
                        }
                }

        //Reconstrucción de matriz completa
        for(j = 0; j < n; j++)
                for(i = j+1; i < n; i++)
                        Ac[j+i*n]=Ac[i+j*n];

        fclose(f);

        fin = dsecnd();
        printf("Read time: %g \n",fin-ini);

        printf("Norma A-LL'  = %.3e\n",checkNorm(A,Ac,n));
        printf("Norma A-LL' L  = %.3e\n",checkNorm(L,Lc,sbTam));



        return 0;
}

// Extrae la triangular inferior de G, calcula GG', resta A-GG' y devuelve la 2norma de la matriz
resultante.
double checkNorm(double* A, double* G, long n)
{
        double* L =  (double*)calloc(n * n,sizeof(double));
        double* M =  (double*)calloc(n * n,sizeof(double));
        long i,j;
        double norm;

        extraeTriInf(G,L,n,n,n);
```

```
	cblas_dgemm (CblasColMajor, CblasNoTrans, CblasTrans, n, n, n, 1, L, n, L, n, 0, M, n);

	for(i=0;i<n;i++)
		for(j=0;j<n;j++)
			M(i,j)=M(i,j)-A(i,j);

	norm = cblas_dnrm2(n*n,M,1);
	//printMatrix(M,n);

	free(L);free(M);

	return norm;

}
```

ANEXO IV

En el presente anexo se muestra el código del algoritmo de la descomposición de Cholesky out-of-core que es el corazón de este trabajo. El código fuente completo se encuentra en el archivo adjunto *choleskyOoC.c:*

```
int main(int argc, const char* argv[])
{
	int i, j,k,x,y, info;
	unsigned long n,nB,tB,pos, bytesB,bytesL,bytesU;
	FILE* f;
	double ini,fin,t;
	const char* fname;
	double* B,*Aij,* Aii,* Aik, *Ajk,* T,* S;

	if(argc != 2)
	{
		printf("Uso: cholesky problem_file");
		exit(1);
	}
	else
```

```
{
        fname = argv[1];
}

/******** reserva de memoria para las matrices y creación de A*******/
ini = dsecnd();

f = fopen(fname,"rb+");
fread(&n,sizeof(unsigned long),1,f);
fread(&tB,sizeof(unsigned long),1,f);

nB = n/ tB;
bytesB = tB*tB*sizeof(double);
bytesL = tB*(tB+1)/2*sizeof(double);
bytesU = tB*(tB-1)/2*sizeof(double);
printf("n = %d, tB = %d, nB = %d\n",n,tB,nB);

//primera columna no necesita nada de columnas anteriores
j = 0;
Aii = (double *) mkl_malloc( bytesL ,64);
fread(Aii,sizeof(double),tB*(tB+1)/2,f);

info = LAPACKE_dpftrf(LAPACK_COL_MAJOR,'N','L',tB,Aii);
if (info != 0)
        printf("Error en lapack(dpftrf): %d\n",info);

fseek(f,16,SEEK_SET);
fwrite(Aii,sizeof(double),tB*(tB+1)/2,f);

T = (double*)mkl_malloc(tB*tB * sizeof(double),64);
info = LAPACKE_dtfttr(LAPACK_COL_MAJOR,'N','L',tB,Aii,T,tB);
if (info != 0)
        printf("Error en lapack(dpftrs): %d\n",info);
mkl_free(Aii);

for(i = 1; i < nB; i++)
{
        //printf("[%d,%d]pos: %d\n",i,j,ftell(f));
        Aij = (double*)mkl_malloc(tB*tB * sizeof(double),64);
        fread(Aij,sizeof(double),tB*tB,f);

        cblas_dtrsm(CblasColMajor,CblasRight,CblasLower,CblasTrans,
        CblasNonUnit,tB,tB,1,T,tB,Aij,tB);

        fseek(f,(i+j*nB)*bytesB-(j+1)*bytesU+16,SEEK_SET);
        fwrite(Aij,sizeof(double),tB*tB,f);

        mkl_free(Aij);
}
mkl_free(T);
```

```
//resto de columnas
for(j = 1; j < nB; j++)
{
	for(i = j; i < nB; i++)
	{
		if( i == j)
		{

			S = (double*)mkl_malloc(tB*tB * sizeof(double),64);
			for(k = 0; k < j;k++)
			{
				Aik = (double*)mkl_malloc(tB*tB * sizeof(double),64);
				fseek(f,(i+k*nB-(k*(k+1)/2))*bytesB-(k+1)*bytesU +16,
				SEEK_SET);
				//printf("[%d,%d] Aik pos: %d\n",i,j,ftell(f));
				fread(Aik,sizeof(double),tB*tB,f);
				cblas_dgemm(CblasColMajor,CblasNoTrans,CblasTrans,tB,tB
				tB,1,Aik,tB,Aik,tB,1,S,tB);
				mkl_free(Aik);
			}
			pos = (i+j*nB-(j*(j+1)/2))*bytesB-(j)*bytesU+16;
			Aii = (double *) mkl_malloc( bytesL ,64);
			fseek(f,pos,SEEK_SET);
			//printf("[%d,%d] Aii pos: %d\n",i,j,ftell(f));
			fread(Aii,sizeof(double),tB*(tB+1)/2,f);

			//resta Aii = Aii - S
			B = (double *) mkl_malloc( bytesL ,64);
			LAPACKE_dtrttf(LAPACK_COL_MAJOR,'N','L',tB,S,tB,B);

			cblas_daxpy(tB*(tB+1)/2,-1,B,1,Aii,1);
			mkl_free(B);
			//
			info = LAPACKE_dpftrf(LAPACK_COL_MAJOR,'N','L',tB,Aii);
			if (info != 0)
				printf("[%d,%d]Error en lapack(dpftrf): %d\n",i,j,info);

			fseek(f,pos,SEEK_SET);
			fwrite(Aii,sizeof(double),tB*(tB+1)/2,f);

			T = (double*)mkl_malloc(tB*tB * sizeof(double),64);
			info = LAPACKE_dtfttr(LAPACK_COL_MAJOR,'N','L',tB,Aii,T,tB);

			mkl_free(Aii);
			mkl_free(S);
		}
		else
		{
```

```
                                S = (double*)mkl_malloc(tB*tB * sizeof(double),64);
                                for(k = 0; k < j;k++)
                                {
                                        Aik = (double*)mkl_malloc(tB*tB * sizeof(double),64);
                                        fseek(f,(i+k*nB-(k*(k+1)/2))*bytesB-(k+1)*bytesU+16,
                                        SEEK_SET);
                                        fread(Aik,sizeof(double),tB*tB,f);

                                        Ajk = (double*)mkl_malloc(tB*tB * sizeof(double),64);
                                        fseek(f,(j+k*nB-(k*(k+1)/2))*bytesB-(k+1)*bytesU+16,
                                        SEEK_SET);
                                        fread(Ajk,sizeof(double),tB*tB,f);

                                        cblas_dgemm(CblasColMajor,CblasNoTrans,CblasTrans,tB,
                                        tB,tB,1,Aik,tB,Ajk,tB,1,S,tB);
                                        mkl_free(Aik);
                                        mkl_free(Ajk);
                                }
                                pos = (i+j*nB-(j*(j+1)/2))*bytesB-(j+1)*bytesU+16;
                                Aij = (double*)mkl_malloc(tB*tB * sizeof(double),64);
                                fseek(f,pos,SEEK_SET);
                                fread(Aij,sizeof(double),tB*tB,f);

                                cblas_daxpy(tB*tB,-1,S,1,Aij,1);

                                cblas_dtrsm(CblasColMajor,CblasRight,CblasLower,CblasTrans,
                                CblasNonUnit,tB,tB,1,T,tB,Aij,tB);

                                fseek(f,pos,SEEK_SET);
                                fwrite(Aij,sizeof(double),tB*tB,f);

                                mkl_free(Aij);
                                mkl_free(S);
                        }

                }
                mkl_free(T);
        }
  fin = dsecnd();

        printf("Cholesky ooC time: %g \n",fin-ini);

        return 0;
}
```