# Caso de estudio 1
# La descomposición de Cholesky

Por: Hugo Montoto

# 1. Definición del problema a estudiar

En este caso de estudio se va a mostrar varias formas de manejar la factorización de Cholesky. La descomposición o factorización de Cholesky define que una matriz simétrica definida positiva puede ser descompuesta como el producto de una matriz triangular inferior por su traspuesta. Se estudiará la eficiencia de la implementación de esta descomposición desde distintos modelos de programación así como también desde distintos lenguajes de programación. Se comenzará por la programación secuencial del algoritmo, la más básica. A partir de aquí se irá mejorando la eficiencia de la factorización de Cholesky mediante la división por bloques . Esta técnica mostrará como mejora en términos de tiempo de ejecución el algoritmo, tanto en secuencial como en paralelo. Para utilizar la división por bloques haremos uso de un parámetro adicional, el tamaño de bloque. Variar el tamaño del bloque producirá distintos tiempos de ejecución y es objetivo de este estudio obtener el valor adecuado para la arquitectura utilizada.
El siguiente paso en nuestro estudio es el uso de librerías externas especializadas, en nuestro caso se hará uso de rutinas BLAS/LAPACK. Concretamente haremos uso de las siguientes rutinas: dpotrf, dtrsm, dgemm y dsyrk.

El resultado deberá reflejar una eficiencia mayor con respecto a las anteriores tentativas gracias al uso de librerías especializadas.
Con esto queda concluido una parte del estudio, el análisis secuencial del algoritmo. El proceso continuará con la programación en paralelo de la descomposición de Cholesky. El trato en paralelo de este algoritmo se hará desde distintos puntos de vista. Primero se hará una implementación mediante el uso de directivas OpenMP básicas, sin el uso de tasks. Una vez se midan tiempos ya se incluirá el uso de tasks y se concluirá con una implementación del algoritmo utilizando Threading Building Blocks.

La máquina sobre la que efectuaremos las pruebas es el clúster Kahan. Consta de 6 nodos biprocesador conectados mediante una red InfiniBand. Cada nodo consta de:
·2 procesadores AMD Opteron 16 Core 6272, 2.1GHz, 16MB
·32GB de memoria DDR3 1600
·Disco 500GB, SATA 6 GB/s
·Controladora InfiniBand QDR 4X (40Gbps, tasa efectiva de 32Gbps)
En total: 12 procesadores, 192 núcleos, 192 GB

# 2. Planteamiento de la solución

El resultado que queremos obtener es la implementación más eficiente de la descomposición de Cholesky. Al partir de unos algoritmos secuenciales podemos afirmar que estos no serán la solución mas óptima. Sin embargo esta conclusión no significa que todo algoritmo paralelo va a resultar más rápido que su homónimo secuencial. En nuestro caso concreto sí es así porque habiendo un producto de matrices podemos intuir que este se puede dividir en subtareas mas pequeñas sin dependencias entre ellas, para luego, recopilando todos los datos, obtener un resultado conjunto. De manera más genérica una algoritmo se puede acelerar con paralelismo solo si lo soporta, ya sea a nivel de datos, de tareas, de instrucciones o incluso a nivel de bit.
El resultado que vamos a obtener de las dos implementaciones paralelas(OpenMP y TBB) no es tan fácil de pronosticar sin realizar ninguna prueba. El uso de OpenMP requiere de la elección del tipo de planificador que se quiera, sin embargo TBB hace esto de una manera automática, esto hace que

si el algoritmo no se adecúa bien a alguno de los perfiles del planificador puede que TBB obtenga un mejor rendimiento.

# 3. Evaluación experimental

La primera pareja de algoritmos que compararemos son las diferentes implementaciones del algoritmo escalar en C y en Octave. Ambos programas serán ejecutados fuera del kahan puesto que no se necesita una gran carga de trabajo para compararlas. Aplicándolos sobre una matriz(n =30) obtenemos:

Algoritmo en C:

Error = 1.3925e-14

Tiempo del algoritmo tamaño 30 --> 0.0000590000 seg

Algoritmo en Octave:

octave:5> tic;chol_escalar(A);toc;

Elapsed time is 0.09919 seconds.

Se puede comprobar como el algoritmo implementado en C es mucho más eficiente. Es más a valores más altos el tiempo que consume el algoritmo de Octave se dispara:

Algoritmo en C: Con **N = 1000**

Error = 1.1486e-11

Tiempo del algoritmo tamaño 1000 --> 1.9910110000 seg

Algoritmo en Octave: Con **N = 100**

Elapsed time is 2.458 seconds.

Aún con una carga 10 veces menor Octave sigue quedando por detrás en rendimiento.

Una vez visto la diferencia entre estas dos implementaciones, nos centraremos más en los diferentes enfoques tomados dentro de C.

La siguiente batería de pruebas se hará con el algoritmo 2, secuencial, con el uso de las rutinas BLAS/LAPACK.

Algoritmo 1, secuencial:

| N | 1500 | 3000 | 5000 |
|---|---|---|---|
| Tiempo en segundos | 7.151877 | 72.899927 | 344.40426 |

Algoritmo 2, secuencial:

Tiempo del algoritmo tamaño 1500 y b = 50 --> 0.5038140000 seg
**Tiempo del algoritmo tamaño 1500 y b = 100 --> 0.4683890000 seg**
Tiempo del algoritmo tamaño 1500 y b = 200 --> 0.4807140000 seg
Tiempo del algoritmo tamaño 1500 y b = 300 --> 0.4742870000 seg
Tiempo del algoritmo tamaño 1500 y b = 400 --> 0.4702920000 seg
Tiempo del algoritmo tamaño 1500 y b = 500 --> 0.4730170000 seg
Tiempo del algoritmo tamaño 1500 y b = 1000 --> 0.4730290000 seg

Tiempo del algoritmo tamaño 3000 y b = 50 --> 3.7842620000 seg
Tiempo del algoritmo tamaño 3000 y b = 100 --> 3.5628650000 seg
**Tiempo del algoritmo tamaño 3000 y b = 200 --> 3.5461620000 seg**
Tiempo del algoritmo tamaño 3000 y b = 300 --> 3.5768360000 seg
Tiempo del algoritmo tamaño 3000 y b = 400 --> 3.5502170000 seg
Tiempo del algoritmo tamaño 3000 y b = 500 --> 3.5567290000 seg
Tiempo del algoritmo tamaño 3000 y b = 600 --> 3.5778780000 seg
Tiempo del algoritmo tamaño 3000 y b = 1000 --> 3.5703220000 seg

Tiempo del algoritmo tamaño 5000 y b = 100 --> 16.2341210000 seg
Tiempo del algoritmo tamaño 5000 y b = 200 --> 16.1126100000 seg
Tiempo del algoritmo tamaño 5000 y b = 300 --> 16.2451030000 seg
**Tiempo del algoritmo tamaño 5000 y b = 400 --> 16.0663080000 seg**
Tiempo del algoritmo tamaño 5000 y b = 500 --> 16.1332260000 seg
Tiempo del algoritmo tamaño 5000 y b = 600 --> 16.1730040000 seg
Tiempo del algoritmo tamaño 5000 y b = 1000 --> 16.1710680000 seg
Tiempo del algoritmo tamaño 5000 y b = 2500 --> 16.2969530000 seg

Tiempo del algoritmo tamaño 7000 y b = 100 --> 44.3280180000 seg
Tiempo del algoritmo tamaño 7000 y b = 200 --> 43.8787710000 seg
Tiempo del algoritmo tamaño 7000 y b = 300 --> 44.2472870000 seg
**Tiempo del algoritmo tamaño 7000 y b = 400 --> 43.7693310000 seg**
Tiempo del algoritmo tamaño 7000 y b = 500 --> 43.8742680000 seg
Tiempo del algoritmo tamaño 7000 y b = 600 --> 44.1211840000 seg
Tiempo del algoritmo tamaño 7000 y b = 1000 --> 43.9237090000 seg
Tiempo del algoritmo tamaño 7000 y b = 2500 --> 44.2340940000 seg

Se puede observar como el tiempo de ejecución mejora drásticamente en el algoritmo 2. Independientemente del tamaño de bloque,b , el segundo algoritmo siempre es más rápido.
Otro aspecto que queríamos resolver era que tamaño de bloque era el idóneo para esta arquitectura. Se aprecia que no existe un tamaño de bloque perfecto que encaje en todas las ocasiones, si no que depende también del tamaño de la matriz. Sin embargo a valores altos de n, que es lo que nos interesa podemos ver como el valor ideal es cercano a 400. Incluso para matrices más pequeñas, como por ejemplo de tamaño 1500 se ve que aunque el mejor valor de b no es 400, si se consigue un tiempo cercano al mejor.

La siguiente toma de tiempos realizada es a la implementación en OpenMP del algoritmo1. Los resultados son los siguientes:

Algoritmo 1, OpenMP:

Tiempo del algoritmo tamaño 600 --> 0.6587280000 seg
Tiempo del algoritmo tamaño 1500 --> 9.8130710000 seg
Tiempo del algoritmo tamaño 3000 --> 86.6175960000 seg
Tiempo del algoritmo tamaño 5000 --> 409.8685890000 seg

Procedemos a la toma de tiempos del algoritmo 2 implementado mediante directivas #pragma omp parallel for:

Algoritmo 2, OpenMP:

Tiempo del algoritmo tamaño 1500 y b = 50 --> 1.1567170000 seg
Tiempo del algoritmo tamaño 1500 y b = 100 --> 0.6303400000 seg
**Tiempo del algoritmo tamaño 1500 y b = 500 --> 0.4818930000 seg**
Tiempo del algoritmo tamaño 1500 y b = 1000 --> 0.5790620000 seg

**Tiempo del algoritmo tamaño 3000 y b = 50 --> 0.6336650000 seg**
Tiempo del algoritmo tamaño 3000 y b = 100 --> 0.6577990000 seg
Tiempo del algoritmo tamaño 3000 y b = 500 --> 2.1097440000 seg
Tiempo del algoritmo tamaño 3000 y b = 1000 --> 3.3548450000 seg

Tiempo del algoritmo tamaño 5000 y b = 100 --> 4.5591180000 seg

**Tiempo del algoritmo tamaño 5000 y b = 200 --> 3.4056540000 seg**
Tiempo del algoritmo tamaño 5000 y b = 300 --> 3.6803720000 seg
Tiempo del algoritmo tamaño 5000 y b = 500 --> 5.7917300000 seg
Tiempo del algoritmo tamaño 5000 y b = 1000 --> 9.6407570000 seg
Tiempo del algoritmo tamaño 5000 y b = 2500 --> 16.6415780000 seg

Tiempo del algoritmo tamaño 7000 y b = 50 --> 11.3129840000 seg
**Tiempo del algoritmo tamaño 7000 y b = 100 --> 7.0231300000 seg**
Tiempo del algoritmo tamaño 7000 y b = 200 --> 7.1338880000 seg
Tiempo del algoritmo tamaño 7000 y b = 300 --> 8.9765080000 seg
Tiempo del algoritmo tamaño 7000 y b = 400 --> 9.2491670000 seg
Tiempo del algoritmo tamaño 7000 y b = 500 --> 12.0958320000 seg
Tiempo del algoritmo tamaño 7000 y b = 1000 --> 19.9502590000 seg
Tiempo del algoritmo tamaño 7000 y b = 2500 --> 34.5229110000 seg

Al igual que en su versión secuencial nos fijamos en cómo afecta el tamaño de bloque al tiempo de ejecución. En este caso se puede apreciar que el cambiar el tamaño de bloque tiene un gran impacto en el tiempo de ejecución. Mientras que en el algoritmo secuencial solo se veía afectado unas décimas de segundo aquí la diferencia alcanza la de varios segundos. El valor de b que ofrece mejores resultados en matrices grandes son los valores cercanos al 100.
Otro aspecto que se puede observar es como en el algoritmo paralelo en comparación con el secuencial es más lento para tamaños pequeños de la matriz, esto es debido a la sobrecarga generada cuando se crean muchos hilos cada uno con una carga de trabajo muy pequeña.

La siguiente implementación es del algoritmo 2 con tareas de OpenMP.
Algoritmo 2 , OpenMP Task:

Tiempo del algoritmo tamaño 1500 y b = 50 --> 0.0567890000 seg
**Tiempo del algoritmo tamaño 1500 y b = 100 --> 0.0531080000 seg**
Tiempo del algoritmo tamaño 1500 y b = 200 --> 0.0653210000 seg
Tiempo del algoritmo tamaño 1500 y b = 300 --> 0.0782130000 seg
Tiempo del algoritmo tamaño 1500 y b = 400 --> 0.1361710000 seg
Tiempo del algoritmo tamaño 1500 y b = 500 --> 0.1707140000 seg
Tiempo del algoritmo tamaño 1500 y b = 1000 --> 0.4613980000 seg

Tiempo del algoritmo tamaño 3000 y b = 50 --> 0.3527690000 seg
**Tiempo del algoritmo tamaño 3000 y b = 100 --> 0.3206920000 seg**
Tiempo del algoritmo tamaño 3000 y b = 200 --> 0.3653390000 seg
Tiempo del algoritmo tamaño 3000 y b = 300 --> 0.3484400000 seg
Tiempo del algoritmo tamaño 3000 y b = 400 --> 0.3592430000 seg
Tiempo del algoritmo tamaño 3000 y b = 500 --> 0.4221710000 seg
Tiempo del algoritmo tamaño 3000 y b = 1000 --> 1.5844380000 seg

Tiempo del algoritmo tamaño 5000 y b = 100 --> 1.4076260000 seg
**Tiempo del algoritmo tamaño 5000 y b = 200 --> 1.3799900000 seg**
Tiempo del algoritmo tamaño 5000 y b = 300 --> 1.4058170000 seg
Tiempo del algoritmo tamaño 5000 y b = 400 --> 1.4035420000 seg
Tiempo del algoritmo tamaño 5000 y b = 500 --> 1.4814830000 seg
Tiempo del algoritmo tamaño 5000 y b = 1000 --> 2.2076380000 seg
Tiempo del algoritmo tamaño 5000 y b = 2500 --> 8.5439550000 seg

Tiempo del algoritmo tamaño 7000 y b = 100 --> 3.6254310000 seg
Tiempo del algoritmo tamaño 7000 y b = 200 --> 3.6469440000 seg

Tiempo del algoritmo tamaño 7000 y b = 300 --> 3.6947000000 seg
**Tiempo del algoritmo tamaño 7000 y b = 400 --> 3.6415240000 seg**
Tiempo del algoritmo tamaño 7000 y b = 500 --> 3.7302520000 seg
Tiempo del algoritmo tamaño 7000 y b = 1000 --> 4.3358030000 seg
Tiempo del algoritmo tamaño 7000 y b = 2500 --> 12.1955300000 seg
Tiempo del algoritmo tamaño 7000 y b = 2500 --> 12.2528180000 seg

Los tiempos obtenidos con el uso de las task de openmp son algo más bajos que los obtenidos con tan solo las directivas #pragma omp parallel for. Los valores de que obtienen los mejores resultados se encuentran en el rango de [100, 400], sin embargo para tamaños de matriz pequeños este rango se ve disminuido por su límite superior.

# 4. Conclusiones.

Con este estudio hemos comprobado como se puede acelerar un algoritmo con el uso de librerías especializadas y paralelismo. También ha mostrado como la paralelización no es siempre la solución, puesto que hemos obtenido en casos específicos mejores resultados con el algoritmo secuencial que con el paralelo (cuando la cantidad de datos a tratar es pequeña,).

# 5.Fuentes.

Los archivos fuentes están adjuntados a la tarea.