Implementación de la Factorización Cholesky con openMP

Alejandro Nivón[1], Uriel Miranda[2], Héctor Corro[3]

Maestría en Ciencia de Datos
Instituto Tecnológico Autónomo de México

1. Introducción

Objetivo: aprovechar las oportunidades creadas o generadas por las tecnologías actuales para la paralelización de rutinas del cálculo numérico, para cuestiones de este trabajo y para la implementación de alguna de los paradigmas de programación vistos en la clase de Análisis Numérico y Computo Científico. Se hará uso de openMP para la factorización Cholesky aprovechando las siguientes características de openMP:

- Paralelización for-loops secuenciales de forma simple
- Paralelización de tarea y sincronización explícita de threads
- Permite una paralelización del código secuencial de forma paulatina (incremental parallelism)

Factorización Cholesky Una matriz simétrica nxn A es positiva definida si su forma cuadrática X^TAX es positiva definida para todos los vectores no-cero x o, equivalentemente si todos los eigenvalores de A son positivos. Las matrices positivas definidas pueden ser expresadas de la forma $A = X^TX$ para una matriz X no singular. La factorización Cholesky es una forma particular de factorizar X, en la que X es la matriz triangular superior con elementos positivos en su diagonal; generalmente es escrito como: $A = R^TR$ o $A = LL^T$ de una matriz definida A, en la que R es una matriz triangular superior con elementos positivos en su diagonal es una herramienta fundamental para la computación matricial. El algoritmo estándar para su cómputo data de principios del siglo pasado y es uno de los métodos numéricos más estables de todos los algoritmos matriciales. La factorización Cholesky existe y es única si A es positiva definida. [**?**]

Lemma 1.1: Sea A positiva semi-definida de rango r.

1. Existe al menos una matriz triangular superior R con elementos no negativos en su diagonal tal que $A = R^TR$.

1. Hay una permutación $\prod$ tal que $\prod^T A \prod$ tiene una única factorización cholesky que toma la forma:

$$\prod^T A \prod = R^TR, \tag{1}$$

$$R = \begin{pmatrix} r_{1,1} & r_{1,2} \\ 0 & 0 \end{pmatrix}$$

2. Aplicaciones

La descomposición de Cholesky se usa principalmente para hallar la solución numérica de ecuaciones lineales $Ax = b$. Si a es simétrica y positiva definida, entonces se puede solucionar $Ax = b$ calculando primero la descomposición de Cholesky $A = LL^T$, luego resolviendo $Ly = b$ para y, y finalmente resolviendo $L^T x = y$ para x.

- **Mínimos cuadrados lineales:**

Sistemas de la forma $Ax = b$ con A simétrica y definida positiva aparecen a menudo en la práctica. Por ejemplo las ecuaciones normales en problemas de mínimos cuadrados lineales son problemas de esta forma.

- **Simulación de Montecarlo:**

La descomposición de cholesky se usa comúnmente en el método de Montecarlo para simular sistemas con variables múltiples correlacionadas: la matriz de correlación entre variables es descompuesta, para obtener la triangular inferior L. Aplicando ésta un vector de ruidos simulados incorrelacionados, u produce un vector Lu con las propiedades de covarianza del sistema a ser modelado.

- **Filtro de Kalman:**

Los filtros de Kalman usan frecuentemente la descomposición de Cholesky para escoger un conjunto de puntos sigma. El filtro de Kalman sigue el estado promedio de un sistema como un vector x de longitud n y covarianza dada por una matriz P de tamaño nxn. La matriz P es siempre semidefinida positiva y puede descomponerse como LL^T. Las columnas de L pueden ser adicionadas y restadas de la media x para formar un conjunto de $2N$ vectores llamados puntos sigma. Estos puntos sigma capturan la media y la covarianza del estado del sistema.

- **Estabilidad del proceso de Cholesky**

En aritmética exacta se sabe que una matriz simétrica positiva definida tiene una factorización Cholesky. Asimismo, si el proceso Cholesky que se mostrará posteriormente en este trabajo realiza su iteración hasta su terminación con raíces cuadradas estrictamente positivas, entonces la matriz A es positiva definida. Por lo tanto, para encontrar si una matriz A es positiva definida, solamente se intentará de computar la factorización Cholesky usando el método citado a continuación.

3. Calculo

La Factorización Cholesky puede ser calculada por una forma de eliminación gaussiana que toma ventaja de la simetría y definición. Iterando (i, j), elementos en la ecuación $A = R^T R$ como se ve a continuación:

$$\text{J} = i \qquad a_{ii} = \sum_{k=1}^{i} r_{ki}^2 \tag{2}$$

$$\text{J} > i \qquad a_{ij} = \sum_{k=1}^{i} r_{ki} r_{kj} \tag{3}$$

```
for j = 1:n
    for i = 1:j − 1
        r_ij = (a_ij − Σ_{k=1}^{i−1} r_ki r_kj)/r_ii
    end
    r_jj = (a_jj − Σ_{k=1}^{j−1} r_kj^2)^{1/2}
end
```

Figura 1: Algoritmo: Factorización Cholesky

Estas ecuaciones pueden ser resueltas para producir R columnas a la vez, de acuerdo con el siguiente algoritmo:

Que A sea positiva definida garantiza que el argumento de la raíz cuadrada en este algoritmo es siempre positivo y por lo tanto que R tiene una diagonal positiva. Este algoritmo requiere $n^3/3 + O(n^2)$ flops y n raíces cuadrada, donde un flop es cualquiera de las 4 operaciones aritméticas elementales escalares $+, -, *$ y $/$. [**?**]

4. Algoritmo

Algoritmo: Después de investigar diferentes aproximaciones y formas de desarrollar la factorización de Cholesky, así como su implementación, se decidió paralelizar con base en el cálculo de filas. Para esto lo más importante a considerar es el orden en que se realizan los cálculos.

Regresando al algoritmo original de la factorización de Cholesky se tienen dos fórmulas, una para los elementos en la diagonal y otra para los elementos bajo la diagonal. [**?**]

- **Elementos en la diagonal**

$$L_{ij} = \sqrt{A_{ij} - \sum_{k=1}^{j-1} L_{jk}^2} \tag{4}$$

- **Elementos bajo la diagonal**

$$L_{ij} = \frac{1}{L_{jj}}\left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk}\right) \tag{5}$$

El algoritmo para realizar el proceso de manera paralela se deriva de la forma y dependencia de los elementos que se requieren para realizar el cálculo de cada uno de los elementos. Iniciando por el elemento l_{11}:

Aplicando la fórmula para elementos en la diagonal, tenemos:

$$l_{11} = \sqrt{a_{11}} \tag{6}$$

Para calcular el elemento l_{21}:

$$l_{21} = \sqrt{a_{21}} \tag{7}$$

Por lo cual para cada elemento de la columna j=1 no requiere dependencia de ningún otro elemento, por lo cual se puede realizar en paralelo.

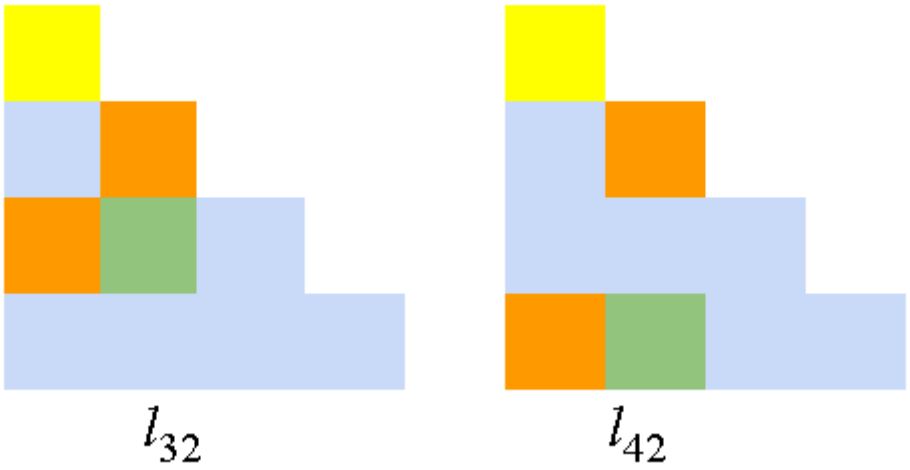
Siguiendo con esa línea para calcular el elemento l_{32}

$$l_{32} = \frac{1}{l_{33}}(a_{32} - l_{31}l_{21}) \tag{8}$$

Comparamos con el cálculo del elemento l_{42}:

$$l_{42} = \frac{1}{l_{44}}(a_{42} - l_{41}l_{21}) \tag{9}$$

La dependencia de elementos de la matriz L para los dos elementos de la misma columna es:



Los colores en este caso tienen los siguientes distintivos:

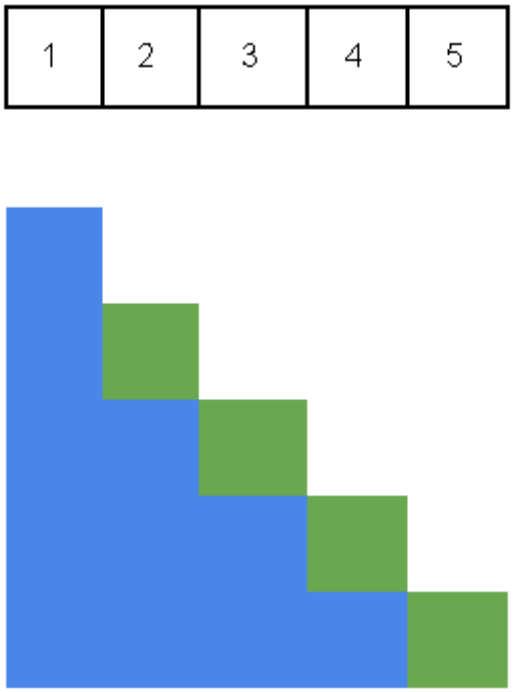
- Verde: El elemento que se está calculando.
- Naranja: Los elementos de los que depende directamente el elemento calculado.
- Amarillo: Los elementos de los que depende indirectamente el elemento calculado.

Con los ejemplos anteriores podemos ver que solamente se requiere el cálculo de los elementos de la fila (anteriores a la columna del elemento que se desea calcular) y el elemento que cruza con la diagonal de la misma columna.

El algoritmo para el cálculo en paralelo será:

1. Se calcula el elemento de la diagonal l_{kk}.
2. Se calculan en paralelo los elementos para cada i con la j fija.
3. Se repite para toda $k <= n$ donde n es la dimensión de la matriz, dado que es simétrica tendremos matrices de nxn.

En el primer punto el cálculo depende únicamente de los elementos de la fila k, en una matriz de $n = 5$ el orden de cálculo será:



El orden es de $1 \rightarrow 5$ y empezando por el elemento en verde, siendo los elementos en azul ,para cada columna, los que se podrán calcular en paralelo. [**?**]

5. Implementación

Después de analizar los algoritmos vistos anteriormente estos se implementaron en C y OpenMP para explotar las características de esta API. Una de las caracteristicas corresponde a la forma simple de establecer que un bloque de código sea ejecutado en paralelo, algunas otras se citan a continuación:

- Paralelizar for-loops secuenciales de forma simple
- Paralelización de tareas y sincronización explícita de threads
- Permite una paralelización del código secuencial de forma paulatina (incremental parallelism)

Algunas observaciones de OpenMP: openMP es una API que provee de directivas para shared-memory. En términos de C, esto significa que se tiene instrucciones especiales para el procesador. Para openMP se usa *#pragma*. Los pragmas son añadidos al sistema para permitir comportamientos que no son parte de la especificación básica de C. Los pragmas de openMP siempre comienzan con: *#pragma omp*.

1. Se diseñó y programó en Python un script que genera matrices para su posterior implementación de la Factorización Cholesky en C.
    - El nombre del archivo que contiene el script es **gen_mat.py**.

- Se ejecuta con el comando: python **gen_mat.py n**, en el cual se debe sustituir **n** por la dimensión de la matriz cuadrada positiva definida a generar.
- El resultado se almacenará en el archivo **matrizSPD.txt** en una sola columna, para después ser el insumo del algoritmo de factorización de cholesky con **chol.c**.
- Para tener una mejor apreciación de la matriz se estructurará en el archivo **MATRIZ.txt** solo como referencia.

2. El script **chol.out** realizará mediante standar input la ingesta de los elementos de la matriz y posteriormente imprimir la matriz factor en el archivo: **fact.txt**. Se dejan los archivos .txt como ejemplo con matrices de dimensión $50x50$.

3. El script **Ejercicio.sh** realiza una nueva iteración del proceso. Ejecutar como: **source Ejercicio.sh**.

4. Por último los scripts **cholesky_1.c** y **chol_seq.c** son ejemplos de aplicaciones de matrices pequeñas en el algoritmo de la factorización cholesky tanto secuencial como en paralelo con tiempos de ejecución y matrices introducidas a mano en el script.

6. Código

A continuación se muestran listados los códigos enunciados en la sección anterior de este trabajo escrito.

1. Python script para las matrices:

```
import sys
import pandas as pd
import numpy as np
import random

def gen_matSPD(dim):
        dim = int(dim)
        mat = pd.DataFrame()

        for i in range(1,dim+1):
                list=[random.randrange(-10000, 10000)/100 for j in range(i)]
                for n in range(dim-i):
                        list.append(0)
                list = pd.DataFrame(list)
                mat=pd.concat([mat,list], axis=1)

        mat = np.dot(mat, np.transpose(mat))

        with open('matrizSPD.txt','wb') as f:
            for line in mat:
                np.savetxt(f, line, fmt='%.2f')
        f.close()

if __name__ == "__main__":
gen_matSPD(sys.argv[1])
```

2. Código para cholesky_1.c:

```
#include <stdlib.h>\
#include <math.h>\
#include <omp.h>\
#include <time.h>\
void cholesky(float **A, float **L, int n);\
void printm(float **M, int n);\


int main(int argc, char **argv)\{\
/*[Tiempo] */\

double time_p = 0;\


/*[Test] Values */\
float tm[25] = \{29,5,9,5,6,5,29,10,8,7,9,10,23,4,5,5,8,4,26,6,6,7,5,6,30\};
/*float tm[9] = \{25, 15, -5,15, 18,  0,-5,  0, 11\};*/\
int n =5;\

/*Matrix A and Matrix L */\
float  *mA = (float *)malloc(n*n*sizeof(float));\
float  *mL = (float *)calloc(n*n,sizeof(float));\

float **A;\
float **L;\

/* Assign values to the array mA*/\
int i,j, count = 0;\
for (i = 0;i<n;i++)\
        for (j = 0;j < n ;j++)\
                *(mA + i*n + j) = tm[count++];\
/* Print the matrix values */\

A = &mA;\
L = &mL;\

printf("A =\\n");\
printm(A,n);\
clock_t start = clock();\
cholesky(A,L,n);\
clock_t end = clock();\

time_p = (double)(end-start);\
printf("tiempo: %f\\n",time_p);\
printf("L =\\n");\
printm(L,n);\
free(mA);\
free(mL);\
```

```
return 0;\
\}\



void cholesky(float **A, float **L, int n)\{\
\
int k,j,i,l;\
float s=0;\
int thr =1;\
\
for(j=0;j<n;j++)\{\
        s=0;\
        for(k=0;k<j;k++)\
                s+= *(*L+j*n+k)* *(*L+j*n+k);\
        /*printf("A:%f\\n s: %f\\n",*(*A+j*n+j),s);*/\
        *(*L+j*n+j) = sqrt( *(*A+j*n+j) -s);\
        thr = n-(j+1);\
        #pragma omp parallel for num_threads(thr) private(i,l,s)\
        for (i=j+1;i<n;i++)\{\
                s =0;\
               /* printf("Thread: %d de %d para calcular L(%d,%d)\\n",omp_get_thr
                for (l=0;l<j;l++)\{\
                        /*printf("s: %f ,",s);\
                        printf("(%d, %d,%d, %d, %d, %f, %f) ",i,j,l,i*n+l,j*
                        s+= *(*L+i*n+l) * *(*L+j*n+l);\
                \}\
                /*printf("i: %d, l: %d ,s: %f :",i,l,s);*/\
                *(*L+i*n+j) = (1.0/ *(*L+j*n+j) )*( *(*A+i*n+j) - s);\
                /*printf("%f \\n",*(*L+i*n+j));*/\
        \}\
\}\
\
\}\
\
/*Print matrix values in format | | */\
void printm(float **M, int n)\{\
int i,j;\
for (i=0; i<n;i++)\{\
        printf("| ");\
        for (j=0;j<n;j++)\
                printf("%f\\t",*(*M+i*n+j));\
        printf("|\\n");\}\
\
\}}
```

3. Código para **chol.c**

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>

// PROPIAMENTE SE CALCULA LA FACTORIZACION CHOLESKY
double *cholesky(double *A, int n) {
    double *L = (double*)calloc(n * n, sizeof(double));
    if (L == NULL)
        exit(EXIT_FAILURE);

    for (int i = 0; i < n; i++)
        for (int j = 0; j < (i+1); j++) {
            double s = 0;
            for (int k = 0; k < j; k++)
                s += L[i * n + k] * L[j * n + k];
            L[i * n + j] = (i == j) ?
                           sqrt(A[i * n + i] - s) :
                           (1.0 / L[j * n + j] * (A[i * n + j] - s));
        }

    return L;
}

 // SE IMPRIME LA MATRIZ
void show_matrix(double *A, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            printf("%2.5f ", A[i * n + j]);
        printf("\n");
    }
}

// SE OBTIENE LA DIMENSION DE LA MATRIZ A FACTORIZAR
int get_dim(){
    int n=0;
    char c;

    while((c = getchar()) != EOF) {
        if(c == '\n') {
            n++;
        }
    }
    return(sqrt(n));
}

// SE PUEBLA LA MATRIZ
double *arr_mat(int n, double *arr) {

    FILE * matriz;
    double num;
    int i;
```

```
    matriz = fopen("matrizSPD.txt", "r");

    for (i=0; i<n; i++) {
        fscanf(matriz, "%lf", &num);
        arr[i]=num;
    }

    return arr;
}

int main() {
    int dim,n,i;
    dim = get_dim();
    n=pow(dim,2);
    double *matap = malloc(sizeof(double)*n);
    double matriz[n];

    matap = arr_mat(n, matap);

    for(i=0;i<n;i++){
        matriz[i]=*matap;
//          printf("%f\n", matriz[i]);
        matap++;
    }

    double *fact = cholesky(matriz, dim);
    show_matrix(fact, dim);
    printf("\n");
    free(fact);

    return 0;
}
```

Bibliografía

[1] Jaeyoung Choi, Jack J Dongarra, L Susan Ostrouchov, Antoine P Petitet, David W Walker, and R Clint Whaley. Design and implementation of the scalapack lu, qr, and cholesky factorization routines. *Scientific Programming*, 5(3):173–184, 1996.

[2] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[3] Nicholas J Higham. *Analysis of the Cholesky decomposition of a semi-definite matrix*. Oxford University Press, 1990.

[4] Nicholas J Higham. Cholesky factorization. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(2):251–254, 2009.