Métodos numéricos - Tarea 8 Método de la Jacobi Cómputo paralelo

Salim Vargas Hernández

7 de octubre del 2018

1. Introducción

El método de Jacobi para el cálculo de valores y vectores propios es un método iterativo que permite aproximar la diagonalización de la matriz A simétrica, es decir, aproximar la forma $P^TAP = \Lambda$ donde Λ es una matriz diagonal que contiene en la diagonal a los eigenvalores de A y P es una matriz ortogonal con los vectores propios de A.

La paralelización es un procedimiento que permite aprovechar los cores de una computadora multicore, de forma que una operación o cálculo que se realiza normalmente en serie, se puede realizar a trozos en cada uno de los cores de la máquina, para depués colapsar el resultado en uno solo. La ventaja de la paralelización es que, para problemas muy grandes, permite realizar estas operaciones en un menor tiempo que el que tomaría hacerlas en serie.

2. Desarrollo

2.1. Método de Jacobi

El método de Jacobi consiste en aplicar rotaciones estratégicas en ciertos puntos de la matriz simétrica *A* para diagonalizarla. El algoritmo 1 muestra el pseudocódigo del método de Jacobi.

3. Resultados

3.1. Método de Jacobi

Se probó el método de Jacobi con las matrices mat5a.txt, mat5b.txt y mat10.txt. Para lograr la convergencia se utilizaron los siguientes parámetros:

Parámetro	mat5a.txt	mat5b.txt	mat10.txt
tolerancia	0.0001	0.0001	0.001
maxIter	1,000,000	1,000,000	1,000,000

Para cada matriz se imprimían los eigenvalores obtenidos. Los resultados se muestran en la figura 1.

```
Algoritmo 1 Método de Jacobi para valores y vectores propios
```

Entrada:

```
Matriz simétrica A tolerancia tol máximo de iteraciones maxIter
```

Salida: Matriz diagonal *C* con los valores propios

Matriz ortogonal P con los vectores propios

```
1: C = A
2: P = I_N
3: Mientras haya valores arriba de la tolerancia fuera de la diagonal y las iteraciones sean menores que maxlter hacer
       max = máximo en valor absoluto de C
       [k, l] = posición de max en C
 5:
       Si |max| > tol entonces
6:
          Si C_{kk} == C_{ll} entonces
 7:
 8:
           Si no
 9:
             \theta = \arctan\left(\frac{2C_{kl}}{C_{kk} - C_{ll}}\right)
10:
           Fin Si
11:
          P^{(i)} = I_N
12:
           P_{kk}^{(i)} = cos(\theta)
13:
           P_{II}^{(i)} = cos(\theta)
14:
           P_{kl}^{(i)} = -\sin(\theta)
15:
           P_{lk}^{(i)} = sin(\theta)
16:
           C = P^{(i)T} CP^{(i)}
17:
           P = PP^{(i)}
18:
       Fin Si
19:
20: Fin Mientras
21: Devolver C, P
```

```
Nombre del archivo: mat5a.txt Nombre del archivo: mat5b.txt Tolerancia: 0.0001 Tolerancia: 0.0001 Tolerancia: 0.0001 Máximo de iteraciones: 10000000 Máximo de iteraciones: 10000000 Eigenvalores

1 5 3 2 4 Eigenvalores

9 7 10 6 8 Figure 10 6 8 Figure 10 6 8 -8 2 -4 8 -6 -10 -2
```

Figura 1: Resultados del método de Jacobi

3.2. Cómputo paralelo

Se hizo uso de OpenMP para paralelizar algunas operaciones con vectores y matrices, como fueron:

- 1. Suma de vectores
- 2. Producto de vectores elemento a elemento
- 3. Producto punto entre dos vectores
- 4. Producto matriz-vector
- 5. Producto matriz-matriz

Se tomó en cuenta el tiempo de ejecución de las funciones paralelas y el método en serie, así como el speed-up y la eficiencia

Se generaban vectores y matrices aleatorias de tamaño especificado por el usuario; asimismo se pedía al usuario el número de *threads* a utilizar en la paralelización.

Los resultados se muestran en las figuras 2 a la 6.

```
Operación a realizar: 1
Dimensión de los vectores: 100000000
Número de threads: 1
        Tiempo de ejecución en paralelo = 2.69585 segundos
        Tiempo de ejecución en serie = 1.59681 segundos
        Speed-up = 0.592322
        Eficiencia = 59.2322\%
Operación a realizar: 1
Dimensión de los vectores: 100000000
Número de threads: 2
        Tiempo de ejecución en paralelo = 7.61055 segundos
        Tiempo de ejecución en serie = 1.35906 segundos
        Speed-up = 0.178576
        Eficiencia = 8.9288%
Operación a realizar: 1
Dimensión de los vectores: 100000000
Número de threads: 4
       Tiempo de ejecución en paralelo = 11.4487 segundos
       Tiempo de ejecución en serie = 1.34734 segundos
       Speed-up = 0.117685
        Eficiencia = 2.94213%
```

Figura 2: Resultados de la operación 1

```
Operación a realizar: 2
Dimensión de los vectores: 100000000
Número de threads: 1
        Tiempo de ejecución en paralelo = 2.5835 segundos
       Tiempo de ejecución en serie = 1.3399 segundos
        Speed-up = 0.518637
        Eficiencia = 51.8637%
Operación a realizar: 2
Dimensión de los vectores: 100000000
Número de threads: 2
        Tiempo de ejecución en paralelo = 10.0701 segundos
        Tiempo de ejecución en serie = 1.32713 segundos
        Speed-up = 0.131788
        Eficiencia = 6.58941\%
Operación a realizar: 2
Dimensión de los vectores: 100000000
Número de threads: 4
       Tiempo de ejecución en paralelo = 11.5287 segundos
       Tiempo de ejecución en serie = 1.32898 segundos
       Speed-up = 0.115275
       Eficiencia = 2.88188%
```

Figura 3: Resultados de la operación 2

```
Operación a realizar: 3
Dimensión de los vectores: 100000000
Número de threads: 1
       Tiempo de ejecución en paralelo = 1.98251 segundos
       Tiempo de ejecución en serie = 0.655635 segundos
       Speed-up = 0.33071
        Eficiencia = 33.071%
Operación a realizar: 3
Dimensión de los vectores: 100000000
Número de threads: 2
        Tiempo de ejecución en paralelo = 8.25783 segundos
       Tiempo de ejecución en serie = 0.666994 segundos
        Speed-up = 0.0807711
       Eficiencia = 4.03855%
Operación a realizar: 3
Dimensión de los vectores: 100000000
Número de threads: 4
        Tiempo de ejecución en paralelo = 12.3339 segundos
       Tiempo de ejecución en serie = 0.666639 segundos
       Speed-up = 0.0540495
        Eficiencia = 1.35124%
```

Figura 4: Resultados de la operación 3

```
Operación a realizar: 4
Dimensión 1 de la matriz: 10000
Dimensión 2 de la matriz: 10000
Número de threads: 1
        Tiempo de ejecución en paralelo = 2.03109 segundos
        Tiempo de ejecución en serie = 0.645681 segundos
        Speed-up = 0.317899
        Eficiencia = 31.7899%
Operación a realizar: 4
Dimensión 1 de la matriz: 10000
Dimensión 2 de la matriz: 10000
Número de threads: 2
        Tiempo de ejecución en paralelo = 2.25068 segundos
        Tiempo de ejecución en serie = 0.680713 segundos
        Speed-up = 0.302448
        Eficiencia = 15.1224%
Operación a realizar: 4
Dimensión 1 de la matriz: 10000
Dimensión 2 de la matriz: 10000
Número de threads: 4
        Tiempo de ejecución en paralelo = 2.7608 segundos
        Tiempo de ejecución en serie = 0.649015 segundos
        Speed-up = 0.235082
        Eficiencia = 5.87706%
```

Figura 5: Resultados de la operación 4

```
Operación a realizar: 5
Dimensión 1 de la matriz 1: 1000
Dimensión 2 de la matriz 1: 1000
Dimensión 2 de la matriz 2: 1000
Número de threads: 1
         Tiempo de ejecución en paralelo = 51.79 segundos
         Tiempo de ejecución en serie = 21.0065 segundos
         Speed-up = 0.405609
         Eficiencia = 40.5609%
Operación a realizar: 5
Dimensión 1 de la matriz 1: 1000
Dimensión 2 de la matriz 1: 1000
Dimensión 2 de la matriz 2: 1000
Número de threads: 2
         Tiempo de ejecución en paralelo = 51.6593 segundos
Tiempo de ejecución en serie = 21.632 segundos
         Speed-up = 0.418744
         Eficiencia = 20.9372%
Operación a realizar: 5
Dimensión 1 de la matriz 1: 1000
Dimensión 2 de la matriz 1: 1000
Dimensión 2 de la matriz 2: 1000
Número de threads: 4
         Tiempo de ejecución en paralelo = 51.8278 segundos
         Tiempo de ejecución en serie = 20.9003 segundos
         Speed-up = 0.403264
         Eficiencia = 10.0816%
```

Figura 6: Resultados de la operación 5