Entrega 2

Andrés Milla - 14934/6
Jara, Rodrigo - 14854/8

Enunciado

Dada la siguiente expresión:

$$D = d.ABC$$

$$d = \frac{maxA \cdot maxB \cdot maxC - minA \cdot minB \cdot minC}{avgA \cdot avgB \cdot avgC}$$

donde:

- **A,B,C y D** son matrices **NxN.**
- **minA y maxA** son el mínimo y el máximo valor de los elementos de la matriz **A**, respectivamente; ídem para las matrices **B y C.**
- **AvgA** es el valor promedio de los elementos de la matriz **A**; ídem para **B y C.**

Algoritmo secuencial

Primero se calcula multiplicación entre matrices ABC aprovechando el recorrido se busca el mínimo, máximo y se sumariza sus elementos para luego calcular el promedio. Luego, se calcula d con todo los valores obtenidos.
Y finalmente se realiza la multiplicación escalar por la matriz resultante ABC, para obtener la matriz final D.

Pseudocódigo

```
1º Calcular AB = A.B // mínimo, máximo y suma total de A y B

for i=1..N (Ver sección de optimización)
      for j=1..N
            // Calcular Mínimo, Máximo y Suma total B
            // Calcular Mínimo, Máximo y Suma total de A.
            for k=1..N
                  // Calcular multiplicación AB = A.B

2º Calcular ABC = AB.C // mínimo, máximo y suma total de C

for i=1..N
      for j=1..N
            // Calcular Mínimo, Máximo y Suma total C
            for k=1..N
                  // Calcular multiplicación ABC = AB.C

3º Calcular d

4º Calcular D = d.ABC
for i=1..N
      for j=1..N
            // Calcular multiplicación D = d.ABC
```

Decisiones de optimización

Para aprovechar la localidad de la caché se organizó por filas a las matrices A, AB y D, y se organizó por columnas a las matrices B y C.

Para aprovechar el recorrido de la multiplicación se calcula el máximo, mínimo y total de las matrices A, B, C en la multiplicación $AB = A.B$ y luego en $ABC = AB.C$ respectivamente.

Para no realizar conversiones de datos innecesarias en el cálculo de d se declararon como doubles todos los valores que componen la ecuación.

Se decidió no utilizar el *for* para calcular la segunda multiplicación $AB.C$ para aprovecha la localidad de la caché, ya que en el supuesto caso que se calcule primero una fila de AB y luego una fila de ABC se podrían pisar datos ya alojados de la matriz A y B por datos de la matriz B lo que provocaría un acceso salteado a memoria en las siguientes iteraciones. Además, en las primeras iteraciones no aprovecharía la localidad ya que solo tendría un valor o muy pocos de la matriz intermedia AB para utilizar.

Tiempos

N	Tiempo en segundos
512	2.189936
1024	17.772299
2048	142.090770
4096	1144.890040

Algoritmos paralelos

Pthread

El algoritmo paralelo consiste en dividir las iteraciones en partes iguales para resolver la ecuación en un menor tiempo. Además, se aprovecha que cada thread no deba esperar el cálculo total de la primera multiplicación $A.B$ ya que la misma parte de la cual está obteniendo resultado será la que utilice en la segunda multiplicación $AB.C$.

Para ello, cada thread tendrá su *strip* (subconjunto de la matriz original) de un tamaño de N/T, siendo N la dimensión de la matriz y T la cantidad de threads disponibles.

Primero cada thread realizará el cálculo $AB = A.B$ en el *strip* dado y también determinará los máximos, mínimos y suma total de A y B en **forma local**.

Luego, al tener el *strip* de AB resuelto, cada thread realizará el cálculo de ABC en el *strip* dado junto con los máximos, mínimos y la suma total de C en **forma local**.

Una vez terminado esto, cada thread deberá comunicar sus mínimos, máximos y total de A, B y C. Para realizar esta comunicación evaluamos la granularidad de los locks, con 9 locks obtendríamos mayor grado de paralelismo pero mucho costo de recursos. Con 1 lock para comunicar las 9 variables tendríamos un menor grado de paralelismo y menor gasto de recursos. Por lo tanto elegimos utilizar 3 locks, cada uno asociado a la matriz correspondiente. De esta forma se logra el balance entre grado de paralelismo y utilización de recursos.

Al finalizar cada uno esperará en un barrera a que el resto termine y que el cálculo de d no sea erróneo.

Después, el thread con $id = 0$ será encargado de realizar el cálculo de d mientras los demás esperan en una barrera.

Finalmente, cuando el thread con $id = 0$ termine el cálculo de d, todos los threads realizarán su *strip* correspondiente para multiplicar $D = d.ABC$.

Pseudocódigo

```
:: producto(tid)
// Realizar strip AB=A.B y max/min/suma de A y B
```

```
// Realizar strip ABC=AB.C y max/min/suma de C
// Comunicar max/min/suma de A, B y C
barrier(T);
if (tid == 0) → calcular d
barrier(T);
// Realizar strip D=d.ABC

:: main()
threads[T];

for i=1..T
      threads[i].start(producto);

for i=1..T
      threads[i].join();
```

Tiempos

2 threads

N	Tiempo secuencial	Tiempo paralelo utilizando Pthreads	SpeedUp	Eficiencia
512	2.189936	1.101275	1.9885460035	0.99427300175
1024	17.772299	8.934931	1.9890807215	0.99454036075
2048	142.090770	71.399487	1.9900811052	0.9950405526
4096	1144.890040	574.638446	1.99236589193	0.99618294596

4 threads

N	Tiempo secuencial	Tiempo paralelo utilizando Pthreads	SpeedUp	Eficiencia
512	2.189936	0.551348	3.97196688843	0.9929917221
1024	17.772299	4.502103	3.94755495376	0.98688873844
2048	142.090770	35.937157	3.95386785883	0.9884669647
4096	1144.890040	288.757224	3.96488795723	0.9912219893

8 threads

N	Tiempo secuencial	Tiempo paralelo utilizando Pthreads	SpeedUp	Eficiencia
512	2.189936	0.275790	7.94059247979	0.99257405997
1024	17.772299	2.243278	7.92246836995	0.99030854624
2048	142.090770	18.238183	7.79084023885	0.97385502985
4096	1144.890040	144.919299	7.90019029833	0.98752378729

OpenMP

Se utiliza el mismo método que en Pthread, es decir, el algoritmo paralelo consiste en dividir las iteraciones en partes iguales para resolver la ecuación en un menor tiempo y se aprovecha la división de las matrices en *strips* para que los threads no esperen que la primera multiplicación $A.B$ se termine completa para realizar la siguiente.

Primero se utiliza el constructor *for* con schedule *static sin chunk* especificado para que se repartan las misma cantidad de iteraciones a los mismos threads para calcular la multiplicación $AB = A.B$. Se aprovecha este *for*, para realizar las reducciones de suma, máximo y mínimo de las matrices A y B, utilizando la cláusula *reduction*. También se especifica la cláusula *nowait* para que los threads puedan continuar luego de haber terminado de resolver esta parte.

Luego, se utiliza el constructor *for* con schedule s*tatic sin chunk* especificado para que se repartan las misma cantidad de iteraciones a los mismos threads para calcular la multiplicación $AB.C$. También se aprovecha este *for* para realizar la reducción de suma, máximo y mínimo de la matriz C, utilizando la cláusula *reduction*. En este caso no se especifica la cláusula *nowait*, ya que necesitamos la barrera implícita que proporciona el constructor *for* para luego calcular d.

Una vez que todos los threads finalizaron de realizar $AB.C$ y calcularon sus máximos, mínimos y totales, se opta por el constructor *single* para que sólo 1 thread resuelva la ecuación de d. Los demás threads quedarán esperando hasta que se termine de calcular d, ya que *single* tiene una barrera implícita.

Finalmente, se utiliza el constructor *for* con schedule *static sin chunk* especificado para que se repartan las misma cantidad de iteraciones a los mismos threads para calcular la multiplicación $D = d.ABC$.

Pseudocódigo

```
:: main()
// omp parallel {
      // omp for static nowait con reducción de suma, max, min A y B
      AB=A.B y max/min/suma de A y B

      // omp for static con reducción de suma, max, min C
      ABC=AB.C y max/min/suma de C
```

```
        // omp single
        Calcular d

        // omp for static
        Calcular D=d.ABC
}
```

Tiempos

2 threads

N	Tiempo secuencial	Tiempo paralelo utilizando OpenMP	SpeedUp	Eficiencia
512	2.189936	1.104932	1.98196450098	0.99098225049
1024	17.772299	8.973084	1.98062327289	0.99031163644
2048	142.090770	71.794761	1.9791244935	0.98956224675
4096	1144.890040	576.863470	1.98468112394	0.99234056197

4 threads

N	Tiempo secuencial	Tiempo paralelo utilizando OpenMP	SpeedUp	Eficiencia
512	2.189936	0.552917	3.96069572829	0.99017393207
1024	17.772299	4.520739	3.93128181034	0.98282045258
2048	142.090770	36.010075	3.94586154014	0.98646538503
4096	1144.890040	289.940922	3.94870110815	0.98717527703

8 threads

N	Tiempo secuencial	Tiempo paralelo utilizando OpenMP	SpeedUp	Eficiencia
512	2.189936	0.276766	7.91259041934	0.98907380241
1024	17.772299	2.253379	7.88695510165	0.9858693877
2048	142.090770	18.036760	7.87784335989	0.98473041998
4096	1144.890040	146.033583	7.83990926252	0.97998865781

Comparación

Tiempos

Cantidad de threads = 2							
		OpenMP			Pthreads		
N	Tiempo secuencial	Tiempo paralelo	SpeedUp	Eficiencia	Tiempo paralelo	SpeedUp	Eficiencia
512	2.189936	1.104932	1.98196450098	0.99098225049	1.101275	1.9885460035	0.99427300175
1024	17.772299	8.973084	1.98062327289	0.99031163644	8.934931	1.9890807215	0.99454036075
2048	142.090770	71.794761	1.9791244935	0.98956224675	71.399487	1.9900811052	0.9950405526
4096	1144.890040	576.863470	1.98468112394	0.99234056197	574.638446	1.9923658919	0.99618294596

Cantidad de threads = 4							
		OpenMP			Pthreads		
N	Tiempo secuencial	Tiempo paralelo	SpeedUp	Eficiencia	Tiempo paralelo	SpeedUp	Eficiencia
512	2.189936	0.552917	3.96069572829	0.99017393207	0.551348	3.9719668884	0.9929917221
1024	17.772299	4.520739	3.93128181034	0.98282045258	4.502103	3.9475549537	0.98688873844
2048	142.090770	36.010075	3.94586154014	0.98646538503	35.937157	3.9538678588	0.9884669647
4096	1144.890040	289.940922	3.94870110815	0.98717527703	288.757224	3.9648879572	0.9912219893

Cantidad de threads = 8							
		OpenMP			Pthreads		
N	Tiempo secuencial	Tiempo paralelo	SpeedUp	Eficiencia	Tiempo paralelo	SpeedUp	Eficiencia
512	2.189936	0.276766	7.91259041934	0.98907380241	0.275790	7.9405924797	0.99257405997
1024	17.772299	2.253379	7.88695510165	0.9858693877	2.243278	7.9224683699	0.99030854624
2048	142.090770	18.036760	7.87784335989	0.98473041998	18.238183	7.7908402388	0.97385502985
4096	1144.890040	146.033583	7.83990926252	0.97998865781	144.919299	7.9001902983	0.98752378729

Conclusión

El algoritmo paralelo tarda menos comparado con el secuencial, ya que cada thread realiza N/T pasos paralelamente. Lo cual consigue un *speedup* de aproximadamente T en todos los casos probados anteriormente. Por lo que se podría decir que el algoritmo paralelo es aproximadamente T veces mejor que el algoritmo secuencial, es decir, se logra aproximadamente un *speedup* óptimo.

En cuanto a la *eficiencia* de los algoritmos paralelos, en promedio su *eficiencia* fue:

Pthreads → 0.9903223082458333 ≃ 0.9903
OpenMP → 0.9874578341883332 ≃ 0.9875

Esto muestra que los algoritmos paralelos aprovechan casi al máximo el uso de los threads, ya que la *eficiencia* es casi 1, lo que indica que el tiempo en el cual los mismos están ociosos es casi nulo.

En cuanto a la diferencia entre Pthreads y openMP, se puede ver que es casi nula en cuanto *eficiencia* en la ejecución, esto se debe a que openMP utiliza la librería Pthreads de fondo. La ventaja de openMP está en los constructores y cláusulas que brinda, las cuales permiten un mayor nivel de abstracción que ahorra la declaración explícita de muchas variables de sincronización y variables para el cálculo de los valores mínimo, máximo y total de cada matriz.