Entregable 1 - Bonifacio Augusto 751/1, Gauna Julián

Ejercicio 1

Resolver con Pthreads y OpenMP la siguiente expresión:

$$R=AA$$

Donde A es una matriz de NxN. Analizar el producto AA y utilizar la estrategia que proporcione el mejor tiempo de ejecución.

Interpretación:

La multiplicación de matrices puede ser representado mediante un algoritmo que tome cada fila de la primera Matriz A para multiplicar y sumarlo con cada fila de la segunda matriz A respetando el siguiente esquema:

$$C = A \cdot B = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1l} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{ml} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1m}b_{m1} & \cdots & a_{11}b_{1l} + \cdots a_{1m}b_{ml} \\ \vdots & \ddots & \vdots \\ a_{n1}b_{11} + \cdots + a_{nm}b_{m1} & \cdots & a_{n1}b_{1l} + \cdots + a_{nm}b_{ml} \end{pmatrix}$$

Respetando la fórmula anterior, podemos llegar a la conclusión que este algoritmo puede paralelizarse. Esta paralelización se desarrollará en la solución del problema.

Solución:

Para resolver el problema partimos de una solución algorítmica secuencial que puede tener la siguiente forma para realizar la operación $A_1 A_2$ (ambas matrices cuadradas):

```
Inicializar las matrices con valores iniciales.
Por cada Fila “f“de A1
     Por cada columna “c” de A2
          R(c,f) = Σ_{i=0}^{N} c_i × f_i
```

Una forma de optimizar el cálculo de una multiplicación de matrices AxB es ordenar la matriz B por columnas, es decir, calculando su transpuesta. Esto aumenta la tasa de aciertos de caché porque accede más veces a memoria contigua. Si se ordena el segundo argumento de manera diferente la multiplicación también debe hacerse de otra manera: el código que calculaba la suma de los productos de una fila con una columna ahora calculará la suma de los productos de una fila con otra fila. Esta estrategia se utilizó con todas las multiplicaciones de matrices cuadradas que aparecen en el trabajo, tanto de manera secuencial como paralela.

```
sum +=  A[i*N+k] * B[j*N+k]; // con transpuesta
sum +=  A[i*N+k] * B[k*N+j]; // sin transpuesta
```

En el código se inicia todos los elementos de la matriz A con un “1”, lo que nos da como resultado una matriz en la que todos sus elementos son iguales (en este caso, N). Esto facilita la verificación del resultado, ya que si todos los elementos de la matriz son iguales, entonces el código está bien implementado.

Otro punto a tener en cuenta es que se va a monitorear el tiempo que toma la ejecución del programa. Esto se realiza con la función dwalltime(), la cual nos devuelve el tiempo en segundos actual que el reloj del sistema posee. Esto nos permite guardar en una variable el tiempo al iniciar el programa, y, una vez finalizado, obtener el tiempo que tomo la ejecución del mismo mediante la diferencia del tiempo actual y el tiempo inicial guardado.

El tiempo de ejecución de este programa, dependiendo del tamaño de la matriz, se resume en el siguiente cuadro que compara el tiempo que toma la ejecución del programa aplicando la transpuesta a la matriz y sin aplicarla:

Tamaño de Matriz	Tiempo (seg) con transpuesta	Tiempo (seg) sin transpuesta
512x512	0,394625	0,426825
1024x1024	3,182928	16,514881
2048x2048	25,170461	157.322624

Queda en evidencia así, cómo mejora el tiempo de ejecución del programa aplicando la transpuesta de la matriz con respecto a la multiplicación ordinaria a medida que el tamañnodel problema crece.

Teniendo estos resultados, ahora se mejorará la performance del programa paralelizando el código en el que se utilizó la matriz transpuesta. La primera herramienta a utilizar para ello, es la API POSIX Threads (Pthreads) modificando el algoritmo secuencial y utilizando las funciones que la API nos facilita.

El algoritmo modificado es el siguiente:

```
Inicializar las matrices con valores iniciales.
Crear los hilos.
Por cada Hilo “h” del programa
```

```
Aplicar transpuesta de A1 y  guardarlo en A2
Por cada Fila “f“ de A1 desde fxh hasta (f+1)xh
      Por cada columna “c” de A1
```

$$R(c,f) = \sum_{i=0}^{N} A1(f,i) \times A2(c,i)$$

Teniendo en claro el funcionamiento de Pthreads, una vez creado cada hilo, cada uno de ellos tomará una cantidad de filas de A1 igual al tamaño de una fila sobre la cantidad de hilos que tiene el programa. Así, por ejemplo, para multiplicar dos matrices de 512x512 con 2 hilos, el primer hilo tomar desde la fila 1 hasta la número 256 y el segundo hilo tomará desde la fila 257 hasta la número 512. Luego, cada hilo tomará cada una de sus filas para multiplicar y sumar con cada elemento de cada columna de A2 guardando el resultado correspondiente en la matriz C.

Una de las desventajas que se pueden mencionar de esta solución, es que es muy probable que dos hilos quieran acceder al mismo elemento de A2. Para ejemplificar, y siguiendo con el ejemplo anterior, puede suceder que la fila 1 (hilo número 1) y la fila 257 (hilo número 2) requieran en acceso del primer elemento de la primera fila de A2, y, ya que el acceso a la matriz es compartido por todos los hilos, uno de los dos hilos deberá esperar a que el otro acceda a dicha dirección, demorando su acceso. El punto favorable es que a la dirección que quieren acceder estos dos hilos no la modificarán, por lo que los hilos solo tendran una demora y no una incoherencia de datos.

Aún así, se evidencia la mejora del programa en el siguiente cuadro en cuanto al tiempo de ejecución:

Matriz/Hilos	2	4
512x512	0,207165	0,113704
1024x1024	1,674356	0,847455
2048x2048	12,979342	8,035707

Sabiendo que podemos comparar estos resultados con los del programa secuencial mediante la *mejora o speedup* (Tiempo secuencial/Tiempo Paralelo):

Matriz/Hilos	2	4
512x512	1,904882582	3,470634278
1024x1024	1,963600931	3,879577087

2048x2048	1,997758669	3,226796721

Existe otra métrica, además de la mejora que nos puede dar una idea de qué magnitud del programa está siendo paralelizado, que es la *eficiencia* (Mejora del Programa/ Cantos hilos o procesos se utilizaron). Esto pone en evidencia cuánto mejora la performance del programa a medida de que se suman hilos al mismo, dejando en claro la cantidad de hilos pueden utilizarse hasta que la mejora deje de ser lineal, es decir, hasta que no sea necesaria la inclusión de más hilos.

Matriz/Hilos	2	4
512x512	0,952441291	0,86765857
1024x1024	0,981800465	0,96989427
2048x2048	0,998879335	0,80669918

Como se puede apreciar en los resultados, resulta que a medida de que se suma la cantidad de procesadores, la tasa de crecimiento de la performance disminuye. Esto se debe a que el programa posee una porción en la que se realiza tareas secuenciales (creación de hilos, inicialización, demás tareas que se deben hacer en forma secuencial) que tiende a ser constante a medida que crece la cantidad de procesadores o hasta puede llegar a crecer (la creación de hilos es una tarea secuencial que crece a medida que se suman hilos).

Otra de las herramientas que utilizaremos es otra API llamada OpenMP. La misma, a diferencia de pthreads, solo se debe indicar con la cantidad de hilos que se van a utilizar, y mediante directivas del compilador (*#pragma parallel* por ejemplo), se concede el trabajo de mapear las tareas a los hilos al compilador.

El algoritmo que describe la paralelización de este problema es el siguiente:

```
Inicializar las matrices con valores iniciales.
Realizar la transpuesta de la Matriz A2
Configurar cantidad de hilos a utilizar.
Realizar en forma paralela
Por cada Fila “f“de A1
     Por cada columna “c” de A2
```

$$R(c,f) = \sum_{i=0}^{N} A1(f,i) \times A2(c,i)$$

Para realizar los loops en forma paralela se debe incluir la directiva *#pragma omp parallel for*. Esto indica al compilador que se realizará un loop en el que cada iteración del mismo será ejecutado por uno de los hilos disponibles. Al agregar a la directiva *private(i,j,k,tmp,sum),* se le indica al compilador que las variables i, j, k, tmp y sum serán variables privadas de cada hilo (no serán iguales a las de otro hilo y tampoco otro hilo podrán acceder a ellas).

En teoría, este algoritmo tendría el mismo problema que se mencionó con la solución en Pthreads (probablemente dos hilos intentarán acceder a la misma posición de memoria). Aún así, se puede ver el tiempo que toma la ejecución del programa., teniendo en cuenta el tamaño del mismo y la cantidad de hilos a utilizar, en el siguiente cuadro:

Matriz/Hilos	2	4
512x512	0,209003	0,118918
1024x1024	1,633963	0,860968
2048x2048	13,187110	7,009767

Además, podemos observar la mejora del programa en el siguiente cuadro:

Matriz/Hilos	2	4
512x512	1,065865083	1,873299248
1024x1024	1,947980462	3,696917888
2048x2048	1,908716997	3,590769993

El rendimiento del programa en cuanto a la eficiencia es el siguiente:

Matriz/Hilos	2	4
512x512	0,532932542	0,468324812
1024x1024	0,973990231	0,924229472
2048x2048	0,954358499	0,897692498

Se evidencia nuevamente que la tasa de crecimiento de la mejora va disminuyendo a medida que se aumenta la cantidad de procesadores y esto tiene la misma explicación que en el ejemplo anterior. Aún así, los resultados arrojados por el programa en openmp evidencian una mejora.

Ejercicio 2:

Realizar un algoritmo Pthreads y otro OpenMP que resuelva la expresión:

$$M = \overline{u.l}.AAC + \bar{b}LBE + \bar{b}DUF$$

Donde *A, B, C, D, E* y *F* son matrices de NxN. *L* y *U* son matrices triangulares de NxN inferior y superior, respectivamente. $\bar{b}$ es el promedio de los valores de los elementos de la matriz B y $\overline{u.l}$ es el producto de los promedios de los valores de los elementos de las matrices U y L, respectivamente. Evaluar N=512, 1024 y 2048.

Interpretación:

Habiendo creado un algoritmo para la multiplicación de matrices parte del problema está prácticamente resuelto. Por ende, resta buscar la forma de multiplicar una matriz por un escalar, que en este caso será promedio de una matriz, y hacer la suma entre dos matrices (o tres). Tanto la suma de matrices como la multiplicación de un escalar y una matriz puede entenderse en la siguientes fórmulas:

$$C = \alpha A = \alpha \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} = \begin{pmatrix} \alpha a_{11} & \cdots & \alpha a_{1m} \\ \vdots & \ddots & \vdots \\ \alpha a_{n1} & \cdots & \alpha a_{nm} \end{pmatrix}$$

$$S = A + B = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} + \begin{pmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nm} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & \cdots & a_{1m} + b_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nm} + b_{nm} \end{pmatrix}$$

Otra operación debe contemplar la solución, es la del promedio de una matriz. Este promedio lo obtenemos mediante la suma de todos los elementos de la matriz sobre la cantidad de elementos que tiene la matriz.

Buscaremos, además, cómo paralelizar todas estas operaciones para obtener una mejor performance en la ejecución del problema.

Solución:

En primer lugar, inicializamos las matrices con todos sus elementos igual a 1, a excepción de las matrices triangulares superiores e inferiores. Con estas matrices debemos buscar la forma de que, para las matrices triangulares superiores, todos los elementos de la diagonal y la parte superior de la diagonal de la matriz sean 1 y, para las matrices triangulares superiores inferiores, los elementos de la

diagonal y la parte inferior a la diagonal sea 1. En definitiva, el algoritmo que inicializa una matriz "A" triangular superior es el siguiente:

```
Por cada fila "f" de A
      Por cada columna "c" de A
            Si c es mayor o igual que f
                  A_{c,f}=1
            Sino
                  A_{c,f}=0
```

Análogamente, el algoritmo que inicializa una matriz "A" triangular inferior es el siguiente:

```
Por cada fila "f" de A
      Por cada columna "c" de A
            Si c es mayor o igual que f
                  A_{c,f}=1
            Sino
                  A_{c,f}=0
```

Una vez que se tiene en cuenta como se debe inicializar las matrices, debemos crear un algoritmo que realice la suma de dos matrices. Dicho algoritmo no resulta difícil, ya que para sumar dos matrices "A" y "B" y guardar el resultado en "C" solo deberíamos realizar los siguientes pasos:

```
Por cada fila "f" de A y B
      Por cada columna "c" de A y B
            C_{c,f} = A_{c,f} + B_{c,f}
```

Otra operación a utilizar es la multiplicación de una matriz "A" con un escalar "b" y guardar el resultado en la matriz "C", la cual podemos explicarla con el siguiente algoritmo:

```
Por cada fila "f" de A
      Por cada columna "c" de A
            C_{c,f} = A_{c,f} x b
```

Ya con las operaciones básicas implementadas, sólo resta definir el algoritmo final que define la operación final, utilizando matrices transpuestas para mejorar la performance del programa en cuanto al tiempo. La misma se puede sintetizar con las siguientes acciones:

Inicializar las matrices A, B, C, D, E, F, L y U
Obtener el promedio de B y guardarlo en $\overline{b}$
Obtener el promedio de L y guardarlo en $\overline{l}$
Obtener el promedio de U y guardarlo en $\overline{u}$
Guardar la transpuesta de A en At multiplicando cada valor de A por u_l
Guardar la multiplicación entre A y At en A_A
Guardar la transpuesta de C en Ct
Guardar la multiplicación entre A_A, y Ct en A_A_C
Guardar la multiplicación entre L, y B en L_B multiplicando cada valor de L_B por el escalar b
Guardar la transpuesta de E en Et
Guardar la multiplicación entre L_B y Et en L_B_E
Guardar la multiplicación entre U y F en U_F multiplicando cada valor de U_F por el escalar b
Guardar la transpuesta de U_F en U_Ft
Guardar la multiplicación entre D y U_Ft y guardarlo en D_U_F
Sumar A_A_C, L_B_E y D_U_F y guardarlo en M

Se realiza la multiplicación de del escalar u_l a la hora de calcular la transpuesta de A para aprovechar las iteraciones de ese algoritmo, debido que:

$$\alpha A A^T = A(\alpha A^T)$$

De la misma forma, la multiplicación del escalar b sobre el término LBE se realiza a la hora de multiplicar L y B,así no sólo se aprovechan las iteraciones sin que también se reducen las multiplicaciónes a aquellas en los cuáles los elemento de L no sean 0.

Se realizó una prueba en MATLAB para detectar un patrón sobre el resultado y se encontró que los elementos de la matriz M resultante son iguales dentro de cada fila de esta. Es decir, que

$$M_{i0} = x \Rightarrow M_{ij} = x \quad \forall j$$

Por lo que se realizó una modificación en el proceso de verificación de manera de comprobar que los elementos de cada fila de la matriz son iguales a el primer elemento de la misma fila.

Para verificar el tiempo que toma este programa secuencial, nuevamente hacemos uso de la función dwaltime() implementada en la solución del primer problema. El tiempo que toma la ejecución del programa secuencial dependiendo del tamaño de las matrices se resume en el siguiente cuadro:

Tamaño de Matriz	Tiempo (seg)
512x512	3,357519
1024x1024	70,944696
2048x2048	636,047661

Una vez creado el código secuencial. crearemos el algoritmo paralelizado para mejorar la performance del mismo. Por ello se utilizará Pthreads y se detallará que operaciones del algoritmo secuencial es paralelizable.

De esta forma, el cálculo del promedio de la matriz "A" sigue los siguientes pasos:

```
Crear los hilos.
Crear un arreglo "a" del tamaño de la cantidad de hilos
Por cada hilo "h"
      Por cada Fila "f" de A1 desde fxh hasta (f+1)xh
            Por cada columna c
                  a_h = a_h + A_c,f
Sumar todos los elementos de a y dividirlos por N
```

Además, la suma de dos matrices "A" y "B" y guardarlo en "C" se realiza de la siguiente forma:

```
Crear los hilos.
Por cada hilo "h"
      Por cada Fila "f" de A1 desde fxh hasta (f+1)xh
            Por cada columna c
                  C_c,f = A_c,f + B_c,f
```

Por último, el producto de las matrices se realiza de la misma forma que en el ejercicio 1, a excepción de las matrices triangulares que solo iteran sobre los elementos no nulos.

Matriz/Hilos	2	4
512x512	1,185051	0,886806
1024x1024	13,350790	6,778590
2048x2048	131,919315	69,346842

Con ello podemos resumir el Speedup y la Eficiencia del código en el siguiente cuadro:

	Speedup		Eficiencia	
Matriz/Hilos	2	4	2	4
512x512	2,833227431	3,786080608	1,416613715	0,946520152
1024x1024	5,313894983	10,465996026	5,232998013	2,616499006
2048x2048	4,82149	9,17198	2.41075	2.293

Para el caso de Openmp, solo basto utilizar la directiva *#pragma parallel for* por cada loop que se realizó en el programa. Como se estableció en el ejercicio 1, esto mapea los hilos para que cada uno realice cierta cantidad de iteraciones. A su vez, en el caso de calcular el promedio se puede utilizar *reduction(+: parámetro),* el cual establece de antemano una variable que debe tratarse con exclusión mutua (parámetro) para poder sumar todos los valores necesarios del arreglo y asignarlo en dicha variable sin tener conflictos de mutuo acceso.

Contemplado esto y aplicándolo al algoritmo secuencial, podemos ejecutarlo y obtener los siguientes tiempos de ejecución

Matriz/Hilos	2	4
512x512	1,494959	1,207178
1024x1024	18,512570	8,188769
2048x2048	157,241046	82,562986

Nuevamente podemos calcular los parámetros de Speedup y Eficiencia y volcarlos en el siguiente cuadro :

	Speedup		Eficiencia	
Matriz/Hilos	2	4	2	4

512x512	2,2458937	2,781295716	1,12294685	0,695323929
1024x1024	3,832244578	8,663658237	1,916122289	2,165914559
2048x2048	4,045048524	7,703787034	2,022524262	1,925946758

Ejercicio 3:

Paralelizar con OpenMP un algoritmo que cuente la cantidad de número pares en un vector de N elementos. Al finalizar, el total debe quedar en una variable llamada pares.

Evaluar con valores de N donde el algoritmo paralelo represente una mejora respecto al algoritmo secuencial.

Interpretación:

Para este problema, se lo resolverá diseñando directamente el algoritmo diferente. Es decir, establecer qué variables auxiliares utilizar, diseñar las tareas que van a realizar cada hilo.

Solución:

Para comenzar debemos establecer cuál sería el algoritmo que cuenta las cantidad de números pares de un arreglo:

```
Por cada elemento “i” de un arreglo A
      Sí A[i] % 2 = 0
            pares = pares + 1
```

Este algoritmo resulta fácil de paralelizar openmp, ya que el con las directivas *#pragma* los hilos se repartirán las iteraciones del for, encargándose cada uno de una porción del arreglo. Para optimizar tanto el código secuencial como el paralelo, se reemplaza la operación

```
A[i] % 2 == 0
```

Por

```
!(A[i]&1)
```

Dado que la primera realiza la división del número para quedarse con el resto mientras que la segunda realiza un AND lógico sobre el número A[i]. La división puede tomar más de un ciclo, en cambio el AND es 1 operación básica de la ALU y es igual de válida porque revisa si el último bit es un 1 (impar) o 0 (par).

Los tiempos de la ejecución secuencial fueron:

N	Tiempo
10000000	0.030941
100000000	0.307031
1000000000	3.160152

Los tiempos de la ejecución paralela:

Matriz/Hilos	2	4
10000000	0.016934	0.011658
100000000	0.166957	0.095733
1000000000	1.668221	0.942759

Las mejoras y eficiencias quedarían

	Speedup		Eficiencia	
Matriz/Hilos	2	4	2	4
10000000	1.82715	0,061638	0.91358	0.66352
100000000	1.83898	3.20716	0.919149	0.80179
1000000000	1.89432	3.35915	0.94716	0.83979

Para obtener una mejora en la performance respecto al algoritmo secuencial se necesitó crear un vector de 10.000.000 elementos, dado que el trabajo a realizar es muy sencillo en cuestiones de instrucciones por elemento. Entonces, para números muy menores a 10.000.000 el algoritmo paralelo lo resuelve con mayor latencia, debido al overhead de crear y posteriormente sincronizar los hilos.

Conclusión

Nos han quedado speedups y eficiencias que superan lo teórico, si bien es posible, nos resulta extraño que sea de tal magnitud, pero no pudimos detectar ningún error a la hora de realizar los algoritmos secuenciales. Además los resultados de las operaciones resultaron exitosos en todos los casos.

Aun así queda en evidencia la gran mejora que resulta de la aplicación de algoritmos paralelos sobre códigos secuenciales hasta un cierto punto que puede ser investigado.