PC4 - INFORME

Integrantes:
- Barreto Zavaleta, Jeanlee Roy
- Solano Vergara, Felix Jesus

INTRODUCCIÓN:

Multiplicar matrices es una de las cosas tediosas que hemos hecho en las escuelas. A medida que aumenten las dimensiones de una matriz, también aumentará el tiempo necesario para completar el cálculo. Incluso para las computadoras, el problema está ahí. Sin embargo, se puede utilizar la concurrencia para mejorar la performance de la computadora al multiplicar matrices y esto se logra a través de hilos.
El objetivo del presente informe es realizar una comparación entre el algoritmo concurrente y el algoritmo no concurrente para la multiplicación de matrices, analizar el rendimiento de dichos algoritmos y determinar en qué condiciones es más favorable utilizar cada uno de ellos.

MULTIPLICACIÓN DE MATRICES:

La multiplicación de matrices se realiza de la siguiente manera:
Si *A = a[i][k]* es una matriz de tamaño *2 x 3* y *B = b[k][j]* es una matriz de tamaño *3 x 2*. Entonces, *A * B* es una matriz de *C = c[i][j]* de tamaño *2 x 2*, en donde:

El primer elemento de la matriz c será:
*c[1][1] = a[1][1] * b[1][1] + a[1][2] * b[2][1] + a[1][3] * b[3][1].*
El segundo elemento de la matriz c será:
*c[2][1] = a[2][1] * b[1][1] + [2][2] * b[2][1] + a[2][3] * b[3][1].*
El tercer elemento de la matriz C será:
*c[1][2] = a[1][1] * b[1][2] + a[1][2] * b[2][2] + a[1][3] * b[3][2].*
Y el cuarto elemento de la matriz C será:
*c[2][2] = a[2][1] * b[1][2] + a[2][2] * b[2][2] + a[2][3] * b[3][2].*

Cabe recalcar que para que dos matrices se puedan multiplicar es importante que las columnas de la primera matriz sean iguales a las filas de la segunda matriz, de lo contrario no será posible realizar la multiplicación.

Además, el número de total multiplicaciones que se requiere para obtener la matriz C es: (columnas de A) * (filas de C * columnas de C). Finalmente el el número total de sumas que se requieren para obtener la matriz C es: (columnas de A - 1) * (filas de C * columnas de C).

EXPERIMENTO:
Para comparar los algoritmos, concurrente y no concurrente, se diseñaron dos clases matrices con una sobrecarga en el operador de multiplicación (*). Con la finalidad de tomar el tiempo de ejecución, lo que tarda cada algoritmo en realizar la multiplicación, se utilizó la librería *chrono*.

Multiplicación con algoritmo no concurrente:
La clase **Matriz** utiliza un algoritmo no concurrente para realizar la multiplicación matricial, a continuación se presenta la implementación de la sobrecarga de su operador (*).

```
Matriz<T> operator*(Matriz<T> other) const{
    Matriz<T> temp(filas,other.get_columnas());
    for (int i = 0; i < filas; ++i) {
        for (int j = 0; j < other.get_columnas(); ++j){
            T valor = 0;
            for (int k = 0; k < columnas; ++k) {
                valor += matriz[i][k] * other.get(k,j);
            }
            temp.set(i,j, valor);
        }
    }
    return temp;
}
```

En este caso, se crea una objeto temporal *Matriz<T> temp* a quien se le va asignando el resultado de las operaciones para la multiplicación. Cabe resaltar que estas operaciones se llevan a cabo de manera secuencial para cada elemento de la matriz resultante.
En el programa principal se realiza la prueba y se crean las tres matrices *m1, m2 y m3*; para después hacer la operación con el operador sobrecargado.

```
Matriz<int> m1(1000,1000), m2(1000,1000), m3(1000,1000);
m1.print_matriz(); m2.print_matriz();
auto start = high_resolution_clock::now();
m3 = m1*m2;
auto stop = high_resolution_clock::now();
```

Así mismo, se toma el tiempo antes y después de la operación para registrar cuánto tardó realizar esta multiplicación.

Multiplicación con algoritmo concurrente:
La clase **MatrizConcurrente** es quien utiliza el algoritmo concurrente para realizar la multiplicación de matrices. Para realizar dicha tarea se deben de emplear hilos, a quienes se les va a delegar partes de la operación para obtener la matriz resultante. A continuación se presenta la implementación de la sobrecarga de la operación.

```
MatrizConcurrente<T> operator*(MatrizConcurrente<T> other){
    MatrizConcurrente<T> matrizTemp(filas,other.get_columnas());
    std::vector<std::thread> hilos;

    for (int i = 0; i < NUM_HILOS; ++i) {
        hilos.emplace_back(&MatrizConcurrente<T>::multiplicarF, this, i,std::ref(other),std::ref(matrizTemp) );
    }
    for (int j = 0; j < NUM_HILOS; ++j) {
        hilos[j].join();
    }
    return matrizTemp;
}
```

Al igual que en el caso anterior, se crea una matriz temporal y se le van asignando los resultados. Sin embargo, en este caso se utiliza un vector de hilos quienes dependiendo de su posición se van a encargar de una parte de la matriz resultante. Cada hilo llama a la función multiplicarF, la cual le dice a cada hilo que parte de la matriz se tiene que encargar de operar. A continuación se presenta la implementación de la función multiplicarF.

```
void multiplicarF(int idHilo, MatrizConcurrente<T>& Matriz2, MatrizConcurrente<T>& Matriz3){
    int inf, sup, extra;
    extra = this->filas % NUM_HILOS;
    inf = idHilo * (this->filas / NUM_HILOS);
    sup = (idHilo+1) * (this->filas / NUM_HILOS);
    if (idHilo == NUM_HILOS - 1 && extra != 0) {
        sup += extra;
    }
    for (int i = inf; i < sup; ++i) {
        for (int j = 0; j < Matriz2.columnas; ++j){
            T valor = 0;
            for (int k = 0; k < Matriz2.filas; ++k) {
                mtx.lock();
                valor += this->matriz[i][k] * Matriz2.matriz[k][j];
                mtx.unlock();
            }
            Matriz3.matriz[i][j] = valor;
        }
    }
}
```

Se puede observar que cada hilo tiene encargado una cantidad de filas para operar, las cuales están acotadas por los límites *inf* y *sup*. En simples palabras, usando esta declaración se delega partes del bucle for para diferentes hilos. En el caso de que el número de filas no sea múltiplo del número de hilos, entonces las últimas filas sobrantes son añadidas a la operación del último hilo. De este modo se obtiene una multiplicación completa.

En el programa principal se realiza la prueba y se crean las tres matrices *m11, m21 y m31*; para después hacer la operación con el operador sobrecargado.

```
// CONCURRENTE
MatrizConcurrente<int> m11(1000,1000), m21(1000,1000), m31(1000,1000);
m11.print_matriz(); m21.print_matriz();
auto start = high_resolution_clock::now();
m31 = m11*m21;
auto stop = high_resolution_clock::now();

auto duration = duration_cast<microseconds>(stop - start);

m31.print_matriz();
cout << "Time: " << duration.count() << endl;
```

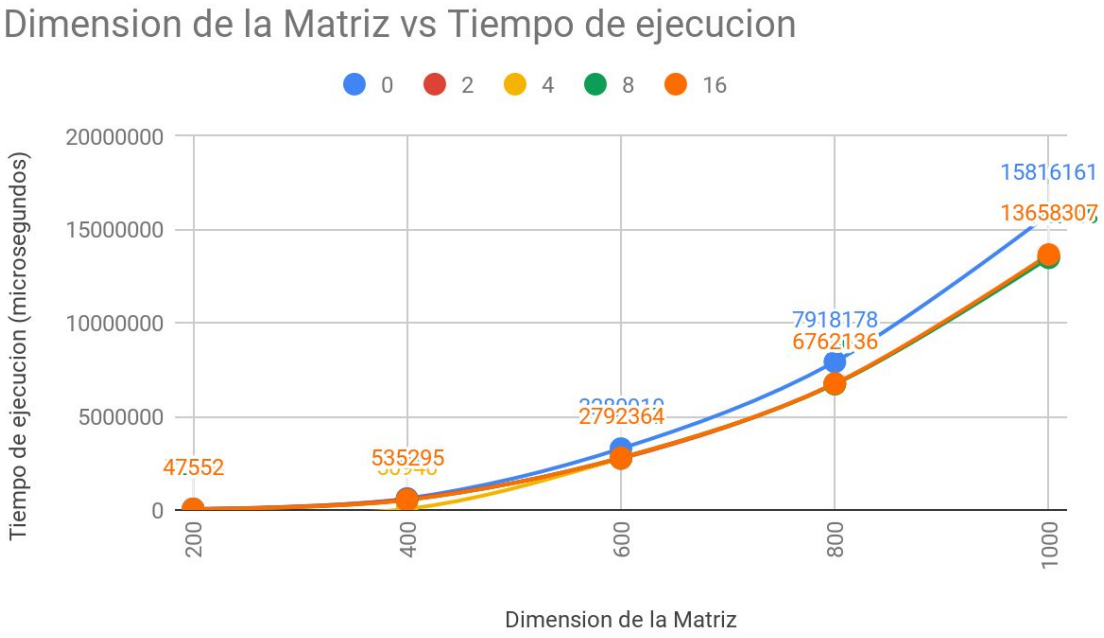
Asimismo, se toma el tiempo antes y después de la operación para registrar cuánto tardó realizar esta multiplicación.

RESULTADOS:

Luego de haber realizado las pruebas para matrices cuadradas con una dimensión de 200, 400, 600, 800 y 1000 respectivamente, se obtuvieron los resultados para cada algoritmo empleado. La siguiente tabla contiene el tiempo necesario para cada dimensión en cada enfoque y los resultados se encuentran en microsegundos.

Dimension	Sin Hilos	2 hilos	4 hilos	8 hilos	16 hilos
200	58000	43086	41620	42631	47552
400	613578	580355	50940	534613	535295
600	3280910	2746241	2767975	2790947	2792364
800	7918178	6751593	6724674	6734643	6762136
1000	15816161	13537811	13563301	13482648	13658307

Para tener una visión general de lo que está pasando, a continuación se muestran los datos en una gráfica Dimensión vs. Tiempo.

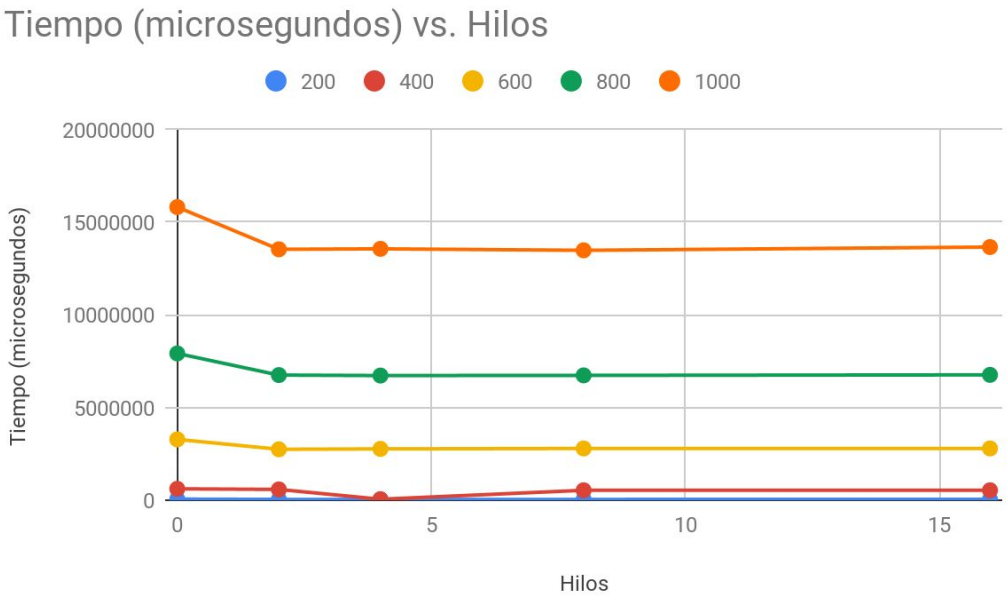


Como se puede observar en la gráfica, el uso de hilos hace que el tiempo de ejecución de la multiplicación sea menor. Esto se evidencia más a medida que la dimensión de las matrices van aumentando. Sin embargo, esta visión muy limitada ya que solo estamos trabajando con una cantidad moderada de datos y no sabemos cómo es el comportamiento de las líneas de tiempo a medida que se va aumentando la dimensión de las matrices.

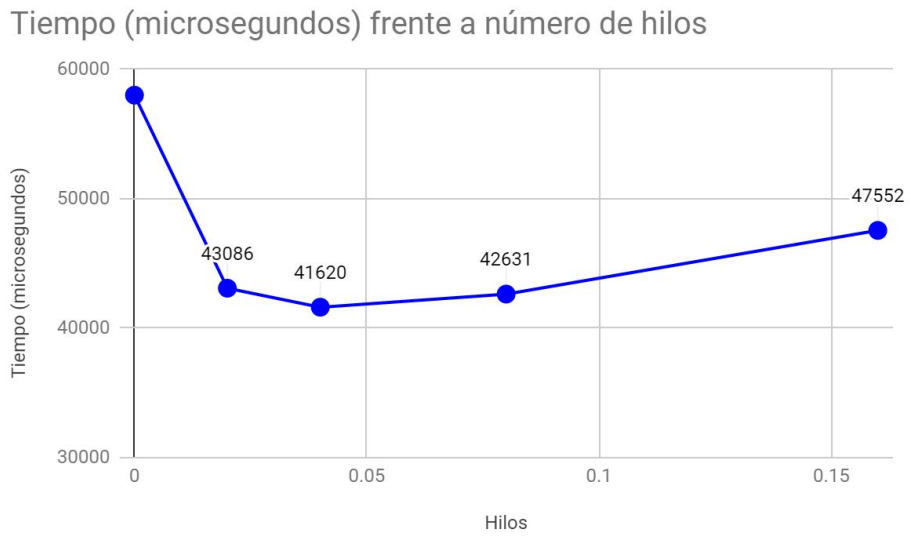
Otra enfoque que se puede obtener a partir de los datos es el tiempo que demora la multiplicación debido al número de hilos usados para realizarla.

	Dimension				
Hilos	200	400	600	800	1000
0	58000	613578	3280910	7918178	15816161
2	43086	580355	2746241	6751593	13537811
4	41620	50940	2767975	6724674	13563301
8	42631	534613	2790947	6734643	13482648
16	47552	535295	2792364	6762136	13658307

El gráfico del tiempo empleado en cada tipo matriz (según su dimensión) frente al número de hilos se muestra a continuación.



Se puede ver que la diferencia en el tiempo de ejecución entre el algoritmo concurrente y el algoritmos no concurrente es más notorio a medida que el tamaño de la matriz aumenta. Ahora vamos a enfocar el resultado para el caso de las matrices de dimensión 200x200.



Se puede observar en la gráfica que el tiempo de ejecución, desde la multiplicación que se realiza con el algoritmo no concurrente hasta la operada con cuatro hilos, va disminuyendo. Sin embargo, una vez que se aumenta el número de hilos mayores a cuatro el tiempo va aumentando. Con esta información podemos ver hasta qué punto es más favorable utilizar el algoritmo concurrente y cuando es mejor usar el no concurrente en la máquina utilizada.

En este experimento se midió el tiempo de ejecución usando solo matrices cuadradas debido a la falta de tiempo. Sin embargo, el código también funciona para medir el tiempo de ejecución de matrices rectangulares.

CONCLUSIONES:

- El uso de más hilos implica un mayor performance en el código, siempre y cuando el número de hilos empleados no supere al número de hilos que posee la máquina para el proceso.
- El tiempo de ejecución aumentó al probar con 8 y 16 hilos debido a que la laptop en la que medimos los tiempos dispone solo de 4 hilos y reutilizo sus hilos cuando le pedimos que trabaje con 8 y 16 hilos.
- El último hilo es el que hace más trabajo, esto es debido a que hemos definido los límites en el código del algoritmo para que eso suceda.
- La diferencia en el tiempo de ejecución entre el algoritmo concurrente y el algoritmos no concurrente es más notorio a medida que el tamaño de la matriz aumenta. Evidenciando una mucho mejor performance la utilización de hilos para nuestro caso, donde trabajamos con una cantidad moderada de datos.

Código completo

La implementación completa de los códigos usados en este informe están disponible en el siguiente repositorio de Github https://github.com/FJ98/PC4 o en https://github.com/utec-cs1103-2019-01/unidad-6-programacion-concurrente-JeanleeRoy