CS1103 – Programación Orientada a objetos II
PC 4 – Programación concurrente
Piero Morales
Angel Motta
Jorge Nicho

---

1. Introducción

   En el presente trabajo se desarrollan dos algoritmos para realizar operaciones de multiplicación de matrices, con el fin de mostrar los beneficios de performance de un algoritmo concurrente versus un algoritmo tradicional. Para esto se procesarán diferentes volúmenes de datos (dimensiones de matrices) para determinar cuando es más conveniente usar concurrencia y cuando un algoritmo tradicional.

2. Multiplicación de matrices con algoritmo tradicional

   Para el algoritmo tradicional sobrecargamos el operador "*" para ejecutar la multiplicación de 2 matrices de la siguiente forma.

```
Mimatriz<TipoDato> m3(4,4);
t0 = clock();
m3 = m1 * m2;
t1 = clock();
```

   El algoritmo principalmente recorre las filas de la matriz de resultante, en la segunda iteración recorre las columnas de la matriz resultante y en la tercera iteración se realiza la multiplicación de ambas matrices y se almacena el resultado en la matriz resultante. La complejidad de tiempo de este algoritmo es O (N^3)

```
Mimatriz<T> operator*(Mimatriz<T> m2) {
    Mimatriz<T> m3(this->row, this->col);
    for (int i = 0; i < this->row; i++) {
        for (int j = 0; j < this->col; j++) {
            for (int k = 0; k < this->row; k++) {
                m3.arr[i][j] += arr[i][k] * m2.arr[k][j];
            }
        }
    }
    return m3;
}
```

3. Multiplicación de matrices con algoritmo concurrente

   Para el desarrollo de un algoritmo concurrente se utilizó la implementación de pthread disponible con el compilador g++. En el desarrollo de este algoritmo definimos una variable "MAX" para definir la dimensión de matriz cuadrada 2D de esa longitud. "MAX_THREAD" para definir el numero de hilos a utilizar.

```
// Tamaño máximo de la matriz Columnas y Filas
#define MAX 4

// Máximo número de hilos
#define MAX_THREAD 4

Mimatriz<TipoDato> matA(MAX,MAX);
Mimatriz<TipoDato> matB(MAX,MAX);
Mimatriz<TipoDato> matC(MAX,MAX);
```

Definimos un array del tipo “pthread” de longitud MAX_THREAD (4 para este caso) iteramos de acuerdo con la cantidad de hilos disponibles e invocamos la ejecución de la función “multi” encargada de ejecutar la multiplicación matricial.

```
// Se declaran los 4 hilos
pthread_t threads[MAX_THREAD];

// Se crean 4 hilos, cada uno calcula 1 fila del resultado
inicio = clock();
for (int i = 0; i < MAX_THREAD; i++) {
    int* p;
    pthread_create(&threads[i], NULL, multi, (void*)(p));
}

// Espera a que todos los hilos terminen
for (int i = 0; i < MAX_THREAD; i++) {
    pthread_join(threads[i], NULL);
}
fin = clock();
```

Con el uso de la variable global step_i, se realiza la iteración del numero de filas que se encargará cada hilo (bucle exterior) de tal forma que cada hilo se encargará de ejecutar la multiplicación de la cuarta parte del total de filas a procesar.

```
int step_i = 0;

void* multi(void* arg) {
    int core = step_i++;

    // Cada hilo va a calcular 1/4 de la multiplicación
    for (int i = core * MAX / 4; i < (core + 1) * MAX / 4; i++) {
        for (int j = 0; j < MAX; j++) {
            for (int k = 0; k < MAX; k++) {
                matC.arr[i][j] += matA.arr[i][k] * matB.arr[k][j];
            }
        }
    }
}
```

```
// Espera a que todos los hilos terminen
for (int i = 0; i < MAX_THREAD; i++) {
    pthread_join(threads[i], NULL);
}
fin = clock();
```

4. Descripción del experimento

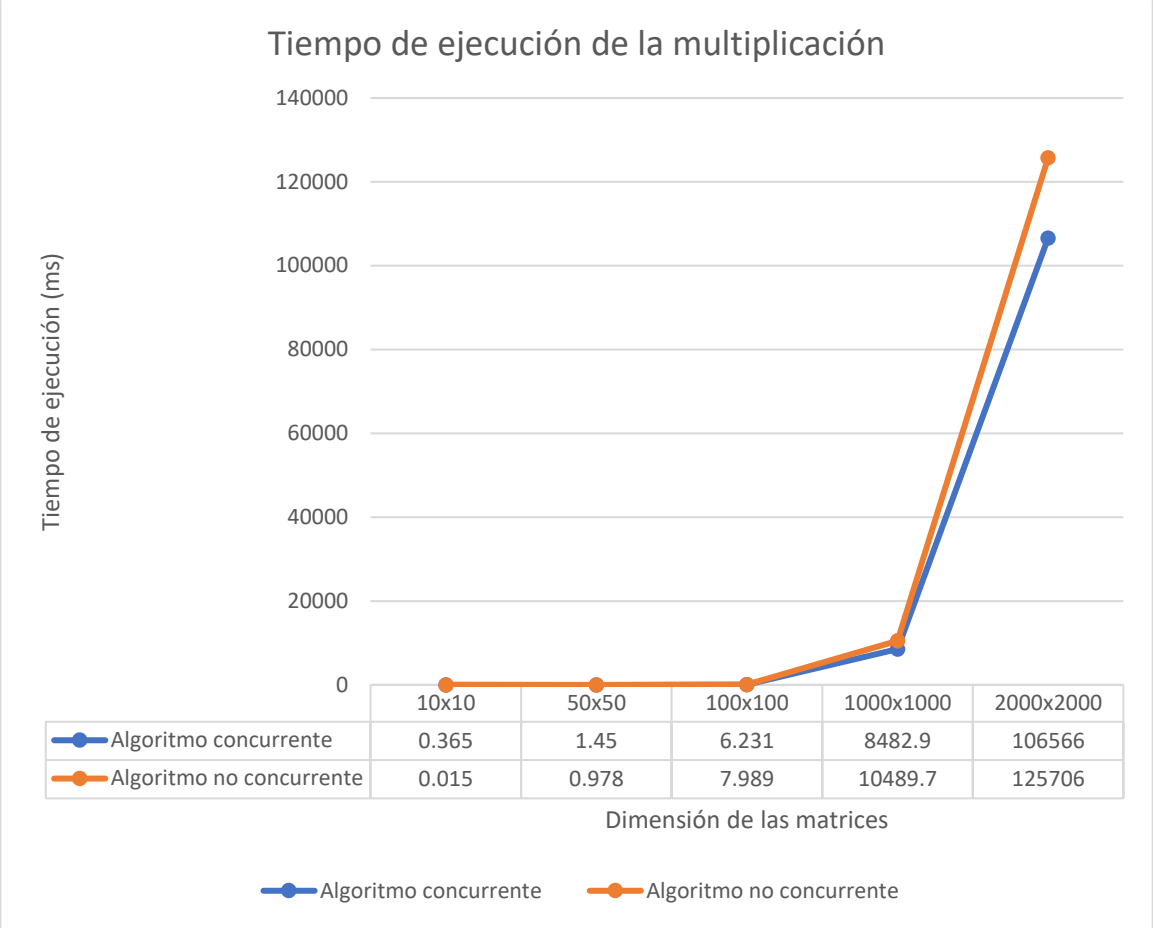
   Estos algoritmos han sido probados en un equipo con las siguientes características:

   Equipo: Acer Predator PH315-51-78NP
   Sistema Operativo: Linux Deepin 15.10
   Tipo de S.O: 64 bits
   Procesador: i7-8750H
   Memoria RAM: 16 GB
   Disco: 1 TB
   Lenguaje de Programación: C++
   Compilador: g++ 4.2.1

5. Resultados

   Se ejecutó la multiplicación de matrices de acuerdo con las dimensiones indicadas obteniendo lo siguientes tiempos de ejecución para cada tipo de algoritmo.

Dimensiones de matriz 2D	Tiempo de ejecución Algoritmo Concurrente	Tiempo de ejecución Algoritmo No Concurrente
10x10	0.365 ms	0.015 ms
50x50	1.45	0.978
100x100	6.231 ms	7.989
1000x1000	8482.9 ms	10489.7 ms
2000x2000	106566 ms	125706 ms



6. Conclusiones

- De acuerdo con las pruebas realizadas, determinamos que a partir de matrices de dimensiones superiores a 100 x 100, resulta más beneficioso utilizar algoritmos basados en concurrencia, ya que de esta forma se aprovecharían de manera eficiente los recursos de computo.
- Concluimos que la ejecución serializada mediante algoritmos tradicionales es valida y esta justificada en escenarios donde no hay grandes volúmenes de datos o en aplicaciones donde se podría tolerar un tiempo de espera, ya que incrementar el performance conlleva un incremento de recursos computacionales que a su vez se traduce en mayores costos de infraestructura.