Programación orientada a objetos II

PC 4

Desarrollado por

Württele Igari, Stephano Fabrizio

Sobrados Burgos, Enrique Francisco

Madera Garcés, César Antonio

Profesor

Rivas Medina, Rubén Demetrio

Sección

2.00

Barranco, junio 2019

Algoritmo concurrente vs Algoritmo secuencial

Introducción

A lo largo de los años la tecnología a avanzado en pasos agigantados, tal y como fue predicho por la ley de Moore. Este crecimiento, sin embargo, ha alcanzado un límite físico. Por ello se ha tenido que innovar para romper límites anteriormente trazados en cuanto a rendimiento. Dada esta necesidad, se ideó la programación concurrente, método que consiste en utilizar más de un procesador a la vez para maximizar la eficiencia de instrucciones por unidad de tiempo. En el presente informe buscamos explorar la verdadera eficiencia de usar este método de programación. Para ello, evaluaremos cantidad de tareas por realizar con número de Hilos utilizados a través del uso de threads. Finalmente, concluimos con una justificación sobre la eficiencia o utilidad de usar hilos y programación concurrente, y a partir de qué límites.

Objetivo

Demostrar bajo qué circunstancias programación concurrente puede optimizar los tiempos de compilación en un programa de multiplicación de matrices.

El código

El código realizado por el equipo se basó en el algoritmo de multiplicación de matrices de una manera iterativa. Para ello tuvimos que:

1. Implementar una clase matriz, donde se construye y destruye respectivamente.

2. Sobrecargar el operador de multiplicación en la clase para poder hacer multiplicacion matricial.
3. Crear una función adicional donde se reparta la concurrencia para cada uno de los hilos uniformemente.

Las matrices como tal están definidas con un puntero doble que guarda la primera posición de arreglos de memoria, formándose un contenedor de dos dimensiones a las que podemos acceder con dos direcciones.

El algoritmo para la sobrecarga de la multiplicación matricial está definido con un algoritmo simple de multiplicación de matrices en el que se recorre cada parte de cada una y se multiplican entre ellas.

A continuación, se explicará la concurrencia.

Para realizar la concurrencia creamos 4 threads y separamos la función calcular(función en la cual se realiza la multiplicación de matrices) en 4 partes.

```
#define NUMHILOS 4
```

```
void calcular(matriz<int> temp,matriz<int> matriz1,matriz<int> matriz2,int id){
    int lim_inf=id*(matriz1.filas)/NUMHILOS;
    int lim_sup=(id+1)*(matriz1.filas)/NUMHILOS;
    for (int i = lim_inf; i < lim_sup; i++) {
        for (int j = 0; j < matriz2.columnas; j++){
            for (int z = 0; z < matriz2.filas; z++) {
                temp.matrizz[i][j] += matriz1.matrizz[i][z] * matriz2.matrizz[z][j];
            }
        }
    }
}
```

La función calcular tiene como parámetros 3 objetos que son instancias de la clase matriz.

```
template <typename A>
class matriz{
    A **matrizz;
    int filas;
    int columnas;
public:
    friend void calcular(matriz<int> temp,matriz<int> matriz1, matriz<int> matriz2,int id);
    A** get_matriz()const {
        return matrizz;
    }
    int get_filas() const {
        return filas;
    }
    int get_columnas()const {
        return columnas;
    }
    void imprimir_matriz(){
        for (int i=0;i<filas;i++){
            for (int j=0;j<columnas;j++){
                cout<<matrizz[i][j]<<" ";
            }
            cout<<endl;
        }
```

La clase matriz tiene un doble puntero el cual guarda valores de tipo genérico y se le da la forma de una matriz al momento de ser construida.

¿Cómo funciona el algoritmo?

```
matriz<A> operator*( const matriz<A>& matriz2){
    matriz temp(this->filas,matriz2.columnas);
    if(this->columnas== matriz2.filas) {
        for (int i = 0; i < this->filas; i++) {
            for (int j = 0; j < matriz2.columnas; j++){
                for (int z = 0; z < matriz2.filas; z++) {
                    temp.matrizz[i][j] += this->matrizz[i][z] * matriz2.matrizz[z][j];
                }
            }
        }
    }
    else{
        cout<<"las filas y columnas que se ingresaron fueron diferentes"<<endl;
        return temp;
    }
```

El algoritmo recorre todas las filas de la primera matriz, todas las columnas de la segunda, genera la multiplicación y la guarda el resultado de toda una fila por una columna dentro del espacio determinado en la tercera matriz.

El algoritmo que se utilizó es el más común a la hora de desarrollar una multiplicación matricial. El costo computacional de este modelo se magnifica mientras ambas matrices sean de mayor tamaño.

En el caso de la multiplicación matricial concurrente, se tiene que dividir las partes de la matriz que se desean utilizar correctamente para evitar el uso de mutex y de guard_lock, así se puede ahorrar tiempo. Para el presente trabajo, se decidió realizar esta división a través de un id que se pasa como parámetro, lo cual definirá el límite superior e inferior para cada hilo en el cual puede trabajar.

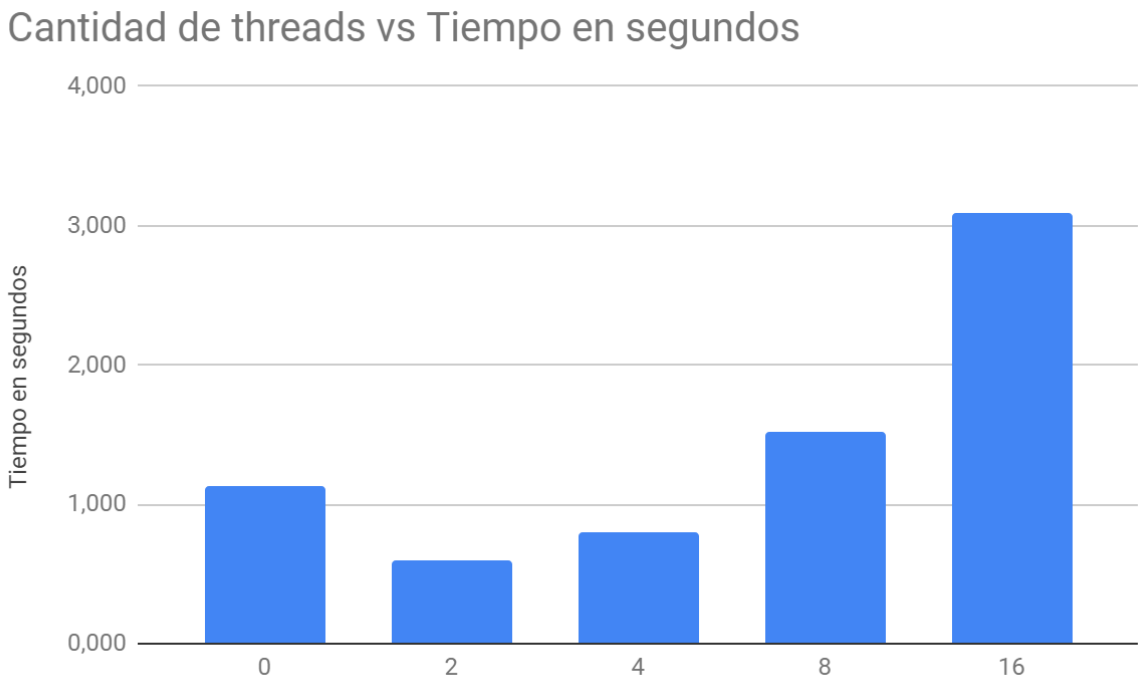
```
void calcular(matriz<int> temp,matriz<int> matriz1,matriz<int> matriz2,int id){
    int lim_inf=id*(matriz1.filas)/4;
    int lim_sup=(id+1)*(matriz1.filas)/4;

    for (int i = lim_inf; i < lim_sup; i++) {
        for (int j = 0; j < matriz2.columnas; j++){
            for (int z = 0; z < matriz2.filas; z++) {
                temp.matrizz[i][j] += matriz1.matrizz[i][z] * matriz2.matrizz[z][j];
            }
        }
    }
}
```

Supongamos que el id ingresado sea 0, el límite inferior será 0, mientras que el límite superior será ¼ del tamaño de la matriz. Por otra parte, este límite inferior determinará, en el caso de la primera y tercera matriz, las filas, las cuales serán rellenadas o usadas, mientras que en la segunda, determinarán las columnas usadas.

De esta manera, no es necesario utilizar algún método de bloqueo de información debido a que nunca se utilizará un dato dos veces o más, en ese sentido, se está mejorando el rendimiento del programa al evitar el uso del mutex u otro elemento de bloqueo .

Resultados calculados

El rendimiento de este programa que multiplica matrices de 500x500 de acuerdo a criterios de número de hilos y tiempo en microsegundos se da de la siguiente forma:



Como se ve en esta gráfica, el rendimiento usando threads en casos con pocos hilos, el tiempo se ve significativamente reducido en comparación a no utilizarlos. Sin embargo, su uso como tal también implica algún uso de memoria para separar y luego juntar nuevamente los hilos, por lo que para operaciones pequeñas o procesos no tan complicados o estructurados, usar muchos hilos resulta ser más engorroso y costosa en cuanto a tiempo para el computador.

Conclusiones

Como pudimos observar en las gráficas anteriores depende de la magnitud de operaciones que realicemos que la efectividad que buscan los threads sea probada. Necesitamos utilizar grandes cantidades de datos para gran magnitud de uso de threads eficientes. En realidad, como vemos en el aumento exponencial de el uso de threads innecesarios, probablemente una cantidad apropiada de threads a usar generalmente sea de alrededor de 8, a menos que sean procesos realmente complejos (cálculos para despegues espaciales simultáneos, por ejemplo). Sin embargo, para procesos no tan pequeños como con los que probablemente trabajemos nuestro día a día, no siempre va a beneficiarnos utilizar threads simplemente por utilizarlos. Debe ser de manera medida y con distribución controlada. De no ser así, su uso puede ser incluso perjudicial.