Instituto Tecnológico de Buenos Aires

TP2

73.62 - Programación Paralela

Autor:

Ballerini, Santiago[1] - 61746

Docente:

Cristian Maximiliano Mateos Diaz

Fecha de entrega: 14 de Octubre de 2024

[1] sballerini@itba.edu.ar

Índice

1. Introducción

La multiplicación de matrices es un problema altamente paralelizable. En este trabajo, se analizará cómo el tiempo de ejecución de una multiplicación de matrices varía al modificar el número de hilos utilizados. Para este estudio, se emplearon tres variantes del algoritmo básico de multiplicación: una variante secuencial, una variante paralela estándar y una variante paralela basada en el modelo *fork-join*. Estas variantes se presentarán con mayor detalle en secciones posteriores. Todo el desarrollo se realizó utilizando el lenguaje de programación Java.

2. Algoritmos

A continuación se detallan los algoritmos utilizados.

2.1. Secuencial

El algoritmo secuencial utilizado para la multiplicación de matrices se presenta a continuación. Este algoritmo se ejecuta exclusivamente en el hilo principal de la aplicación, lo que significa que todas las operaciones se realizan de forma lineal, sin aprovechar la capacidad de múltiples núcleos de procesamiento.

```
1  public void multiply(double[][] a, double[][] b, double[][] c) {
2      int n = a.length;
3
4      for (int i = 0; i < n; i++) {
5          for (int j = 0; j < n; j++) {
6              double sum = 0;
7              for (int k = 0; k < n; k++) {
8                  sum += a[i][k] * b[k][j];
9              }
10             c[i][j] = sum;
11         }
12     }
13 }
```

Listado 1: Algoritmo Secuencial

2.2. Paralelo

El algoritmo paralelo que se presenta a continuación se ejecuta sobre un grupo fijo de N hilos. Se decidió paralelizar el cálculo de las filas, de modo que cada hilo sea responsable de obtener una fila completa de la matriz resultante de la multiplicación. El hilo principal de la aplicación espera a que todos los hilos del grupo terminen su ejecución antes de finalizar.

```
public void multiply(double[][] a, double[][] b, double[][] c) {
    int n = a.length;

    for (int i = 0; i < n; i++) {
        final int finalI = i;
        pool.execute(() -> {
            for (int j = 0; j < n; j++) {
                double sum = 0;
                for (int k = 0; k < n; k++) {
                    sum += a[finalI][k] * b[k][j];
                }
                c[finalI][j] = sum;
            }
        });
    }

    pool.shutdown();
    try {
        if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
            pool.shutdownNow();
        }
    } catch (InterruptedException e) {
        pool.shutdownNow();
    }
}
```

Listado 2: Algoritmo Paralelo

2.3. Fork-Join

El algoritmo *fork-join* sigue el principio de *Divide and Conquer*, este enfoque recursivo divide la tarea de multiplicación en subtareas más pequeñas que se pueden procesar en paralelo. En este caso Si la cantidad de filas a procesar está por debajo de un umbral definido, se utiliza un método de multiplicación directa. De lo contrario, la tarea se divide en dos subtareas.

```
protected void compute() {
    if (endRow - startRow <= threshold) {
        multiplyDirectly();
    } else {
        int midRow = (startRow + endRow) / 2;

        MultiplyTask upperTask = new MultiplyTask(a, b, c,
           startRow, midRow, size, threshold);
        MultiplyTask lowerTask = new MultiplyTask(a, b, c, midRow,
           endRow, size, threshold);

        upperTask.fork();
        lowerTask.compute();
        upperTask.join();
    }
}

private void multiplyDirectly() {
    for (int i = startRow; i < endRow; i++) {
        for (int j = 0; j < size; j++) {
            double sum = 0;
            for (int k = 0; k < size; k++) {
                sum += a[i][k] * b[k][j];
            }
            c[i][j] = sum;
        }
    }
}
```

Listado 3: Algoritmo Fork-Join

3. Resultados

Para comparar los algoritmos, se varió el número de threads desde 1 hasta 16, que corresponde al número de núcleos lógicos del procesador utilizado en las pruebas. A continuación, se detallan las especificaciones de la computadora en la que se ejecutó el programa.

- **CPU:** Ryzen 7 3800x
  - **Cores:** 8
  - **Threads:** 16
  - **Frecuencia base:** 3.9 Ghz
  - **Frecuencia máxima:** 4.5 Ghz
- **Memoria:**
  - **Capacidad:** 16GB
  - **Frecuencia:** 3200Mhz

A continuación, se llevarán a cabo dos análisis: el primero, para identificar el valor óptimo del parámetro *threshold* en el algoritmo *fork-join*, y el segundo, para comparar el rendimiento de las tres variantes de algoritmos. Para esto se multiplicaron dos matrices de 1024×1024

3.1. Búsqueda del parámetro *threshold* ideal

El algoritmo *fork-join* recibe un parámetro clave que es el *threshold*. Este parámetro define el punto en que la tarea deja de subdividirse y comienza a ser computada directamente. Es importante determinar el valor óptimo de *threshold* antes de comparar el rendimiento de este algoritmo con los otros dos (secuencial y paralelo) en función del número de hilos. Para ello, se probaron valores de *threshold* que son potencias de 2, desde 1 hasta N, donde N es el tamaño de la matriz. Se realizaron 10 iteraciones y se promediaron los resultados. Los resultados se presentan en la Figura 1, divididos en cuatro grupos de hilos para una mejor visualización.

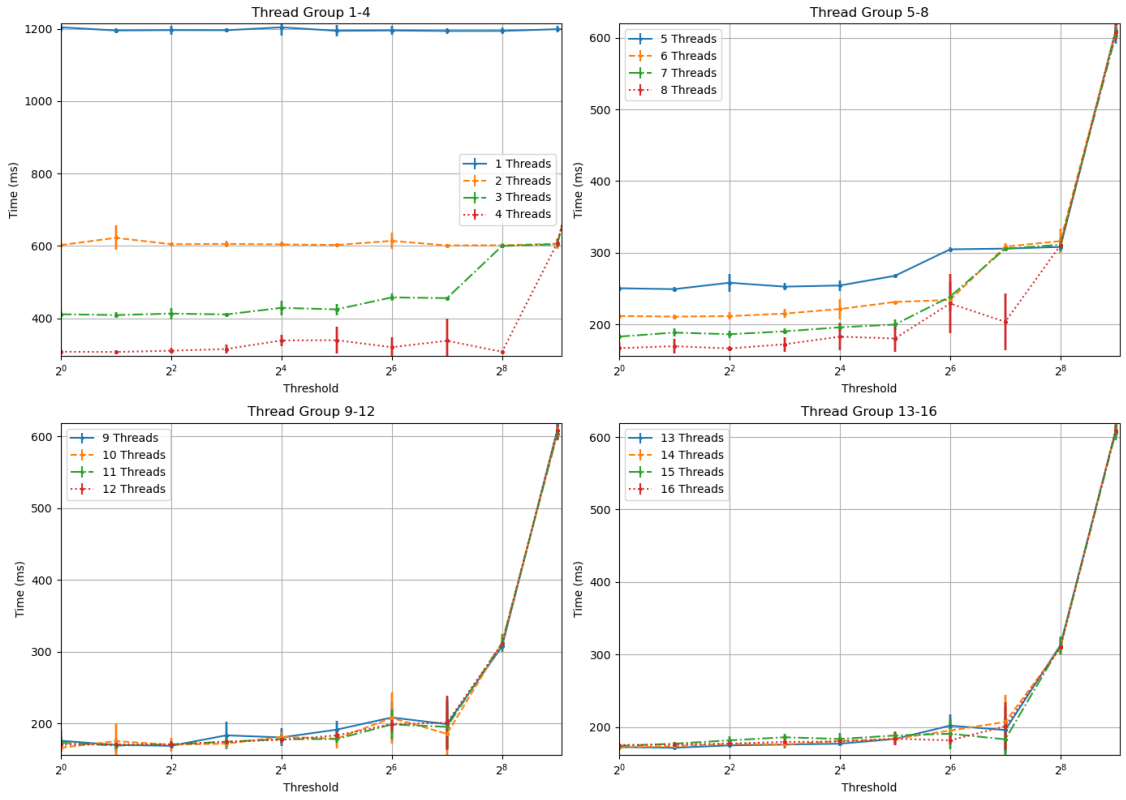

Figura 1: Tiempo de ejecución del algoritmo *fork-join* variando el parámetro *threshold*

Al analizar la Figura 1, se observa que el valor óptimo del parámetro *threshold* generalmente se encuentra entre 2^4 y 2^6 para la mayoría de los grupos de hilos. En configuraciones con pocos hilos (1-4), un *threshold* más bajo (2^2 a 2^4) proporciona un rendimiento más estable, mientras que para grupos más grandes (5-16 hilos), el rendimiento se mantiene estable con *thresholds* más amplios, pero siempre empeora significativamente cuando el *threshold* excede 2^7.

3.2. Comparación del algoritmos

Una vez determinados los mejores *thresholds*, se esta en condiciones de comparar las 3 variantes del algoritmo. Para esta comparación también se promedio el resultado de 10 corridas.

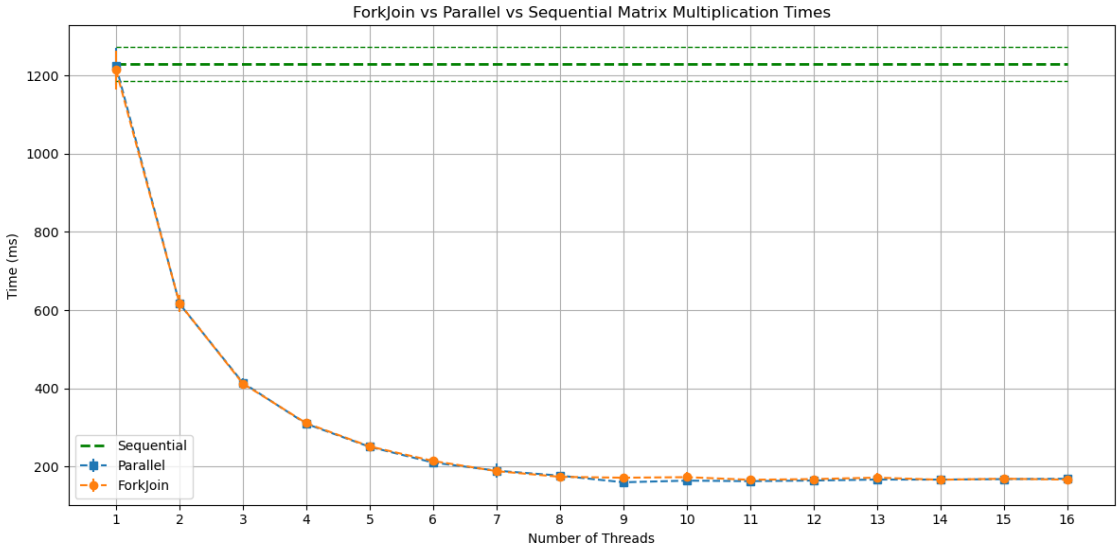

Figura 2: Comparación de las 3 variantes

En la Fig. 2 se puede observar que, a medida que aumenta el número de hilos, tanto el algoritmo paralelo como el *fork-join* reducen drásticamente el tiempo de ejecución en comparación con el algoritmo secuencial. A partir de 4 hilos, el rendimiento de ambos algoritmos paralelos es prácticamente indistinguible y continúa mejorando hasta llegar a 8 hilos, donde la ganancia de rendimiento se estabiliza. Esto indica que la paralelización es altamente efectiva hasta cierto punto, después del cual la sobrecarga de gestionar más hilos no produce mejoras significativas.

También hay que destacar las limitaciones impuestas por el hardware del procesador utilizado, que cuenta con 8 núcleos físicos y 16 lógicos. Es posible que el rendimiento se estanque en 8 hilos, dado que los hilos lógicos comparten los recursos físicos del núcleo, lo que genera una disminución de eficiencia en comparación con tener más núcleos físicos.

4. Conclusiones

Los resultados obtenidos demuestran que la multiplicación de matrices es una tarea altamente paralelizable, y tanto el algoritmo paralelo como el modelo *fork-join* ofrecen mejoras significativas en el tiempo de ejecución en comparación con el algoritmo secuencial. Sin embargo, no se encontró una mejora notable entre el modelo estándar y el modelo *fork-join*, lo que sugiere que, en esta implementación específica, el enfoque de *fork-join* no aporta ventajas adicionales en términos de rendimiento. También se pudo identificar los valores óptimos del parámetro *threshold* para el algoritmo *fork-join*, este se encuentra entre 2^4 y 2^6, dependiendo del número de hilos utilizados.

Al aumentar el número de hilos, el rendimiento de los algoritmos paralelos mejora sustancialmente, especialmente hasta alcanzar los 8 hilos, que coincide con la cantidad de núcleos físicos del procesador utilizado. A partir de este punto, la ganancia en rendimiento se estabiliza.

En conclusión, este trabajo resalta las mejoras que trae la paralelización en la multiplicación de matrices, aunque estas se ven limitadas por el entorno de ejecución.