COMPARACIÓN DE LA MULTIPLICACIÓN DE MATRICES: ENFOQUES SECUENCIAL Y CONCURRENTE EN JAVA

Datos del autor: Ramil Elías
Repositorio con el trabajo: https://github.com/elidev72/TP-PC-1C2025.git
E.mail: elidev72@gmail.com

RESUMEN (ABSTRACT)

Este estudio compara el rendimiento de la multiplicación de matrices cuadradas utilizando enfoques de programación secuencial y concurrente en Java. El objetivo principal es cuantificar las mejoras en el tiempo de ejecución que ofrece la concurrencia frente a la ejecución tradicional, especialmente para matrices de gran dimensión. Se implementaron ambos algoritmos: una versión secuencial clásica y una versión concurrente que distribuye las operaciones por filas utilizando un pool de hilos. El algoritmo concurrente fue de elaboración propia. Los resultados preliminares indican que la implementación concurrente logra una reducción significativa en el tiempo de procesamiento al aprovechar los recursos de hardware multinúcleo disponibles, demostrando la eficacia de la paralelización para tareas computacionalmente intensivas. Este análisis busca proveer una comprensión clara de los beneficios y consideraciones al adoptar la programación concurrente en escenarios de cálculo matricial.

Keywords: Multiplicación de matrices, Programación concurrente, Programación secuencial, Java, Rendimiento, Hilos

1. INTRODUCCIÓN

La multiplicación de matrices es una operación fundamental en diversas áreas científicas y de ingeniería, incluyendo gráficos por computadora, simulación, procesamiento de señales y aprendizaje automático. Dada la naturaleza de esta operación, donde cada

elemento de la matriz resultante es el producto escalar de una fila de la primera matriz por una columna de la segunda, su complejidad computacional es inherentemente alta, específicamente $O(n^3)$ para matrices cuadradas de dimensión N×N. Esta complejidad la convierte en un candidato ideal para explorar técnicas de optimización de rendimiento.

Tradicionalmente, la multiplicación de matrices se ha abordado de forma secuencial, donde las operaciones se ejecutan una tras otra sin paralelismo. Sin embargo, con la evolución de la arquitectura de hardware hacia procesadores multinúcleo y la creciente demanda de procesamiento de grandes volúmenes de datos, la programación concurrente emerge como una alternativa prometedora para reducir significativamente los tiempos de ejecución. La concurrencia permite dividir una tarea grande en subtareas más pequeñas que pueden ejecutarse simultáneamente, aprovechando así la capacidad de procesamiento paralelo del hardware.

Este estudio se centra en comparar el rendimiento de dos implementaciones del algoritmo de multiplicación de matrices cuadradas en Java: una versión secuencial y una versión concurrente. El objetivo principal es cuantificar las diferencias en el tiempo de ejecución entre ambos enfoques a medida que aumenta la dimensión de las matrices, demostrando los beneficios potenciales de la paralelización para este tipo de carga de trabajo. La implementación del algoritmo secuencial, de elaboración propia, puede consultarse en: https://github.com/elidev72/TP-PC-1C2025/blob/master/TP_PC/src/matrices/Secuencial.java

La Multiplicación de Matrices

La multiplicación de dos matrices, A y B, que resulta en una matriz C=AxB, se define tal que cada elemento C_ij se calcula como el producto escalar de la i-ésima fila de A y la j-ésima columna de B (González Medina, 2016).

2. IMPLEMENTACIÓN CONCURRENTE

Dada la naturaleza computacionalmente intensiva de la multiplicación de matrices, la programación concurrente fue elegida para optimizar el tiempo de ejecución del algoritmo. El objetivo principal de la paralelización es distribuir la carga de trabajo entre múltiples unidades de procesamiento (hilos o procesos), permitiendo que porciones independientes del cálculo se realicen simultáneamente.

En la multiplicación de matrices, cada fila de la matriz resultante se calcula de forma independiente de las demás. Esto significa que podemos dividir el trabajo por filas, asignando a cada hilo un rango específico de filas para calcular. Así, mientras un hilo calcula las filas 0 a 9, otro puede estar calculando las filas 10 a 19, y así sucesivamente, aprovechando la independencia de estas operaciones. Las partes que no son concurrentes incluyen la inicialización de las matrices de entrada y la recolección final de los resultados.

Mecanismo de Concurrencia en Java

Para gestionar la concurrencia, se utilizó java.util.concurrent de Java, específicamente un pool de hilos a través de la interfaz ExecutorService y la clase Executors. Este enfoque simplifica la gestión del ciclo de vida de los hilos, ya que no es necesario crear y destruir hilos manualmente para cada tarea, reduciendo el *overhead* de gestión.

El número de hilos utilizados se determinó dinámicamente en tiempo de ejecución mediante Runtime.getRuntime(). availableProcessors(), que devuelve el número de procesadores disponibles para la máquina virtual Java. Esto permite que el algoritmo se adapte al hardware subyacente, buscando maximizar el aprovechamiento de los núcleos de CPU.

El trabajo se dividió en tareas (representadas por la clase TareaMultiplicacionMatrices, que implementa la interfaz Runnable). Cada instancia de esta

tarea recibe las matrices de entrada (matrizA, matrizB), la matriz resultante (matrizResultante) y el rango de filas (iFilaInicio, iFilaFin) que le corresponde calcular.

Esta estrategia asegura que cada hilo opere sobre una porción de la matriz resultante que es independiente de las operaciones de otros hilos, eliminando la necesidad de mecanismos explícitos de sincronización para evitar conflictos de datos durante el cálculo de elementos individuales.

El código fuente completo de la implementación concurrente, junto con la secuencial, está disponible para su consulta en: https://github.com/elidev72/TP-PC-1C2025/blob/master/TP_PC/src/matrices/Concurrente.java

3. COMPARATIVA Y DESEMPEÑO

Para evaluar y comparar el rendimiento de las implementaciones de multiplicación de matrices secuencial y concurrente, se diseñaron una serie de casos de prueba variando la dimensión de las matrices cuadradas (N). El objetivo es observar cómo se comportan ambos algoritmos ante cargas de trabajo crecientes y cuantificar la mejora de rendimiento que ofrece la concurrencia.

Casos de Prueba

Se realizaron pruebas con las siguientes dimensiones para las matrices cuadradas (N×N): 100, 200, 400, 800 y 1000. Para cada dimensión N, las matrices de entrada se inicializaron con números enteros aleatorios dentro de un rango específico (definido en MatrizUtils), asegurando que las condiciones de prueba fueran consistentes. Se realizaron múltiples ejecuciones para cada caso y se promediaron los tiempos para mitigar fluctuaciones puntuales del sistema operativo o la JVM.

Especificaciones del Entorno de Prueba

Los experimentos se llevaron a cabo en el siguiente entorno de hardware y software

para garantizar la reproducibilidad y contextualización de los resultados:

- **Procesador (CPU)**: Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
    - **Núcleos Físicos**: 2
    - **Hilos Lógicos**: 4
- **Memoria RAM**: 8 GB DDR3
- **Sistema Operativo**: Linux Mint 22 x86_64
- **Versión de Java (JDK)**: OpenJDK 21.0.7

Resultados Obtenidos

A continuación, se presentan los tiempos de ejecución promedio en milisegundos (ms) para cada algoritmo y dimensión de matriz probada.

Tabla Comparativas

Dimensión (N)	Tiempo Secuencial (ms)	Tiempo Concurrente (ms)	Factor de Mejora (Secuencial / Concurrente)
100	15	19	0.79
200	38	34	1.12
400	159	135	1.18
800	1948	1440	1.35
1000	6703	4765	1.41

4. CONCLUSIÓN

Este estudio comparativo ha demostrado la eficacia de la programación concurrente como una estrategia para optimizar el rendimiento en operaciones computacionalmente intensivas, como la multiplicación de matrices cuadradas. Los resultados obtenidos, a partir de implementaciones secuenciales y concurrentes en Java, revelan una clara tendencia: a medida que la dimensión de las matrices aumenta, la versión concurrente supera consistentemente a su contraparte secuencial.

Particularmente, para dimensiones pequeñas (N=100), se observó que el algoritmo concurrente fue marginalmente más lento (factor de mejora de 0.79). Este fenómeno es atribuible al overhead inherente a la gestión de hilos y la distribución de tareas, que en problemas de baja complejidad puede superar los beneficios del paralelismo. Sin embargo, a partir de N=200, la versión concurrente comenzó a mostrar ventajas significativas. Para N=1000, la implementación concurrente fue aproximadamente 1.41 veces más rápida, reduciendo el tiempo de ejecución de 6703 ms a 4765 ms.

Estos hallazgos confirman que la concurrencia es una herramienta valiosa para aprovechar la arquitectura de hardware multinúcleo moderna. La capacidad de dividir el problema de multiplicación de matrices en subtareas independientes y ejecutarlas en paralelo permitió una mejora sustancial en el tiempo de procesamiento. A pesar de los costos iniciales asociados a la concurrencia en tareas pequeñas, los beneficios se vuelven indiscutibles conforme la carga de trabajo y el tamaño del problema escalan.

En resumen, la elección entre un enfoque secuencial y uno concurrente debe considerar la escala del problema. Para matrices de gran tamaño, la implementación concurrente ofrece una ventaja de rendimiento crítica, validando su aplicación en contextos

donde la eficiencia computacional es primordial.

REFERENCIAS

González Medina, R. (2016). *Fundamentos de matemáticas de 3er año* (U7). R. González Medina.