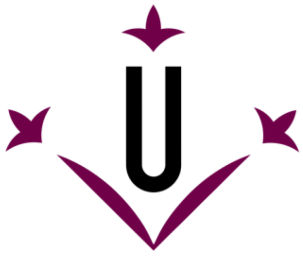

Universitat de Lleida

Multiplicación de matrices concurrente

Segura Paz, Aleix

Serrano Ortega, Aniol

Fecha: 18/12/2023

Práctica 4

Sistemas Concurrentes y Paralelos

Escuela Politécnica Superior

Índice

Índice de Figuras

1. Introducción

El objetivo de esta practica es el de implementar la versión concurrente de dos formas distintas de calcular la multiplicación de matrices en C. Este calculo es entre dos matrices cuadradas de orden n. Es decir, el numero de elementos de cada matriz es de $n \times n$. Además, se debe adaptar la estructura del proyecto para esta nueva versión.

La primera forma de calcular el resultado entre estas dos matrices es el método estándar, es decir, filas por columnas. Este método también es conocido como el método *ijk* o *ikj* en su versión optimizada.

La segunda forma de calcular esta multiplicación, es mediante el algoritmo de *Schönhage-Strassen*, también llamado método de *Strassen*. Este método emplea llamadas recursivas para dividir las tareas y mejorar así la eficiencia.

Para realizar esta implementación se usara una librería de mas bajo nivel respecto a las empleadas en Java llamada `pthread.h`.

2. Diseño de la implementación concurrente

La implementación concurrente que se ha realizado se trata sobre matrices regulares de orden n. Estas matrices se leen desde los ficheros de texto contenidos en el directorio `Input`. Estos ficheros contienen el tamaño de la matriz y los números de la que forman parte. Cabe destacar que para cada matriz A de orden n existe su matriz B con el mismo orden.

Los resultados de estas multiplicaciones se muestran en el directorio `Results`. Así mismo, existe una serie de *tests* que comprueban que los resultados son correctos y deterministas. Para implementar la versión concurrente de la multiplicación de matrices de tamaño $n \times n$, se ha hecho mediante dos métodos, estándar y Strassens.

2.1. Multiplicación de matrices tradicional

La primera parte consiste en realizar la multiplicación de matrices empleando el método tradicional, es decir, la multiplicación *ijk*. Este método es computacionalmente costoso, ya que es $O(n^3)$.

En el proyecto también esta definida la operación de multiplicación de matrices *ikj*. Esta operación es mas eficiente porque los datos en memoria están mas juntos al estar repartidos por filas, es decir, se generan menos fallos. A pesar de que esta operación es más eficiente, sigue siendo computacionalmente costosa.

Por otro lado, cabe destacar que la repartición de tareas es homogénea (excluyendo el ultimo hilo). Esto es debido a que los datos de entrada son conocidos por el programa antes de su ejecución, y se pueden repartir homogéneamente entre los hilos.

Para compartir el numero de hilos pasados por parámetro o por defecto (4), se ha empleado una variable global que en caso que no se pasen los hilos por parámetro, se le asignan los hilos por defecto. Mediante la declaración como variable global externa en el fichero receptor de la variable (`Standard_MultMat.c`) se ha podido compartir este valor.

Cabe destacar, que el grado de concurrencia máximo es n, dado que es el numero de filas de las matrices. Por tanto, en caso que los hilos pasados par parámetro sean mayores a n, simplemente se modifican los hilos empleados a n. Naturalmente, para la siguiente iteración se debe volver a comprobar esta casuística y en caso de no ser así, modificar los hilos empleados a los que se han pasado por parámetro.

La estrategia empleada para ejecutar la versión concurrente de este programa subyace en la creación de una estructura llamada `section_data` para almacenar los datos de la matriz específicos para cada hilo. Dicha estructura tiene los siguientes campos:

```
typedef struct {
    float **matrixA;
    float **matrixB;
    int n;
    int start_row;
    int end_row;
    float **result;
} section_data;
```

Como resulta evidente, las matrices A y B son de tipo *float* que son pasadas por referencia para manejar mas eficientemente la memoria. El campo n representa la dimensión de estas matrices, y los campos `start_row` y `end_row` sirven para definir el rango de números que debe calcular cada hilo. Finalmente, el resultado se almacena en la posición de memoria apuntada por la variable de la matriz de resultado llamada `result`.

A continuación, se emplea esta estructura de datos para asignar a cada campo su tarea correspondiente para cada hilo. Utilizando la variable global `threads`, se itera para asignar a cada hilo los datos correspondientes en su posición de la lista y así crear el hilo.

Para asignar los datos correspondientes a cada campo de la estructura, se deben establecer los datos de las matrices y los límites de las filas que cada hilo procesará. Los campos

`start_row` y `end_row` deben ser calculados específicamente. Para el inicio de cada sección, simplemente se multiplica el índice del hilo actual (i) por el número de filas asignadas a cada hilo.

La función auxiliar `get_end_row` se utiliza para determinar el final de la sección de cada hilo. Esta función considera si el hilo es el último en la lista; en tal caso, se le asigna el procesamiento de todas las filas restantes. Si no es el último hilo, el final de su sección se calcula como el índice siguiente ($i + 1$), multiplicado por el número de filas por sección (`rows_per_section`).

Una vez asignados el conjunto de datos para cada hilo de deben crear, para eso se emplea la llamada `pthread_create`. En esta llamada se le pasa la referencia al hilo a crear, representada por `&threads_list[i]`, un valor `NULL` para las configuraciones por defecto del hilo, la función `process_section` que define la tarea a realizar por cada hilo, y una referencia a los datos que necesita cada hilo, que en este caso es `&data[i]`.

En caso que al crear los hilos se de un error, se deben cancelar todos los que se han creado hasta ese punto y salir del programa mediante la función `Error(char *msg)`. Para eso, se emplea una función auxiliar en donde se le pasa el numero de hilos creados hasta el error (i) y se cancelan todos esos. En caso que al cancelar se genere un error, este se muestra y se termina el programa igualmente.

La función `process_section` es la responsable de procesar la sección de las matrices asignadas a cada hilo, utilizando los datos suministrados en `data[i]`. Esta función realiza la multiplicación de matrices tradicional **ijk**, pero empleando las secciones determinadas por los campos `start_row` y `end_row`.

El resultado calculado por estas sub-matrices se almacena en la posición i j de `result`. Como que dos mismos hilos no pueden modificar la misma posición i j por la división de tareas previamente explicada, no se producirán condiciones de carrera o resultados no deterministas.

Finalmente, se hace el `join` de estos hilos, en caso de error se cancelan todos los hilos y se sale del programa. Los archivos resultado generados empleando este método tienen la extensión `std`.

2.2. Método *Strassens*

El método *Strassens* es un algoritmo de multiplicación de matrices basado en el patrón *Divide and Conquer*. El coste computacional de este algoritmo es de $O(n^{\log_2 7})$, que es aproximadamente $O(n^{2.8074})$. Por consiguiente, este algoritmo es computacionalmente menos costoso que el estándar.

Este algoritmo se basa en calcular recursivamente las sub-matrices hasta que la dimensión de esta sea inferior a una cota dada. En caso que la n sea inferior a esa cota, se hace el calculo estándar de multiplicación de matrices.

Para implementar este método concurrentemente, se ha definido que en cada nivel de recursividad se emplearan 7 hilos para calcular las 7 tareas (m1, m2, ..., m7). Para eso, se emplea la siguiente estructura:

```
typedef struct {
    float **matrixA;
    float **matrixB;
    float **result;
    int n;
    int mx;
} matrix_data;
```

Los campos de esta estructura son los mismos que los empleados en la versión estándar pero eliminando los campos `start_row` y `end_row` y añadiendo un campo llamado `mx`. Este campo representa el calculo que deberá realizar ese hilo en concreto.

A continuación, se crean los 7 hilos y en caso de error, se cancelan los hilos ya creados y se sale del programa. Al crear estos hilos se llama a la función `calculate_mx` con una `mx` distinta para cada hilo. La primera `mx` es la $i + 1$, es decir 1, la segunda `mx = 2` y así consecutivamente hasta `m7`.

Para mejorar las prestaciones, se ha identificado que no hace falta calcular todos los campos (`a11`, `b11`, etc). Por eso, se ha dividido cada caso en una función distinta, así se evita que cada hilo haga cálculos innecesarios. Por lo tanto, la función `calculate_mx` tiene un *switch-case* que llama a la función de calculo especifica.

De esta forma, se definen 7 funciones que hacen los cálculos de división y hacen las llamadas recursivas a `strassensMultRec`. Luego, se liberan los recursos de esas matrices empleando la función `free_matrix(float **matrix, int n)`.

Para ejemplificar visualmente las llamadas de este método, se ha diseñado el siguiente diagrama:

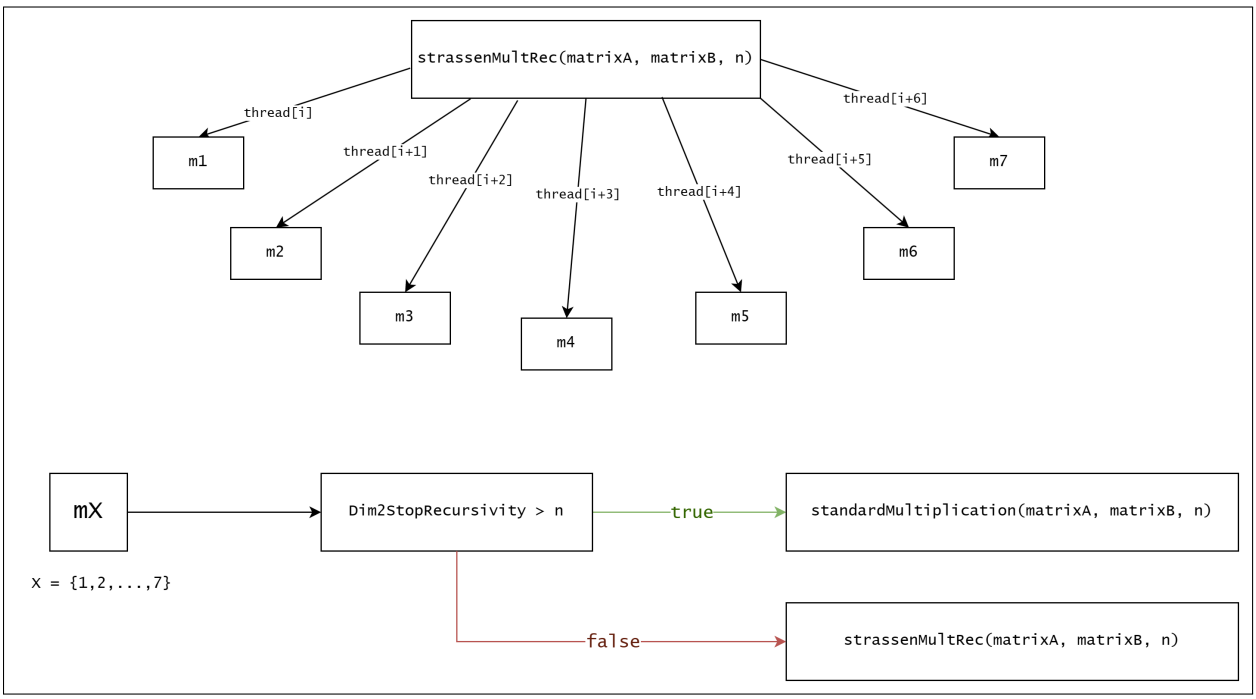

Figura 1: Diagrama conceptual de la llamada de funciones de *Strassens*

Finalmente, se hace el *join* de los hilos y el control de errores. Una vez finalizados los hilos, se hacen los cálculos de `compose` restantes y se liberan los recursos.

3. Rendimiento de la aplicación

Para comprobar la diferencia de rendimiento entre el programa de forma secuencial y de forma concurrente, se han realizado una serie de ejecuciones y se han medido los tiempos de ejecución en ms. Las pruebas se han realizado en un equipo con las siguientes características:

- **Procesador:** *Intel Core i7 12700K*, con velocidad base de *3,61 GHz*, *12 núcleos* y *20 procesadores lógicos*.
- **Disco:** *SSD 1TB M.2* de hasta *5000MB/s*.
- **Memoria RAM:** *32 GB DDR4*.

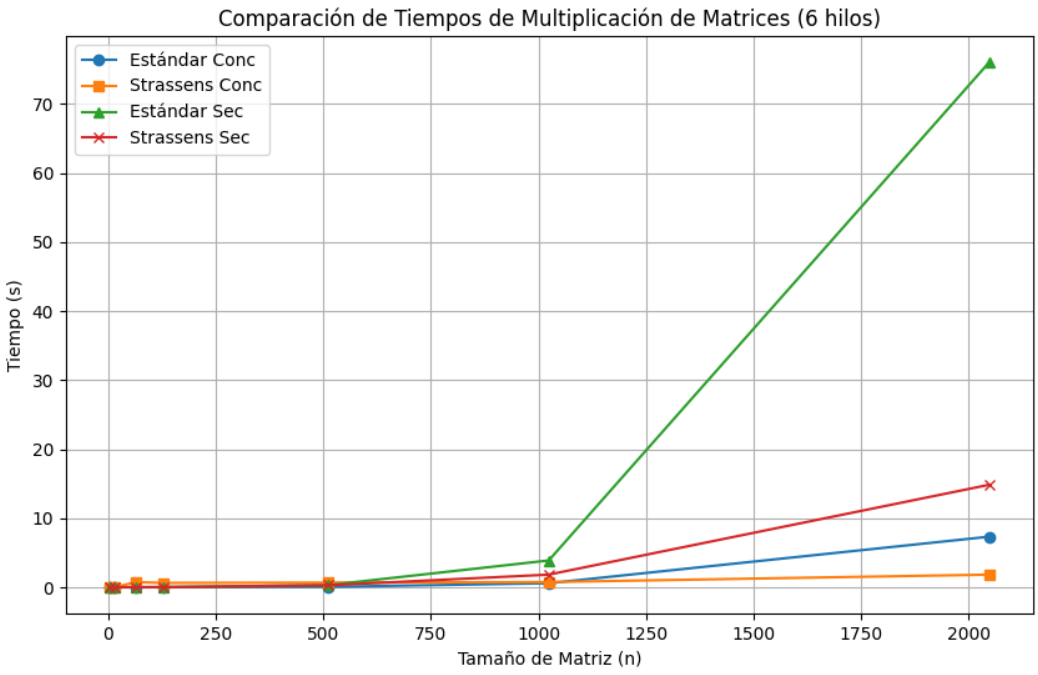

Figura 2: Comparativa de tiempos con 6 hilos

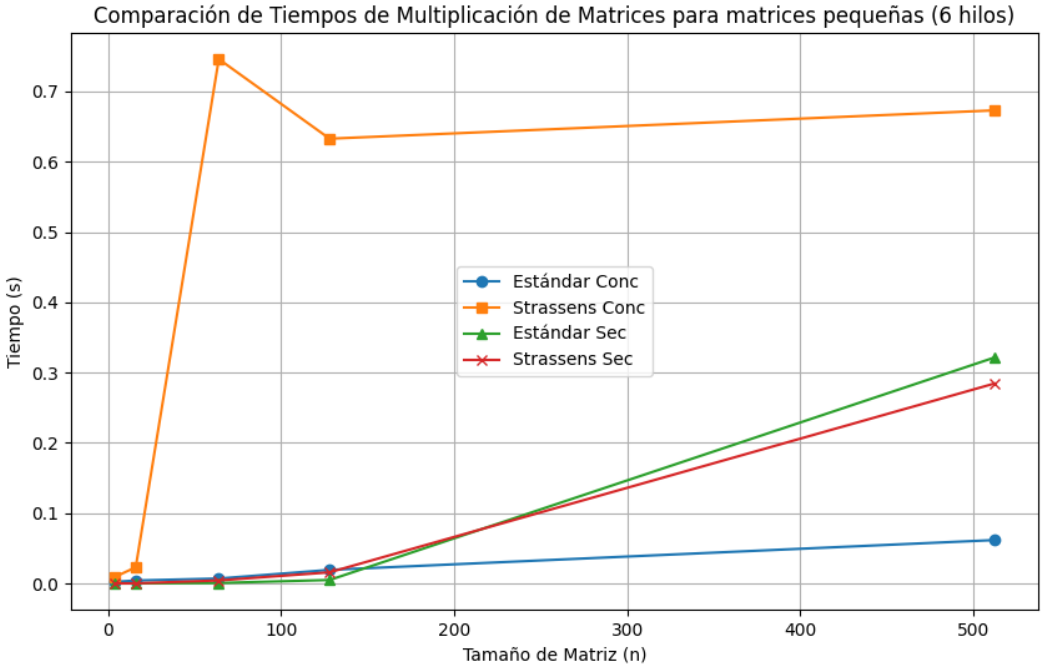

Figura 3: Comparativa de tiempos para matrices pequeñas con 6 hilos

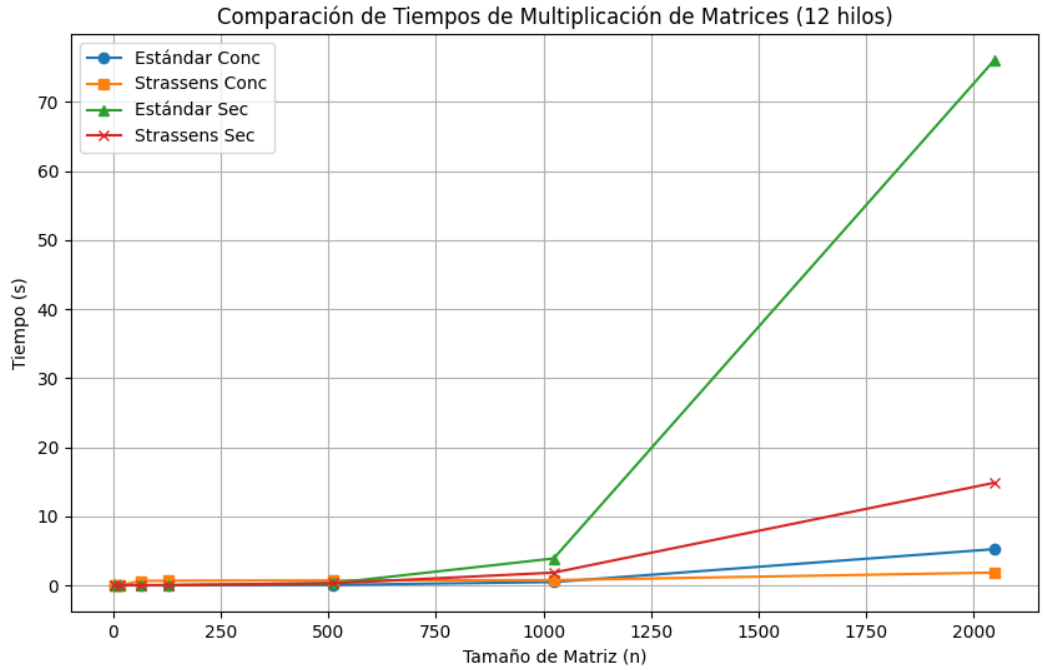

Figura 4: Comparativa de tiempos con 12 hilos

Primeramente, en estos gráficos se puede ver la mejoría de los tiempos en las implementaciones concurrentes contra las secuenciales. Aun y así, se deben remarcar ciertos aspectos.

Como se puede ver en la figura 3, en el caso en el que la matriz sea pequeña, el *overhead* de crear los hilos no compensa por el pequeño coste computacional que estas tienen. Este hecho se remarca especialmente en el caso del *Strassens* concurrente, en el que hay llamadas recursivas que crean 7 hilos cada, por lo que aumenta el coste computacional.

En particular, en el caso de *Strassens* concurrente con $n = 4$, es decir, para la matriz de 4×4, como no se entra en la condición de crear hilos porque la dimensión es muy pequeña ($n < 10$), se calcula el resultado empleando el calculo estándar.

En el caso de matrices grandes, se puede ver como la implementación de *Strassens* concurrente les gana a las demás implementaciones. La implementación concurrente estándar, también tiene buenos tiempos, pero pero para matrices grandes *Strassens* al usar el patrón *Divide and Conquer* resulta ser mas eficiente.

Finalmente, cabe destacar que a el hecho de usar mas hilos mejora las prestaciones de la versión estándar, ya que puede dividir el trabajo entre mas hilos, y como que el grado de concurrencia máximo en ese caso es n, siempre va a mejorar en caso que $threads < n$.

4. Problemas encontrados

En primera instancia, para determinar el numero de *threads* empleados para el calculo concurrente de multiplicación de matrices tradicional se empleaba una variable global en el fichero `Standard_MultMat.c`. De esta forma, no se podían cambiar los *threads* de forma ágil, para solventar este problema se ha empleado una variable global en el programa *main* (`MultMat_Conc.c`).

Esta variable global definida en el archivo del *main*, se ha compartido mediante la declaración de una variable externa siguiendo la siguiente sintaxis:

```
extern int threads;
```

De esta forma, el numero de *threads* se puede pasar por parámetro, y en caso que no se pasen se usan 4 por defecto.

También, el hecho de usar el lenguaje `C` y al ser un lenguaje de mas bajo nivel, provoca una sintaxis considerablemente distinta a la de Java. El hecho de trabajar con punteros y estructuras, ha dificultado la implementación de la práctica.

5. Conclusiones

Durante la realización de esta práctica se ha puesto en prueba los conocimientos de la librería de `pthreads.h`. Además, se ha tenido en cuenta el control de errores de estos hilos y evitar condiciones de carrera sobre las variables. Una mala implementación de esto podría provocar resultados no deterministas, pero mediante un riguroso análisis de la implementación y mediante el uso de los *tests*, se ha podido verificar que los resultados de estos métodos son correctos y deterministas.

Por el otro lado, se ha podido comprobar como para matrices de gran dimensión el coste computacional de crear los hilos compensa para calcular mas eficientemente los resultados, resultando así en menores tiempos de ejecución. En el caso de matrices pequeñas este caso no se da, ya que no compensa crear tantos hilos para calcular matrices tan pequeñas.