Universidad
Carlos III de Madrid

PROGRAMACIÓN DE ALTAS PRESTACIONES

LAB 1 - TIPOS ABSTRACTOS DE DATOS

Asier Sampietro y Andoni Alcelay

MÁSTER UNIVERSITARIO EN TECNOLOGÍAS DEL SECTOR FINANCIERO: FINTECH

Introducción

En las siguientes páginas se hará un resumen de lo conseguido en en laboratorio 1. En este ejercicio se ha creado la clase Matriz para poder darle un uso algebraico sobrecargando así los operadores de suma, multiplicación y paréntesis para implementar la suma y el producto de matrices y el acceso a una posición de la matriz respectivamente. Una vez conseguido se ha calculado la expresión A*B+C siendo cada una de ellas una matriz independiente y la suma de todos los valores de la diagonal principal del resultado.Esta operación se ha probado con matrices NxN donde N coge los valores 10, 100, 1000 y 10000.

Objetivo

El objetivo del laboratorio 1 es practicar las siguientes competencias:

- Diseño de tipos abstractos de datos.
- Constructores y destructores
- Mecanismos de copia y movimiento
- Gestión de memoria dinámica
- Sobrecarga de operadores

Descripción del código fuente

En lo siguientes puntos se analizará brevemente la utilidad e implementación de cada uno de los tres ficheros del código fuente.

main.cpp

En este fichero se sitúa la función main o principal del programa y la usada para rellenar las matrices. El programa recibe varios argumentos. El primero debe ser un número natural,

que determina la dimensión que tendrá la matriz, siendo 1000^2 por defecto. Añadiendo un "-b" en segundo lugar se calcula la potencial de la matriz usando un algoritmo por bloques, siendo este más rápido que la multiplicación estándar por el uso optimizado que hace de la memoria caché. Se puede recurrir al método normal sustituyéndolo por una "-m" o dejandolo vacio, ya que por defecto se usará el modo normal. Por último, añadiendo un "-d" en el tercer lugar se mostrarán en pantalla todas las matrices, teniendo esto un gran impacto negativo y pudiendo evitarse pasando un "-n". Por defecto se evitará esta impresión. Estos parámetros han de meterse en orden, y la entrada no está comprobada. A continuación se inicializan las matrices A, B, y C con la dimensión definida y se rellena la matriz con ceros. También se define una matriz D que no ocupa espacio, que es donde se copiará el resultado. Las tres primeras matrices se rellenan de números aleatorios usando un generador de números normales. Luego empieza a calcular la operación D = A * B + C y mide los tiempos que tarda. Después, suma la diagonal de la matriz obtenida, y muestra el resultado en pantalla. Una vez terminado saca un registro por consola de lo que le ha llevado el cálculo de la matriz D y el tiempo total de ejecución.

matriz.h

En esta cabecera se definen las variables y declaraciones de las funciones que tendrá la clase Matriz. Esta clase dispone de tres miembros dato: dos enteros que guardan el número de filas y columnas, y un puntero a un array de decimales de doble precisión con la dimensión de filas por columnas. La en la clase también se definen constructores, destructores y demás utilidades necesarias para el planteamiento del problema, aparte de sobrecargas para facilitar el acceso o impresión. Por último, también incluye varias sobrecargas de operadores fuera de la clase.

matriz.cpp

En este fichero de definen las funciones que se declaran en la cabecera de la clase matriz. Se implementa el constructor dando el tamaño dado en la creación de este objeto y en caso de no hacerlo, se genera uno vacío, cuyo valor se podrá copiar o mover. Para rellenar las matrices se utiliza el generador de números normales con una media de 2.5 y desviación de 5. También se desarrollan las sobrecargas de operadores declaradas haciendo la suma y producto de matrices. Este último fue optimizado sobre el desarrollado inicialmente, ya que se ha detectado un cuello de botella importante a medida que escala el tamaño de la matriz resultante. En análisis sobre esto se hará más adelante.

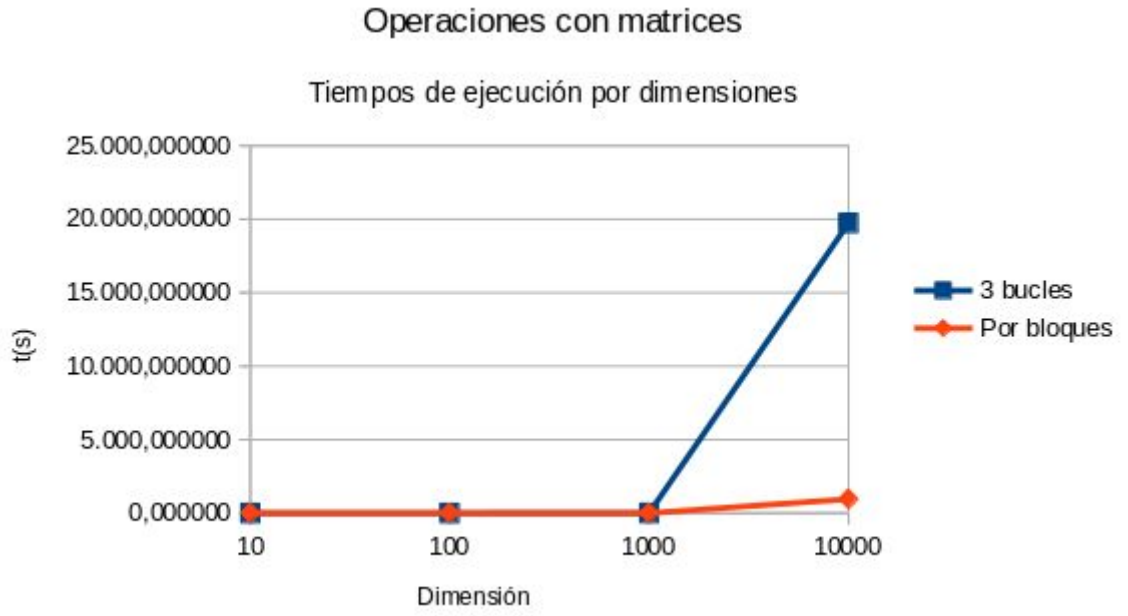
Análisis

El programa se ha ejecutado para matrices de tamaños 10^2, 100^2, 1000^2 y 10000^2, usando diferentes formas de optimizar su funcionamiento. Las primeras dos pruebas han sido prácticamente instantáneas y ha sido a partir del 1000^2 donde se ha podido hacer un análisis más específico sin tener que recurrir a mediciones avanzadas. En un inicio el proceso de multiplicación ha sido el algoritmo con tres bucles para ir rotando la matriz resultado con las dos a multiplicar. La ejecución se ha demorado de forma exponencial usando este metodo, asi que, conservandolo para comparaciones, se ha implementado un nuevo algoritmo por bloques..

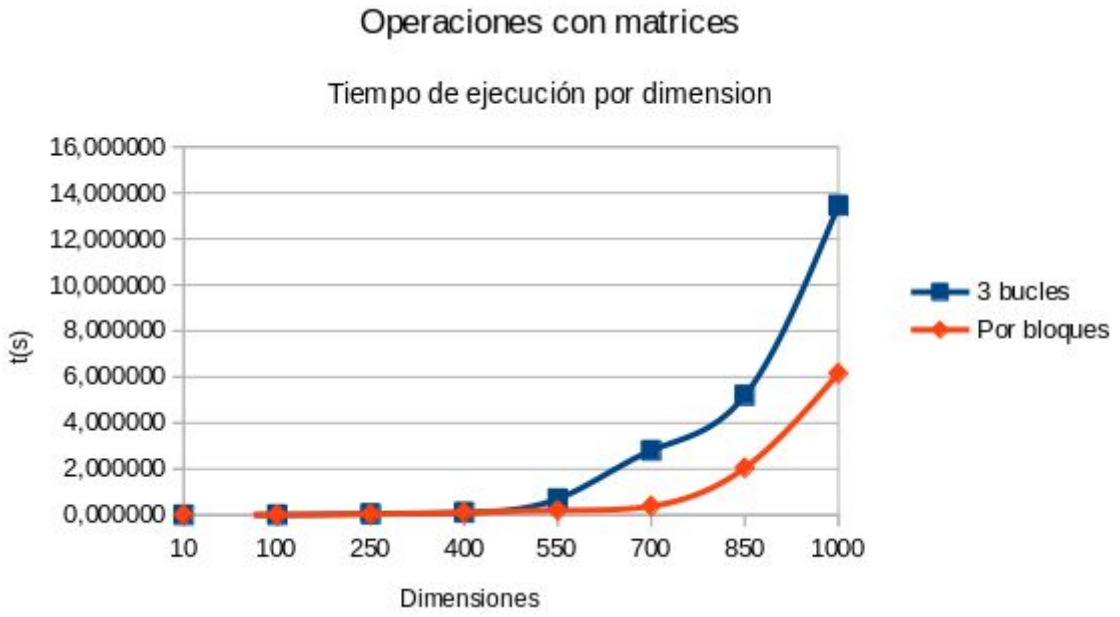
Gracias a conocimientos de álgebra lineal, se ha optado por el método de obtener el producto por bloques. Distintas fuentes de veracidad comprobada de internet indican que esto se debe a que la caché del procesador se llena tras almacenar un proceso tan grande como este. Para ello se ha decidido dividir la matriz en trozos más pequeños asignando un bucle más por cada bloque de matriz, obteniendo un total de 5 bucles. Pese a que la cantidad de bucles haya aumentado, la cantidad de accesos a memoria que se hacen se no aumenta. Al usar mejor el tamaño de la caché, el proceso acelera, sobre todo en matrices de dimensiones grandes.

Además de todo esto, se ha reducido el tiempo de ejecución optimizando la compilación. Se ha compilado añadiendo el parámetro -DNDEBUG para evitar la creación de código de debug, ya que es mucho más lento que el de release. También se ha recurrido al uso de la optimización al nivel 3 con el parámetro -O3, ya que no se confía en las situaciones indefinidas, y es seguro optar por ello. Con estas mejoras, en matrices de 1000^2, se han obtenido mejoras de hasta 4 segundos usando el cálculo de potenciales de iteración lineal.

Los tiempos de ejecución son los siguientes, en comparación a los dos siguientes modos: potencial de tres bucles y potencial por bloques.



Cogiendo el tiempo de la ejecución de la matriz de 10000^2, no se puede precisar nada desde el gráfico, porque los valores escalan demasiado (ejecución de 5 horas), así que se eliminará este valor del diagrama y se añadirán más para tener una distribución en el eje Y más uniforme.



En este gráfico se puede observar ya de una manera más explícita el escalado que se obtiene con ambos algoritmos. Cuando se calcula mediante tres bucles, en dimensiones como 700^2 ya se sobrepasa de los dos segundos cuando el algoritmo de bloques no lo sobrepasa hasta llegados a las dimensiones de 1000^2. Esta comparación, pese a no ser necesaria para la evaluación de la tarea, ha resultado un tanto interesante, y se ha concluido en incluirlo.

Conclusiones

Pese a, a nuestro parecer, tratarse de un ejercicio enfocado a la programación eficiente y estudio de accesos, el cuello de botella que se ha identificado ha sido un problema de algoritmia ligado al uso de la caché del CPU.
Con esto se ha observado que pese a centrarse en optimizar el programa con un uso directo de clases primarias como son los array de espacio fijo, sin tener en cuenta el modo de compilación, los accesos a memoria o el uso de los recursos del equipo, se puede llegar a un punto donde no haya más mejora.
Por eso se ha concluido que a la hora de programar, no solo hay que tener en cuenta las latencias causados por bucles y demás código que ralentiza la ejecución, sino que también hay que tener en cuenta el uso que se le da a los recursos usados del equipo donde se ejecuta.