Sistemas Paralelos - TP 1 - Optimización de algoritmos secuenciales

Autores

- Juan Cruz Cassera Botta, 17072/7
- Lautaro Josin Saller, 18696/9

Especificaciones de sistemas utilizados

Equipo hogareño

- Sistema Operativo: Linux, distribución Ubuntu 24.04.1 LTS.
- CPU: Intel i7-8565U, 4 cores, 8 threads, 1.80 GHz base clock, 4.60 GHz turbo clock.
    - Cachè L1 Data: 128 KiB (4 instancias).
    - Cachè L1 Instruction: 128 KiB (4 instancias).
    - Cachè L2: 1 MiB (4 instancias).
    - Cachè L3: 8 MiB (1 instancia).
- Memoria RAM: 16 GB.
- Almacenamiento: WD NVME SN520 NVME 512GB.

Cluster (usaremos el Multicore Blade)

- Nodos: 16.
- Cada nodo posee 8GB de RAM y 2 procesadores Intel Xeon E5405 de cuatro núcleos cada uno a 2.0GHz.
- Cachè L1 DATA: 32K.
- Cachè L1 Instruction: 32K.
- Cachè L2: 6144K.

Ejercicio 1

Resuelva el ejercicio 4 de la Práctica Nº 1 usando dos equipos diferentes:

1. **Cluster remoto**
2. **Equipo hogareño al cual tenga acceso con Linux nativo (puede ser una PC de escritorio o una notebook).**

A) El algoritmo quadratic1.c computa las raíces de esta ecuación empleando los tipos de datos float y double. Compile y ejecute el código. ¿Qué diferencia nota en el resultado?

Resultado ejecución en máquina local

	Resultado 1	**Resultado 2**
Double	2.00032	1.99968
Float	2.00000	2.00000

Resultado de ejecución en Cluster

	Resultado 1	**Resultado 2**
Double	2.00032	1.99968
Float	2.00000	2.00000

Double tiene mayor precisión.
El resultado de ejecución en el cluster fue idéntico al de nuestra máquina local.

B) El algoritmo quadratic2.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución?

Resultado ejecución en máquina local

Usamos optimización -O3 de gcc.

TIMES	Double	Float
50	0.744823	0.630282
100	1.489232	1.252080
150	2.236756	1.890377
300	4.469543	3.753361
600	8.940716	7.502537

En todos los casos, Double tardó más tiempo y a medida que TIMES aumenta, el tiempo parece aumentar de forma lineal.

Resultado de ejecución en Cluster

TIMES	Double	Float
50	3.291320	2.996373
100	6.600459	5.987980
150	9.977142	8.975664
300	19.775044	17.953704
600	41.212935	35.833170

MOTIVO: En este programa, computar la solución usando Double tarda más tiempo que usando Float debido a que Double es un tipo de dato de 64 bits y Float de 32. Que sea del doble de tamaño implica que se usan más ciclos de CPU y que los datos del arreglo mientras se está procesando probablemente se vuelven demasiado grandes para caber en caché.

C) El algoritmo quadratic3.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución? ¿Qué diferencias puede observar en el código con respecto a quadratic2.c?

Diferencias en el código entre quadratic2.c y quadratic3.c:

- En quadratic3.c las constantes A B C están separadas entre float y double (FA; DA | FB; DB | FC; DC).
- Para hacer los cálculos de las potencias y raíces cuadradas, quadratic3.c usa las funciones ***powf*** y ***sqrtf*** respectivamente, **que están optimizadas para trabajar con valores float, en vez de double**.

Usamos optimización -O3 de gcc.

Resultado ejecución en máquina local

TIMES	Double	Float
50	0.625500	0.433745
100	1.235698	0.856239
150	1.847674	1.286391
300	3.706634	2.593702
600	7.370478	5.132100

El tiempo de ejecución con float sigue siendo menor que el de double, como antes. Además, los tiempos de quadratic3.c son menores que los de quadratic2.c. Esto es porque ya no se están haciendo conversiones implícitas de tipo (de float a double) de forma innecesaria como antes (esto ocurría en quadratic2.c cuando al arreglo de floats se le asignan valores “1.0” que son literales de double y no float, ya que no tienen la **f** al final, y también cuando se le pasan valores float a pow y sqrt, que esperan double, lo cual causa conversiones de tipo implícitas también).

Resultado ejecución en Cluster

TIMES	Double	Float
50	3.424575	1.688062
100	6.560412	3.436770
150	9.839409	5.154045
300	20.497089	10.105344
600	39.293804	20.774240

El script que usamos para este ejercicio fue script-ej1.sh (tiene otro nombre en el cluster).

Utilizamos un solo script pero lo fuimos modificando cambiando el nombre de los archivos output y error y el nombre del ejecutable con distintos valores de TIMES.

Ejercicio 2

Desarrolle un algoritmo en el lenguaje C que compute la siguiente ecuación:

$$R = \frac{(MaxA \times MaxB - MinA \times MinB)}{PromA \times PromB} \times [A \times B] + [C \times B^T]$$

- **Donde A, B, C y R son matrices cuadradas de NxN con elementos de tipo double.**
- **MaxA, MinA y PromA son los valores máximo, mínimo y promedio de la matriz A, respectivamente.**
- **MaxB, MinB y PromB son los valores máximo, mínimo y promedio de la matriz B, respectivamente.**
- B^T **es la matriz transpuesta de B.**

Mida el tiempo de ejecución del algoritmo en el cluster remoto. Las pruebas deben considerar la variación del tamaño de problema (N={512, 1024, 2048, 4096}). Por último, recuerde aplicar las técnicas de programación y optimización vistas en clase.

Análisis inicial del problema

- Tenemos que crear e inicializar 4 matrices cuadradas de NxN con valores de tipo double (A, B, C, R), y 2 matrices más de ese estilo para albergar resultados intermedios: mul1 y mul2.
- Para R, se inicializa con todos sus valores en 0.0, ya que almacenará el resultado.
- Para la inicialización de A B y C, primero pensamos asignar 1.0 a todas las celdas ya que sería lo más rápido y sencillo, pero nos damos cuenta que tener una matriz con todos sus valores en 1 haría que el mínimo, máximo y el promedio pierdan sentido (los tres valdrían siempre 1).
- Por esto, decidimos asignar valores **random** de 1 a 4 a las matrices, usando la función rand.
- A la hora de calcular los máximos, mínimos y promedios, pensamos en realizar los 3 cálculos en un mismo bucle para las 3 matrices (A B C). De esta manera evitamos desperdiciar ciclos de CPU haciendo varios loops.

- Pensamos en organizar las matrices A, B y C por filas. Por lo tanto, para acceder a sus elementos usaremos el índice **i * N + j** (siendo i el número de fila y j el número de columna).

Cómputo de la ecuación

- Para empezar, vamos a resolver la ecuación y verificar que el resultado sea correcto.
- Una vez tenemos una solución funcional, veremos cómo se puede optimizar aplicando las técnicas aprendidas en clase.
- Resolución inicial:
    a. Obtener los mínimos, máximos y promedios y realizar la división y guardar el resultado en una variable auxiliar (**cociente**).
    b. Resolver $[A \times B]$ y guardarlo en una matriz auxiliar **mul1 y** resolver $[C \times B^T]$ y guardarlo en una matriz auxiliar **mul2** (todo en un mismo bucle).
    c. Finalmente multiplicar la matriz auxiliar **mul1** por **cociente** y sumarle a eso la matriz **mul2**.
    d. El resultado final se asigna en la matriz **R.**
- Para los argumentos, enviamos el N para el tamaño de las matrices, y un 0 o 1 indicando si se quiere imprimir las matrices en pantalla o no.
- El tiempo de ejecución se evalúa **desde a) hasta d),** no incluye los prints, ni asignación de memoria, ni inicializaciones, ni liberación de memoria
- Si bien esta primera solución no está optimizada, usamos el **flag -O3** al momento de compilar para agilizar la ejecución.
- El código de esta solución inicial se encuentra en el archivo **ejercicio2-sin-optimizar.c.**

Optimización

- Una vez obtuvimos una solución inicial, nos interesa optimizarla para que tome la menor cantidad de tiempo posible en ejecutar.
- Nuestro algoritmo ya inicializa las matrices como un arreglo dinámico como vector de elementos. De esta manera los datos están contiguos en la memoria y podemos acceder a ellos por fila o columna. Por lo tanto, este aspecto del código no requiere más optimización.

- **Técnicas utilizadas para mejorar el tiempo de ejecución** :
    - Crear e inicializar una nueva matriz B_T que será la matriz B pero con sus valores transpuestos. Esto nos permite acceder a B_T por filas en vez de por columnas, lo cual mejora la localidad de datos y por ende mejora la eficiencia al realizar $[C \times B^T]$.
    - Usar la técnica de multiplicación por bloques vista en la práctica 1, donde realizamos las multiplicaciones de a bloques pequeños que caben en la caché para minimizar accesos a memoria principal.
        - El tamaño de bloque se envía como argumento.
    - Tener en cuenta el tamaño del bloque óptimo, según las especificaciones de caché del sistema.
        - Probaremos con tamaños de bloque 32, 64 y 128.
    - Optimización del compilador: usamos el flag **-03** al momento de compilar el archivo, de esta manera obtenemos el máximo rendimiento posible.
    - Usar una variable auxiliar acumuladora dentro de la función **multiplicar_bloque** para evitar accesos continuos a memoria y en su lugar usar registros de la CPU.
- El código de esta solución optimizada se encuentra en el archivo **ejercicio 2-optimizado.c.**
- El script que usamos para este ejercicio fue script-ej2.sh (tiene otro nombre en el cluster).
- Utilizamos un solo script pero lo fuimos modificando cambiando el nombre de los archivos output y error y los parámetros recibidos por el ejecutable: tamaño de N y si se quiere imprimir o no para el no optimizado, y estos dos + tamaño de bloque para el optimizado.

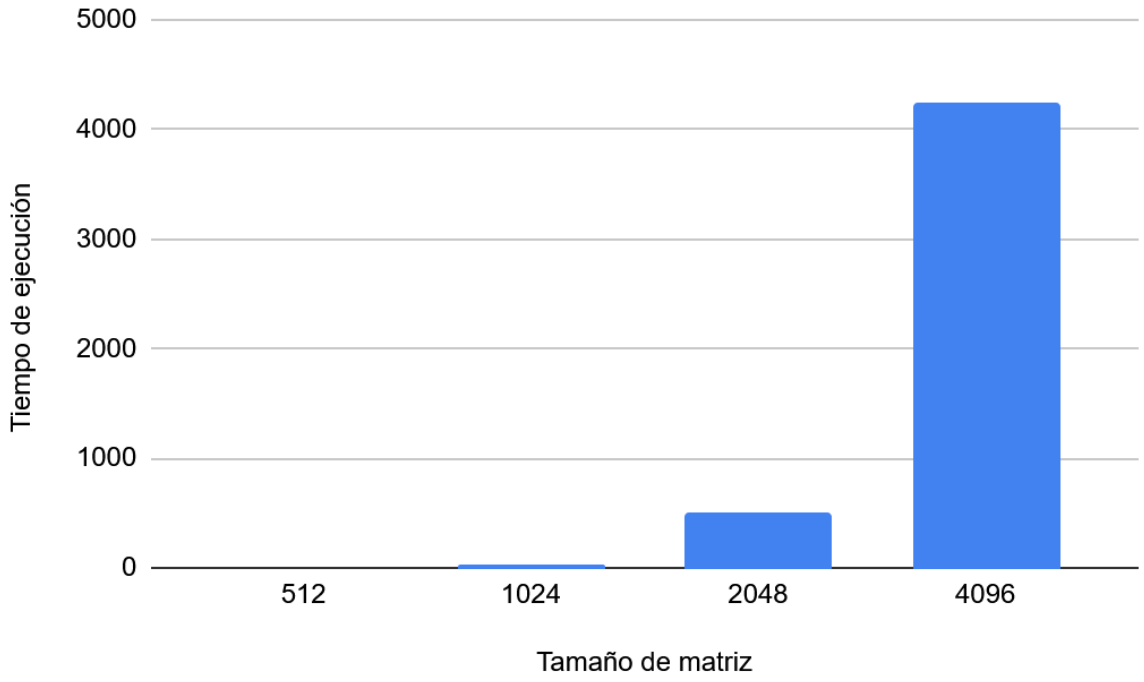
Tiempos de ejecución

Para compilar los programas .c se usaron los siguientes comandos:

- ***gcc -O3 -o ejercicio2-sin-optimizar ejercicio2-sin-optimizar.c -Wall***
- ***gcc -O3 -o ejercicio2-sin-optimizar ejercicio2-optimizado.c -Wall***

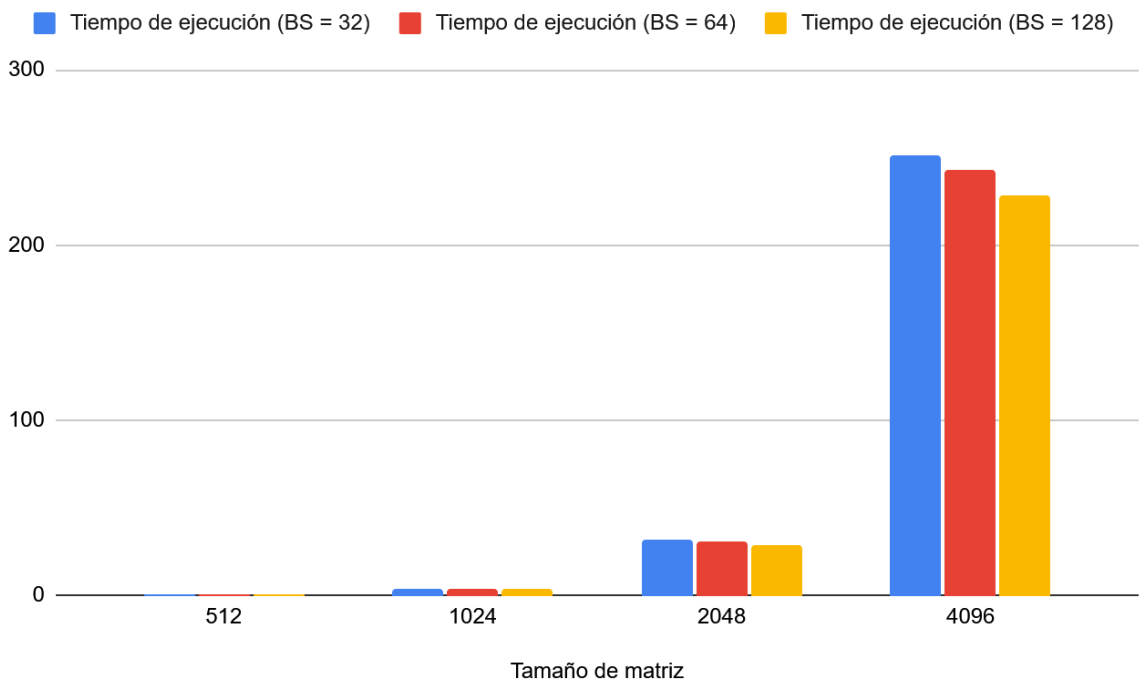
Resultado tiempos de ejecución en Cluster - Sin optimización

Tamaño de matriz	Tiempo de ejecución
512	0.841617
1024	28.599863
2048	509.214289
4096	4245.012602



Resultado tiempos de ejecución en Cluster - Optimizado con multiplicación por bloques (BS = Block Size)

Tamaño de matriz	Tiempo de ejecución (BS = 32)	Tiempo de ejecución (BS = 64)	Tiempo de ejecución (BS = 128)
512	0.492638	0.460000	0.444297
1024	3.997085	3.740928	3.620320
2048	31.659407	30.836544	28.615178
4096	250.859796	243.366505	228.208801



Conclusión sobre el tamaño de bloque

Podemos observar que el tamaño de bloque que mejor resultado nos dió es 128. Para analizar por qué, primero podemos ver que cada bloque pesa, en memoria, 128 (filas) * 128 (columnas) * 8 bytes (tamaño de un double en C) = 131072 bytes, lo que equivale a 131 Kb.

Teniendo en cuenta el tamaño de bloque calculado en nuestro programa y el tamaño de la caché del cluster donde se ejecutó el algoritmo, tiene sentido que los tiempos hayan mejorado ya que se aproximó el tamaño de bloque “formado” (131 kb) al tamaño de bloque máximo de la caché. Produciendo así una mayor cantidad de hits.

Tener en cuenta que el tamaño de la caché L2 del cluster es de 6144K. En base a los resultados, asumimos que el cluster está usando la caché L2 ya que es la que más se ajusta al tamaño de bloque que calculamos (el bloque no entra en la caché L1d ya que esta tiene 32K).