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.

Explicación de la solución en general

- El código de esta solución se encuentra en el archivo **ejercicio 2-optimizado.c.**
- Utilizamos un solo script pero lo fuimos modificando cambiando el nombre de los archivos output y error y los parámetros recibidos por el ejecutable:
  - **a.** Tamaño de N para el no optimizado.
  - **b.** Tamaño de N + tamaño de bloque para el optimizado.
- Argumentos:
  a. N: tamaño de las matrices $N \times N$.
  b. Tamaño de bloque para la multiplicación por bloques.
  c. NOTAS:
    - N y tamaño de bloque deben ser mayor a 0.
    - N debe ser mayor a tamaño de bloque.
    - N debe ser múltiplo de tamaño de bloque.
- Declaramos 7 matrices de tipo double y de tamaño NxN (cuadradas):

a. Matriz **A**.
b. Matriz **B**.
c. Matriz **B_T**.
d. Matriz **C**.
e. Matriz **a_por_b** que alberga el resultado de multiplicar $[A \times B]$.
f. Matriz **c_por_bt** que alberga el resultado de multiplicar $[C \times B^T]$.
g. Matriz **R** que alberga el resultado final, es decir: $Cociente \times [A \times B] + [C \times B^T]$.

- Luego de alocar memoria ([N * N * tamaño de un double] bytes) a todas las matrices, éstas se inicializan de la siguiente manera:
  a. A y C son matrices **identidad** ordenadas por **filas**
  b. B es una matriz ordenada por **columnas** con **valores incrementales**, es decir 1 2 3 4 5...
  c. B transpuesta está ordenada por **columnas** y se inicializa transponiendo la matriz B, pero **esto se hace una vez se empieza a medir el tiempo y no antes**.
  d. R, a_por_b y c_por_bt están ordenadas por **filas** y se inicializan con todas sus celdas en 0.
- Empezamos a medir el tiempo y a realizar los cálculos:
  a. Inicializamos B_T.
  b. Calculamos máximo, mínimo y promedio de A.
  c. Calculamos máximo, mínimo y promedio de B.
  d. Calculamos el cociente.
  e. Calculamos $[A \times B]$.
  f. Calculamos $[C \times B^T]$.
  g. Calculamos $Cociente \times [A \times B] + [C \times B^T]$.
  h. Terminamos de medir el tiempo y lo imprimimos.
- Chequeamos que los resultados que obtuvimos sean correctos.
  a. Chequeamos que el resultado de $[A \times B]$ es correcto verificando que $[A \times B] = B$, ya que A es una matriz identidad.
  b. Chequeamos que el resultado de $[C \times B^T]$ es correcto verificando que $[C \times B^T] = B^T$, ya que C también es una matriz identidad.
  c. Chequeamos que el cociente es correcto sabiendo que:
     - El numerador $[MaxA \times MaxB - MinA \times MaxB]$ tiene que ser igual a $(1 * (N * N)) - (0 * 1)$ debido a cómo inicializamos A y B.

        - El denominador $PromA \times PromB$ tiene que ser igual a $(1.0 / N) * ((N * N + 1) / 2.0)$, nuevamente debido a cómo inicializamos A y B.
        - Usamos un epsilon y una resta con valor absoluto para evitar errores de redondeo con los decimales.
    d. Chequeamos que $Cociente \times [A \times B] + [C \times B^T]$ es correcto.
- Liberamos la memoria alocada a las matrices al principio.

Explicación de las optimizaciones aplicadas en la solución

- Declaramos las matrices como arreglos dinámicos de elementos para asegurar que los datos están contiguos en la memoria y podemos acceder a ellos por fila o columna para aprovechar la localidad de datos.
- Inicializamos a B **ordenada por columnas** intencionalmente, ya que la multiplicación de matrices, matemáticamente, es fila por columna. Esto aprovecha la localidad también, a la hora de acceder a cada columna de B cuando hacemos la multiplicación $[A \times B]$.
- Inicializamos B_T accediendo a B por filas (de esta forma transponemos B).
- Recorremos a la matriz A y a la matriz B, para hallar sus máximos, mínimos y promedios, en bucles for distintos, para aprovechar el uso de la caché (si usamos un solo for quizá no caben ambas matrices en la caché).
    - Además, usamos variables auxiliares **celda_A** y **celda_B** que almacenan el valor de la celda actual de la matriz a medida que la recorremos, para aprovechar los registros de la CPU y minimizar los accesos a memoria (accedemos a memoria una sola vez por cada celda de la matriz, en vez de 5 veces).
- Nuevamente realizamos las multiplicaciones $[A \times B]$ y $[C \times B^T]$ en bucles for separados por la misma razón mencionada.
    - Usamos la técnica de multiplicación por bloques vista en la práctica 1, donde realizamos la multiplicación de a bloques pequeños que caben en la caché (si el tamaño de bloque es el adecuado) para minimizar accesos a memoria principal.
        - Dentro de la función que multiplica los bloques, usamos una variable auxiliar **aux** que acumula la suma de cada fila *

columna de cada bloque, para minimizar drásticamente los accesos a memoria de **bloque_resultado** y en su lugar aprovechar los registros de la CPU.
    - Tener en cuenta el tamaño del bloque óptimo, según las especificaciones de caché del sistema: Probaremos con tamaños de bloque 32, 64 y 128.
- Usamos el **flag -O3** al momento de compilar para agilizar la ejecución.

Solución no optimizada

Para tener una idea de qué tanto afectan las optimizaciones que realizamos, creamos una copia de nuestra solución optimizada pero quitando todas las optimizaciones. Es decir, el resultado termina siendo el mismo pero tarda mucho más en ejecutar.

El código de esta solución se encuentra en el archivo **ejercicio 2-sin-optimizar.c.**

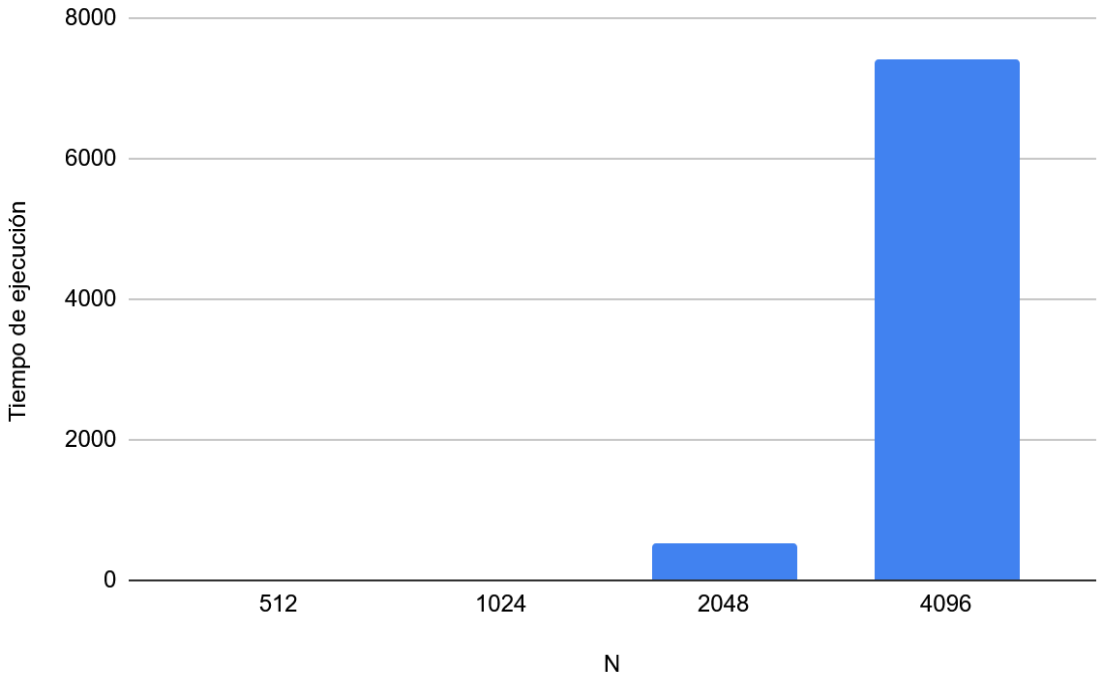
Tiempos de ejecución

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-optimizado ejercicio2-optimizado.c -Wall***

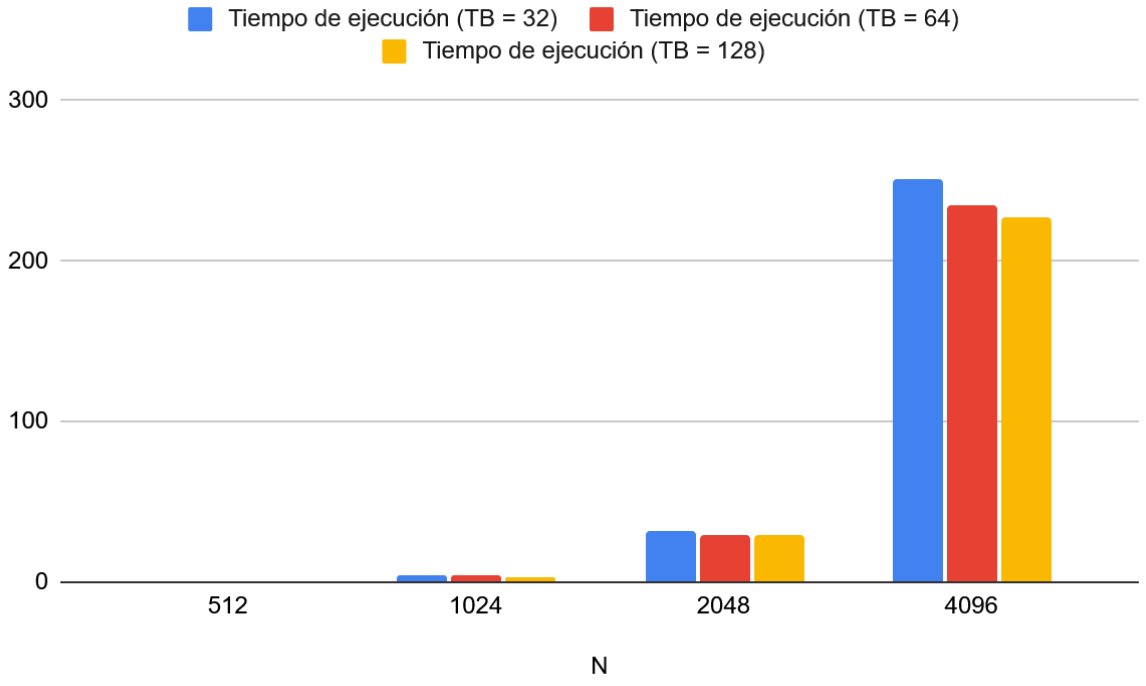
Resultado tiempos de ejecución en Cluster - Sin optimizar

N	Tiempo de ejecución
512	0.777616
1024	18.585473
2048	519.636418
4096	7418.386158



Resultado tiempos de ejecución en Cluster - Optimizado con multiplicación por bloques (TB = Tamaño de Bloque)

N	Tiempo de ejecución (TB = 32)	Tiempo de ejecución (TB = 64)	Tiempo de ejecución (TB = 128)
512	0.497928	0.462729	0.443879
1024	4.003373	3.743643	3.615172
2048	31.716992	29.509436	28.767633
4096	251.421113	234.357358	227.541547



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).