SISTEMAS PARALELOS - 2025

Trabajo Entregable N°1 - Optimización de algoritmos secuenciales

Equipamiento:

- El equipo (PC) general utilizado de forma local cuenta con una memoria RAM de 8GB y un procesador AMD Ryzen 3 3200G [Ficha técnica], usando Kubuntu 24.04.
- Clúster de cómputo proveído por el III-LIDI

Aclaración/Resolución:

- Para el inciso A no se aprovecha la optimización.
- El equipo “hogareño” es llamada de forma sinónima en el trabajo a “Local”.

Sobre optimizadores:

- Se usa el compilador gcc generando analogías de eficiencia.
- GCC provee una serie de flags que permiten configurar su eficiencia:
  - Más información respectiva a los flags y su clasificación

Ejercicio 1

Resolver utilizando: (1) cluster remoto y (2) equipo hogareño al cual tenga acceso con Linux nativo:

Dada la ecuación cuadrática: $x^2 - 4.0000000\,x + 3.9999999 = 0$, sus raíces son r_1 = 2.000316228 y r_2 = 1.999683772 (empleando 10 dígitos para la parte decimal).

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?
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?
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?*

Nota: agregue el flag *-lm* al momento de compilar. Pruebe con el nivel de optimización que mejor resultado le haya dado en el ejercicio anterior.

(a). Precisión de las raíces con `quadatric1.c`

Tipo de dato	Sevidor	Local	Observación
`float`	2.00000	2.00000	Menor precisión, redondeo/truncamiento evidente
`double`	2.00032	1.99968	Mayor precisión, más cercano al valor real

a. Conclusiones:
    i. Se puede apreciar que el resultado de una operación con el tipo float pierde precisión utilizando la función pow().
    ii. Lo ideal para aprovechar la mayor precisión es utilizar variables de tipo Double (justamente refiere su nombre a la doble precisión).

(b). Tiempo de ejecución con `quadatric2.c`

`TIMES`	Double (local)	Float (local)	Double (servidor)	Float (servidor)	Observación
20	5.97s	6.54s	9.23s	9.19s	Diferencia pequeña en tiempos
100	29.97s	32.74s	45.13s	45.63s	`Float` es apenas más lento
200	60.12s	65.64s	90.24s	93.55s	`Double` mantiene ligera ventaja
250	75.27s	82.08s	112.79s	117.05s	A medida que `TIMES` aumenta, la diferencia en tiempos crece

Analogía de procesamiento en el peor caso ejemplificado utilizando optimizadores

Usando diferentes optimizadores del compilador (`-O1`, `-O2`, `-O3`), comparando tiempos para tipos `float` y `double`, tanto en **máquina local** como en el **servidor** (TIMES=250):

Optimizador	Local - Tiempo Double (s)	Local - Tiempo Float (s)	Servidor - Tiempo Double (s)	Servidor - Tiempo Float (s)
`-O1`	1.493598	1.424854	4.767376	7.752602
`-O2`	1.567516	1.380667	4.265237	7.174260
`-O3`	1.526949	1.392410	4.259810	7.047899

b. Conclusiones:

i. Al invocar repetidas veces (aumentar valor de var. TIMES) la función pow() con args. de tipo float requiere múltiples casting de un tipo de dato a otro, causando un retardo notable en la ejecución.
ii. Los resultados aplicando optimizadores vemos que generan una mejora radical de los tiempos de ejecución en cada caso.

Ej. (3) Tiempo de ejecución con `quadatric3.c`

`TIMES`	Double (local)	Float (local)	Double (servidor)	Float (servidor)	Observación
20	5.99s	4.29s	9.02s	13.82s	`Float` más rápido en local, pero más lento en servidor
100	30.00s	21.53s	45.10s	69.13s	`Float` notablemente más lento en servidor
200	60.44s	43.00s	90.20s	138.26s	La diferencia en tiempos aumenta considerablemente
250	75.36s	53.93s	112.83s (Blade)	172.88s (Blade)	En servidores potentes, `Double` es más eficiente

---

Analogía de procesamiento en el peor caso ejemplificado utilizando optimizadores

Usando diferentes optimizadores del compilador (`-O1`, `-O2`, `-O3`), comparando tiempos para tipos `float` y `double`, tanto en **máquina local** como en el **servidor** (TIMES=250):

Optimizador	Local - Tiempo Double (s)	Local - Tiempo Float (s)	Servidor - Tiempo Double (s)	Servidor - Tiempo Float (s)
`-O1`	3.800860	2.397869	4.786592	3.914533
`-O2`	3.789862	2.151910	4.236103	3.731060
`-O3`	3.810149	2.120922	4.232938	3.735020

c. Conclusiones:

i. El mejor optimizador parece ser por las pruebas analizadas en el inciso (b) y (c) -O3 dado que genera un menor tiempo promedio en las pruebas realizadas al ejecutar los 2 algoritmos.
ii. Apreciamos que el algoritmo aprovecha una función específica para el cálculo de potencias de valores tipo float, resultando en una ejecución más eficiente al operar con este tipo de variables(gracias al uso de powf() y ahorrar los casteos de tipo que hacía en el punto anterior).

Resolución del segundo ejercicio:

Se eligieron los flags de optimización -O1, -O2 y -O3 para analizar el rendimiento, el primer objetivo es encontrar cual block size (bs = 16, 32, 64, 128) es ideal a nivel del cluster MultiCore y luego los cálculos a partir de diferentes tamaños de matrices (N =512,1024,2048, 4096) con diferentes optimizaciones.

Al momento de desarrollar el algoritmo para mantener los tiempos de ejecución al mínimo lo mejor es aprovechar una estructura de datos para manejar las matrices que nos permita los siguientes puntos: tener un tamaño máximo por la forma en que está declarado, poder cambiar su tamaño en ejecución, poder elegir cómo se organizan sus datos (por filas o por columnas), tener todos sus datos contiguos en la memoria, así que no podemos tomar un arreglo estatico o dinamico, podríamos elegir un arreglo dinámico como vector de punteros a filas/columnas pero no están todos sus datos contiguos en memoria así que se pierde velocidad al momento de acceder a estos valores pero tenemos una última opción: float * matriz = malloc(N*N*sizeof(float)), un arreglo dinámico como vector de elementos el cual tendremos que elegir un método para acceder a sus elementos siendo por filas: matriz [i*N+j] y por columnas: matriz [j*N+i].

Luego necesitamos una forma de aprovechar la caché del sistema para procesar los datos tomando en cuenta la estructura que íbamos a necesitar y la más eficiente sería por bloques al enfocarse en utilizar submatrices (bloques) para aprovechar el acceso y disminuir los fallos de página durante los accesos a elementos de matrices y para esto requerimos definir un block size (bs) adecuado, el tamaño de este depende de la arquitectura del CPU que lo esté ejecutando, otra condición que debe cumplir el mismo es que debe ser múltiplo del tamaño de la matriz total, es decir N mod BS = 0 y en esta estructura por bloques intentamos en lo máximo posible hacer un acceso lineal a los datos de cada matriz.

Guardamos resultados previamente calculados para evitar tener que hacer operaciones de más, por ejemplo cuando se calculan los máximos, mínimos y promedios de las matrices A y B para luego acceder a ellos en tiempo constante cuando está ejecutando la multiplicación de matrices y de esta forma aprovechar a su vez que ya se están obteniendo los valores por

los cuales se deberá multiplicar el escalar obtenido y evitar hacer más bucles para acceder a esos valores.

Se hizo lo posible para evitar que el tiempo de ejecución se viera afectado por alocación y liberación de memoria, impresión en pantalla (printf), inicialización de estructuras de datos o impresión y verificación de resultados.

En la **figura 1.1** tenemos una gráfica que nos muestra el rendimiento del cluster MultiCore sobre las distintas optimizaciones elegidas con 1024 de tamaño de matrices N y los bs para conseguir el más apropiado (que tarde menos tiempo de ejecución).

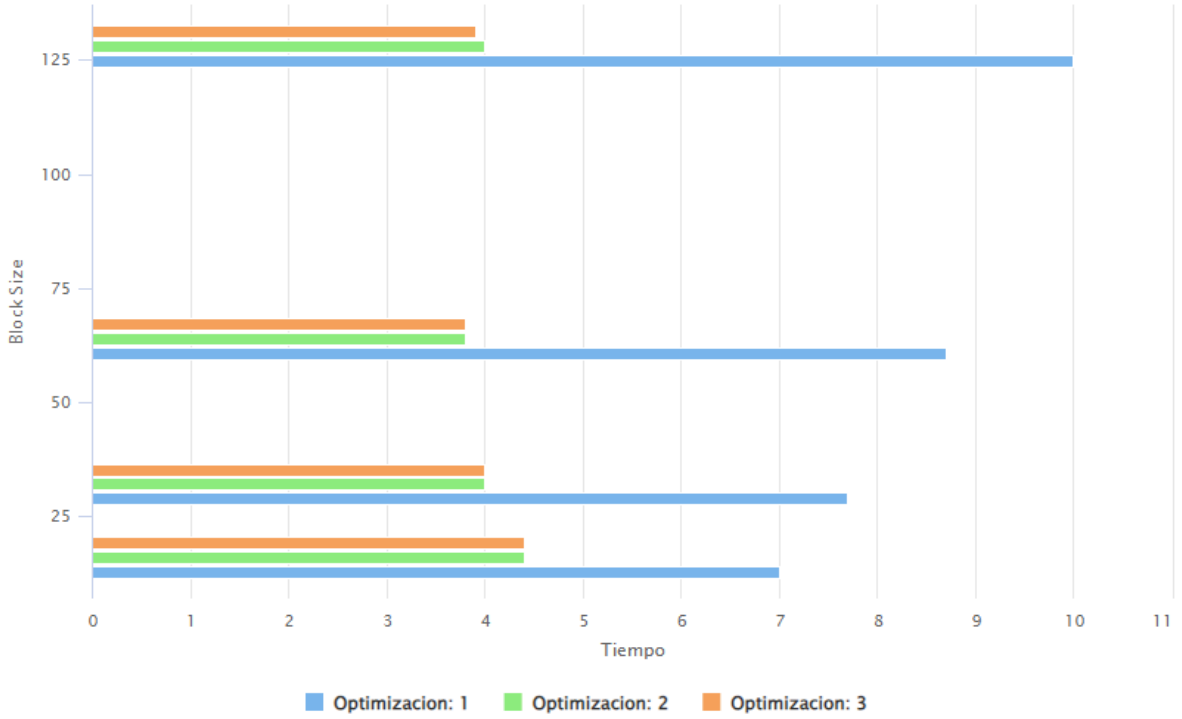

Figura 1.1 comparativa entre los distintos block size ante un mismo tamaño de matriz (1024) entre el Cluster y la PC local con diferentes optimizaciones.

De estos resultados obtenemos que el peor tiempo se lo lleva si ocupamos el -O1 y el mejor para ocupar esta entre el O2 y el O3 que dieron resultados casi idénticos.
Tenemos para mencionar también que la optimización 1 sube de forma significativa su tiempo de ejecución, O2 y O3 se mantuvieron casi idénticos bajando a medida que se aumenta el bs, debido a que en bs = 16 la Optimización 1 tiene su menor valor y en bs=128 O2 junto a O3 tuvieron sus menores valores por lo cual tomaremos esos dos valores para bs (16, 128).

Una vez ya obtenidos los block sizes ahora para el tiempo de ejecución para el cluster MultiCore ya podemos calcular con diferentes tamaños de matrices (512, 1024, 2048, 4096), En la **figura 1.2** tenemos la comparativa entre los tiempos del cluster con 16bs.

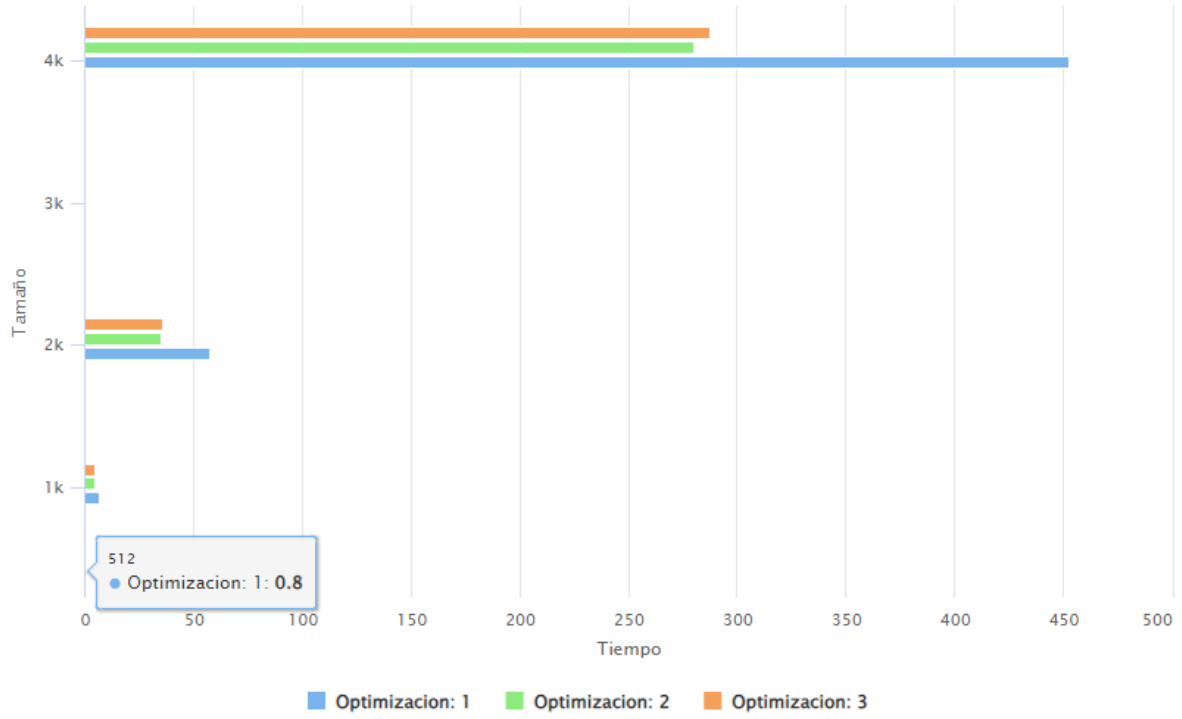

Figura 1.2. Comparativa entre los distintos tamaños de N con respecto al tiempo de ejecución para bs = 16

En esta **figura 1.2** tenemos que notar 2 cosas, la forma en como la optimización que tiene el peor rendimiento en el tiempo es otra vez O1 mientras que O2 va teniendo un mejor rendimiento que O3 ligeramente y que O1 por mucho y también que al ser N = 512 el tiempo de ejecución es tan pequeño en comparación (en especial al N=4096) que no se pueden observar bien pero también allí el más lento fue O1.

Ahora en la figura **1.3** vamos a comparar los tamaños para el bs = 128.

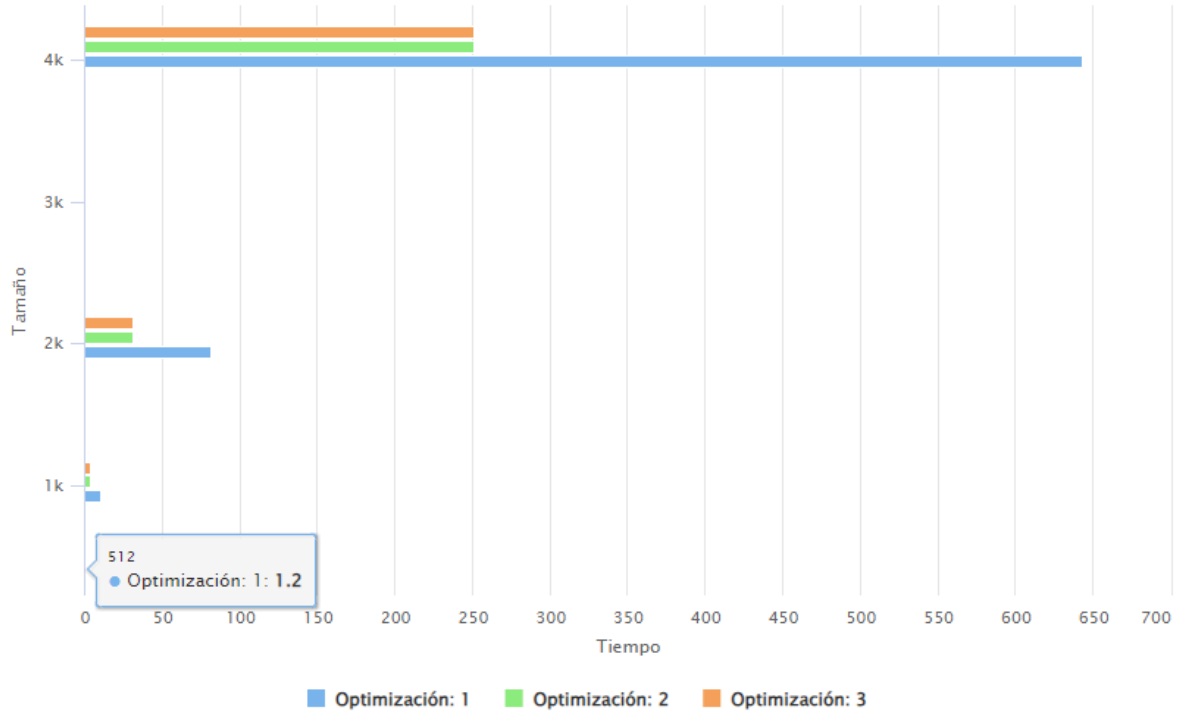

Figura 1.3 Comparativa entre los distintos tamaños de N con respecto al tiempo de ejecución para bs = 128.

Al observar la **Figura 1.2** y la **1.3** se puede notar que el valor más afectado es el O1, el O2 y O3 también se vieron afectados mejorando sus tiempos ocasionando un menor tiempo de ejecución con respecto al de la **figura 1.2** al aumentar el bs de 16 a 128 pero no de forma tan drástica como O1 pero que en su caso fue en sentido contrario, es decir, en lugar de mejorar su tiempo este empeoro causando un mayor tiempo de ejecución.

Se puede observar también un gran crecimiento del tiempo con respecto a la entrada (N) en las figuras **1.2** y **1.3**, esto se debe a que el algoritmo es de O(n^3) por las multiplicaciones de matrices NxN y el resto de operaciones que tienen un O(n^2) como el máximo, mínimo, promedio, potencia y suma de matrices, que aunque sean menores en orden todas contribuyen al tiempo final.