Sistemas Paralelos -2025- Grupo 14 - Ferreyra Coronel, Agustín y Nogueira, Joaquina

Trabajo Práctico Nº 1

Optimización de algoritmos secuenciales

Características del hardware y software utilizados

Equipo con Linux nativo

- Hardware
  - Procesador: Intel (R) Core (TM) i5- 2310 CPU
  - Núcleos: 4
  - Arquitectura: x86_64
  - Memoria RAM: 7.7 GB
- Software
  - Sistema Operativo: Ubuntu 22.04.1
  - Compilador: GCC version 11.3.0

Cluster

- Multicore (partición Blade) conformado por 16 nodos.
- Cada nodo posee 8GB de RAM
- 2 procesadores Intel Xeon E5405 de cuatro 4 cores cada uno que operan a 2.0GHz.

Punto 1

Resuelva el ejercicio 4 de la Práctica Nº 1 usando dos equipos diferentes: (1) cluster remoto y (2) equipo hogareño al cual tenga acceso con Linux nativo (puede ser una PC de escritorio o una notebook).

Ejercicio 4: Dada la ecuación cuadrática: x 2 − 4.0000000 x + 3.9999999 = 0, sus raíces son r1 = 2.000316228 y r2 = 1.999683772 (empleando 10 dígitos para la parte decimal).

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. 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?

Para hacer el test de los ejercicios tanto en la ejecución con linux nativo como en el cluster, elegimos el nivel de optimización -O3 el cual mostró mejores resultados en ambos entornos. Esta optimización habilita todas las optimizaciones disponibles, incluyendo

vectorización y expansión de funciones, que son ideales para operaciones de alto cálculo como la multiplicación de matrices. Es el nivel más alto de optimización posible.

gcc -o salidaEjecutable quadratic1.c -lm -O3

./salidaEjecutable

	Ejecución cluster remoto		Ejecución Linux nativo	
Solución Float	2.00000	2.00000	2.00000	2.00000
Solución Double	2.00032	1.99968	2.00032	1.99968

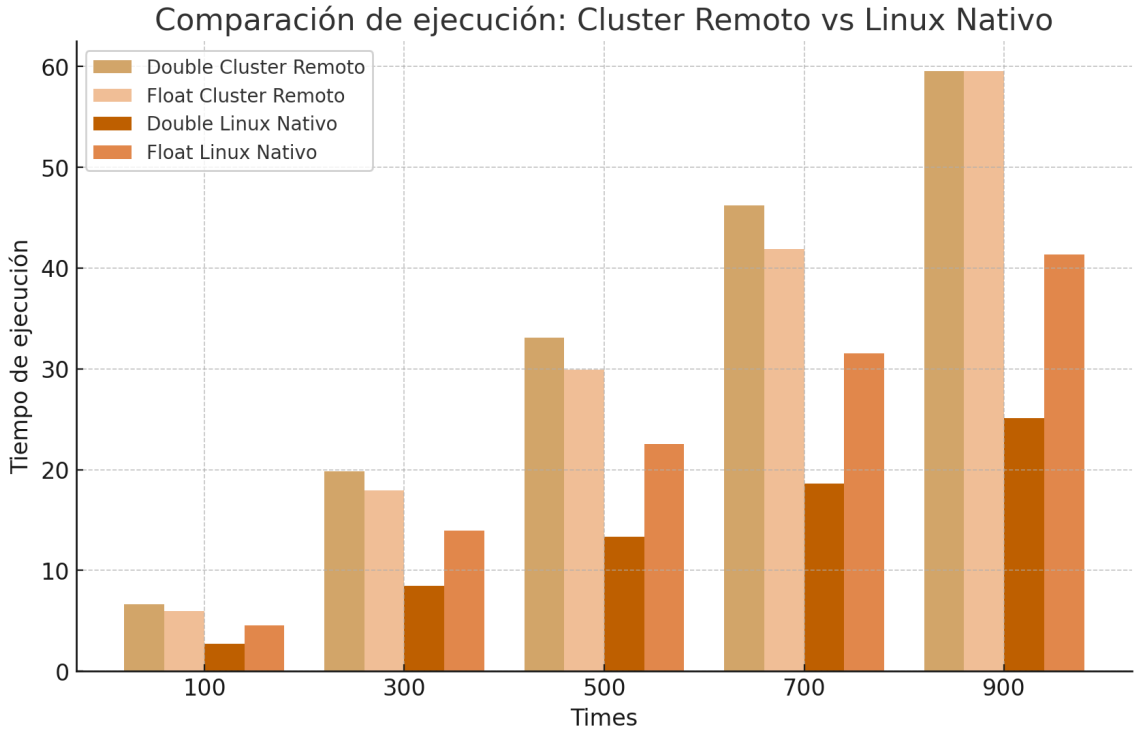
Se puede observar como la solución float redondea y pierde precisión en cambio la solución double es más precisa. Esto se debe a la cantidad de bits que utilizan cada tipo para representar valores. Float utiliza 32 bits y tiene menor capacidad para manejar decimales, mientras que double usa 64 bits, permitiendo mayor precisión.
Los resultados que obtuvimos tanto en el cluster como en el equipo con linux nativo fue el mismo.

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?

TIMES	Ejecución cluster remoto	Ejecución Linux nativo
100	Double: 6.616167 Float: 5.975222	Double: 2.712084 Float: 4.533772
300	Double: 19.853748 Float: 17.954810	Double: 8.484459 Float: 13.919733
500	Double: 33.064950 Float: 29.943480	Double: 13.317772 Float: 22.513854
700	Double: 46.229224 Float: 41.891094	Double: 18.604299 Float: 31.531023
900	Double: 59.537546 Float: 59.537546	Double: 25.127629 Float: 41.343257

Tiempos obtenidos cambiando la constante TIMES y compilando con -O3

Lo que podemos notar es que en el cluster los tiempos con float son mejores a pesar de que quadratic2.c trabaje con las funciones pow() y sqrt() que están preparadas para double. Esto puede ser por varios motivos.
- Los float ocupan la mitad del espacio en memoria que los double, esto significa que en caché pueden caber más elementos reduciendo los accesos a memoria principal.
- La transferencia de datos entre memoria y cpu es más eficiente ya que se pueden mover más valores float en la misma cantidad de tiempo.
- Las operaciones con double pueden requerir más pasos internos para garantizar precisión, lo que ralentiza su ejecución. En cambio, las operaciones con float, al ser menos precisas, pueden completarse más rápido incluso con conversiones.
En cuanto a la ejecución en Linux nativo lo que podemos ver es que a diferencia de los tiempos del cluster, en este ambiente double tiene mejores tiempos respecto a float. Esto podría ser por la arquitectura del hardware, en el cluster hay múltiples nodos y un sistema de caché distribuido. La latencia de acceso a memoria puede ser diferente debido a la arquitectura de múltiples núcleos. En linux nativo, la cpu tiene un mayor rendimiento en cálculos de double precision, favoreciendo a los cálculos double.



Viendo el gráfico de barras podemos observar mejor la diferencia de tiempos que hay en cuanto a la ejecución de un mismo algoritmo en diferentes ambientes. En linux los tiempos de ejecución son menores.

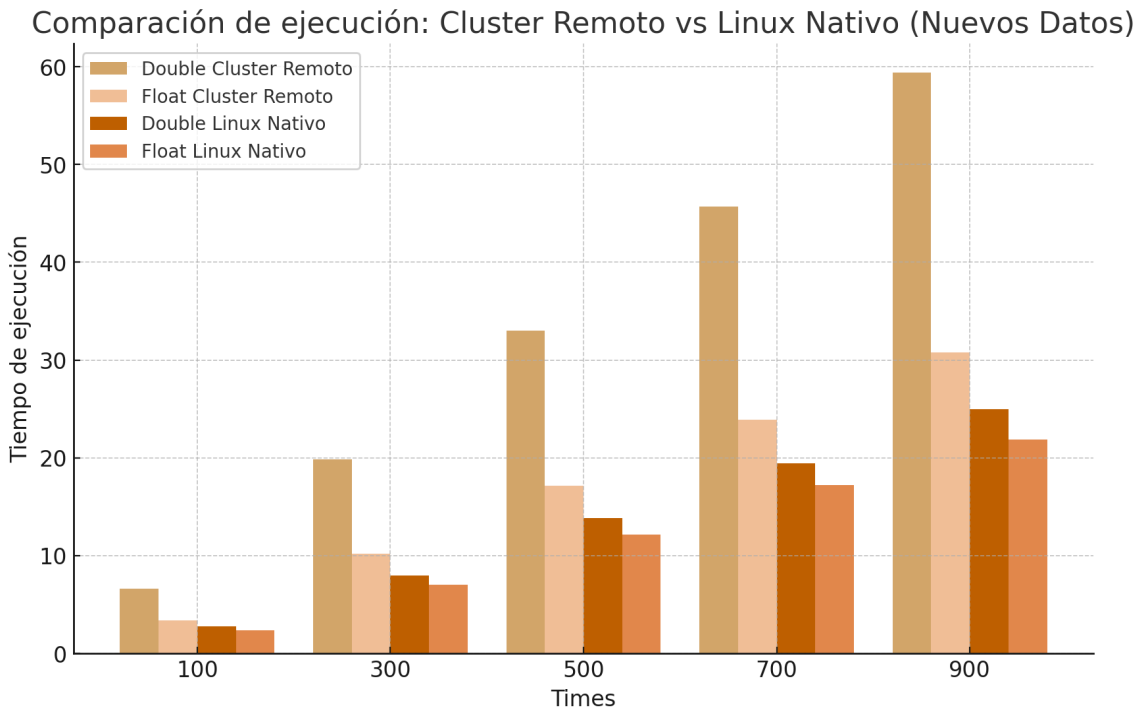
c. El algoritmo quadratic3.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución? ¿Qué diferencias puede observar en el código con respecto a quadratic2.c?

TIMES	Ejecución cluster remoto	Ejecución Linux nativo
100	Double: 6.615937 Float: 3.396658	Double: 2.755859 Float: 2.403522
300	Double: 19.857030 Float: 10.173402	Double: 7.980176 Float: 7.022919
500	Double: 32.979219 Float: 17.176818	Double: 13.872316 Float: 12.137436
700	Double: 45.710713 Float: 23.911458	Double: 19.421114 Float: 17.219294
900	Double: 59.384804 Float: 30.803511	Double: 24.966581 Float: 21.85808

Tiempos obtenidos cambiando la constante TIMES y compilando con -O3

Lo que podemos observar es que float tiene mejor tiempo que double. Mirando el código de quadratic3.c vemos que se utilizan las funciones powf() y sqrtf(), las cuales operan con float, y esto las hace más rápidas ya que no hacen falta conversiones implícitas como en el código de quadratic2.c.

La elección entre float o double depende de lo que se necesite. Es decir, si se busca velocidad y no precisión, usar float. Si se busca más precisión en los cálculos y no se necesita mucha velocidad, usar double.



Se puede notar en la gráfica que a medida que aumenta el TIMES el tiempo de ejecución también.

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

Consideraciones tenidas en cuenta para la optimizacion del codigo realizado:

-Definición de matrices: Decidimos usar la alternativa 4 mostrada en la teoría "arreglos dinámicos como vector de elementos" para poder recorrer las matrices por filas, aprovechando que todos sus datos son almacenados contiguos en memoria.

-Localidad de datos: Para la multiplicación de matrices, se tuvo en cuenta el orden en el que se almacenaban los datos de cada matriz. Este ordenamiento funciona debido al aprovechamiento del principio de localidad de datos empleado por la caché del procesador, en donde al recibir una petición a un dato sobre memoria, toma el dato solicitado y trae, además, los datos adyacentes al solicitado. En nuestro caso usamos un orden por filas, usando a la matriz A y BT (transpuesta de B) para convertir accesos no secuenciales (columnas de B) en accesos secuenciales (filas de BT). Así, al multiplicar AxBT, ambas matrices se recorren por filas, mejorando la localidad temporal al mantener los bloques en caché. También se ve este aprovechamiento de localidad temporal en la multiplicación por bloques ya que las matrices se dividen en bloques de tamaños BSxBS para que entren en caché si el tamaño de la matriz es grande (ej: 4096), la multiplicación tradicional tendría muchos cache misses. Por otro lado se puede ver el aprovechamiento de la localidad espacial en multiplicar_bloques(), donde los índices siguen el patrón i*n + k (filas) y j*n + k (filas de BT) teniendo un acceso contiguo en memoria y evitar reiterados accesos a memoria. Esta función multiplicar_bloques() divide a la matriz en submatrices, como si fuese una matriz independiente maximizando el uso de cada uno de los datos mientras esté en caché.

-Almacenamiento temporal: Usamos variables temporales para almacenar cuentas o valores que se van a usar repetidamente. Usar estas variables temporales es una forma de reducir accesos a memoria. Usamos esta técnica en los índices para acceder a las matrices con el cálculo de desplazamiento, también con la variable size que indica el tamaño de la matriz y en la cuenta escalar.

Cluster

gcc -o salidaPunto2 ecuacion.c -O3

./miScript.sh **N BS**

	N = 512	***N = 1024***	***N = 2048***	***N = 4096.***
BS = 4	0.439854	3.596584	28.668960	224.680538
BS = 8	0.432656	3.451135	27.203272	214.534361
BS = 16	0.403311	3.246216	26.326223	268.892923
BS = 32	0.384853	3.060990	30.545282	237.975997
BS = 64	0.420277	3.550350	28.172195	224.096790
BS = 128	0.404141	3.251167	25.458181	203.549092
BS = 256	0.381458	3.028249	23.588884	217.624953
BS = 512	0.363335	2.965218	24.577411	203.058313

N = Tamaño de las matrices (NxN) BS = Tamaño de los bloques (submatrices)

Capturas mostrando los resultados obtenidos para las tablas:
https://docs.google.com/document/d/16wP1j90q_zRO7oNNsNGDdp9vvLzL_FgRLTcJbbE8jqM/edit?usp=sharing