Año 2024

Sistemas Paralelos

Trabajo Práctico N.º 1 –

Optimización de algoritmos secuenciales

Grupo 3
AGUSTINA SOL ROJAS – ANTONIO FELIX GLORIOSO CERETTI

Características de hardware y software

Equipo hogareño:

- Sistema Operativo: Linux Mint 21 Cinnamon
- Versión de Cinnamon: 5.4.12
- Núcleo Linux: 5.15.0-101-generic
- Procesador: AMD Ryzen 3 3200G with Radeon Vega Graphics x 4
- Memoria: 3.8 GiB
- Discos duros: 54.8 GB
- Compilador: gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0

Más información

Cluster remoto: Cluster Multicore (partición Blade) conformado por 16 nodos. Cada nodo posee 8GB de RAM y 2 procesadores Intel Xeon E5405 de cuatro 4 cores cada uno que operan a 2.0GHz. Compilador gcc (Debian 8.3.0-6) 8.3.0

Consideraciones respecto a la compilación y ejecución

La compilación de los algoritmos se realizó con el nivel de optimización O3 ofrecido por gcc a pesar de que Ofast fue el que dio mejores resultados en la Práctica 1, dado que el mismo no es fiable a la hora de realizar operaciones matemáticas.

En el siguiente grafico se puede observar las particularidades de cada nivel de optimización:

Opción	Nivel de Optimización	Tiempo de ejecución	Tamaño de código	Uso de memoria	Tiempo de compilación
-O0	Optimización para el tiempo de	+	+	-	-

	compilación (predeterminado)				
-O1 u -O	Optimización para tamaño de código y tiempo de ejecución	-	-	+	+
-O2	Optimización más enfocada en tamaño de código y tiempo de ejecución	--		+	++
-O3	Optimización aún más enfocada en tamaño de código y tiempo de ejecución	---		+	+++
-Os	Optimización para tamaño de código		--		++
-Ofast	O3 con cálculos matemáticos rápidos pero menos precisos	---		+	+++

En la tabla, los símbolos indican lo siguiente:

+: Aumenta.

++: Aumenta aún más.

+++: Aumenta considerablemente.

-: Reduce.

--: Reduce aún más.

---: Reduce considerablemente.

Referencia: https://www.rapidtables.com/code/linux/gcc/gcc-o.html

Enunciado

Punto N.º 1

Resuelva el ejercicio 6 de la Práctica N.º 1 usando dos equipos diferentes: (1) cluster remoto y (2) equipo hogareño al cual tenga acceso (puede ser una PC de escritorio o una notebook)

Ejercicio 6 – 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).

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.

Respuesta

a)

(1) Ejecución cluster remoto:

Soluciones Float: 2.00000 2.00000

Soluciones Double: 2.00032 1.99968

(2) Ejecución equipo hogareño:

Soluciones Float: 2.00000 2.00000

Soluciones Double: 2.00032 1.99968

Como se puede observar, se retornaron los mismos resultados tanto en el cluster remoto como en el equipo hogareño. La solución que utiliza el tipo de variable double (64 bits) es la más precisa en ambos ya que al utilizar más bits en memoria con respecto a float (32), puede representar el resultado de una forma más exacta. En la solución con float uno podría interpretar que la ecuación cuadrática tiene una sola raíz, ya que nos da el mismo resultado dos veces, cuando en realidad tiene dos raíces. Esto sucede porque, como se mencionó anteriormente, float no tiene la exactitud necesaria.

b)

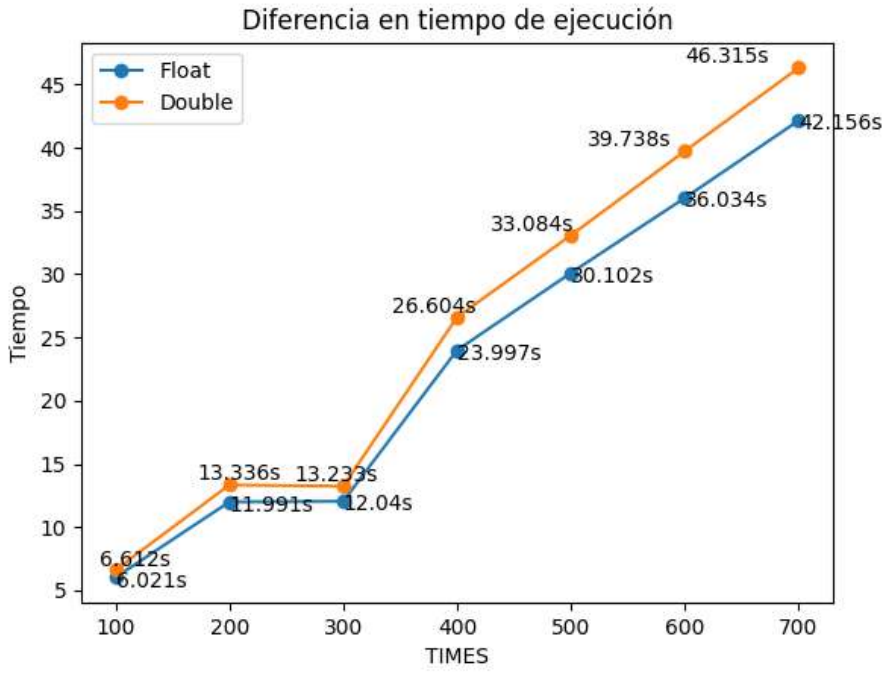
(1) Ejecución cluster remoto:

Tabla con los tiempos de ejecución obtenidos en el cluster remoto al compilar quadratic2.c con nivel de optimización O3

VARIABLE TIMES	*DOUBLE*	*FLOAT*
100	6.611579	6.020683
200	13.336373	11.990925
300	13.232912	12.040084
400	26.604315	23.996530
500	33.083846	30.102182
600	39.738415	36.034014

700	46.314868	42.156240

Gráfico con los tiempos de ejecución obtenidos en el cluster remoto al compilar quadratic2.c con nivel de optimización O3



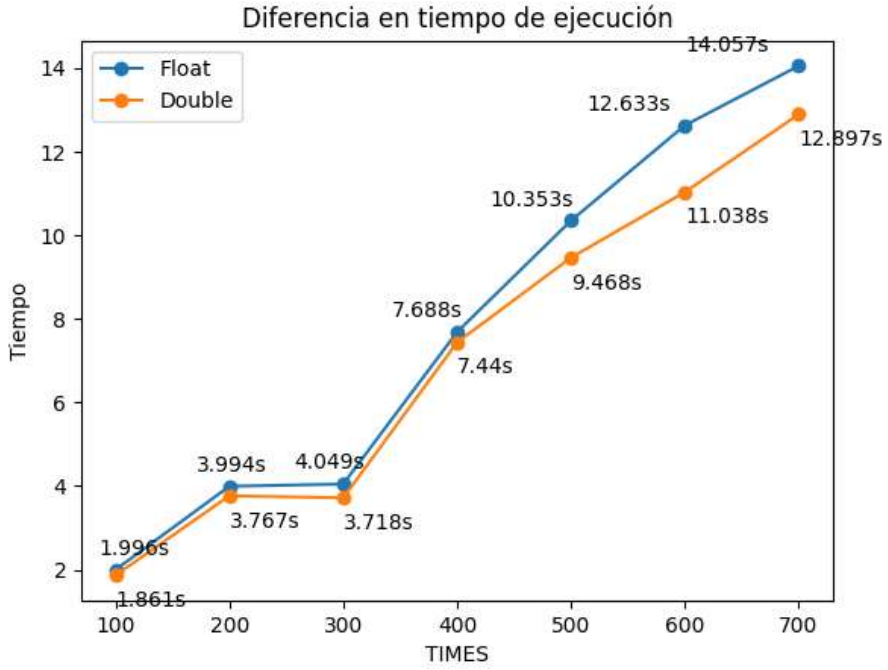
(2) Ejecución equipo hogareño:

Tabla con los tiempos de ejecución obtenidos en el equipo hogareño al compilar quadratic2.c con nivel de optimización O3

VARIABLE TIMES	*DOUBLE*	*FLOAT*
100	1.861056	1.996371
200	3.766665	3.994405
300	3.717867	4.048621
400	7.439918	7.687838

500	9.467614	10.353291
600	11.037553	12.632964
700	12.896527	14.057112

Gráfico con los tiempos de ejecución obtenidos en el equipo hogareño al compilar quadratic2.c con nivel de optimización O3



Se puede observar como la ejecución en el cluster remoto tarda más con variables de tipo double, a diferencia del equipo hogareño donde tarda más en ejecutarse con tipo float. Tanto en el equipo hogareño como en el cluster, el tiempo de ejecución de ambas variables crece de forma casi lineal. A su vez se puede observar como el tiempo de ejecución en el cluster es notablemente más alto que en el equipo hogareño.

El resultado genérico debería dar a float con menor tiempo de ejecución, ya que es el que posee menos bits, caso que sucede en el cluster remoto. Por el contrario,

en el equipo hogareño, la ejecución con variables de tipo double tarda menos. Esto puede deberse a distintos factores tales como:

1) La arquitectura: puede que la arquitectura tenga mejoras con respecto a las variables de tipo double.
2) Conversiones implícitas: en el código de quadratic2.c se utilizan las funciones “pow()” y “sqrt()” que esperan valores de tipo double. Estas funciones, cuando reciben variables de tipo float, realizan una conversión implícita que puede tener impacto en el rendimiento, haciendo que float tarde más.

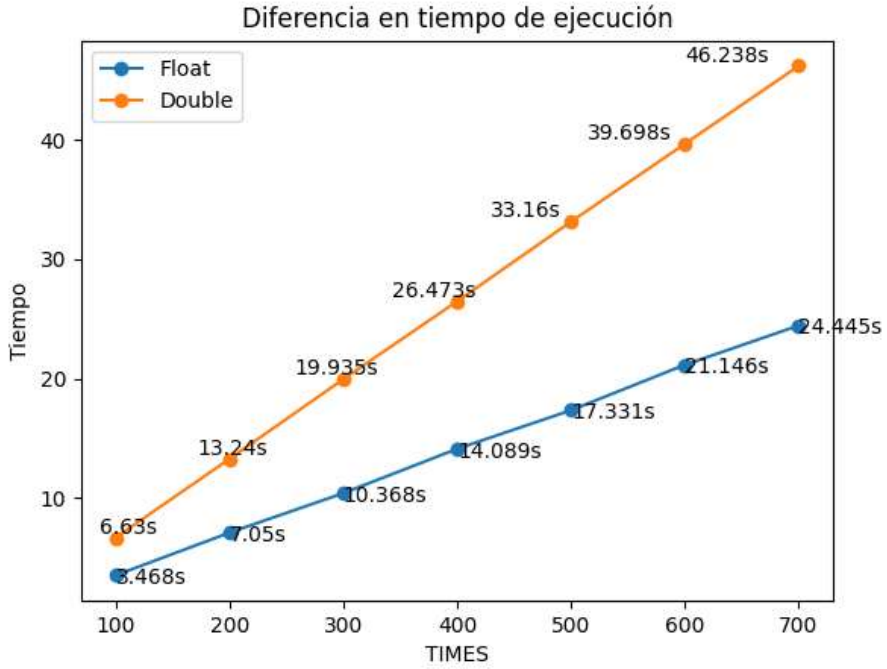
c)

(1) Ejecución cluster remoto:

Tabla con los tiempos de ejecución obtenidos en el cluster remoto al compilar quadratic3.c con nivel de optimización O3

VARIABLE TIMES	*DOUBLE*	*FLOAT*
100	6.629785	3.467720
200	13.239543	7.050341
300	19.935007	10.368365
400	26.472664	14.089075
500	33.159594	17.331257
600	39.698077	21.145925
700	46.238054	24.445106

Gráfico con los tiempos de ejecución obtenidos en el cluster remoto al compilar quadratic3.c con nivel de optimización O3



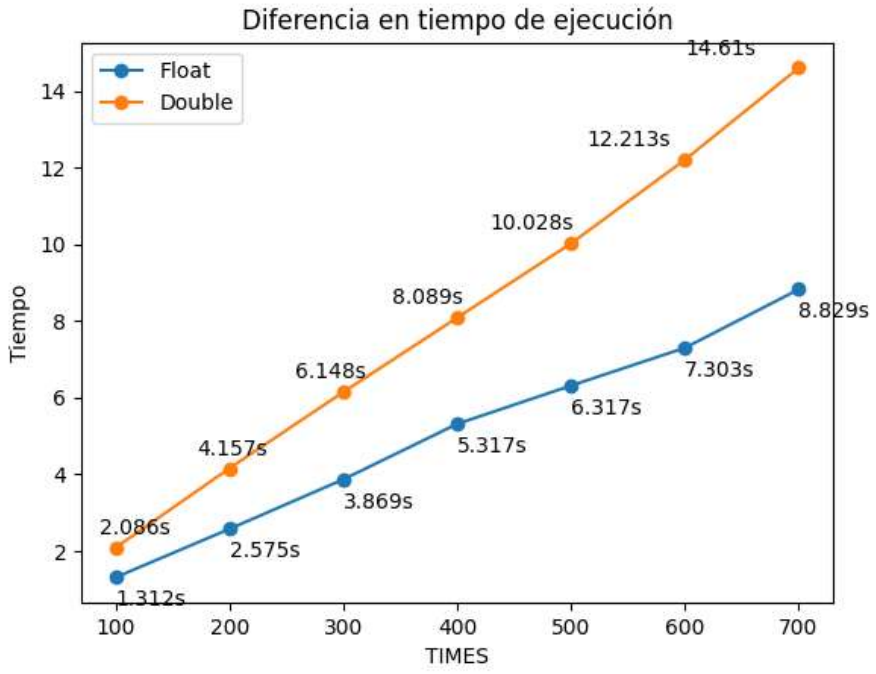
(2) Ejecución equipo hogareño:

Tabla con los tiempos de ejecución obtenidos en el equipo hogareño al compilar quadratic3.c con nivel de optimización O3

VARIABLE TIMES	*DOUBLE*	*FLOAT*
100	2.085823	1.311677
200	4.157121	2.574928
300	6.147948	3.868934
400	8.088706	5.317170
500	10.028385	6.317141
600	12.212687	7.302915

700	14.610324	8.829329

Gráfico con los tiempos de ejecución obtenidos en el equipo hogareño al compilar quadratic3.c con nivel de optimización O3



Se puede observar como en el cluster remoto y en el equipo hogareño la ejecución con variables de tipo double tarda casi lo mismo que en el ejercicio anterior. A su vez, en ambos, tarda menos la ejecución de float en cuanto a double, esto puede deberse a la diferencia presente en el código que recae en dos puntos:

1) Las funciones utilizadas para calcular las potencias y las raíces cuadradas son diferentes: en quadatric2 se utiliza “pow()” y “sqrt()”, que implementan sus números con valores de tipo double, y en quadatric3 se utilizan “powf()” y “sqrtf()”, que implementan sus números con valores de tipo float. Esto hace que en quadratic3 no haya conversiones en el caso de float, favoreciéndolo.

2) Los números flotantes están declarados explícitamente como float, esto puede favorecer al tiempo de ejecución ya que el compilar puede usar tácticas referidas a este tipo concretamente.

Al igual que en el punto anterior, el tiempo de ejecución en el cluster es notablemente más alto que en el equipo hogareño, y en ambos aumenta de forma lineal.

Punto N.° 2

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

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

- Donde A, B, C, D 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.

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.

Respuesta

Algoritmo y consideraciones de diseño

El algoritmo realiza lo siguiente:

1. Declaración e implementación de funciones y variables necesarias para el cómputo y el cálculo de tiempo de ejecución.
2. Verificación de los parámetros de entrada. Los parámetros de entrada deben ser N (tamaño de la matriz) y BS (tamaño de los bloques en los que se subdividirá la matriz para el proceso de multiplicación en bloques)

3. Alocación de memoria para las matrices A, B, C, D, R y resultMatriz.
4. Inicialización de matrices: se inicializan las matrices A, B, C, D con los valores 1.0, y R y resultMatriz con los valores 0.0. La matriz resultMatriz será la que se utilizará para almacenar el resultado de la multiplicación de A con B.
5. Cálculo de mínimos, máximos y promedios de las matrices A y B.
6. Cálculo del escalar con los mínimos, máximos y promedios obtenidos en el punto 5.
7. Multiplicación de las matrices A y B. El resultado de este cálculo se almacena en la matriz resultMatriz.
8. Multiplicación de las matrices C y D. El resultado de este cálculo se almacena en la matriz R.
9. Suma de las matrices resultMatriz y R, y se almacena el resultado en R. Cuando se realiza esta suma a su vez se hace el producto de la matriz resultMatriz con el escalar obtenido en el punto 6.
10. Se imprime el tiempo de ejecución obtenido.
11. Se libera el espacio de memoria de las matrices.

Consideraciones de diseño:

1. Definición de Matrices: las matrices fueron definidas como un arreglo dinámico como vector de elementos principalmente para aprovechar la localidad de datos a futuro, debido a que se puede elegir como se organizan las mismas (por filas o por columnas).
2. Multiplicación de matrices por bloques: se implementó la multiplicación de matrices utilizando la técnica de multiplicación por bloques vista en la Práctica 1, dado que tiene mejores tiempos de ejecución con respecto a la multiplicación de matrices sin utilizar dicha técnica.
3. Localidad de datos: en el código las matrices A y C se inicializan en orden de filas, mientras que las matrices B y D se inicializan en orden de columnas. Esto se realiza de esta manera para aprovechar la localidad espacial cuando se accede a los datos a la hora de realizar la multiplicación de matrices. En la multiplicación de matrices presente en la ecuación, se multiplica cada elemento de una fila de las matrices A y C con cada elemento correspondiente a una columna de las

matrices B y D, respectivamente. Luego se suman estos productos para obtener el elemento correspondiente en las matrices resultMatriz y R.
Para seguir respetando el orden de las matrices cuando se calculan los mínimos, máximos y promedios de las matrices A y B, se accede por filas y por columnas, respectivamente.

4. Evitar repetición de cómputo: cuando se aloca memoria para las matrices, para evitar las repeticiones del cálculo de N * N, se lo almacena en una variable “size”. Esta variable también es posteriormente usada para los cálculos de promedio.
A su vez, a la hora de acceder a las matrices, ya sea en orden por filas o por columnas, se debe realizar un cálculo de desplazamiento. Cuando se realiza dicho calculo hay ciertas partes de este que dentro de las iteraciones son siempre iguales y aun así se calculan, para evitar esto se utilizan las variables “offsetI” y “offsetJ”.

Análisis del algoritmo y conclusiones

Tabla con los tiempos de ejecución obtenidos en el cluster remoto al compilar matricesCalculo.c con nivel de optimización O3

BS / *N*	*16*	*32*	*64*	*128*
512	0.620876	0.556636	0.529749	0.537963
1024	5.038937	4.451540	4.232484	4.301166
2048	40.270436	35.564812	33.683052	34.273572
4096	315.150536	281.989872	268.523230	273.105520

Gráfico con los tiempos de ejecución obtenidos en el cluster remoto al compilar matricesCalculo.c con nivel de optimización O3 y N=512

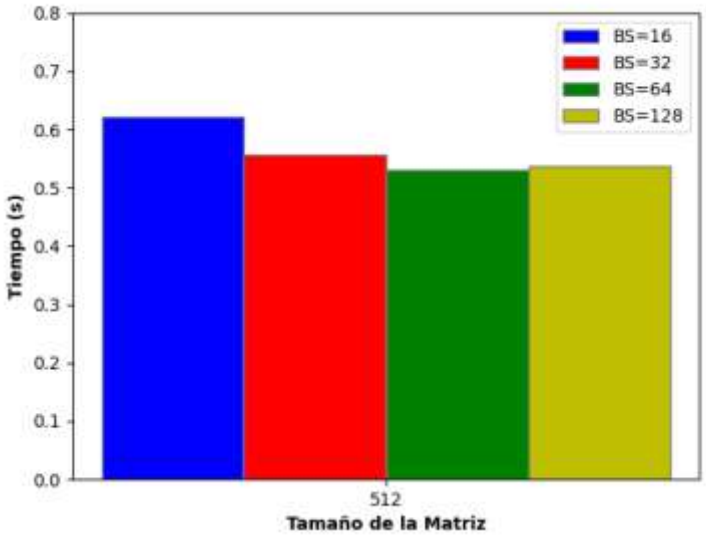


Gráfico con los tiempos de ejecución obtenidos en el cluster remoto al compilar matricesCalculo.c con nivel de optimización O3 y N=1024

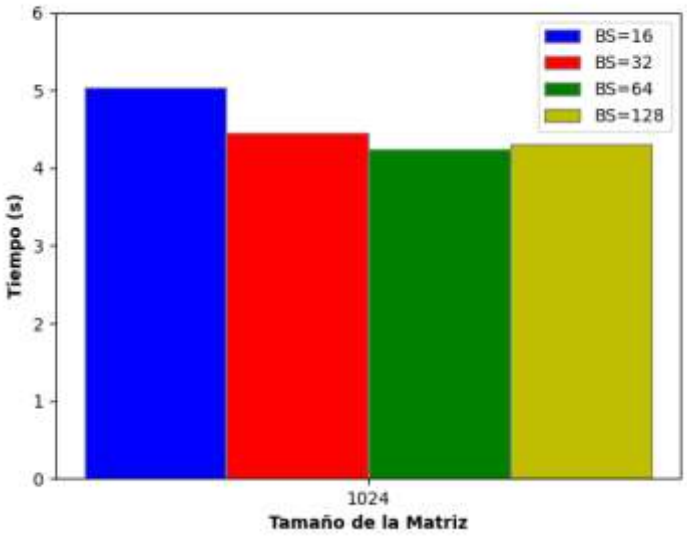


Gráfico con los tiempos de ejecución obtenidos en el cluster remoto al compilar matricesCalculo.c con nivel de optimización O3 y N=2048

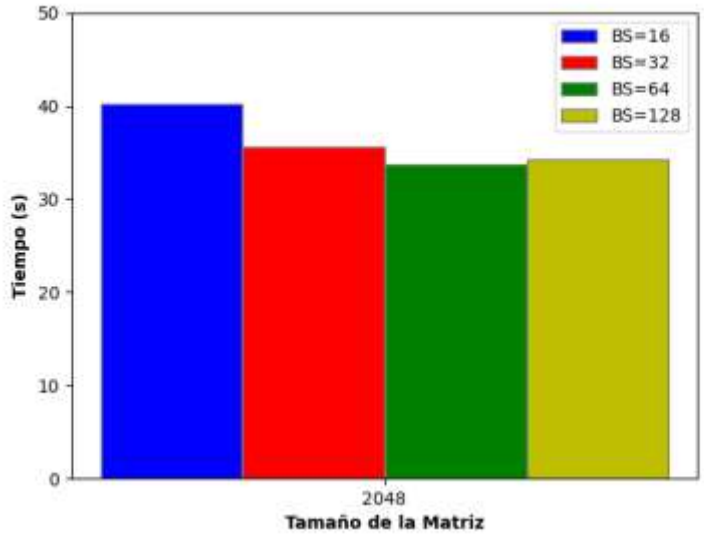
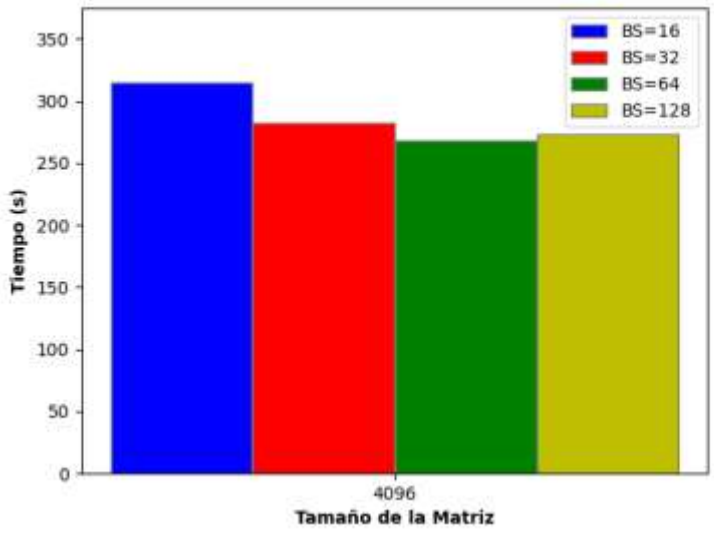


Gráfico con los tiempos de ejecución obtenidos en el cluster remoto al compilar matricesCalculo.c con nivel de optimización O3 y N=4096



Se puede observar que el tamaño de bloque optimo es 64 ya que es el que tiene mejores tiempos de ejecución. Esto está relacionado con el tamaño máximo de cache y la forma

en la que se aprovecha la localidad temporal al traer todos los datos necesarios a este componente para la multiplicación de los bloques, ya que se evitan futuros accesos repetidos a memoria al tener todos los datos necesarios en la cache.

También se puede observar que se mantiene la relación de diferencia de tiempo de ejecución entre los distintos bloques y los diversos tamaños de N.