# Sistemas Paralelos

## Trabajo Práctico 1

Optimización de

algoritmos secuenciales

Grupo 13

AÑO 2023 – 1° Semestre

**Alumnos**

- Garofalo, Pedro 17136/5
- Morena, Nahuel 16290/1

Referencias:

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

https://wiki.gentoo.org/wiki/GCC_optimization/es

Características del equipo hogareño utilizado para las siguientes consignas:

- Procesador: AMD Ryzen 9 5950X (16 núcleos, 32 hilos)
- Memoria: 64Gb DDR4 3.600MHz
- Sistema Operativo: Arch Linux (Linux 6.2.10)
- Compilador: GCC 12.2.1

**1) Resuelva el ejercicio 6 de la práctica 1 usando dos equipos diferentes:**

- **Cluster remoto**
- **Equipo hogareño al cual tenga acceso (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?**

Los resultados obtenidos fueron, tanto en el equipo hogareño como en el cluster:

```
Soluciones Float:     2.00000    2.00000
Soluciones Double:    2.00032    1.99968
```

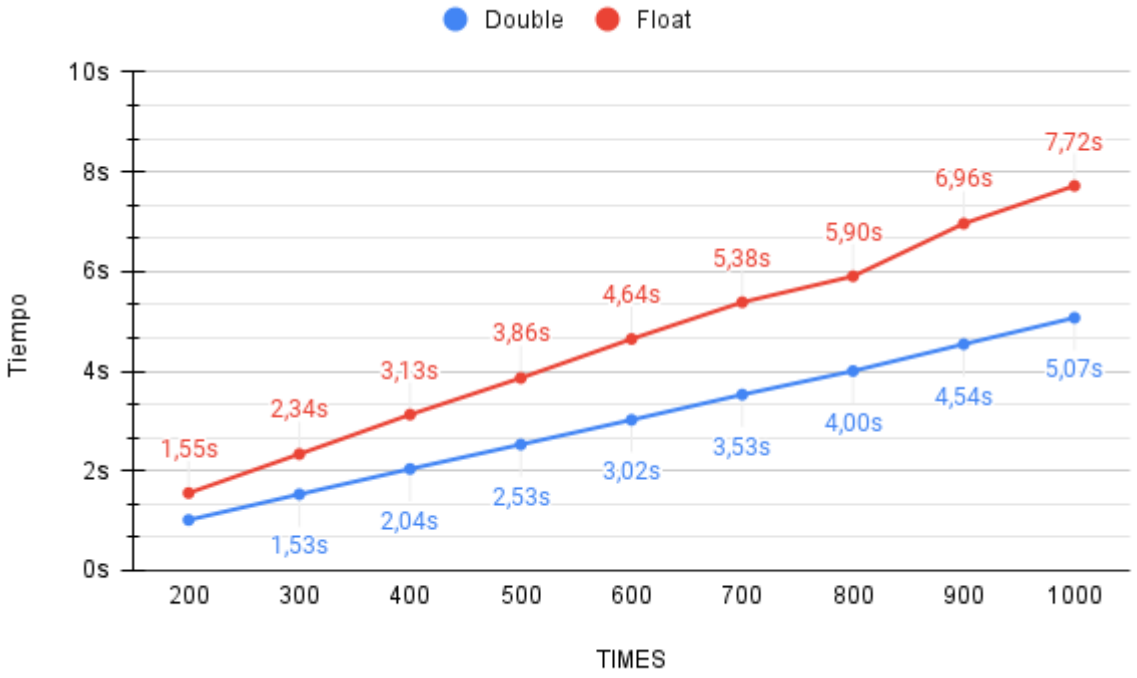
Esto muestra la diferencia de precisión entre los tipos de datos float (32 bits) y double (64 bits) de C. Si realizáramos este tipo de cálculo utilizando solamente float llegaríamos a la conclusión errónea de que esta ecuación cuadrática tiene una sola raíz en lugar de dos, demostrando la importancia de la precisión de los tipos de datos para resolver cierto tipos de problemas.

**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?**

**Equipo hogareño**

| TIMES | Double | Float |
|---|---|---|
| 200 | 1,014s | 1,554s |
| 300 | 1,529s | 2,336s |
| 400 | 2,037s | 3,128s |
| 500 | 2,528s | 3,862s |
| 600 | 3,022s | 4,645s |
| 700 | 3,529s | 5,382s |
| 800 | 4,003s | 5,904s |
| 900 | 4,541s | 6,960s |
| 1000 | 5,070s | 7,718s |

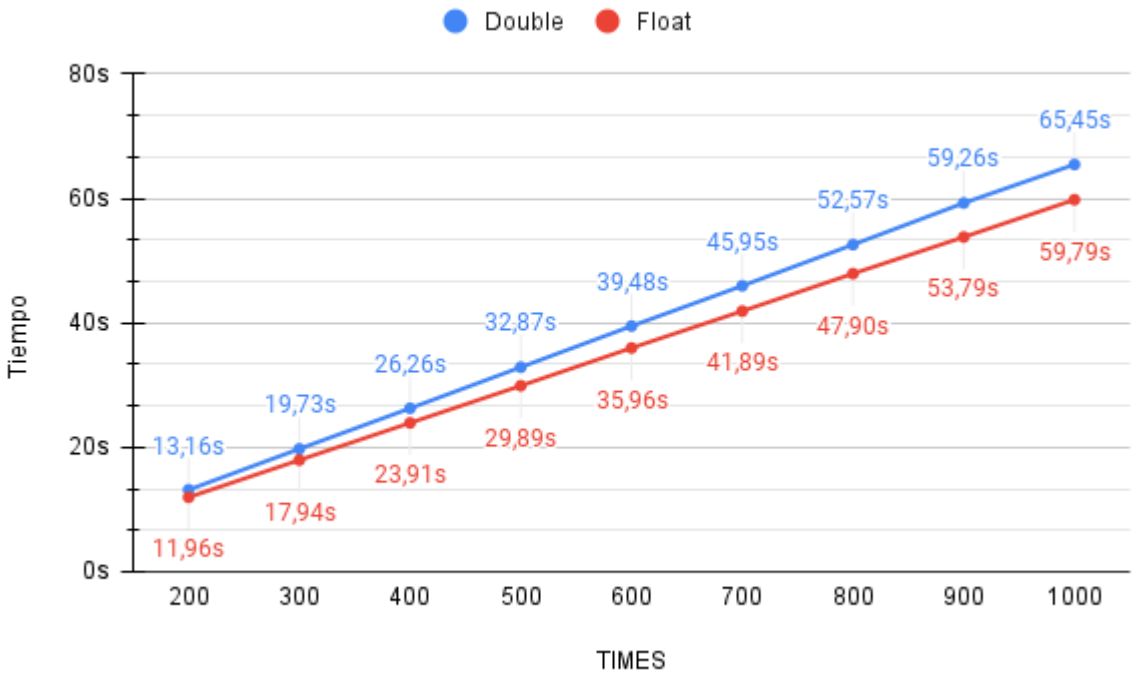
Tiempos obtenidos en equipo hogareño variando la constante TIMES y compilando con -O3



### Cluster remoto

| TIMES | Double | Float |
|---|---|---|
| 200 | 13,157s | 11,959s |
| 300 | 19,735s | 17,937s |
| 400 | 26,259s | 23,913s |
| 500 | 32,867s | 29,887s |
| 600 | 39,478s | 35,958s |
| 700 | 45,949s | 41,887s |
| 800 | 52,565s | 47,903s |
| 900 | 59,256s | 53,789s |
| 1000 | 65,452s | 59,786s |

Tiempos obtenidos en cluster remoto variando la constante TIMES y compilando con -O3



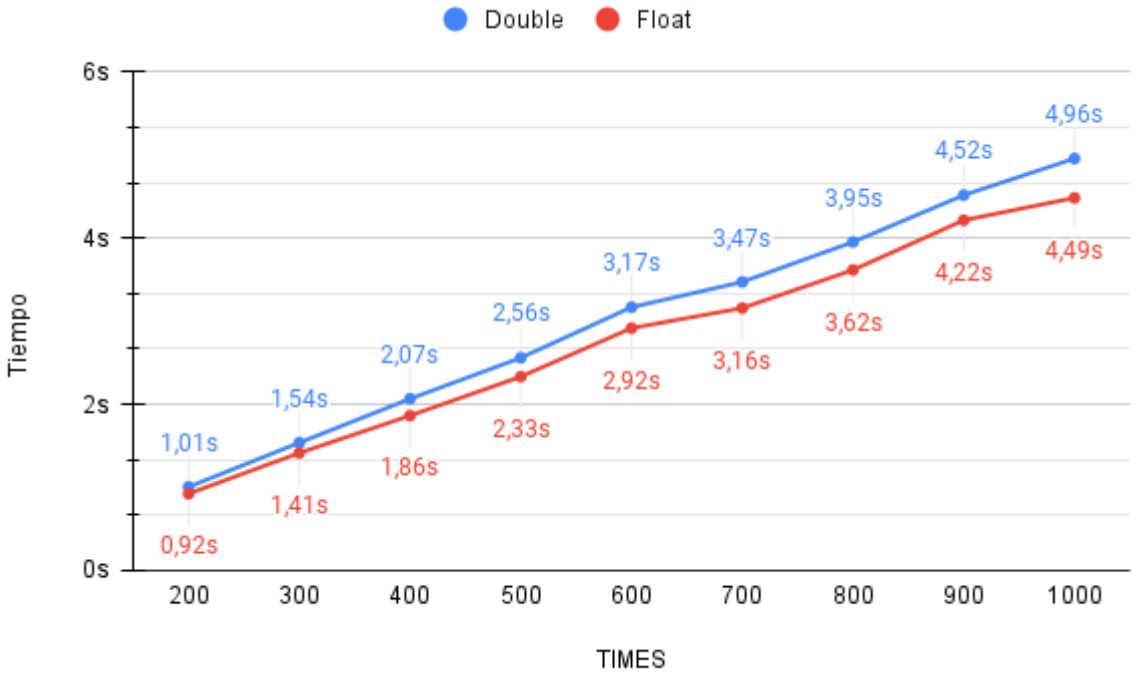

Con los datos anteriores pudimos ver que el tiempo de ejecución de *quadratic2* crece linealmente al incrementar el valor de *TIMES*. Además, pudimos calcular que en promedio la solución con *float* es un 52,4% más lenta que la solución con *double* en el equipo hogareño. En cambio en el cluster remoto es un 9,8% más rápida.

**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 diferencia puede observar en el código con respecto a quadratic2.c?**

**Equipo hogareño**

| TIMES | Double | Float |
|---|---|---|
| 200 | 1,006s | 0,924s |
| 300 | 1,539s | 1,415s |
| 400 | 2,067s | 1,864s |
| 500 | 2,561s | 2,334s |
| 600 | 3,170s | 2,918s |
| 700 | 3,474s | 3,159s |
| 800 | 3,955s | 3,618s |
| 900 | 4,518s | 4,217s |
| 1000 | 4,959s | 4,485s |

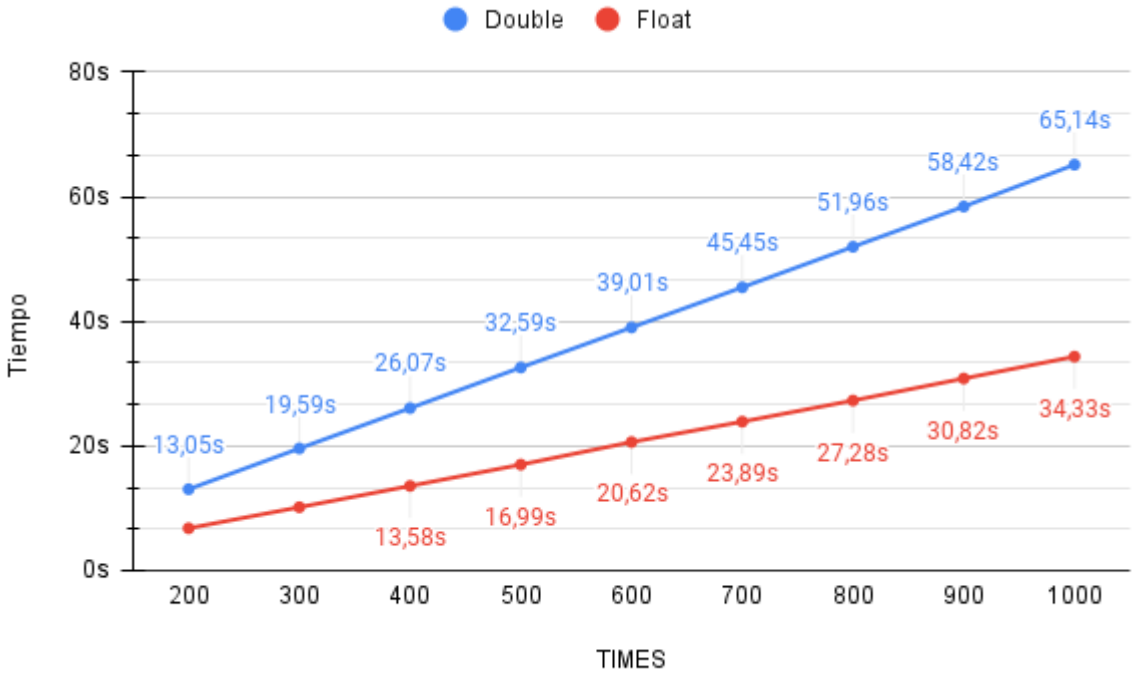
Tiempos obtenidos en equipo hogareño variando la constante TIMES y compilando con -O3



## Cluster remoto

| TIMES | Double | Float |
|---|---|---|
| 200 | 13,047s | 6,781s |
| 300 | 19,593s | 10,158s |
| 400 | 26,066s | 13,577s |
| 500 | 32,585s | 16,994s |
| 600 | 39,011s | 20,619s |
| 700 | 45,454s | 23,892s |
| 800 | 51,959s | 27,283s |
| 900 | 58,417s | 30,818s |
| 1000 | 65,140s | 34,328s |

Tiempos obtenidos en cluster remoto variando la constante TIMES y compilando con -O3



Al igual que con *quadratic2,* pudimos observar que el tiempo de ejecución incrementa linealmente al aumentar *TIMES*.
Lo que pudimos distinguir con respecto al punto anterior es que el tiempo de ejecución de la solución con *float* es en promedio un 9,3% más rápido que la solución con *double* en el equipo hogareño y un 90,9% más rápido en el clúster remoto. Observando el código pudimos ver de dónde viene la diferencia de rendimiento entre ambas implementaciones: esto se debe a que *quadratic2* utiliza las funciones de la librería estándar de C *pow()* y *sqrt()* para la solución con *float*, y estas operan con valores de tipo *double*. En cambio, en *quadratic3* se utilizan las funciones *powf()* y *sqrtf()*, las cuales operan con *float*, y esto probablemente las hace más rápidas en la mayoría de las implementaciones al tener que trabajar con menos precisión. Además se ahorran las conversiones de tipo implícitas que probablemente sea lo que cause que en *quadratic2* la solución con *float* resulte más lenta que la de *double* en el equipo hogareño.

## Optimizadores

En las pruebas realizadas se utiliza el optimizador -O3, este es uno de varios brindados por el compilador utilizando el flag -O para cambiar el nivel de optimización que se emplea en el código, variando de esta forma en el tiempo y memoria que se tomará en la compilación siendo estas mayores a medida de que se incrementa el nivel de optimización.

Algunos de los diferentes niveles de optimización son los siguientes:

- -O0: Es el predeterminado si no se especifica ningún nivel -O. Lo que significa que el código no recibirá ningún tipo de optimización

- -O1: Es el nivel de optimización más básico, se encarga principalmente de generar un código rápido y pequeño en tamaño sin generar demasiadas demoras en el tiempo de compilación.

- -O2: Posee todas las características brindadas por -O1 y además, intenta aumentar el rendimiento del código sin comprometer el tamaño y el tiempo de compilación empleado.

- -O3: Posee todas las características brindadas por -O2 y además activa optimizaciones que son caras en términos de tiempo de compilación y uso de memoria. No garantizando una mejora de rendimiento, pudiendo producir el efecto contrario al ralentizar un sistema por el uso de binarios de gran tamaño generados. Una de las mejoras es la vectorización dentro de los bucles utilizados.

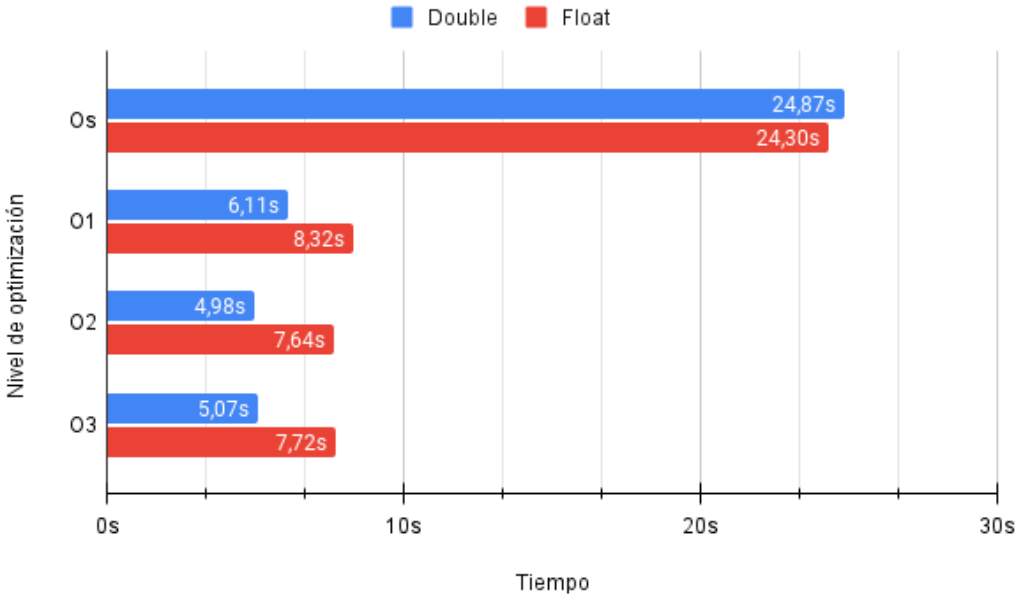
- -Os: Se encarga principalmente de que el compilador de prioridad en realizar más optimizaciones diseñadas para reducir el tamaño del código en lugar de preocuparse por la velocidad de ejecución.

| option | optimization level | execution time | code size | memory usage | compile time |
|---|---|---|---|---|---|
| -O0 | optimization for compilation time (default) | + | + | - | - |
| -O1 or -O | optimization for code size and execution time | - | - | + | + |
| -O2 | optimization more for code size and execution time | -- | | + | ++ |
| -O3 | optimization more for code size and execution time | --- | | + | +++ |
| -Os | optimization for code size | | -- | | ++ |

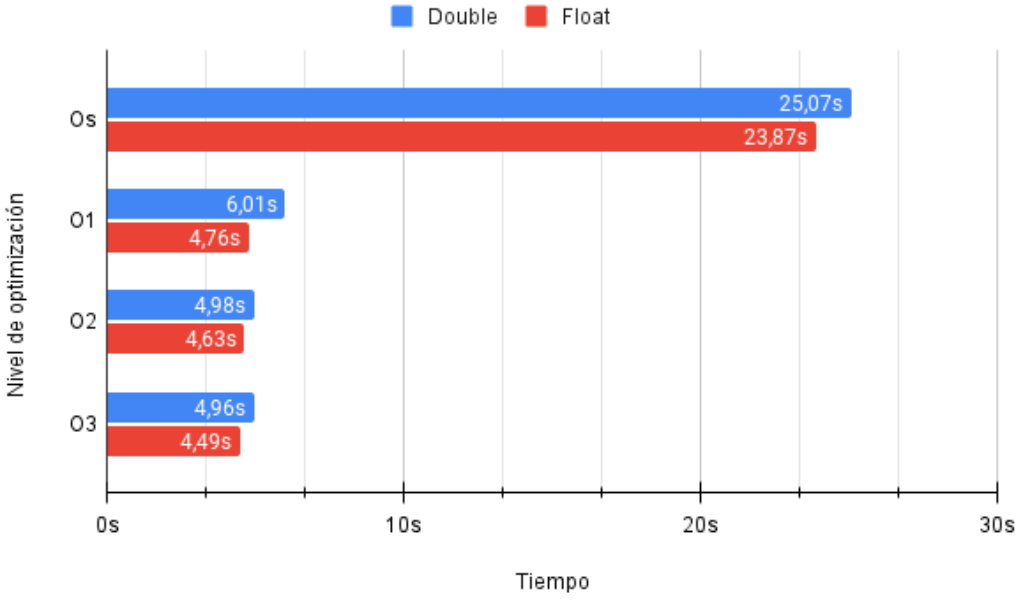
Tabla de tipos de optimización de GCC

En donde se observa en el cuadro una comparación entre los distintos niveles de optimización brindados por el compilador *gcc* mostrando que los símbolos '+' indican un aumento y el símbolo significa '-' una disminución de lo indicado en cada una de las columnas (al presentarse más de un símbolo en una casilla significa un mayor cambio significativo con respecto a las demás optimizaciones).

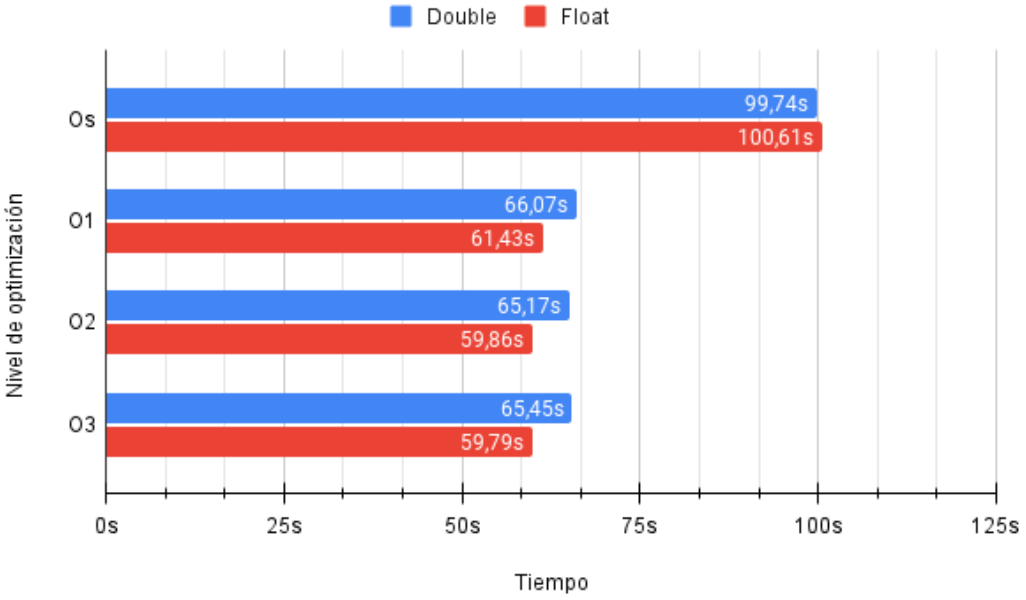
**Comparación de niveles de optimización**



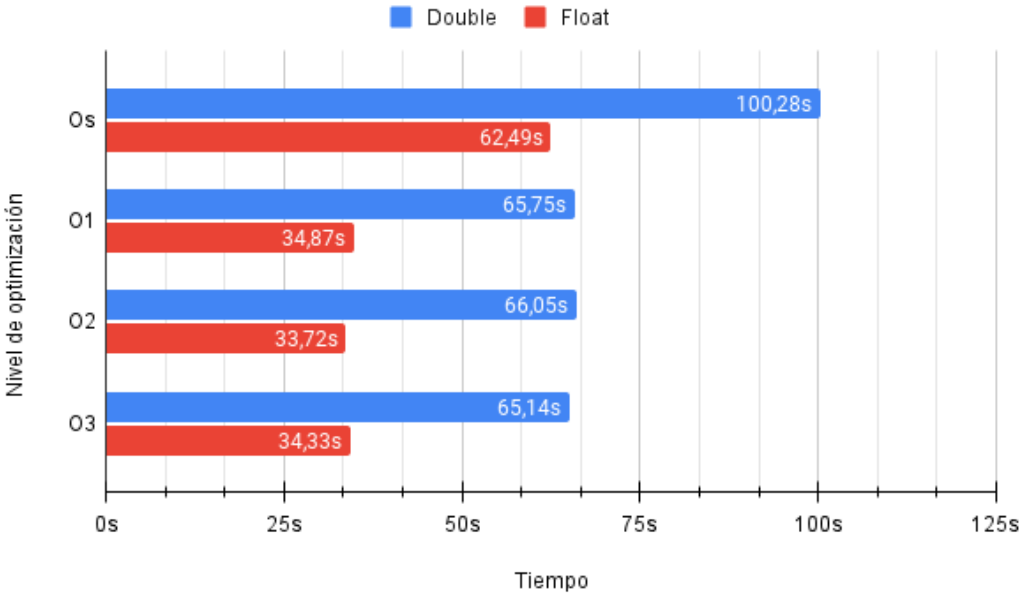

Tiempos de *quadratic2* con TIMES = 1000 para distintos niveles de optimización en equipo hogareño




Tiempos de *quadratic3* con TIMES = 1000 para distintos niveles de optimización en equipo hogareño




Tiempos de *quadratic2* con TIMES = 1000 para distintos niveles de optimización en clúster remoto




Tiempos de *quadratic3* con TIMES = 1000 para distintos niveles de optimización en clúster remoto

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

$$R = \frac{(MaxA \times MaxB - MinA \times MinB)}{PromA \times PromB} \times [A \times B] + [C \times Pot2(D)]$$

- **Donde A, B, C y R son matrices cuadradas de NxN con elementos de tipo double.**
- **D es una matriz de enteros de NxN y debe ser inicializada con elementos de ese tipo (NO float ni double) en rango de 1 a 40.**
- **MaxA, MinA, y PromA son los valores máximo, mínimo y promedio de la matriz A.**
- **MaxB, MinB y PromB son los valores máximo, mínimo y promedio de la matriz B.**
- **La función Pot2(D) aplica potencia de 2 a cada elemento de la matriz D.**

**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.**

El algoritmo desarrollado para resolver la consigna está conformada por una sucesión de pasos a seguir para brindar la solución esperada y devolviendo el tiempo que se empleó en resolver cada una de las variaciones de matrices de tamaño NxN.

Pasos realizados:

- Tomar los parámetros de tamaño “N” para dimensionar las matrices.
- Declarar todas las variables, como índices o matrices.
- Inicializar las matrices.
- Llevar a cabo la resolución de la ecuación brindada (tomando el tiempo en cada una de las operaciones realizadas).
- Finalizar y guardar el contador final.
- Liberar el espacio de memoria de las matrices.
- Brindar como salida del programa el resultado obtenido.

Decisiones de diseño:

Se desarrolló un script en bash llamado *’matrices.sh’* que permite automatizar las pruebas sobre el código *‘matrices.c’* con cada una de las variaciones de tamaño solicitadas en la consiga y permitiendo al final de la prueba, visualizar los resultados en un archivo que generará con el nombre ‘*resultado.csv*’.

Para la toma de tiempo del algoritmo, se decidió que estas sean tomadas y almacenadas por separado por cada una de las operaciones realizadas individualmente y que al final de la ejecución, todos los tiempos obtenidos sean sumados para obtener el tiempo total de toda la ejecución dedicada al cálculo de la ecuación. Decisión tomada principalmente para poder monitorear y evaluar el comportamiento en los tiempos obtenidos en cada una de las operaciones individuales a la hora de aplicar o probar nuevas técnicas de optimización sobre el código.

Consideraciones tenidas en cuenta para la optimización del código realizado:

- **Definición de matrices:** Se decidió definir a las matrices como *'arreglos dinámicos como vector de elementos'* principalmente para poder beneficiarse de las características que esta ofrece, como lo son el permitir recorrer la matriz por filas o columnas o el aprovechar que todos sus datos sean almacenados contiguos en memoria mediante diferentes algoritmos de optimización.

- **Localidad de datos:** Por la metodología empleada en el cálculo de multiplicación de matrices, se decidió tener en cuenta el orden en el que se almacenaban los datos de cada una de las matrices. Esto se debe a que en el producto matricial, para cada valor de la matriz resultante, el recorrido realizado para el primer factor es realizado mediante filas mientras que el segundo factor se recorre por columnas. Por lo que las matrices A y C (al ser el primer factor de sus respectivas multiplicaciones) se ordenan por filas, y las matrices B y D se recorren por columnas.
  Este ordenamiento de matrices funciona debido al aprovechamiento del principio de localidad de datos empleado por la caché del procesador, en donde este al recibir una petición a un dato sobre memoria, toma el dato solicitado y trae además, los datos adyacentes al solicitado.

- **Minimizar cálculos:** Se decidió el utilizar variables auxiliares para recudir el cálculo de los datos que se requieren repetidamente, como la multiplicación de ciertos índices para indicar la posición de las matrices a trabajar o el mismo dato que posteriormente es utilizado como escalar para una de las multiplicaciones de matrices.

- **Multiplicación de matrices por bloque:** Para aprovechar mucho más la localidad de los datos y evitar reiterados accesos a memoria, se utiliza esta técnica en donde se centra en dividir la matriz en submatrices, permitiendo trabajar con estas como si fueran independientes y maximizando de esta forma el uso de cada uno de los datos de la matriz mientras este se encuentre en caché.

Resultados obtenidos:

**Equipo hogareño**

| | N = 512 | N = 1024 | N = 2048 | N = 4096 |
|---|---|---|---|---|
| bs = 4 | 0,098s | 0,770s | 6,162s | 48,175s |
| bs = 8 | 0,085s | 0,654s | 5,294s | 43,109s |
| bs = 16 | 0,090s | 0,726s | 6,215s | 55,678s |
| bs = 32 | 0,103s | 0,797s | 6,891s | 56,604s |
| bs = 64 | 0,116s | 0,913s | 7,495s | 60,725s |
| bs = 128 | 0,141s | 1,124s | 10,029s | 75,593s |
| bs = 256 | 0,158s | 1,320s | 10,460s | 84,173s |
| bs = 512 | 0,162s | 1,366s | 10,717s | 88,211s |

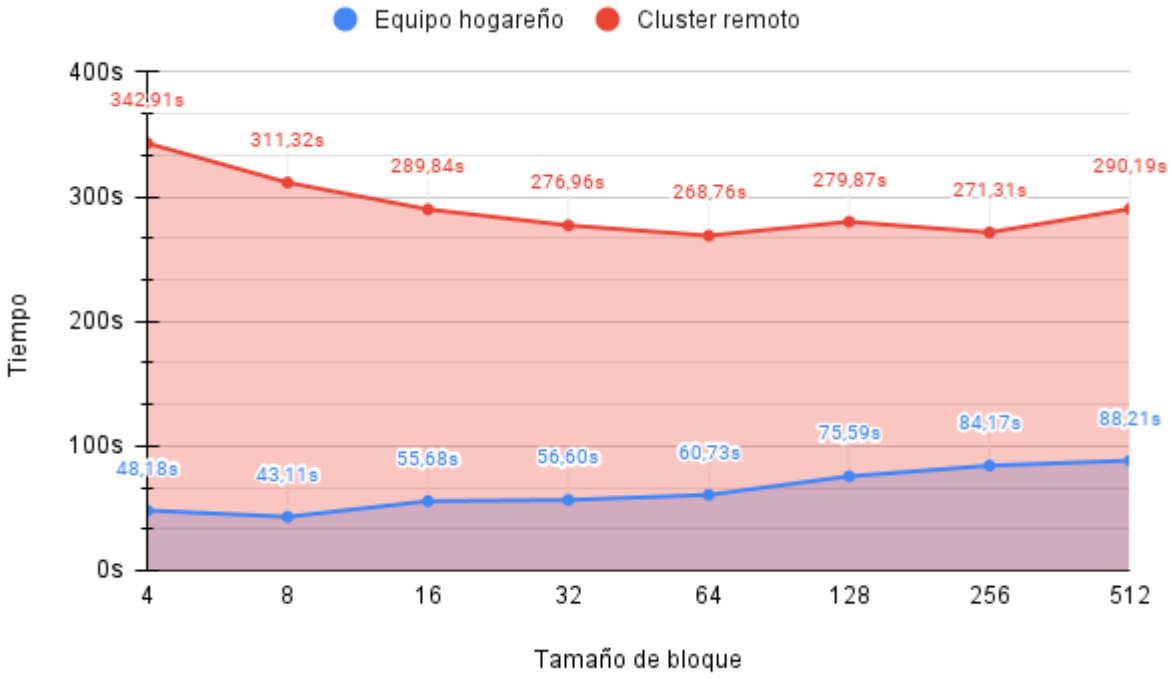
Tiempos obtenidos en equipo hogareño variando N y bs compilando con -O3
(N: tamaño de las matrices, bs: tamaño de las submatrices)

### Cluster remoto

| | N = 512 | N = 1024 | N = 2048 | N = 4096 |
|---|---|---|---|---|
| bs = 4 | 0,655s | 5,289s | 42,644s | 342,906s |
| bs = 8 | 0,617s | 4,946s | 39,571s | 311,320s |
| bs = 16 | 0,557s | 4,548s | 36,377s | 289,840s |
| bs = 32 | 0,543s | 4,342s | 34,649s | 276,964s |
| bs = 64 | 0,527s | 4,211s | 33,663s | 268,764s |
| bs = 128 | 0,548s | 4,373s | 34,900s | 279,867s |
| bs = 256 | 0,531s | 4,241s | 33,907s | 271,313s |
| bs = 512 | 0,523s | 4,185s | 33,401s | 290,193s |

Tiempos obtenidos en cluster remoto variando N y bs, compilando con -O3
(N: tamaño de las matrices, bs: tamaño de las submatrices)

### Comparación de tiempos entre equipo hogareño y clúster remoto (N = 4096)




Analizando los resultados obtenidos pudimos determinar que en el cluster, el tamaño ideal de las submatrices para la multiplicación es de 64x64, a diferencia del equipo hogareño que da un mejor rendimiento con submatrices de 8x8. También cabe destacar que esta conclusión es la misma para todos los tamaños de matrices probados (512, 1024, 2048 y 4096).