Sistemas Paralelos 2021
Informe - Entrega I

Alumnos:

- Julian Marques de Abrantes 15966/0
- Felipe Dioguardi 16211/4

Características de la computadora hogareña:

- Intel Core i5-7200U 2.50GHz
- 8GB de memoria RAM

Ejercicio 1:

1. Resuelva el ejercicio 6 de la Práctica Nº 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).

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?

Resultados de la ejecución del cluster y hogareña:

- Soluciones Float: 2.00000 2.00000
- Soluciones Double: 2.00032 1.99968

La diferencia que notamos es que la solución con el Double (que tiene el doble de dígitos) es más precisa que la solución con el Float, que da un resultado redondeado.

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?

Para esta prueba no utilizamos aún ninguna opción de optimización.

Resultados de la ejecución en segundos:

TIMES=500	Hogareña	Cluster
Double	176.836005	225.498923
Float	208.486519	234.082213

Al ejecutarlo la primera vez, se nota que el tiempo de ejecución del cálculo del Double es menor al tiempo de ejecución del Float. Al aumentar la constante TIMES al 1000, el tiempo de ejecución de ambos aumentó de gran manera, aunque la diferencia entre el tiempo de ejecución de el Float y el Double se mantiene, siendo Double aun el más rápido.

Al investigar, encontramos que el Float puede tardar más tiempo debido al redondeo realizado, sin embargo el Double debería tardar aún más, ya que al ser de doble precisión, los cálculos son más costosos.

También el tiempo de ejecución de estas operaciones varía dependiendo las arquitecturas de las computadoras; algunas optimizan las operaciones con Double haciéndolas más rápidas que las operaciones con Float (como parece ser el caso, tanto para la arquitectura hogareña como para la del cluster).

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?

Para esta prueba no utilizamos aún ninguna opción de optimización.

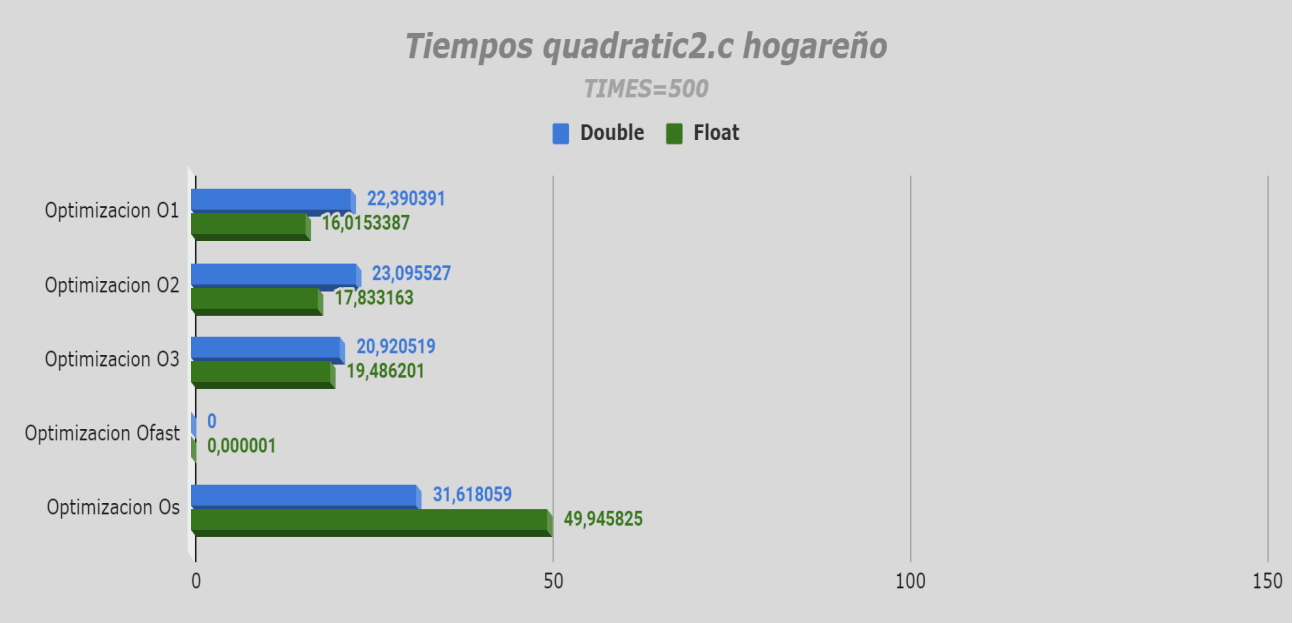
Resultados de la ejecución en segundos:

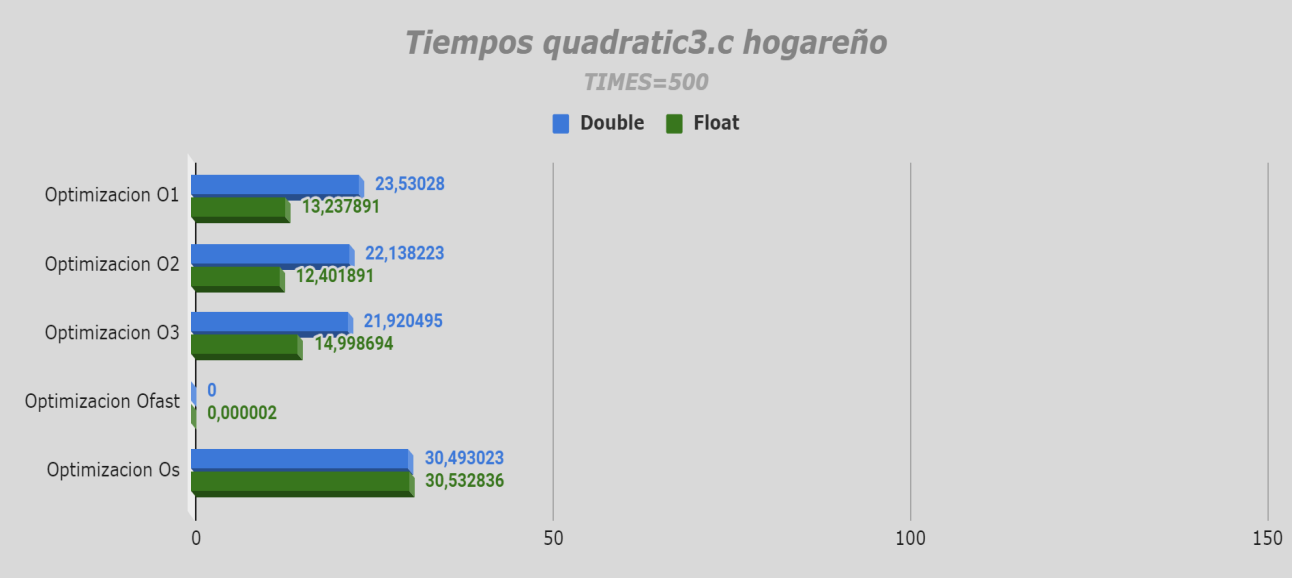
TIMES=500	Hogareña	Cluster
Double	176.588161	225.581337
Float	117.679778	345.607705

En este caso el código está más optimizado para realizar operaciones con Float (porque usa las funciones powf() y sqrtf(), que mejoran el rendimiento con ese tipo de datos). Esto parece haber sido útil para la arquitectura hogareña, donde se aprecia que el cálculo de Float es más rápido que el del Double. Sin embargo, al ejecutarlo en el cluster el tiempo de ejecución de Double fue de nuevo menor al de Float.

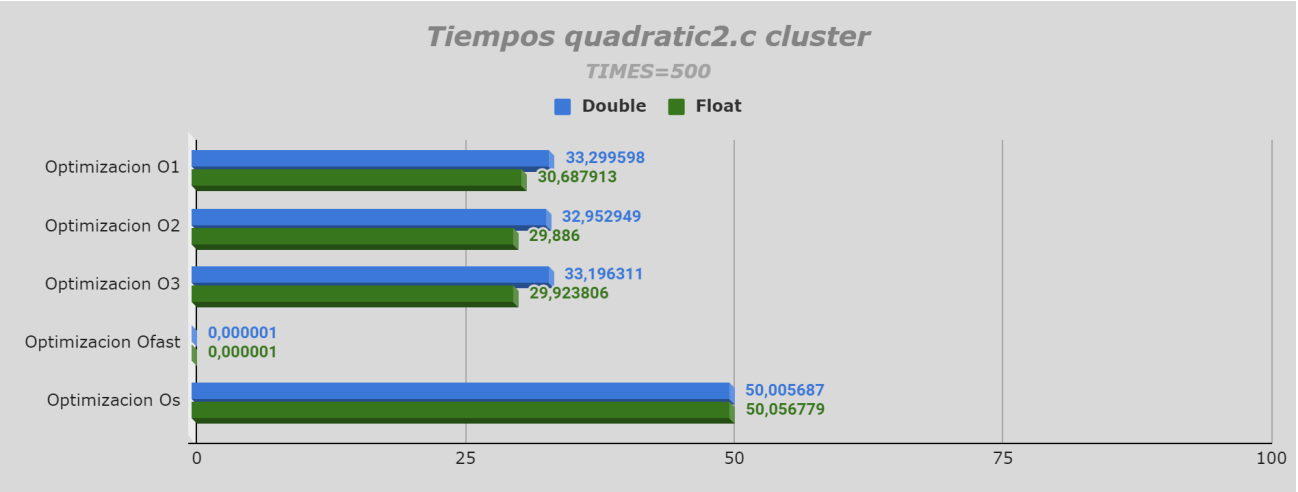
Luego de realizadas estas pruebas bases, corrimos los “quadratic2.c” y “quadratic3.c” con cada optimización del compilador. Estos fueron los resultados:

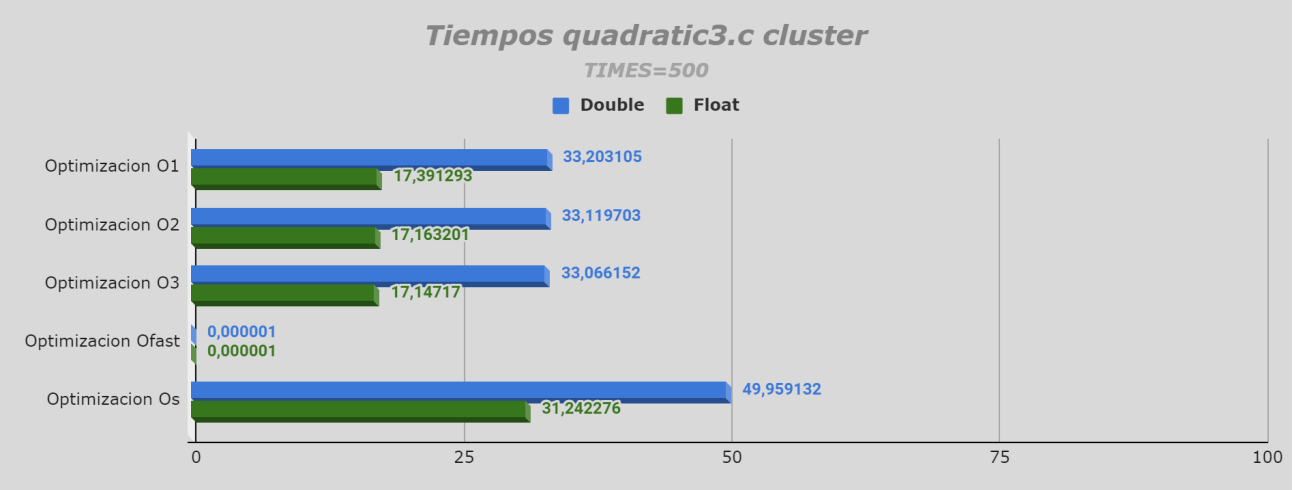
Hogareño


Tiempos quadratic2.c hogareño
TIMES=500
Double
Float
Optimizacion O1
22,390391
16,0153387
Optimizacion O2
23,095527
17,833163
Optimizacion O3
20,920519
19,486201
Optimizacion Ofast
0
0,000001
Optimizacion Os
31,618059
49,945825
0
50
100
150


Tiempos quadratic3.c hogareño
TIMES=500
Double
Float
Optimizacion O1
23,53028
13,237891
Optimizacion O2
22,138223
12,401891
Optimizacion O3
21,920495
14,998694
Optimizacion Ofast
0
0,000002
Optimizacion Os
30,493023
30,532836
0
50
100
150

Cluster





Investigamos qué optimizacion realiza cada opción del compilador (omitiendo aquellas específicas de debugging):

- O1 intenta reducir el tamaño del código y el tiempo de ejecución sin realizar ninguna optimización que afecte muy gravemente el tiempo de compilación.
- O2 incluye todas las mejoras de O1, realizando casi todas las optimizaciones soportadas que no involucran un desbalance entre espacio y tiempo.
- O3 incluye todas las mejoras de O2, y optimizaciones de loops.
- Os incluye todas las mejoras de O2, salvo las que incrementan el tamaño del código.
- Ofast incluye todas las mejoras de O3, pero con cálculos matemáticos no seguros / imprecisos.

Al ejecutar en el cluster los programas con las optimizaciones, todas las operaciones de Double registraron un tiempo de ejecución superior a las de Float, cumpliendo así con lo descrito acerca de las funciones powf() y sqrtf().

Ejercicio 2:

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

$$C = T + avg_R(RA + RB)$$

Donde A, B, C, T y R son matrices cuadradas de NxN. avgR es el valor promedio de los elementos de la matriz R. El elemento (i,j) de la matriz R debe calcularse como:

$$Ri,j = (1 - T_{i,j})(1 - cos\theta_{i,j}) + T_{i,j} sin\theta_{i,j}$$

Donde $T_{i,j}$ es el elemento en la posición (i,j) de la matriz T. El angulo $\theta_{i,j}$, en radianes, se obtiene de la posición (i,j) de una matriz M de NxN. Los valores de los elementos de la matriz M están comprendidos en un rango entre 0 y 2π. 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.

Nuestra solución de este ejercicio tiene varios pasos:

- Declarar las variables/punteros a matrices (arreglos).
- Controlar los argumentos pasados al problema (solo recibe un entero que representa el valor de N).
- Inicializar las variables, y alocar memoria para las matrices.
- Inicializar el randomizador como se indica en la práctica.
- Inicializar las matrices con valores aleatorios.
- Calcular la matriz R y su promedio.
- Calcular la matriz C.
- Liberar la memoria de las matrices.

Para optimizar nuestro código tuvimos en cuenta lo siguiente:

- **Reformular el cálculo de C:** La fórmula $C = T + avg_R(RA + RB)$ es equivalente a $C = T + avg_R * R(A + B)$. Realizar la cuenta de la primera forma

implica realizar 2 multiplicaciones y una suma, mientras que hacerlo de la segunda conlleva una multiplicación y una suma (la multiplicación matricial es la operación que más tiempo le consume al algoritmo. Remover una mejora sustancialmente la eficiencia). **Somos conscientes de que se puede realizar. Decidimos evitarlo al aclarar en la consulta sincrónica que este no es el objetivo del trabajo.**

- **Evitar fallos de caché:** La fórmula del cálculo de C implica realizar 2 multiplicaciones de matrices. Realizar ambas en un mismo loop triple puede provocar una grave caída en el rendimiento al tener que acceder tantas veces a posiciones de memoria que no se encuentran en la caché (reemplazando los valores de una de las matrices con los valores de la otra en cada iteración). Como solución decidimos calcular primero la multiplicación de R*A en un conjunto de bucles for, luego calcular la multiplicación R*B de la misma forma, y finalmente realizar el resto de la ecuación para calcular C.

- **Explotación de la localidad de datos:** Según como estaba planteada la ecuación, decidimos ordenar las matrices A y B por columnas, ya que estas dos son el segundo operando del producto matricial. En cambio, como la matriz R es la primer operando en la multiplicación, la ordenamos por filas. Como para calcular cada valor de la matriz resultante se necesita una fila del primer operando y una columna del segundo, ordenarlas de esta forma nos permite optimizar el acceso a los datos de cada matriz (porque el sistema operativo se aprovecha del principio de la localidad de datos y en cada acceso a memoria suele traer los elementos adyacentes al que fue a buscar originalmente).

- **Multiplicación de matrices por bloques:** Este método divide las matrices en varios bloques, y realiza las operaciones sobre cada uno de estos (como si fueran matrices independientes). Esta técnica permite utilizar cada dato lo más posible mientras estos se encuentran en la caché, efectivamente reduciendo la cantidad de accesos a memoria.
  La multiplicación por bloques alcanza su mayor potencial cuando estas particiones son del tamaño justo para que llenen la caché (que depende de la máquina donde se vaya a ejecutar), permitiendo usar la mayor cantidad de datos por acceso a memoria. **Somos conscientes de que se puede realizar. Decidimos evitarlo al aclarar en la consulta sincrónica que no es necesario para el trabajo.**

- **Evitar repetición de cálculos:** Identificamos cálculos repetitivos y los realizamos solo cuando este cambiaría, guardando el resultado en variables auxiliares para reducir el tiempo de ejecución.

- **Minimizar el acceso a la heap*:*** Creamos variables auxiliares guardadas en en memoria estática, donde acumulamos los resultados parciales que se generan durante la multiplicación matricial. Una vez finalizado el cálculo, guardamos el valor final en la posición correspondiente de la *matriz resultado* (que está alocada en la heap). Hicimos esto debido a que el acceso a la heap suele ser más costoso que el acceso a memoria estática.

Resultados de la ejecución en segundos:

Entrada/Arquitectura	Hogareña	Cluster
512	1.054347	2.525227
1024	8.355151	20.327265
2048	70.222854	162.461309
4096	605.043639	1302.174156