Entrega 1 - Sistemas Paralelos

Alumnos: Gaitan Catalina, Lizondo Santiago

Especificaciones equipo hogareño:

Especificaciones del hardware:

- CPU: Intel Haswell Core i3 1115G4, 3.00GHz
  - Cores totales: 2
  - Threads totales: 4
- Cache: 6MB Intel Smart Cache
- Memoria RAM: 12 GB DDR4
- Grafica: Mesa Intel UHD Graphics (TGL GT2)
- SSD: 240GB

Especificaciones del software:

- Sistema Operativo: Ubuntu 22.10 64-bit

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).

   La optimización empleada y elegida es la -O3:

   Si bien, la optimización **-Ofast** dio resultados mas rápidos, estos no resultaron ser fiables. Ocurrieron tiempos extremadamente cortos de menos de 1 segundo e incluso soluciones mas rápidas con Double que con Float. Esto se debe a que -Ofast habilita una opción de optimización adicional llamada -ffast-math, que puede hacer que algunas operaciones aritméticas no cumplan con los estándares matemáticos exactos, pero que son más rápidas.

   a. Ejecución algoritmo quadratic1.c en equipo hogareño con optimización -O3:

	(-b + sd) / (2.0*a)	(-b - sd) / (2.0*a)
Solución Float	2.00000	2.00000
Solución Double	2.00032	1.99968

   Son dos cuentas muy similares, la diferencia es que la primera tiene suma los valores b y sd, y la otra los resta.

   Los resultados dados por las dos funciones dbl_solve y flt_solve deberían ser iguales, pero en este caso son diferentes porque la precisión de float y double es diferente. El tipo de dato float tiene una precisión de 7 dígitos decimales, mientras que double tiene una precisión de 15-16 dígitos decimales. En este caso, los valores de B y C están muy cerca uno del otro y la precisión limitada de float resulta en una pérdida de precisión durante el cálculo, lo que conduce a una diferencia en las soluciones calculadas por las dos funciones.

Ejecución algoritmo quadratic1.c en el cluster con optimización -O3:

	(-b + sd) / (2.0*a)	(-b - sd) / (2.0*a)
Solución Float	2.00000	2.00000
Solución Double	2.00032	1.99968

Debido a que la optimización -O3 no activa flags que puedan modificar los resultados de las operaciones matemáticas, los resultados van a ser los mismos que si lo ejecutáramos sin optimizaciones.

Los resultados son iguales entre la PC hogareña y el cluster, debido a que como son resultados de cuentas numéricas y no de tiempo, no importa la potencia del dispositivo en donde se ejecute el código.

b. Ejecución algoritmo quadratic2.c en equipo hogareño con optimización -O3:

En este algoritmo se trabaja con datos double y se resuelven ambas potencias con pow, sin aprovechar la existencia de la funciones especificas para las variables de tipo float.

TipoDato / TIMES	100	1000	10000	100000
Float	1.292493	12.725648	125.901158	1251.805355
Double	1.470996	14.730939	146.717817	1422.816031

- Si realizamos la division entre los resultados que nos da el float con el resultado que nos da el double con los diferentes valores de TIMES, los resultados que obtendríamos seria: 1.13, 1.15, 1.16 y 1.13. En promedio el float se calcula 1,14 (14%) mas rápido que el double.

Ejecución algoritmo quadratic2.c en el cluster con optimización -O3

TipoDato / TIMES	100	1000	10000	100000
Float	6.028453	60.250481	602.464571	6024.614882
Double	6.955242	69.581198	696.387936	6957.625095

- Ejemplo del codigo utilizado en los scripts:

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --exclusive
#SBATCH -o ./outputs/output_quadatric2.txt
./script_quadatric2.c  #es codigo secuencial
```

Como es un algoritmo secuencial, en las configuraciones para el cluster pusimos: #SBATCH -N1 y “exclusive” para que solo nuestro código pueda correr en ese core.

Esto significa que se va a ejecutar en un solo núcleo. Este tiene 8GB de memoria RAM y dos procesadores Intel Xeon E5405 a 2.0GHz de 4 cores cada uno.

Tabla comparativa:

	Intel Xeon E5405	Intel Core i3 1115G4
CPU class	Server	Laptop
Año de lanzamiento	2007	2020
Clock Speed	2.0 GHz	3.0 GHz
Turbo Speed	Not Supported	Up to 4.1 GHz
# cores físicos	4	2
# threads	4	4
Cache	L1: 256KB L2: 12.0MB L3: No tiene	L1: 320KB L2: 5.0MB L3: 6MB
Intel Hyper-Threading	No	Si
Velocidad del bus	1333 MHz	4 GT/s

Fuentes: Intel Xeon, Intel Core i3.

Los tiempos de ejecución en el equipo hogareño son mas rápidas que en la CPU del cluster debido a que el algoritmo es secuencial. Como este es ejecutado sobre un único nodo del cluster, el intel i3 va a obtener mejores tiempos, ya que es mas potente que el intel Xeon.

Si quisiéramos sacar el aprovechamiento máximo del cluster deberíamos programar paralelamente, porque podríamos distribuir el procesamiento de nuestros algoritmos en los diferentes nodos. De esta manera no es tan crucial la potencia individual de cada nodo, sino la distribución del trabajo que se puede lograr.

c. Ejecución algoritmo quadratic3.c en equipo hogareño con optimización -O3:

TipoDato / TIMES	100	1000	10000	100000
Float	1.002258	10.809179	87.782515	860.719371
Double	1.464961	16.603722	147.130810	1473.942464

Ejecución algoritmo quadratic3.c en cluster con optimización -O3 con el cluster:

TipoDato / TIMES	100	1000	10000	100000
Float	3.435089	34.405444	341.436863	3428.249278
Double	6.596940	66.425398	661.054686	6626.435459

El tiempo requerido para la solución de sistemas de ecuaciones usando datos de tipo double es mayor que el tiempo requerido para la solución usando datos de tipo float, ya que el double utiliza más memoria y requiere más tiempo de procesamiento para realizar cálculos de precisión doble.

Ademas se aprovechan las funciones powf (para calcular la potencia) y sqrtf (para calcular la raíz cuadrada). Estas funciones provistas por la libreria “*math.h*”, son especificas para el

tipo de dato float, las resuelve de manera mas eficiente, y como consecuencia mejorando los tiempos de ejecucion

Otra característica que mejora los tiempos de ejecucion es que se declaran variables globales de tipo float, asi teniendo que evitar un casteo de tipo de datos.

---

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

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

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.

Algoritmos para buscar el máximo, mínimo y promedio de las matrices

Para buscar la forma mas eficiente de encontrar el máximo/mínimo de las matrices se nos ocurrieron 2 alternativas:

- Buscar el máximo, mínimo y promedio de A y de B dentro de los mismos for's
- Buscar el máximo y mínimo de A y de B en diferentes for's

Sin optimización:

Algoritmo / N	512	1024	2048	4096
Mismo for	0.004431	0.018441	0.072107	0.287387
Diferente for	0.005527	0.022472	0.089637	0.359596

Con optimización -O1:

Algoritmo / N	512	1024	2048	4096
Mismo for	0.001244	0.005782	0.022569	0.087825
Diferente for	0.002379	0.009930	0.039450	0.158314

Con optimización -O2:

Algoritmo / N	512	1024	2048	4096
Mismo for	0.000968	0.005390	0.020088	0.078970
Diferente for	0.001616	0.007213	0.028645	0.114907

Con optimización -O3:

Algoritmo / N	512	1024	2048	4096
Mismo for	0.000947	0.005498	0.020637	0.079501
Diferente for	0.001619	0.007299	0.028706	0.114674

Podemos ver que se obtienen mejores tiempos de ejecución calculando los valores de la matriz A y B dentro de un mismo for, sin importar el tipo de optimización.

Nuestra motivación para medir el tiempo de ejecución de obtener los valores de la matriz A y B en diferentes fors, es que al acceder a los valores de cada matriz por separado, podría haberse aprovechado mejor la cache debido a la **localidad de datos**.

Pero al ver los resultados, hemos llegado a la conclusion de que este aprovechamiento de la cache no es lo suficientemente bueno como para sobrepasar el costo de hacer 2 fors extra.

Debido a esto, lo mas optimo es buscar los valores de A y de B dentro de un mismo for.

Realización de la multiplicación de las matrices

Para lograr la optimización maxima de la multiplicación de AxB y de CxD realizamos lo siguiente:

1. Usar variables locales para ir almacenando valores: ir acumulando los resultados de las multiplicaciones en una variable y finalmente cuando tengo el resultado final, volcarlo en la matriz resultante.
    - Esto minimiza los accesos a la matriz, y ayuda en reducir el tiempo de ejecución.
2. Ordenar y acceder a las matrices A y C por filas

    Ordenar y acceder a las matrices B y D por columnas
    - Ordenar y acceder a las matrices según corresponda mejora los tiempos de ejecución. La causa de esto es que porque los datos al estar contiguos en memoria (por la reserva del bloque N*N en la heap), cuando yo traiga a la cache un bloque de datos, esta “información extra” que traiga, seguro van a ser los siguientes elementos que necesite de la matriz. A esto se lo llama **localidad de datos**.
3. Realizar la multiplicación por bloques, para aprovechar el uso de la cache
    - Como mencionamos anteriormente al realizar la multiplicación por bloques permite aprovechar el máximo la cache, asi mejorando los tiempos de ejecución. La optimización empleada es -O3:

Tipo / N	512	1024	2048	4096
No aprovechando el orden (1)	2.544773	51.957232	527.780679	4234.841948
Aprovechando orden (2)	0.219141	2.662488	22.021788	185.288704
Usando estrategia por bloques (3)	0.227133	1.805920	14.351750	115.834424

    En el cuadro se usan 3 formas distintas para resolver una multiplicación:

    1. No aprovechando el orden: Consiste en inicializar una matriz A por filas y una matriz B por filas y luego acceder a sus elementos de forma que no se aprovecha el principio de localidad (en A hay acceso secuencial en la memoria, en B hay saltos).

Por como funciona la cache se traen datos contiguos que no voy a necesitar en cada salto.

2. Aprovechando el orden: Consiste en inicializar una matriz A por filas y una matriz B por columnas y luego acceder a sus elemento aprovechando el principio de localidad, es decir que cuando se accede a la memoria se traen los datos contiguos que efectivamente voy a utilizar para el siguiente elemento minimizando la cantidad de accesos a memoria y causando menos “fallos de paginas”
3. Aprovechando el orden y usando estrategia por bloques: Igual que el caso numero 2 pero agregando la estrategia por bloques con tamaño 32. Se explica a continuación en detalle el uso de esta técnica.

Buscar el tamaño de bloque optimo

La técnica de multiplicación de matrices por bloque consiste en dividir las matrices en bloques más pequeños, generalmente en forma de submatrices, y luego realizar las multiplicaciones utilizando estas submatrices. Luego, los resultados parciales se combinan adecuadamente para obtener el producto final de la multiplicación de matrices.

Esta técnica suele ser mas efectiva en matrices con un N grande, que en matrices con una N chica. Esto se debe a que si la matriz es muy pequeña seguramente esta quepa completa en la cache y no tendría sentido separarla por bloques. Por otro lado si es grande, no va a caber dentro de la cache y por esto debo separarla en submatrices mas pequeñas que entren completas y asi aprovechar al máximo el uso de la cache.

Cuando hablamos del *tamaño del bloque* nos referimos al tamaño de las submatrices en las que se dividen las matrices. Entonces un gran dilema que se tiene siempre es: ¿Cual es el tamaño de bloque ideal?

Hay dos respuestas:

- La teórica, que consiste en realizar:

  > Tamaño de bloque * tipo de dato = cantidad de bytes que ocupa ese bloque

  Al resultado de esta multiplicación deberíamos compararlo con la cache de nuestro microprocesador

  El parámetro de comparación es el siguiente: el resultado de esta multiplicación debería ser menor a la cache de nivel2 (para que el bloque entre entera en la cache), pero no demasiado chica como para acceder demasiadas veces para buscar los valores a memoria.

  Debido a que este calculo teórico no siempre se aproxima a los verdaderos valores ideales, decidimos buscar el tamaño de bloque ideal de la forma practica:

- La práctica: Consiste en en ejecutar el algoritmo de multiplicación de matrices por bloque, variando el tamaño de N y del bloque, para buscar el numero ideal. El siguiente cuadro refleja los tiempos de ejecución de una multiplicación de matrices cambiando estas variables:

Pruebas ejecutadas con -O3:

Tamaño bloque / N	512	1024	2048	4096
4	0.172003	1.433299	11.376213	94.409737
8	0.123297	1.008424	8.091210	65.258525
16	0.179751	1.456238	11.738114	94.141216
32	0.198157	1.579175	12.601108	100.834985
64	0.233529	1.857430	14.879430	118.274667
128	0.227094	1.810520	14.362459	115.489313
256	0.232049	1.851035	14.739783	209.200762
512	0.227864	1.797492	22.025097	187.623296

Segun los resultados provistos podemos deducir que el tamaño optimo de bloque es de 8.

Calculo de las potencias

Dos opciones:

- Realizar la potencia de 2 manualmente (multiplicando el valor por si mismo)
- Usar la función pow, provista por la librería math: Esta función permite elevar un número a una potencia, especificados por parámetro
- Vector pre-cargado: debido a que los valores que va a tomar cada celda de esta matriz va de 1 a 40, se carga un vector con cada valor elevado al cuadrado, y asi cada vez que ese valor aparece, simplemente se le asignaria el valor correspondiente del vector.

Sin optimización:

Técnica / N	512	1024	2048	4096
Sin pow	0.003955	0.013448	0.050707	0.200212
Con pow	0.006931	0.024885	0.097272	0.385900
Pre-cargando resultados en vector	0.004288	0.014393	0.055142	0.217765

Con optimización -O3:

Algoritmo / N	512	1024	2048	4096
Sin pow	0.001955	0.005712	0.019897	0.077079
Con pow	0.000788	0.005582	0.019745	0.076468
Pre-cargando resultados en vector	0.002210	0.005963	0.021816	0.083125

El algoritmo que ofrece mejores resultados sin optimizacion es multiplicando ese valor por si mismo. Esto puede suceder debido que realizar una multiplicacion es menos costoso que hacer el llamado a la funcion “pow” para que ejecute su codigo.

Pero si se prueba con la optimizacion -O3, sucede que la funcion “pow” arroja mejores resultados. Lo que podria suceder es que este active flags que optimicen lo que sucede dentro del codigo de la funcion.

Como la optimizacion que vamos a utilizar para ejecutar la ecuacion final es -O3, en el codigo de este, utilizaremos el pow para calcular la potencia.

Ecuación final

Tiempos de ejecución de la ecuación con tamaño de bloque = 8

Optimización / N	512	1024	2048	4096
-O3	0.603101	4.907956	38.796849	305.858869

Archivos entregados:

1. Ecuacion.c: contiene al el algoritmo completo para calcular la ecuación.
2. comp_BuscarMax_Min_Prom.c: contiene los códigos de prueba para la elección del mejor algoritmo para calcular estos valores.
3. mult_matrices.c: contiene los códigos para comprobar la manera mas veloz de realizar la multiplicacion de matrices.
4. prueba_porBloques.c: contiene el código para comprobar el tamaño de bloque optimo en la multiplicación de matrices por bloque.
5. prueba_potencia.c: contiene los códigos para comprobar el algoritmo mas veloz para calcular la potencia de todos los elementos de una matriz.