Entrega 1

Sistemas Paralelos 2020

Milla, Andrés - 14934/6
Jara, Rodrigo Jose - 14854/8

Para tomar el tiempo de los ejercicios se utilizó:

- **CPU** → **Intel I7-5500u**.
    - Caché → 4MB
    - Cantidad de núcleos → 2
    - Cantidad de subprocesos → 4
    - Frecuencia básica → 2.40 Ghz
    - Frecuencia máxima → 3.00 GHz

Ejercicio 1

Resolver los ejercicios 1 y 2 de la práctica 1. Evaluar para N=512, 1024 y 2048.

1. Analizar el algoritmo matrices.c que resuelve la multiplicación de 2 matrices cuadradas de NxN. ¿Dónde cree que se producen demoras?

En el algoritmo original se organizaba la matriz B por filas, esto es malo, ya que en la multiplicación de matrices por cada iteración se está multiplicando filas de A * columnas de B, por lo tanto el acceso a memoria es "salteado" y no estaríamos aprovechando la localidad de la caché.
También se está haciendo N^3 escrituras sobre C[i,j], esto lo optimizamos utilizando una variable local para acumular el producto de A[i,k]*B[k,j] se reduce a N^2 escrituras sobre C[i,j].

¿Cómo podría optimizarse el código? Implementar una solución optimizada y comparar los tiempos probando con diferentes tamaños de matrices.

Solución → Organizar a B por columnas así se puede traer los bloques contiguos a caché evitando acceder salteadamente como se hacía anteriormente. *El código se puede ver en el archivo **opt-ejercicio-01.c***.

Comparación de Tiempos:

N	**Tiempo del original (s)** *original-ejercicio-01-1.c*	**Tiempo del optimizado (s)** *opt-ejercicio-01-1.c*
512	3.160290	1.105641
1024	50.350494	9.110086
2048	467.846483	71.414711

2. Describir brevemente cómo funciona el algoritmo multBloques.c que resuelve la multiplicación de matrices cuadradas de NxN utilizando una técnica de multiplicación por bloques. Ejecutar el algoritmo utilizando distintos tamaños de matrices y distintos tamaños de bloque.

El algoritmo resuelve una matriz nxn dividiéndola en NxN submatrices rxr. Al hacer esta subdivisión trata a las submatrices como si fueran elementos de una matriz y realiza la multiplicación entre estos.

Según el tamaño de las matrices y de bloque elegido ¿Cuál es más rápido? ¿Por qué? ¿Cuál sería el tamaño de bloque óptimo para un determinado tamaño de matriz?

El tamaño de bloque óptimo debería ser ⅓ del tamaño de la caché, así se puede acceder a los datos de las 3 submatrices (bloques) en menos accesos de memoria y aprovechar la localidad de la caché. Por esta razón los resultados marcados en rojo de la tabla de abajo son los más rápidos, ya que esos tamaños de bloque se adecuan mejor a la caché y con solo 1 acceso a memoria principal se podrían traer una submatriz de la matriz original a caché.

Por lo tanto:
- Se requieren 3 submatrices en memoria caché (A,B,C)
- Tamaño de la caché = C **bytes**
- Dimensión del bloque = r
- Tamaño de Double = 8 **bytes**

La dimensión del bloque óptimo teórico sería: $r = \frac{\sqrt{C}}{8 \cdot 3}$

Cada bloque ocupará r^2 ***bytes*** *en memoria.*

Comparación:

N	Dimensión de c/bloque	Tiempo (s)
Matriz = 512x512		
1	512	1.085978
2	256	1.002733
4	128	0.663164
8	64	0.623769
16	32	0.596768
32	16	0.611352

64	8	0.673682
128	4	0.741657
256	2	1.061447
512	1	2.341459
Matriz = 1024x1024		
1	1024	28.088428
2	512	10.082286
4	256	8.585843
8	128	5.484099
16	64	4.869360
32	32	4.732319
64	16	4.817532
128	8	5.088901
256	4	5.693456
512	2	8.397084
1024	1	46.324505
Matriz = 2048x2048		
1	2048	238.914689
2	1024	211.216877
4	512	69.885065
8	256	64.032486
16	128	41.541137
32	64	39.582692
64	32	37.870940
128	16	39.335203
256	8	41.080991

512	4	48.039681
1024	2	101.217858
2048	1	402.683766

Ejercicio 2

Utilizando las optimizaciones realizadas en el ejercicio 1 de la práctica 1, resolver la multiplicación de matrices: D=ABC.
¿Cómo debe ordenar en memoria las matrices A, B, C y D para alcanzar un mejor rendimiento? Evaluar para N=512, 1024 y 2048.

Las matrices se deben organizar de la siguiente manera:

- La matriz A por filas
- La matriz B por columnas
- La matriz AB por filas
- La matriz C por columnas
- La matriz D por filas

La solución se encuentra en el archivo ***ejercicio-02.c***

N	Tiempo (s)
512	2.289638
1024	17.717014
2048	143.614827

Ejercicio 3

Resolver el ejercicio 5 de la práctica 1

a)

```
andres@andres ~/Documentos/programacion/facultad/sp/p1/e5
  % ./quadatric1
2.000000 2.000000Soluciones Float: 2.00000      2.00000
2.000316 1.999684Soluciones Double: 2.00032     1.99968
```

Los resultados de float están erróneos porque solo soporta 6 decimales, en cambio el double soporta hasta 14 decimales.
Por lo tanto, en el caso de float toma a C = 3.999999 como C = 4.0, en cambio en double sigue igual.

Si hacemos las cuentas paso a paso, se puede ver cómo difieren:

```c
void dbl_solve(double a, double b, double c)
{
    double d = pow(b,2) - 4.0*a*c;      // d  = 4^2 - 4*1*3.99.. = 0, 000 000 399 999 999
    double sd = sqrt(d);                // sd = 0.00063245553
    double r1 = (-b + sd) / (2.0*a);    // r1 = (4 + sd) / 2 = 2.00031622776
    double r2 = (-b - sd) / (2.0*a);    // r2 = (4 - sd) / 2 = 1.99968377224
    printf("Soluciones Double: %.5f\t%.5f\n", r1, r2);
}

void flt_solve(float a, float b, float c)
{
    float d = pow(b,2) - 4.0*a*c;       // d  = 4^2 - 4*1*4 = 0
    float sd = sqrt(d);                 // sd = 0
    float r1 = (-b + sd) / (2.0*a);     // r1 = ((4 + 0) / 2) = 2
    float r2 = (-b - sd) / (2.0*a);     // r2 = ((4 - 0) / 2) = 2
    printf("Soluciones Float: %.5f\t%.5f\n", r1, r2);
}
```

b) Se puede notar que la solución de Double tarda menos que la solución Float.
Si bien el tamaño de cada Float es de 4 Bytes, mientras que el de Double es de 8 Bytes y esto hace que se puede alojar más Floats en un bloque de memoria que el cpu trae a caché, se puede ver que en la solución float se deben realizar muchas conversiones de tipo de datos lo que provoca que tarde más.
Esto solo se puede ver si se ejecuta sin optimizaciones, ya que si se ejecuta con optimizaciones, el compilador va a optimizar estas conversiones por nosotros.

TIMES	Con optimización (-O3)		Sin optimización (-O0)	
50	**Float**	**Double**	**Float**	**Double**
	1.1358655	1.2969685	10.179949	9.280958
100	**Float**	**Double**	**Float**	**Double**
	2.252166	2.6207003	19.2649985	18.172425
150	**Float**	**Double**	**Float**	**Double**
	3.6537465	4.3139655	29.889712	28.093564

c) La diferencia es que en *quadractic3.c* se realiza la solución Float con todos números float, y en la anterior no.
Esto provoca que se tengan que hacer menos conversiones en *quadratic3.c*, comparemos:

```c
// computar soluciones usando float
tick = dwalltime();

for (j=0; j<TIMES ; j++)
    for (i=0; i<N ; i++) {
//      flt_solve(fa[i], fb[i], fc[i]);
        float d = powf(fb[i],2.0f) - 4.0f*fa[i]*fc[i];
        float sd = sqrtf(d);
        float r1 = (-fb[i] + sd) / (2.0f*fa[i]);
        float r2 = (-fb[i] - sd) / (2.0f*fa[i]);
    }
```

```c
// computar soluciones usando float
tick = dwalltime();

for (j=0; j<TIMES ; j++)
    for (i=0; i<N ; i++) {
//      flt_solve(fa[i], fb[i], fc[i]);
        float d = pow(fb[i],2) - 4.0*fa[i]*fc[i];
        float sd = sqrt(d);
        float r1 = (-fb[i] + sd) / (2.0*fa[i]);
        float r2 = (-fb[i] - sd) / (2.0*fa[i]);
    }
```

i) *quadratic2.c:*

(1) `float d = pow(fb[i],2) - 4.0*fa[i]*fc[i];`

(2) Se realiza la siguiente operación `double - double * float * float` para luego el resultado castearlo a Float.

(3) `sqrt(d)` recibe un float, y debe castearse a Double.

(4) Luego en el cálculo de las 2 raíces se hace: `double + float / (double * double).` Otra vez tendremos conversiones.

ii) *quadratic3.c:*

(1) `float d = powf(fb[i],2.0f) - 4.0f*fa[i]*fc[i];`

(2) Se realiza la siguiente operación `float - float * float * float` y se guarda el resultado como Float. No hay conversiones.

(3) `sqrtf(d)` recibe un float, por lo tanto No hay conversiones.

(4) Luego en el cálculo de las 2 raíces se hace: `float + float / (float * float).` No hay conversiones

Por lo tanto el tiempo de ejecución de quadratic3.c es menor ya que demanda menos conversiones que quadratic2.c:

	Tiempo quadratic2.c		Tiempo quadratic3.c	
Sin optimización -O0	**Float**	**Double**	**Float**	**Double**
	19.41	18.01	17.10	18.63
Con optimización -O3	**Float**	**Double**	**Float**	**Double**
	2.28	2.55	1.45	2.54

** Fueron corridos con TIMES=100*

Ejercicio 4

Resolver el ejercicio 12 de la práctica 1.

Algoritmo	SECUENCIA	PRIMOS	PARES	IMPARES	UNIFORME CRECIENTE	UNIFORME DECRECIENTE	ALEATORIA
Cantidad de listas = **512** - Cantidad mínima = 512 - Cantidad máxima = 1024							
Merge múltiple	0.721063	0.798857	0.702054	0.702512	1.181418	1.499064	0.842761
Merge incremental	0.396107	0.392985	0.388073	0.393920	0.530117	0.529524	0.383708
Merge de a pares	0.012277	0.022341	0.012278	0.012062	0.019720	0.021005	0.022374
Cantidad de listas = **1024** - Cantidad mínima = 512 - Cantidad máxima = 1024							
Merge múltiple	2.919777	3.300644	2.943994	2.953998	4.828477	5.972972	3.683111
Merge incremental	1.579485	1.595699	1.578967	1.586754	2.127085	2.137698	1.536507
Merge de a pares	0.027153	0.050717	0.027238	0.026984	0.043793	0.044269	0.051689
Cantidad de listas = **2048** - Cantidad mínima = 512 - Cantidad máxima = 1024							
Merge múltiple	13.126934	15.450503	13.16991	13.25933	27.271473	33.665075	19.628355
Merge incremental	6.280526	6.344511	6.398389	6.373853	8.561198	8.463890	6.194578
Merge de a pares	0.063820	0.115964	0.064191	0.059705	0.101325	0.100649	0.108423

El algoritmo que saca un mayor rendimiento es el “**Merge de a pares”**, esto sucede porque aprovecha mejor la memoria caché, ya que en el primer paso mergea un par de listas y en las siguientes etapas mergea cada par de listas restantes hasta que quede una sola con todos los elementos ordenados. En el mejor de los casos por cada iteración aloja cada par de listas en caché y se trabaja allí con sus elementos, lo que provoca tener que acceder menos veces a

memoria principal.
La cantidad de pasos para resolver este algoritmo es del $O(\log_2(N))$.

En cambio la estrategia “**Merge incremental”** sigue un algoritmo de tipo pipe, en donde primero se mergean 2 buffers y dicha salida del buffer es la entrada de otro buffer, lo que nos da una cantidad de pasos de $O(\log_2(N))$. Pero por cada iteración se están recorriendo todos los elementos desde el principio del buffer actual, por lo tanto se produce un acceso salteado a memoria por cada fracción de elementos que no entren en caché y esto se replica en cada una de las etapas, lo que termina dando una.

Por otro lado, el algoritmo **“Merge multiple”** simplemente recorre todas las listas linealmente, determina un mínimo y lo coloca en un arreglo. Lo que termina dando una complejidad de $O(N)$. El otro problema de este algoritmo es que es no tiene en cuenta para nada el principio de localidad de la memoria caché ya que recorre desde el principio al final todas las listas, es decir, recorre diferentes bloques de memoria para determinar un mínimo, y luego se repite el mismo proceso hasta que la lista esté ordenada, lo que provoca un acceso totalmente salteado a memoria y esto provoca que en la mayoría de los casos se esté accediendo a memoria principal desaprovechando los datos almacenados en caché, por eso mismo es el que peor rendimiento produce.