Universidad Nacional de La Plata

Facultad de Informática

Sistemas Distribuidos y Paralelos

Trabajo 1: Programación con memoria compartida

Arambarri, Delfina	724/7
Espinoza, Juan Manuel	801/3

Mayo 2018

Aclaraciones iniciales:

1. Los tiempos se calculan en todos los casos (tanto secuenciales como paralelos) desde que se inicia la multiplicación hasta que termina. Sin considerar inicializaciones o transposiciones.
2. Adjuntamos al final del informe un anexo con las características de la computadora en que fueron ejecutados los algoritmos.

Métricas a utilizar:

Speed up: es una métrica de rendimiento que representa el beneficio relativo alcanzado al ejecutar un programa. Se calcula como Tiempo del algoritmo secuencial (T_s) dividido Tiempo del algoritmo paralelo (T_p).

$$S = \frac{Ts}{Tp}$$

Eficiencia: es una métrica de rendimiento que indica el porcentaje de tiempo en el que los procesadores están realizando trabajo útil. Se calcula como el Speed up dividido la cantidad de procesadores. La misma va desde 0 hasta 1.

$$E = \frac{S}{P}$$

EJERCICIO 1:

Partimos de un **algoritmo secuencial** donde realizamos la multiplicación de este modo:

```
Desde i=0 hasta N (matriz de dimensión NxN)
    Desde j=0 hasta N
        Desde k=0 hasta N
        matriz resultado en la posición [i,j] = Σ A[i,k] * A'[j,k] (siendo ' la
        traspuesta)
```

Pseudocódigo 1.1

Para ello primero tuvimos que inicializar la matriz *A* (matriz a multiplicar) y *resultado*, que inicialmente cada posición está en cero. También alocar memoria para las matrices mencionadas y para guardar la traspuesta, la cual llamamos *At*.

Para trasponer la matriz *A* realizamos lo siguiente:

```
Desde i=0 hasta N
    Desde j=0 hasta N
        guardamos temporalmente el valor de A[i,j]
        asignamos a la posición At[i,j] el valor de A[j,i]
        asignamos a la posición At[j,i] el valor guardado de A[i,j]
```

Pseudocódigo 1.2

Para realizar el algoritmo paralelo en **Open MP** utilizamos la directiva *#pragma omp parallel for collapse(2)* que distribuye las iteraciones del loop proporcionalmente entre los hilos. El *collapse(2)* indica la cantidad de bucles *for* a paralelizar.

Para el algoritmo paralelo en **Pthreads** se requieren más operaciones:

Cada hilo va a ejecutar la función de multiplicación. Para ello debemos obtener qué hilo está ejecutando, mediante su ID. Teniendo este valor calculamos desde y hasta dónde cada hilo debe realizar la multiplicación de la siguiente manera:

Primero debemos saber cual es la dimensión del bloque a multiplicar por cada hilo. Para ello dividimos N por la cantidad de hilos.

Luego calculamos el inicio y el fin:

```
inicio hilo con ID=X = X*tamaño del bloque
fin= inicio + tamaño del bloque
```

Pseudocódigo 1.3

Finalmente realizamos la multiplicación:

```
Desde i=inicio hasta fin
    Desde j=0 hasta N
        Desde k=0 hasta N
        matriz resultado en la posición [i,j] = Σ A[i,k] * A'[j,k] (siendo
        ' la traspuesta)
```

Pseudocódigo 1.4

Calculamos los tiempos para distintos N del algoritmo secuencial, y para los paralelos con 2 y 4 hilos:

Tabla 1: Tiempo ejercicio 1

Valor de N	Secuencial	Open MP		Pthreads	
		2 hilos	4 hilos	2 hilos	4 hilos
512	0.744464	0.498579	0.269702	0.404616	0.225369
1024	5.864660	3.795886	1.984635	3.032588	1.608417
2048	47.011576	30.327124	15.913648	24.077725	12.473785

Luego calculamos las métricas correspondientes:

Tabla 2: Métricas ejercicio 1

Valor de N/N° de hilos	Open MP		Pthreads	
	Speed Up	Eficiencia	Speed Up	Eficiencia
512/2	1,4935	0,7467	1,8399	0,9199
512/4	2,8833	0,7208	3,3033	0,8258
1024/2	1,5450	0,7725	1,9338	0,9669
1024/4	2,9550	0,7387	3,6462	0,91155
2048/2	1,5501	0,7750	1,9524	0,9762
2048/4	2,9541	0,7385	3,7688	0,9422

Conclusiones:

Tanto en Open MP como en Pthreads, el speed up y la eficiencia aumentan a medida que crece N. Respecto a la cantidad de hilos, el speed up máximo se encuentra con el N máximo (2048) y cuatro hilos. Para todas las opciones de N la eficiencia resulta mejor en dos hilos que en cuatro, por lo que la eficiencia máxima se consigue para N=2048 y dos hilos. Además, hay una notable mejora en Pthreads.

EJERCICIO 2:

Para este ejercicio se nos solicitó que realicemos el algoritmo que resuelva la siguiente fórmula matemática:

$$M = \overline{u.l}.AAC + \overline{b}LBE + \overline{b}DUF$$

Dónde A, B, C, D, E y F son matrices de *NxN*; U y L son matrices triangulares de *NxN*, una superior y otra inferior; y b, u y l son los promedios de las matrices B, U y L, respectivamente.

Al igual que en el ejercicio 1 comenzamos con el **algoritmo secuencial** declarando, reservando la memoria suficiente para cada matriz e inicializando las mismas con los valores necesarios para poder llevar a cabo la resolución del ejercicio.

Como la fórmula está compuesta por muchas operaciones matemáticas, se dividió el algoritmo en etapas para ir resolviendo cada parte de la fórmula y luego calcular su resultado final.

Primero comenzamos con en el siguiente pseudocódigo 2.1:

```
Desde i=0 hasta N (matriz de dimensión NxN)
        Sumar variable para el promedio de B
        Si debo sumar
                Sumar variable para el promedio de U
                Sumar variable para el promedio de L
        Desde j=0 hasta N
                Desde k=0 hasta N
                        matriz resultado1 en la posición [i,j] =  ΣA[i,k]*A'[i,k]
                        matriz resultado2 en la posición [i,j] = ΣB[i,k]*E'[i,k]
                        matriz resultado3 en la posición [i,j] = ΣD[i,k]*F'[i,k]
```

Pseudocódigo 2.1

Como se puede apreciar en el pseudocódigo se realizan diferentes cálculos para ir reduciendo el cálculo general de la fórmula. En estos se calcula los resultados de:

- Resultado parcial 1 = A.ATraspuesta
- Resultado parcial 2 = B.E
- Resultado parcial 3 = D.F

Cada resultado es almacenado en una estructura auxiliar que fue inicializada en ceros cuando se realizó la iniciación de cada matriz. A la vez se utilizó el procesamiento de los **for** para realizar la suma en tres variables de los valores que se almacenan en las variables B, U y L, así después se puede obtener el promedio de estas. Para las matrices U y L se obviaron los valores ceros para ahorrar procesamiento.

Una vez calculados estos resultados parciales, se prosigue con el siguiente pseudocódigo:

```
Desde i=0 hasta N (matriz de dimensión NxN)
        Desde j=0 hasta N
                Desde k=0 hasta N
                        matriz resultado4 [i,j]=Σmatriz resutlado1[i,j]*C[j,k]

Desde i=0 hasta N (matriz de dimensión NxN)
        Desde j=0 hasta N
                Desde k=0 hasta j
                        matriz resultado5 [i,j]=Σmatriz resutlado2[i,j]*L[j,k]
                Desde k=j hasta N
                        matriz resultado6 [i,j]=Σmatriz resutlado3[i,j]*U[j,k]
```

Pseudocódigo 2.2

A diferencia de la primer parte en esta distribución de la fórmula se realizan dos **for**. El primero lleva a cabo la multiplicación del *resultado1* por la matriz *C* y el segundo se encarga de multiplicar los resultados parciales 2 y 3 por las matrices *L* y *U*, respectivamente. Fue realizado de esta manera ya que en el segundo **for** no se procesan todas posiciones de las matrices *L* o *U,* ya que gran parte de estas matrices tienen valor cero y su multiplicación no afectaría el resultado final. Con esto nos ahorramos mucho procesamiento de cálculo.

```
Valor a multiplicar = 1/(N*N)
Cálculo de promedios
Desde i=0 hasta N (matriz de dimensión NxN)
        Desde j=0 hasta N
                matriz resultado4 [i,j]= matriz resultado4[i,j]*promedio l.u
                matriz resultado5 [i,j]=matriz resultado5[i,j]*promedio b
                matriz resultado6 [i,j]=matriz resultado6[i,j]*promedio b
```

Pseudocódigo 2.3

Siguiendo el código se calculan los valores de los promedios de las matrices. Para ello tomamos los que sumamos en el pseudocódigo 1 y las multiplicamos por el valor *1/(N*N)*. Al realizar la multiplicación nos ahorramos 3 divisiones ganando tiempo, ya que la multiplicación consume menor tiempo de procesamiento que la división.

Una vez que se tiene el valor de los 3 promedios solicitados se multiplican estos por las diferentes matrices resultados. De esta forma lo único que falta para completar el cálculo de la fórmula es sumar los distintos valores y almacenarlos en una variable. Esto se puede observar en el pseudocódigo 2.4.

```
Desde i=0 hasta N (matriz de dimensión NxN)
        Desde j=0 hasta N
                M[i,j]=resultado4[i,j]+resultado5[i,j]+resultado[i,j]
```

Pseudocódigo 2.4

Al igual que en el ejercicio 1 para realizar el algoritmo paralelo en **Open MP** utilizamos la directiva *#pragma omp parallel for collapse(2)* que distribuye las iteraciones del loop proporcionalmente entre los hilos. El *collapse(2)* indica la cantidad de bucles *for* a paralelizar. Para no provocar que los hilos se duerman cada vez que se termina de procesar un *#pragma omp parallel for collapse(2)* se coloca al inicio del programa la directiva *#pragma omp parallel* la cual provocará que los hilos no se duerman y se mantengan en ejecución continuamente.

Para el algoritmo paralelo en **Pthreads** se requieren más operaciones. Al igual que el ejercicio 1 cada hilo va a ejecutar una función multiplicar y dependiendo de su índice se manejara el procesamiento de cada **for**. El pseudocódigo de este programa se puede observar en el pseudocódigo 2.5.

```
Función multiplicar
        Indices propios
        Pseudocódigo 2.1
        Barrera 1
        Pseudocódigo 2.2
        Barrera 2
        Pseudocódigo 2.3
        Barrera 3
        Pseudocódigo 2.4
```

Pseudocódigo 2.5

Como se ve en el pseudocódigo 2.5 la función multiplicar está compuesta por los pseudocódigos explicados previamente. Se puede observar que se utilizan barreras entre cada pseudocódigo para evitar que un hilo tome valores erróneos de las matrices que se comparten. Ya que diferentes hilos procesaran las matrices, se puede generar que un hilo se ejecute a mayor velocidad que otro y tome valore erróneos debido a que el otro hilo no terminó de procesar. Al tener las barreras se evita esto.

Calculamos los tiempos para distintos N del algoritmo secuencial, y para los paralelos con 2 y 4 hilos:

Tabla 3: Tiempo ejercicio 2

Valor de N	Secuencial	Open MP		Pthreads	
		2 hilos	4 hilos	2 hilos	4 hilos
512	3.528848	2.449501	1.326558	2.045015	1.014006
1024	28.307462	19.887202	10.098734	14.937539	7.734807
2048	279.934849	199.098588	103.709647	148.694910	79.075121

En la tabla 4 se pueden observar las métricas calculadas del ejercicio 2.

Tabla 4: Métricas ejercicio 2

Valor de N/N° de hilos	Open MP		Pthreads	
	Speed Up	Eficiencia	Speed Up	Eficiencia
512/2	1,4406	0,7203	1,7255	0,8627
512/4	2,6601	0,6650	3,4801	0,8700
1024/2	1,4234	0,7117	1,8950	0,9475
1024/4	2,8030	0,7007	3,6597	0,9149
2048/2	1,4060	0,7030	1,8826	0,9413
2048/4	2,6992	0,6748	3,5401	0,8850

Conclusiones:

Para Open MP, en cuanto a speed up, el máximo se obtiene con N=1024 y cuatro hilos, luego comienza a bajar. Respecto a la eficiencia, resulta mejor en todos los casos para dos hilos comparando con cuatro.

En Pthreads, como en el ejercicio anterior, se notan mejoras respecto a Open MP. También el speed up máximo se obtiene con N=1024 y cuatro hilos. La eficiencia, excepto para N=512, resulta mejor con dos hilos que con cuatro.

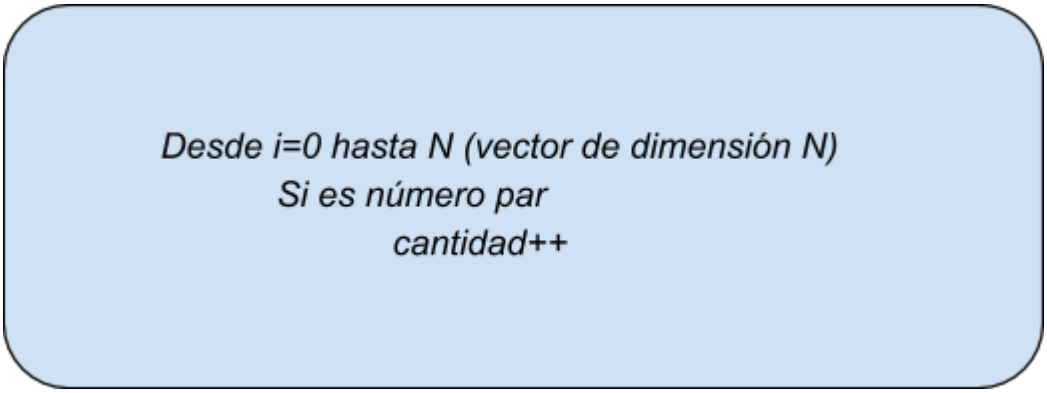
Aclaración:

Mientras se realizaba el informe nos dimos cuenta de que utilizamos mal la propiedad asociativa para matrices. El algoritmo propuesto no es una solución para la fórmula $M = \overline{u}.l.AAC + \overline{b}LBE + \overline{b}DUF$ sino que resuelve $M = \overline{u}.l.AAC + \overline{b}LBE + \overline{b}DFU$.

EJERCICIO 3:

En este ejercicio se nos solicitó calcular la cantidad de números pares que existen en un arreglo de tamaño N. Para ello inicializamos un arreglo con números *random* entre 0 y 99 y una variable que cuenta la cantidad de valores pares en ese arreglo.

El **algoritmo secuencial** está definido por el pseudocódigo 3.1.



Pseudocódigo 3.1

El código es sencillo, se recorre el vector y por cada valor se verifica si es par o no. Para verificar si el número es par se utiliza la función *(número&1) == 0*, ya que en prácticas anteriores se pudo verificar que utilizar *&* es mejor que utilizar el módulo.

El algoritmo en **open MP** utiliza la directiva *#pragma omp parallel for reduction(+:pares)* lo que hace es dividir el **for** en la cantidad de hilos que se generaron y el **reduction** realiza la suma de la variable que contiene la cantidad de pares y la reduce en una variable.

Los tiempos de este ejercicio se observan en la Tabla 5.

Tabla 5: Tiempos ejercicio 3

Valor de N	**Secuencial**	**Open MP**	
		2 hilos	**4 hilos**
10000	0.000145	0.000157	0.006264
100000	0.001445	0.000490	0.004759
1000000	0.011579	0.005248	0.011487
2000000	0.021678	0.011404	0.011549
3000000	0.027743	0.014103	0.009280

Las métricas de este ejercicio se observan en la Tabla 6.

Tabla 6: Métricas ejercicio 3

Valor de N/N° de hilos	Open MP	
	Speed Up	Eficiencia
10000/2	0,9235	0,4617
10000/4	0,0231	0,0057
100000/2	2,9489	1,4744
100000/4	0,3036	0,0759
1000000/2	2,2063	1,1031
1000000/4	1,0080	0,2520
2000000/2	1,9009	0,9504
2000000/4	1,8770	0,4692
3000000/2	1,9671	0,9835
3000000/4	2,9895	0,7473

Conclusiones:

Se puede observar que hay una clara diferencia cuando se ejecuta con dos o cuatro hilos. Cuando se ejecuta con dos y se tiene un valor de N mayor a 10000, el algoritmo paralelo se ejecuta de manera más rápida que el secuencial.

El valor de Speed up es alto en la mayoría de los casos y para N igual a 100000 y 1000000 el valor es mayor a la cantidad de procesadores en uso. Creemos que esto se debe a tener un Speed up superlineal, lo cual se puede estar generando porque la versión paralela del algoritmo realiza menos trabajo que la versión secuencial. Como consecuencia se tiene una eficiencia mayor a 1 en estos dos casos y cercanas a 1 en los demás.

Al ejecutar con cuatro hilos el tiempo que tarda el algoritmo es mayor que el secuencial mientras que el valor de N sea menor a 1000000. Esto se debe al overhead generado en el algoritmo paralelo. Por esto no conviene paralelizar el algoritmo con cuatro hilos para valores de N muy chicos. En este caso el Speed up siempre da menor a la cantidad de procesadores que se están utilizando, por lo que no existe Speed up superlineal.

Anexo:

ARQUITECTURA DE LA COMPUTADORA

Mediante el comando *cat /proc/cpuinfo* podemos observar las características de la computadora y confirmar que posee cuatro cores físicos:

processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 42
model name : Intel(R) Core(TM) i5-2310 CPU @ 2.90GHz
stepping : 7
microcode : 0x25
cpu MHz : 1691.175
cache size : 6144 KB
physical id : 0
siblings : 4
core id : 0
*cpu cores : **4***
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 popcnt tsc_deadline_timer aes xsave avx lahf_lm epb tpr_shadow vnmi flexpriority ept vpid xsaveopt dtherm ida arat pln pts
bugs :
bogomips : 5786.78
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
power management:

processor : 1
vendor_id : GenuineIntel
cpu family : 6
model : 42
model name : Intel(R) Core(TM) i5-2310 CPU @ 2.90GHz
stepping : 7

microcode : 0x25
cpu MHz : 1599.871
cache size : 6144 KB
physical id : 0
siblings : 4
core id : 1
cpu cores : 4
apicid : 2
initial apicid : 2
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 popcnt tsc_deadline_timer aes xsave avx lahf_lm epb tpr_shadow vnmi flexpriority ept vpid xsaveopt dtherm ida arat pln pts
bugs :
bogomips : 5786.78
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
power management:

processor : 2
vendor_id : GenuineIntel
cpu family : 6
model : 42
model name : Intel(R) Core(TM) i5-2310 CPU @ 2.90GHz
stepping : 7
microcode : 0x25
cpu MHz : 1616.976
cache size : 6144 KB
physical id : 0
siblings : 4
core id : 2
cpu cores : 4
apicid : 4
initial apicid : 4
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 popcnt tsc_deadline_timer aes xsave avx lahf_lm epb tpr_shadow vnmi flexpriority ept vpid xsaveopt dtherm ida arat pln pts
bugs :
bogomips : 5786.78
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
power management:

processor : 3
vendor_id : GenuineIntel
cpu family : 6
model : 42
model name : Intel(R) Core(TM) i5-2310 CPU @ 2.90GHz
stepping : 7
microcode : 0x25
cpu MHz : 1667.839
cache size : 6144 KB
physical id : 0
siblings : 4
core id : 3
cpu cores : 4
apicid : 6
initial apicid : 6
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 popcnt tsc_deadline_timer aes xsave avx lahf_lm epb tpr_shadow vnmi flexpriority ept vpid xsaveopt dtherm ida arat pln pts
bugs :
bogomips : 5786.78
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
power management: