Universidad Nacional de La Plata
Facultad virtual de Ingeniería e Informática

Entrega Nº1

Programación con memoria compartida

Sistemas distribuidos y paralelos

12 de mayo de 2018

Andreini, Antonella 822/9
Flores, Brian 795/4

El código de los ejercicios se encuentra disponible en
https://github.com/brianflores/sdp-entrega1.
Debido a problemas de funcionamiento en las computadoras de la sala de PC de posgrado, los tiempos fueron medidos en una notebook con procesador Intel i7-7700HQ de 4 cores y 16gb de ram.

1. Resolver con Pthreads y OpenMP la siguiente expresión:

R=AA

Donde A es una matriz de NxN. Analizar el producto AA y utilizar la estrategia que proporcione el mejor tiempo de ejecución.

Evaluar N=512, 1024 y 2048.

Resolución con un algoritmo secuencial:

Se comienza el ejercicio planteando una solución secuencial donde se inicia la matriz A en el valor uno y se multiplica por la misma matriz A dejando el resultado en R. Una de las observaciones es que al iniciar A ordenada por filas, la multiplicación de ambas matrices se deben realizar por filas.

Código de multiplicación por filas:

```
//Realiza la multiplicacion

for(i=0;i<N;i++){
     for(j=0;j<N;j++){
              R[i*N+j]=0;
     for(k=0;k<N;k++){
              R[i*N+j]= R[i*N+j] + A[k*N+i]*A[k*N+j];
              }
     }
}
```

Entonces se tiene en cuenta una segunda opción que es realizar la transpuesta de la matriz A para poder multiplicar A ordenada por filas y A transpuesta que quedaría ordenadas por columnas.

Para calcular una matriz transpuesta se debe realizar el siguiente cálculo:

$$A = \begin{pmatrix} 2 & 3 & 0 \\ 1 & 2 & 0 \\ 3 & 5 & 6 \end{pmatrix} \qquad A^t = \begin{pmatrix} 2 & 1 & 3 \\ 3 & 2 & 5 \\ 0 & 0 & 6 \end{pmatrix}$$

Desarrollando el código quedaría:

```
//Genera matriz transpuesta
       for(i=0;i<N;i++){
        for(j=0;j<N;j++){
                     temp = At[i*N+j];
                     At[i*N+j]= At[j*N+i];
                     At[j*N+i]= temp;
        }
       }
```

Entonces se puede plantear el código de multiplicación por columnas:

```
//Realiza la multiplicacion

for(i=0;i<N;i++){
     for(j=0;j<N;j++){
              R[i*N+j]=0;
     for(k=0;k<N;k++){
              R[i*N+j]= R[i*N+j] + A[k*N+i]*A[k*N+j];
              }
     }
}
```

Por lo tanto, teniendo en cuenta estas dos resoluciones se lleva a cabo las medidas de tiempo para poder elegir el más optimizado.

Nota: Se incluye en la medida de tiempo del código que utiliza multiplicación por columnas el cálculo de la matriz transpuesta.

N	Multiplicación por filas	Multiplicación por columnas
512	0.960384	0.549563
1024	19.325652	4.157781
2048	197.429430	34.574755

En conclusión, es más rápido el algoritmo que utiliza multiplicación por columnas (A transpuesta) y será la solución optimizada del ejercicio secuencial.

Comando para ejecución (ejemplificado con matriz N=512):

- Multiplicación por filas:
  ./ejercicio1secfilas 512
- Multiplicación por columnas:
  ./ejercicio1sectranspuesta 512

Resolución con un algoritmo paralelizado con Pthreads:

Para resolver el ejercicio 1 con hilos se desarrolla una función multiplicación donde se realiza el cálculo de la matriz transpuesta y luego se multiplica.
Se comienza incluyendo la librería **pthread.h** y se crean las siguientes variables:

- numthreads: cantidad de hilos a usar que se recibe por parámetro.
- elementosThread: variable que se calcula N/numthreads y es la cantidad de elementos a procesar por cada hilo.
- posInicial: se calcula idLocal*elementosThread y sirve para indicar al hilo que parte de los elementos debe recorrer. Por ejemplo, Si el idLocal tiene valor 0, el hilo debe comenzar desde 0.
- posFinal: se calcula (idLocal+1)*elementosThread y al contrario de posInicial indica donde el hilo debe terminar.

Además, en la resolución se tiene en cuenta que para realizar la multiplicación es necesario que A transpuesta ya esté calculada, entonces se utilizó una barrera para evitar que se comience la multiplicación antes de que la misma esté calculada.

Comando para ejecución (ejemplificado con matriz N=512): ./ejercicio1pthread 512 2 donde el 2 indica la cantidad de hilos.

Resolución con un algoritmo paralelizado con OpenMP:

Al igual que el algoritmo secuencial se utilizan dos sentencias for anidadas, se recorre las matrices A y A transpuesta, y se calculan los valores pedidos, pero con la diferencia que se hace uso de la sentencia *#pragma omp parallel for* y se agrega la librería **omp.h**.

Se debe tener en cuenta que cada hilo que se genere debe tener la variable k del for privada para poder realizar el cálculo de la matriz R correctamente. Entonces la multiplicación quedaría:

```
//Realiza la multiplicacion
 #pragma omp parallel for collapse(2) private(k)
for(i=0;i<N;i++){
     for(j=0;j<N;j++){
     R[i*N+j]=0;
     for(k=0;k<N;k++){
             R[i*N+j]= R[i*N+j] + A[i*N+k]*At[k+j*N];
             }
     }
}
```

Donde collapse(2) indica que la cantidad de operaciones anidadas que debe realizar el hilos son 2.

Nota: la sentencia #pragma se ubica arriba, en el primer for repartiendo el trabajo de las filas porque si se hiciera en el segundo for (con las columnas) los hilos se dormirían y se volverían a despertar desperdiciando tiempo de ejecución y haciendo al algoritmo menos eficiente.

Comando para ejecución (ejemplificado con matriz N=512): ./ejercicio1openmp 512 2 donde el 2 indica la cantidad de hilos.

Tiempos de los algoritmos:

N	Secuencial	Pthreads 2h	Pthreads 4h	OpenMP 2h	OpenMP 4h
512	0.549563	0.273813	0.168109	0.388184	0.202194
1024	4.157781	2.154374	1.140860	3.024477	1.619262
2048	34.574755	16.977779	9.099488	24.375815	12.577733

Speedup y eficiencia de los algoritmos respecto al secuencial:

N	Pthreads 2h	Pthreads 4h	OpenMP 2h	OpenMP 4h
512	S = 2.007 E = 0.501	S = 3.269 E = 0.817	S = 1.415 E = 0.353	S = 2.717 E = 0.679
1024	S = 1.929 E = 0.482	S = 3.644 E = 0.911	S = 1.374 E = 0.343	S = 2.567 E = 0.641
2048	S = 2.036 E = 0.509	S = 3.799 E = 0.949	S = 1.418 E = 0.354	S = 2.748 E = 0.687

2. Realizar un algoritmo Pthreads y otro OpenMP que resuelva la expresión:

$$M = u.l.AAC + b.LBE + b.DUF$$

Donde A, B, C, D, E y F son matrices de NxN. L y U son matrices triangulares de NxN inferior y superior, respectivamente. b es el promedio de los valores de los elementos de la matriz B y u.l es el producto de los promedios de los valores de los elementos de las matrices U y L, respectivamente.

Evaluar N=512, 1024 y 2048.

Resolución con un algoritmo secuencial:

Este ejercicio requiere llevar a cabo varias operaciones sobre matrices:

- Calcular valores promedio.
- Multiplicar una matriz por sí misma, como en el ejercicio 1.
- Multiplicar matrices.
- Multiplicar una matriz por una matriz triangular (superior o inferior).
- Sumar matrices.

En primer lugar, se asigna memoria para todas las matrices y se inicializan. Luego, en un mismo *for* anidado, se suman todos los elementos de B, L y U y se divide por el número total de elementos para obtener el promedio de cada matriz:

```
for(i=0;i<N;i++){   //Calcula los promedios
   for(j=0;j<N;j++){
       promedioB+= B[i*N+j];
       promedioL+= L[i*N+j];
       promedioU+= U[i*N+j];
   }
}
promedioB = promedioB / Total;
promedioL = promedioL / Total;
promedioU = promedioU / Total;
```

Después, se genera una matriz transpuesta de A y se realiza la multiplicación de la misma manera que en el ejercicio 1, el resultado es una nueva matriz AA. En este punto es bueno aclarar que se podrían haber realizado varias multiplicaciones de matrices dentro de un mismo *for*, pero esto causaría un aumento en los fallos de caché y por lo tanto bajaría el rendimiento del programa. Por lo tanto, cada multiplicación de matrices se lleva a cabo de manera individual.

Luego se genera la matriz AAC, multiplicando la matriz AA obtenida previamente por la matriz C y por el promedio de las matrices L y U. De esta manera obtenemos el primer término de la expresión:

```
AAC=(double*)malloc(sizeof(double)*N*N); //AAC=(AA*C)*promedio de L y U
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        AAC[i*N+j]=0;
        for(k=0;k<N;k++){
            AAC[i*N+j]= AAC[i*N+j] + AA[i*N+k]*C[k+j*N]*promedioL;
        }
    }
}
```

Los otros dos términos se obtienen de la misma manera, con la salvedad de que D*U y L*B son multiplicaciones que involucran matrices triangulares, por lo que solo es necesario recorrer los elementos no nulos de las mismas:

```
LB=(double*)malloc(sizeof(double)*N*N); //LB= L*B
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        LB[i*N+j]=0;
        for(k=0;k<=j;k++){
            LB[i*N+j]= LB[i*N+j] + B[i*N+k]*L[k+j*N];
        }
    }
}
```

```
DU=(double*)malloc(sizeof(double)*N*N); //DU= D*U
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        DU[i*N+j]=0;
        for(k=j;k<N;k++){
            DU[i*N+j]= DU[i*N+j] + D[i*N+k]*U[k+j*N];
        }
    }
}
```

Finalmente, una vez obtenidas las matrices de cada término, se realiza la suma de los mismos:

```
TOTAL=(double*)malloc(sizeof(double)*N*N); //TOTAL= AAC + LBE + DUF
for(i=0;i<N;i++){
    for(j=0;j<N;j++)
    TOTAL[i*N+j]= AAC[i*N+j] + LBE[i*N+j] + DUF[i*N+j];
}
```

Para probar este algoritmo, ejecutar ./ejercicio2secuencial N siendo N la cantidad de filas y columnas de las matrices.

Resolución con un algoritmo paralelizado con Pthreads:

En la implementación del algoritmo con hilos en Pthreads primero se creó una función llamada multiplicar que es la encargada de realizar todas las operaciones anteriormente mencionadas:

- Calcular la matriz transpuesta.
- Calcular los promedios.
- Realizar las multiplicaciones.
- Sumar el total.

Y luego, al igual que en el ejercicio 1 con pthreads, se crearon las variables para el manejo de hilos:

- numthreads: cantidad de hilos a usar que se recibe por parámetro.
- elementosThread: variable que se calcula N/numthreads y es la cantidad de elementos a procesar por cada hilo.
- posInicial: se calcula idLocal*elementosThread y sirve para indicar al hilo que parte de los elementos debe recorrer. Por ejemplo, Si el idLocal tiene valor 0, el hilo debe comenzar desde 0.
- posFinal: se calcula (idLocal+1)*elementosThread y al contrario de posInicial indica donde el hilo debe terminar.

Una vez que son creados los hilos con las funciones de la librería pthreads, se tiene en cuenta que para poder realizar los promedio correctamente la suma del total se tiene que tratar como una variable compartida, entonces se crean variables aux propias de cada hilo en las que suman su rango de elementos del promedio y una vez terminadas, son sumadas a la variable global donde se asigna un mutex para que los hilos ingresen de a uno. El código para cada hilo quedaría:

```
for(i=posInicial;i<posFinal;i++){   //Calcula los promedios
    for(j=0;j<N;j++){
            auxB+= B[i*N+j];
            auxL+= L[i*N+j];
            auxU+= U[i*N+j];
    }
}
    pthread_mutex_lock(&miMutex);
        promedioB+=auxB;
        promedioL+=auxL;
        promedioU+=auxU;
    pthread_mutex_unlock(&miMutex);
```

Además, se le asignó al hilo con id igual a 0 la operación para realizar la división del total de la suma por el total de los elementos y finalmente calcular los promedios.

```
if(idLocal==0){   //el hilo 0 calcula los promedios
    promedioB = promedioB / Total;
    promedioL = promedioL / Total;
    promedioU = promedioU / Total;
    promedioL = promedioL * promedioU; //En promedio L queda el promedio de L por el de U.
}
```

Por último, para la solución de este ejercicio se tuvo en cuenta que era necesario que los hilos cuenten con ciertos resultados antes de calcular otras operaciones, entonces se asignaron las siguientes barreras:

- barrera: Duerme a los hilos hasta que el total de las sumas para el cálculo de los promedios sean realizadas, ya que es son necesarias antes de que el hilo 0 divida por el total y calcule el promedio. Además, se utiliza esta barrera para realizar el cálculo de A transpuesta, que luego será utilizada en la matriz AA.
- barrera 1: Es la barrera utilizada para terminar los promedios antes que sean utilizados y calcular las siguientes matrices:
  - AA(A*At) antes de que sea multiplicada por matriz C y generar AAC
  - LB antes que sea multiplicada por E para generar LBE
  - DU antes de que sea utilizada para el cálculo de DUF
- barrera2: La última barrera se utiliza antes de calcular la suma de todos los elementos en la matriz TOTAL.

Para probar este algoritmo, ejecutar ./ejercicio2pthread N H siendo N la cantidad de filas y columnas de las matrices y H la cantidad de hilos.

Resolución con un algoritmo paralelizado con OpenMP:

La implementación en OpenMP es similar a la secuencial, pero en este caso todas las operaciones se realizan dentro de un bloque *#pragma omp parallel*, que le indica que al compilador que esa sección se debe paralelizar.

Dentro de este bloque, cada multiplicación de matrices se hace con la sentencia *#pragma omp for*, igual que en el ejercicio 1:

```
#pragma omp for collapse(2) private(k)
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        AA[i*N+j]=0;
        for(k=0;k<N;k++){
            AA[i*N+j]= AA[i*N+j] + A[i*N+k]*At[k+j*N];
        }
    }
}
```

En el caso del cálculo de los promedios, se hace *#pragma omp for* haciendo *reduction* de suma en cada una de las variables, para que cuando terminen los hilos se unan sus resultados. Luego, se hace la división por el número de elementos dentro de un *#pragma omp single*, para que solo un hilo la lleve a cabo:

```
#pragma omp for collapse(2) reduction(+:promedioB,promedioL,promedioU)
for(i=0;i<N;i++){   //Calcula los promedios
    for(j=0;j<N;j++){
        promedioB+= B[i*N+j];
        promedioL+= L[i*N+j];
        promedioU+= U[i*N+j];
    }
}
#pragma omp single
{
    promedioB = promedioB / Total;
    promedioL = promedioL / Total;
    promedioU = promedioU / Total;
    promedioL = promedioL * promedioU; //En promedioL queda el promedio de L por el de U
}
```

Para probar este algoritmo, ejecutar ./ejercicio2openmp N T siendo N la cantidad de filas y columnas de las matrices, y T la cantidad de hilos.

Tiempos de los algoritmos:

N	Secuencial	Pthreads 2h	Pthreads 4h	OpenMP 2h	OpenMP 4h
512	Ts = 2.771670	Tp = 1.481054	Tp = 0.785561	Tp = 1.858552	Tp = 0.966462
1024	Ts = 22.984427	Tp = 11.729690	Tp = 6.157454	Tp = 14.736200	Tp = 7.929398
2048	Ts = 178.868367	Tp = 94.509699	Tp = 51.163887	Tp = 118.638122	Tp = 64.005209

Speedup y eficiencia de los algoritmos respecto al secuencial:

N	Pthreads 2h	Pthreads 4h	OpenMP 2h	OpenMP 4h
512	S = 1.871 E = 0.467	S = 3.528 E = 0.882	S = 1.491 E = 0.372	S = 2.867 E = 0.716
1024	S = 1.959 E = 0.489	S = 3.732 E = 0.933	S = 1.559 E = 0.389	S = 2.898 E = 0.724
2048	S = 1.892 E = 0.473	S = 3.495 E = 0.873	S = 1.507 E = 0.376	S = 2.794 E = 0.698

3. Paralelizar con OpenMP un algoritmo que cuente la cantidad de número pares en un vector de N elementos. Al finalizar, el total debe quedar en una variable llamada pares.
Evaluar con valores de N donde el algoritmo paralelo represente una mejora respecto al algoritmo secuencial.

Resolución con un algoritmo secuencial:

Se genera un arreglo de N elementos y se lo inicializa con números al azar, luego se recorren todos los elementos del arreglo y si el número es par, se suma 1 a la variable pares. Cabe destacar que para averiguar si el número es par, se usa una operación *and* (&) bit a bit, que es más eficiente que la operación *modulo* (%).

```
for(int i=0;i<N;i++) if((arreglo[i] & 1) == 0) pares++;
```

Resolución con un algoritmo paralelizado con OpenMP:

Se usa el mismo algoritmo secuencial, pero paralelizando el *for* que recorre el arreglo, y haciendo *reduction* con suma en la variable pares, para que se una el resultado obtenido por cada hilo.

```
#pragma omp parallel for reduction(+ : pares)
    for(int i=0;i<N;i++) if((arreglo[i] & 1) == 0) pares++;
```

Para probar ambos algoritmos, ejecutar ./ejercicio3 N T siendo N la cantidad de elementos del arreglo y T la cantidad deseada de hilos.

Tiempos de los algoritmos:

N	Secuencial	OpenMP 2 hilos	OpenMP 4 hilos
100000	Ts =0.003299	Tp = 0.001757	Tp = 0.001334
1000000	Ts = 0.012584	Tp = 0.004942	Tp = 0.002764
10000000	Ts = 0.681812	Tp = 0.038404	Tp = 0.020525
100000000	Ts = 0.731198	Tp = 0.367181	Tp = 0.196724
1000000000	Ts = 6.822824	Tp = 3.640580	Tp = 1.932807

Speedup y eficiencia de los algoritmos respecto al secuencial:

N	OpenMP 2 hilos	OpenMP 4 hilos
100000	S = 1.877 E = 0.46925	S = 2.473 E = 0.61825
1000000	S = 2.54633 E = 0.636	S = 4.55282 E = 1.138
10000000	S = 1.8139 E = 0.4535	S = 3.3941 E = 0.848
100000000	S = 1.991 E = 0.497	S = 3.7168 E = 0.929
1000000000	S = 1.874 E = 0.468	S = 3.530 E = 0.882