Práctica 4: Paralelización con OpenMP
Multiprocesadores
Ingeniería Informática

Autor: Héctor Lacueva Sacristán 869637

Fecha: 24/04/2025

Índice

Resumen **3**

Respuestas a preguntas del guión **3**
Pregunta 1 . . . 3
Respuesta . . . 3
Pregunta 2 . . . 3
Respuesta . . . 3
Pregunta 3 . . . 3
Respuesta . . . 3

Parte 1: Resultados obtenidos apartado 3 **4**
Apartado 3.3: Directiva `parallel` . . . 4
Resultado OMP . . . 4
Resultado NoOMP . . . 4
Conclusiones . . . 4
Apartado 3.3.1: Funciones de biblioteca OpenMP . . . 4
Resultado con 2 threads . . . 4
Resultado con 4 threads . . . 4
Conclusiones . . . 4
Apartado 3.3.2: . . . 4
Resultado sin anidamiento . . . 5
Resultado con anidamiento . . . 5
Apartado 3.3.3: Reducción . . . 5
Resultado con 2 threads . . . 5
Resultado con 4 threads . . . 5
Resultado con 8 threads . . . 6
Conclusiones . . . 6
Apartado 3.4: Directiva `for` . . . 6
Resultado para 2 threads . . . 6
Resultado sin directiva `for` . . . 6
Apartado 3.5: Directiva `parallel for` . . . 7
Conclusiones . . . 7
Apartado 3.6: Directiva `sections` . . . 7
Resultado con 1 thread . . . 7
Resultado con 2 threads . . . 7
Resultado con 4 threads . . . 7
Resultado con 8 threads . . . 7
Conclusiones . . . 8
Apartado 3.7: Directiva `single` . . . 8
Resultado con la directiva . . . 8
Resultado sin la directiva . . . 8
Apartado 3.9: Directiva `barrier` . . . 8
Resultados con directiva . . . 8

Resultado sin directiva . . . 9
Apartado 3.11: Directiva `atomic` . . . 9
Apartado 3.12: Directiva `ordered` . . . 9

Apartado 4.1: Análisis de dependencias **10**
Ejecicio 2.a . . . 10
Bucle 1 . . . 10
Bucle 2 . . . 10
Resultados obtenidos . . . 10
Ejercicio 2.b . . . 10
Resultados obtenidos . . . 11
Ejercicio 2.c . . . 11
Resultados obtenidos . . . 11

Apartado 4.2: Privatización **11**
Ejercicio 3.a . . . 11
Resultados obtenidos . . . 11
Ejercicio 3.b . . . 12
Bucle externo . . . 12
Bucle interno . . . 12
Versión con paralelización . . . 12
Resultados obtenidos . . . 12

Apartado 4.3: Sustitución de variables de inducción **12**
Ejercicio 4.a . . . 12
Resultados obtenidos . . . 12

Apartado 4.4: Reducción **13**
Ejercicio 5 . . . 13
Resultados obtenidos . . . 13

Resumen

Respuestas a preguntas del guión

Pregunta 1

Apartado 3.3: Directiva `parallel`.

> Ejecuta parallel.cpp con distintos valores de OMP_NUM_THREADS. **¿Cuál es el número máximo de threads que soporta?**

Respuesta

Soporta un máximo de **191 threads**.

Pregunta 2

Apartado 3.3.2: Paralelismo anidado.

> Vuelve a ejecutar el programa y comprueba la diferencia. Modifica el programa para que varíe el número de threads que ejecutan cada una de las regiones paralelas. **¿Cuál es el número máximo de threads que llegas a ver?**

Respuesta

Con dos threads superiores y 95 threads anidados por cada thread se observan un total de **190 threads**. Si se modifica el 95 a 96 salta un error.

Pregunta 3

Apartado 3.9: Directiva `barrier`

> Compila, ejecuta con 4 threads y observa la salida del programa barrier.cpp. Repite el mismo proceso tras comentar la línea #pragma omp barrier. **¿Podrías conseguir la misma funcionalidad con la directiva single?**

Respuesta

```
/* barrier.cpp */
#include <omp.h>
#include <iostream>

const unsigned int DIM = 12;
double A[DIM], B[DIM], C[DIM], D[DIM];

int main(void){
  int l;
  int nthreads, tnumber;

  #pragma omp parallel shared (l) private(nthreads, tnumber)
  {
    nthreads = omp_get_num_threads();
    tnumber  = omp_get_thread_num();

    #pragma omp master
    {
      std::cout << "Escribe un valor:" << std::endl;
      std::cin >> l;
    }

    //#pragma omp barrier

    #pragma omp single
    {}
```

```
    #pragma omp critical
    {
      std::cout << "Mi numero de thread es: " << tnumber << std::endl;
      std::cout << "Numero de threads: " << nthreads << std::endl;
      std::cout << "Valor de L es: " << l << std::endl;
    }
  }
}
```

De esta manera se consigue un comportamiento similar al de la directiva `barrier` pero usando la directiva `single`. Todos los threads esperan a que todos hayan ejecutado la directiva `single`.

Parte 1: Resultados obtenidos apartado 3

Apartado 3.3: Directiva `parallel`

Para dos threads se obtiene lo siguiente:

Resultado OMP

```
1-0
1-1
```

Resultado NoOMP

```
1
```

Conclusiones

Al compilar sin las opciones de OpenMP, el código se ejecuta de forma secuencial como lo haría cualquier código y, por tanto, solo hay un thread. Mientras que si compilamos con las directivas se ejecuta la parte paralela en los threads especificados.

Apartado 3.3.1: Funciones de biblioteca OpenMP

Resultado con 2 threads

```
Soy el master thread (tid = 0)
Soy el thread con tid 1
```

Resultado con 4 threads

```
Soy el master thread (tid = 0)
Soy el thread con tid 3
Soy el thread con tid 1
Soy el thread con tid 2
```

Conclusiones

El thread principal pasa a ser el thread master, con tid = 0 y el resto de los threads adquieren un tid distinto de 0 de forma secuencial. Al probar con más valores para los threads también se puede ver como el texto `Soy el master thread (tid = 0)` no tiene por qué ejecutarse primero.

Apartado 3.3.2:

El código ejecutado es el siguiente

```
/* nested.cpp */
#include <omp.h>
#include <iostream>

int main(void){
  int i;
  int nthreads, tnumber;
```

```
  omp_set_dynamic(0);  // Ver el manual
  omp_set_max_active_levels(1);  // Sin anidamiento
  //omp_set_max_active_levels(5);  // Hasta 5 niveles de anidamiento
  omp_set_num_threads(2);

  #pragma omp parallel private(nthreads, tnumber)
  {
    tnumber = omp_get_thread_num();
    std::cout << "Primera region paralela: thread " + std::to_string(tnumber) + " de " + std::to_string

    omp_set_num_threads(95);

    #pragma omp parallel firstprivate(tnumber)
    {
      std::cout << "Region anidada paralela (Equipo " + std::to_string(tnumber) + "): thread " + std::t
    }
  }
  return 0;
}
```

Resultado sin anidamiento

Se ejecuta con `omp_set_max_active_levels(1);`.

```
Primera region paralela: thread 0 de 2
Primera region paralela: thread 1 de 2
Region anidada paralela (Equipo 0): thread 0 de 1
Region anidada paralela (Equipo 1): thread 0 de 1
```

Resultado con anidamiento

Se ejecuta con `omp_set_max_active_levels(5);`.

```
Primera region paralela: thread 0 de 2
Primera region paralela: thread 1 de 2
Region anidada paralela (Equipo 0): thread 0 de 95
Region anidada paralela (Equipo 0): thread 1 de 95
Region anidada paralela (Equipo 0): thread 4 de 95
Region anidada paralela (Equipo 0): thread 2 de 95
Region anidada paralela (Equipo 1): thread 0 de 95
Region anidada paralela (Equipo 0): thread 91 de 95
Region anidada paralela (Equipo 0): thread 3 de 95
Region anidada paralela (Equipo 1): thread 6 de 95
...
```

En este caso se ha recortado la salida, pero por cada thread master se ejecuta el siguiente código 95 veces.

Apartado 3.3.3: Reducción

Resultado con 2 threads

```
Thread 0 I = 0 J = 0 K = 0
Thread 1 I = 1 J = 1 K = 1
Thread 0 I = 1 J = 0 K = 1
```

Resultado con 4 threads

```
Thread 0 I = 0 J = 0 K = 0
Thread 1 I = 1 J = 1 K = 1
Thread 3 I = 3 J = 3 K = 3
Thread 2 I = 2 J = 2 K = 2
Thread 0 I = 6 J = 0 K = 3
```

Resultado con 8 threads

```
Thread 0 I = 0 J = 0 K = 0
Thread 3 I = 3 J = 3 K = 3
Thread 1 I = 1 J = 1 K = 1
Thread 4 I = 4 J = 4 K = 4
Thread 2 I = 2 J = 2 K = 2
Thread 7 I = 7 J = 7 K = 7
Thread 5 I = 5 J = 5 K = 5
Thread 6 I = 6 J = 6 K = 6
Thread 0 I = 28 J = 0 K = 7
```

Conclusiones

Como se puede observar en los resultados, la variable `i` al final de la ejecución tiene como resultado la suma de todas las `i` de los distintos threads paralelos, que coincide con el propósito de la `reduction(+:i)`. En el caso de la `j` contiene el valor 0, ya que es la reducción de multiplicación y para el thread master (tid = 0), `j` vale 0 y, como resultado, la reducción siempre será 0, coincide con el propósito de `reduction(*:j)`. En el último caso, la reducción de k `k` obtiene el valor más grande de la variable `k` de los ditintos threads y cumple con la directiva `reduction (max:k)`.

Apartado 3.4: Directiva `for`

Resultado para 2 threads

Con `DIM = 10` y `schedule (static, 4)` la salida es la siguiente.

```
Thread-0 de 2 tiene N=2
Thread-0 de 2 tiene N=3
Thread-0 de 2 tiene N=4
Thread-0 de 2 tiene N=5
Thread-1 de 2 tiene N=6
Thread-1 de 2 tiene N=7
Thread-1 de 2 tiene N=8
Thread-1 de 2 tiene N=9
```

A cada thread se le reparten las iteraciones del bucle de 4 en 4, el thread 0 a recibido del 2 al 5 y el thread 1 del 6 al 9. Si ejecutamos el mismo código con `schedule (static, 2)` el resultado es el siguiente:

```
Thread-0 de 2 tiene N=2
Thread-0 de 2 tiene N=3
Thread-0 de 2 tiene N=6
Thread-0 de 2 tiene N=7
Thread-1 de 2 tiene N=4
Thread-1 de 2 tiene N=5
Thread-1 de 2 tiene N=8
Thread-1 de 2 tiene N=9
```

Ahora se reparten las iteraciones de dos en dos, de tal forma que al thread 0 se le han asignado las iteraciones 2-3 y 6-7, mientras que al thread 1 se le han asignado las iteraciones 4-5 y 8-9.

Resultado sin directiva `for`

```
Thread-0 de 2 tiene N=2
Thread-1 de 2 tiene N=2
Thread-1 de 2 tiene N=3
Thread-1 de 2 tiene N=4
Thread-0 de 2 tiene N=3
Thread-0 de 2 tiene N=4
Thread-1 de 2 tiene N=5
Thread-1 de 2 tiene N=6
Thread-0 de 2 tiene N=5
Thread-0 de 2 tiene N=6
Thread-1 de 2 tiene N=7
Thread-1 de 2 tiene N=8
Thread-1 de 2 tiene N=9
```

```
Thread-0 de 2 tiene N=7
Thread-0 de 2 tiene N=8
Thread-0 de 2 tiene N=9
```

En este caso, las iteraciones no se reparten entre los threads, sino que cada thread ejecuta el bucle de forma independiente.

Apartado 3.5: Directiva `parallel for`

La siguiente tabla muestra el tiempo de ejecución y de CPU para el mismo un mismo problema abordado con diferente número de threads. A su vez también muestra el speedup conseguido al paralelizar.

NºThreads	Time (s)	CPU time (s)	Speedup
1	43.5831	43.5822	1
2	22.3771	44.5197	1.947
4	11.7741	45.8923	3.702
8	6.94633	52.3122	6.272
16	5.03977	76.1535	8.650
32	4.40746	140.441	9.891

Conclusiones

Conforme aumenta el número de hilos, más disminuye el tiempo de ejecución con respecto a la versión secuencial y, como consecuencia, el speedup aumenta. Dado esto, podríamos pensar que cuantos más threads se usen, mejor será el resultado y nada más lejos de la realidad. Conforme aumenta el número de threads, el speedup se aleja cada vez más de los valores ideales. P.e. para 4 threads el valor del speedup ideal sería un 4, pero solo alcanza hasta el 3.7, conforme se aumenta el número de threads esta diferencia aumenta lo que indica que hay un límite a partir del cual no merece la pena usar tantos threads, p.e. para 32 el valor 9.891 está muy alejado del 32 ideal. Además, cuantos más threads se usen, mayor será la sobrecarga que reciba el procesador por organizar tanta cantidad de threads y esto provocará un mayor coste energético.

Por todo esto es conveniente realizar pruebas del código para determinar la cantidad óptima de threads que se debrían usar para su ejecución.

Apartado 3.6: Directiva `sections`

Resultado con 1 thread

```
Esta es la sección 1 ejecutada por el thread 0
Esta es la sección 2 ejecutada por el thread 0
Esta es la sección 3 ejecutada por el thread 0
Esta es la sección 4 ejecutada por el thread 0
```

Resultado con 2 threads

```
Esta es la sección 1 ejecutada por el thread 0
Esta es la sección 2 ejecutada por el thread 1
Esta es la sección 4 ejecutada por el thread 1
Esta es la sección 3 ejecutada por el thread 0
```

Resultado con 4 threads

```
Esta es la sección 4 ejecutada por el thread 3
Esta es la sección 1 ejecutada por el thread 0
Esta es la sección 2 ejecutada por el thread 2
Esta es la sección 3 ejecutada por el thread 1
```

Resultado con 8 threads

```
Esta es la sección 4 ejecutada por el thread 2
Esta es la sección 1 ejecutada por el thread 0
Esta es la sección 3 ejecutada por el thread 3
Esta es la sección 2 ejecutada por el thread 5
```

Conclusiones

Cada `section` dentro de una directiva `parallel` es ejecutada por un solo thread. Cuantos más threads haya, más se repartirán los distintos trozos de código.

Apartado 3.7: Directiva `single`

Resultado con la directiva

```
Resultado de la suma es: 36000000.000000
D[0]=2.000000
D[1]=2.000000
D[2]=2.000000
D[3]=2.000000
D[4]=2.000000
D[5]=2.000000
D[6]=2.000000
D[7]=2.000000
D[8]=2.000000
D[9]=2.000000
D[10]=2.000000
D[11]=2.000000
```

Como resultado, Solo un thread ejecuta ese trozo de código.

Resultado sin la directiva

```
Resultado de la suma es: 36000000.000000
Resultado de la suma es: 36000000.000000
Resultado de la suma es: 36000000.000000
Resultado de la suma es: 36000000.000000
D[0]=2.000000
D[1]=2.000000
D[2]=2.000000
D[3]=2.000000
D[4]=2.000000
D[5]=2.000000
D[6]=2.000000
D[7]=2.000000
D[8]=2.000000
D[9]=2.000000
D[10]=2.000000
D[11]=2.000000
```

En este caso, todos los threads ejecutan esa sección de código.

Apartado 3.9: Directiva `barrier`

Resultados con directiva

Al simplementa ejecutar el comando, en la terminal aparece lo siguiente:

```
Escribe un valor:
```

Al escribir un valor, el resultado es el siguiente:

```
Escribe un valor:
34
Mi numero de thread es: 0
Numero de threads: 4
Valor de L es: 34
Mi numero de thread es: 2
Numero de threads: 4
Valor de L es: 34
Mi numero de thread es: 1
Numero de threads: 4
```

```
Valor de L es: 34
Mi numero de thread es: 3
Numero de threads: 4
Valor de L es: 34
```

Resultado sin directiva

```
Escribe un valor:
Mi numero de thread es: 2
Numero de threads: 4
Valor de L es: 0
Mi numero de thread es: 3
Numero de threads: 4
Valor de L es: 0
Mi numero de thread es: 1
Numero de threads: 4
Valor de L es: 0
34
Mi numero de thread es: 0
Numero de threads: 4
Valor de L es: 34
```

Apartado 3.11: Directiva `atomic`

Resultado con la directiva

```
0 - 0
10989 - 10989
11148 - 11148
10849 - 10849
11098 - 11098
11254 - 11254
11099 - 11099
11102 - 11102
11086 - 11086
11375 - 11375
```

Como se puede apreciar los resultados son correctos.

Resultado sin la directiva

```
0 - 0
3206 - 11086 ERROR
3202 - 11218 ERROR
3151 - 11131 ERROR
3239 - 11174 ERROR
3105 - 11205 ERROR
3186 - 11142 ERROR
3083 - 10800 ERROR
3168 - 11177 ERROR
3133 - 11067 ERROR
```

El resultado no es el correcto.

Apartado 3.12: Directiva `ordered`

Resultado con la directiva

```
Z[0]=1011.236604
Z[1]=986.927376
Z[2]=993.845231
Z[3]=981.143302
Z[4]=1036.442865
Z[5]=999.492768
Z[6]=998.144687
Z[7]=998.793459
Z[8]=1009.409924
Z[9]=1002.866337
Z[10]=1004.759651
Z[11]=989.447304
Z[12]=980.561906
Z[13]=1019.238059
Z[14]=984.462465
Z[15]=1005.896750
Z[16]=1017.862811
Z[17]=1001.402825
Z[18]=989.755025
Z[19]=1000.366513
Z[20]=1017.577449
```

El resultado sale ordenado como era de esperar.

Resultado sin la directiva

```
Z[6]=993.099406
Z[4]=1013.518945
Z[7]=993.860497
Z[0]=998.057209
Z[2]=1021.035953
Z[5]=983.482538
Z[1]=1009.953161
Z[3]=976.946998
Z[8]=1005.169208
Z[10]=991.218632
Z[14]=1001.381247
Z[12]=1001.702103
Z[9]=1001.904190
Z[11]=1007.345792
Z[15]=992.574724
Z[13]=1005.955084
Z[16]=1010.415354
Z[18]=979.415930
Z[20]=1009.632452
Z[17]=968.799767
Z[19]=1007.506892
```

El resultado sale desordenado.

Apartado 4.1: Análisis de dependencias

Todos los ejercicios 2.x han sido compilados tanto con el script compomp.sh como compnoomp.sh y ejecutado en un total de 191 threads. Realmente no es necesario ejecutarlo en los 191 threads, con 50 o menos se podría conseguir resultados parecidos sin tanta sobrecarga.

Ejecicio 2.a

Haciendo la suposición de que el bucle de incialización es solo el primero, tenemos dos bucles por analizar.

Bucle 1

```
for (int64_t i = 0; i < N; ++i) {
    A[i] = A[i] + B[i]/2.0;
}
```

El análisis de dependencias da como resultado que hay una antidependencia a distancia 0 que se puede despreciar. El bucle **es vectorizable y paralelizable**.

Se le añade el `#pragma omp parallel for`.

Bucle 2

```
for (int64_t i=0 ; i < (N/2)-1; ++i){
    A[2*i] = B[i];
    C[2*i] = A[2*i];
    C[2*i + 1] = A[2*i + 1];
  }
```

En este caso, hay una dependencia verdadera entre la s1 y la s2 a distancia 0. Pese a ello, el bucle también **es vectorizable y paralelizable**.

Se le añade el `#pragma omp parallel for`.

Resultados obtenidos

Bucle	Con #pragmas	Sin #pragmas
Bucle 1	0.342048s	0.791793s
Bucle 2	0.316846s	1.03377s

Como se puede ver en la tabla la mejora es significativa, aproximadamente se consigue un **speedup de 2,31 en el primer bucle y un speedup de 3,26 en el segundo bucle** con respecto a la versión secuencial.

Ejercicio 2.b

```
for (int i = 1; i < N; ++i) {
    B[i] = A[i] - A[i-1];
}
```

No hay dependencias que puedan causar riesgos de datos, solo se lee del vector A y se escribe en el vector B. Por lo tanto, es vectorizable y paralelizable.

Se añade la directiva `#pragma omp parallel for`.

Resultados obtenidos

Con #pragmas	Sin #pragmas
1.6823s	8.58428s

El **speedup conseguido en este caso es de aproximadamente 5,10** con respecto a la versión secuencial.

Ejercicio 2.c

```
for (int64_t i = 1; i < N; ++i) {
    A[i] = A[i] - A[i - 1];
}
```

En este código encontramos una dependencia verdadera a distancia 1 y una antidependencia a distancia 0. La antidependencia no afecta pero la dependencia verdadera impide la vectorización del bucle y, por tanto, también su paralelización.

Por lo tanto, el bucle **NO ES VECTORIZABLE NI PARALELIZABLE**.

Resultados obtenidos

El tiempo de ejecución de este bucle secuencialmente es de 8,65353 segundos.

Apartado 4.2: Privatización

Ejercicio 3.a

```
for (int64_t i = 1; i < N; ++i) {
    t = A[i];
    B[i] = t + pow(t, 2);
    C[i] = t + 2.0;
  }
```

Este bucle, tiene dependencias entre todas las iteraciones por la variable `t`. Pese a ello, se puede aplicar privatización sobre la dicha variable, **consiguiendo que sea vectorizable y paralelizable.**

```
#pragma omp parallel for private(t)
  for (int64_t i = 1; i < N; ++i) {
    t = A[i];
    B[i] = t + pow(t, 2);
    C[i] = t + 2.0;
  }
```

Resultados obtenidos

Resultados obtenidos con 50 threads.

Con #pragmas	Sin #pragmas
0.0335464s	0.13329s

Con la paralelización del bucle **se consigue un speedup de 3,97 con respecto a la versión secuencial**.

Ejercicio 3.b

```
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j){
      t[j] = A[i][j] + B[i][j];
      C[i][j] = t[j] + C[i][j];
    }
  }
```

Bucle externo

No es paralelizable en un principio ya que variso threads podrían querer escribir en la misma posición de memoria a la vez. Se podría intentar aplicar privatización pero no daría resultado, ya que t ahora es un vector y la directiva private no sería capaz de crear una copia local para cada thread.

Se podría hacer esto manualmente, y en ese caso, el bucle si que sería paralelizable.

Bucle interno

En el encontramos una dependencia verdadera a distancia 0 y, por tanto, es vectorizable y paralelizable.

Versión con paralelización

```
for (int i = 0; i < N; ++i) {
    #pragma omp parallel for
    for (int j = 0; j < N; ++j){
      t[j] = A[i][j] + B[i][j];
      C[i][j] = t[j] + C[i][j];
    }
}
```

Resultados obtenidos

Las pruebas se han realizado con 5 threads ya que se ha comprobado un buen funcionamiento.

Con #pragmas	Sin #pragmas
2.31384s	8.99125s

Como resultado se ha obtenido **un speedup de 3,88 con respecto a la versión secuencial**.

Apartado 4.3: Sustitución de variables de inducción

Ejercicio 4.a

```
for (i = 0; i < ((N/2)-1); ++i) {
    j = j + 2;
    B[j] = A[i];
}
```

Teóricamente no debería de ser vectorizable ni paralelizable, hay una dependencia entre iteraciones. Pero supongamos que el compilador es capaz de inducir el valor de j. En ese caso, ya no habría dependencia y, por tanto, el código sería vectorizable y paralelizable. Pensando en ello, se le ha añadido la directiva `#pragma omp parallel for private (j)`.

Resultados obtenidos

Los resultados obtenidos muestran que el compilador no es capaz de realizar la inducción, por lo menos, con las opciones de compilación usadas (fichero compomp.sh).

Resultado con la directiva

```
B[152]=68.0459
A[150]=29.4359
```

Resultado sin la directiva

```
B[152]=65.2299
A[150]=29.4359
```

Apartado 4.4: Reducción

Ejercicio 5

```
for (i = 0; i < N; ++i) {
    A[i] = A[i] + B[i];
    q = q + A[i];
}
```

De primeras no es vectorizable ni paralelizable por la dependencia entre iteraciones de q. Pero aplicando las directivas de reducción es posible paralelizar. Para ello, se añade `#pragma omp parallel for reduction (+:q)`. Crea una copia privada en cada thread de q, (tiene que estar inicializado a 0) y al finalizar suma todos los valores en q.

Resultados obtenidos

Resultado con la directiva

```
B[150]=7.31233
A[150]=93.8213
q=3.99957e+09
```

Resultado sin la directiva

```
B[150]=7.31233
A[150]=93.8213
q=3.99957e+09
```

El resultado obtenido es el esperado.