Lab 2: Brief tutorial on OpenMP programming model

Pablo Cebollada Hernández
Iván Díaz Ortega
Par1104
17/10/19

A:

1.hello.c

Q1: How many times will you see the "Hello world!" message if the program is executed with "./1.hello"?

Aparecerá tantas veces como threads haya.

Q2: Without changing the program, how to make it to print 4 times the "Hello world!" message?

Utilizando la opción OMP_NUM_THREADS=4 al ejecutar el programa.

2.hello.c

Q1: Is the execution of the program correct? Add a data sharing clause to make it correct.

Es incorrecta, ya que se repite el número de thread cuando no debería. Para evitarlo hay que poner la cláusula private(id)

Q2: Are the lines always printed in the same order? Why the messages sometimes appear intermixed?

Los mensajes aparecen mezclados por que los threads se ejecutan de forma concurrente y no hay manera de saber el orden de ejecución de estos.

3.how_many.c

If the OMP_NUM_THREADS variable is set to 8 with export OMP_NUM_THREADS=8

Q1: How many "Hello world ..." lines are printed on the screen?

Se pintan 36 "Hello world…"s.

Q2: What does omp_get_num_threads return when invoked outside and inside a parallel region?

Fuera de la región paralela retornará 1, y dentro de ella retornará el número máximo de threads que se pueden crear.

4.data_sharing.c

Q1: Which is the value of x after the execution of each parallel region with different data-sharing attribute (shared, private, firstprivate and reduction)? Explain why, repeating the execution many times if necessary.

Los valores son:

Indefinido para el primero, ya que al declarar la variable como shared ésta se comparte entre todos los threads y se puede producir un data race.

Para el segundo y el tercero es 5, ya que en ambas se crean variables privadas dentro de la región paralela y, por lo tanto, no se modifica el valor de la variable original. En el último caso, al hacer un reduction de la variable en cuestión hay una sincronización para el valor de la variable por lo que la suma de todo se hará correctamente y el valor es 125.

B:

1.shedule.c

Q1. Which iterations of the loops are executed by each thread for each schedule kind? En la primera región se reparten equitativamente las iteraciones entre todos los threads, por lo que el primer thread ejecutará las tres primeras iteraciones, el segundo las tres siguientes, y así. La segunda región hace lo mismo que la primera pero utilizando trozos de dos iteraciones, por lo que en vez de repartir tres repartirá dos iteraciones de la misma manera que en la primera región. En las dos siguientes regiones no se puede saber qué thread ejecuta qué iteración, ya que estas se reparten según van habiendo threads libres.

2.nowait.c

Q1. Which could be a possible sequence of printf when executing the program?

0, 1, 2, 3

Q2. How does the sequence of printf change if the nowait clause is removed from the first for directive? Si se quita la cláusula nowait el programa se esperará a que el primer bucle se ejecutará y el segundo no lo hará hasta que el primero haya acabado.

Q3. What would happen if dynamic is changed to static in the schedule in both loops? (keeping the nowait clause).

Si se utiliza el static en vez de dynamic sabemos seguro que la primera iteración de cada bucle se asignará al thread 0, la segunda al segundo y así.

3.collapse.c

Q1: Which iterations of the loops are executed by each thread when the collapse clause is used?

Cada thread ejecutará tres iteraciones que se irán asignando en orden, aunque como al asignar de tres en tres queda una iteración suelta, esta se asigna a un thread, que es el primero.

Q2: Is the execution correct if we remove the collapse clause?Add the appropriate clause to make it correct.

Si se quita la clausula collapse el problema está en que las variables i y j se comparten entre todos los threads y provoca data race, por lo que hay iteraciones que no se ejecutan. Para que funcione correctamente hay que añadir private(i,j) para que las variables i y j sean privadas para cada thread y no se solapen.

C:

1.datarace.c

Q1: Is the program always executing correctly? Why?

No, ya que como se comparte la variable x,los threads van escribiendo en ella según van llegando, lo que provoca data race y un resultado erróneo.

Q2: Add two alternative directives to make it correct. Explain why they make the execution correct.

La primera alternativa es añadir la cláusula reduction(+:x), lo que haría que se fueran sumando en una variable privada por thread y luego se sincronizan al final. La segunda opción es hacer la operación ++x atomic, añadiendo #pragma omp atomic, haciendo que no se pueda actualizar el valor de la variable mientras se está calculando su nuevo valor.

2.barrier.c

Q1: Can you predict the sequence of printf in this program? Do threads exit from the barrier in any specific order?

Sí que se puede predecir el orden de printf, ya que al ponerse a dormir todos los threads sabemos que los primeros se imprimirán antes que los segundos, aunque los primeros no sabemos en qué orden se ejecutarán entre ellos.

3.ordered.c

Q1: Can you explain the order in which printf appear?

Los printf de fuera del ordered no se puede saber con exactitud cómo se irán ejecutando, ya que según acaben los threads que esten asignados se irán imprimiendo por pantalla, mientras que los printf de dentro del ordered si que se imprimirán en orden, ya que eso lo asegura la cláusula ordered.

Q2: How can you ensure that a thread always executes two consecutive iterations in order during the execution of the ordered part of the loop body?

Lo podemos asegurar añadiendo un schedule(dynamic,2) a la cláusula inicial para obligar al programa a asignar dos iteraciones del bucle al mismo thread.

D:

1.single.c

Q1: Can you explain why all threads contribute to the execution of instances of the single worksharing construct? Why are those instances appear to be executed in bursts?

Los 4 threads participan en la ejecución ya que al estar el single dentro del bucle, te aseguras que cada iteración sea ejecutada solo una vez. Aparece en rafagas por la acción del sleep, al ser un no wait la duración de este sleep es igual para todos los threads por la cual cosa se dormirán y se despiertan al unísono.

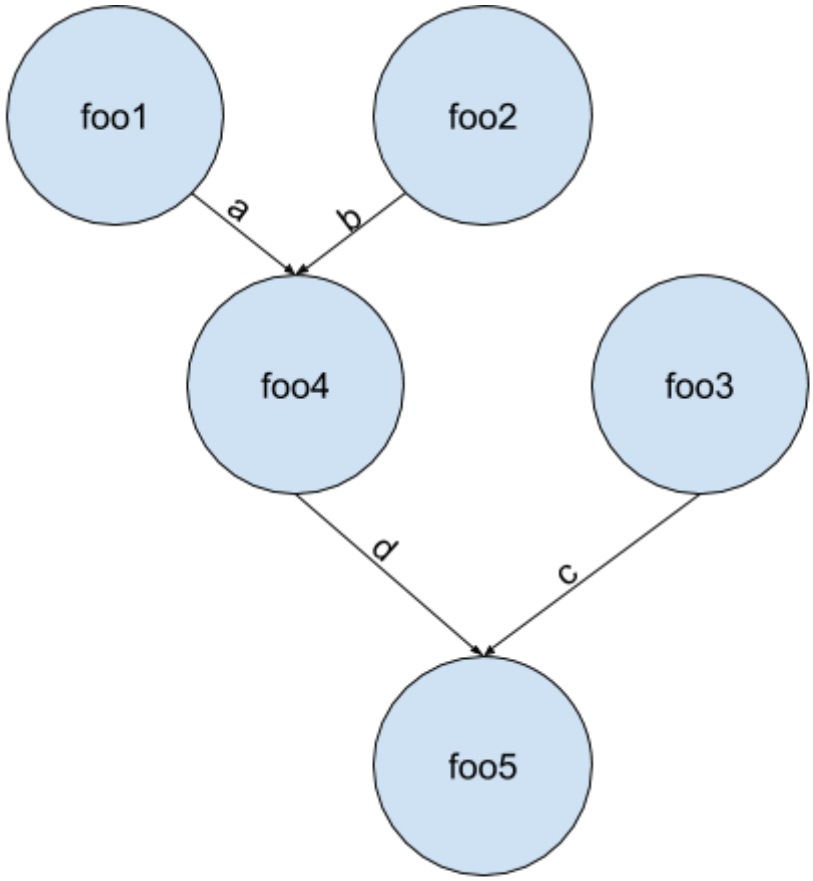
2.fibtasks.c

Q1: Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel?

No se ejecutan en paralelo por que en ningún momento se crea una zona paralelizable, por lo que no se crea ninguna tarea para ejecutarse en paralelo.

3.synchtasks.c

Q1: Draw the task dependence graph that is specified in this program.



Grafo de Dependecias

Q2.Rewrite the program using only taskwait as task synchronisation mechanism (no depend clauses allowed).

Para que siga habiendo el mismo grado de dependencias hay que utilizar el taskwait después de crear la tarea foo3 para que no se cree la tarea foo4 antes de que se ejecute la foo1, foo2 y foo3, y otro taskwait antes de crear foo5 para asegurar de que se ejecute foo4 antes que foo5.

```
#pragma omp parallel
#pragma omp single
{
        printf("Creating task foo1\n");
        #pragma omp task
        foo1();
        printf("Creating task foo2\n");
        #pragma omp task
        foo2();
        printf("Creating task foo3\n");
        #pragma omp task
        foo3();
        printf("Creating task foo4\n");
        #pragma omp taskwait
        #pragma omp task
        foo4();
        printf("Creating task foo5\n");
        #pragma omp taskwait
        #pragma omp task
        foo5();
}
```

4.taskloop.c

Q1:Find out how many tasks and how many iterations each task execute when using the grainsize and num_tasks clause in a taskloop. You will probably have to execute the program several times in order to have a clear answer to this question.

Con grainsize se crean tareas con 5 iteraciones cada una. Como en este caso tenemos sólo 12 iteraciones se crean 2 tareas con 5 iteraciones cada una y las dos que nos sobran se reparten entre estas dos tareas, asignando 6 iteraciones a cada tarea.

Con num_tasks es justo lo contrario, se crean 5 tareas y luego se reparten las iteraciones entre estas tareas. Como tenemos 12 iteraciones tendremos dos tareas con 3 iteraciones y tres con 2 iteraciones.

Q2: What does occur if the no group clause in the first taskloop is uncommented?

El thread que crea las tareas no se espera al final de ningun taskloop, por lo que ejecuta el código hasta el final y por eso no se puede ver un orden en la ejecución de los bucles.

Observing overheads

Si nos fijamos en los datos obtenidos por la ejecución de los diferentes programas podemos llegar a diferentes conclusiones. Primero, si nos fijamos en los tiempos de ejecución de las tablas siguientes podemos ver el impacto que tienen los tiempos de sincronización y de creación de hilos. Al comparar estos tiempos con el tiempo que tarda el programa secuencial vemos como, claramente, el programa que sale peor parado es el que utiliza critical para evitar el data race. Esto se debe a que el utilizar la cláusula critical conlleva un gasto de tiempo que se evita al utilizar atomic, ya que lo que hace critical por software lo hace atomic por hardware. Por esta misma razón es la opción que menos overhead tiene. Al comparar los otros dos programas podemos ver que la diferencia de tiempos es despreciable, aunque si quisiéramos hilar más fino, podríamos decir que la opción que utiliza una variable local para luego sumarlas todas utilizando un critical (pi_sumlocal) conlleva un mayor overhead a causa de este critical, ya que si se utiliza reduction el impacto, supuestamente, ha de ser menor.

	Tiempo de ejecución (s)
pi_sequential	1,79
pi_critical	4,41
pi_atomic	1,798
pi_reduction	1,83
pi_sumlocal	1,83

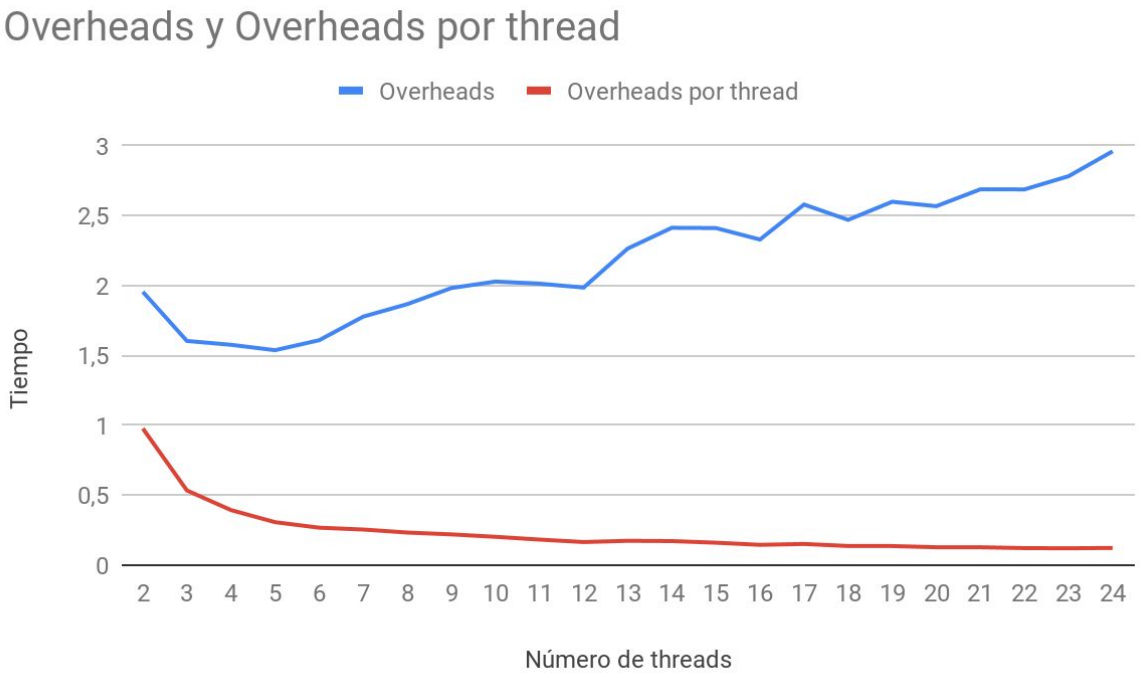
Tabla de tiempos de ejecución con 1 hilo y 100.000.000 iteraciones

Fijándonos ahora en esta otra tabla podemos ver lo que hemos dicho anteriormente, pero también podemos apreciar si el coste de crear estos hilos tiene, realmente, algún impacto en la supuesta mejora de tiempo que conlleva paralelizar el programa secuencial. Como se puede apreciar el caso más extremo de esta tabla es el mismo que en el de la tabla anterior, el programa que utiliza critical para evitar data race. Sin embargo, el caso que en la anterior tabla nos daba un mejor resultado en esta nos da un resultado bastante malo, ya que tarda más que la versión secuencial. Esto se debe a que obliga a todos los threads a hacer la actualización de la variable donde se mantiene el valor de la suma total de uno en uno, creando un tiempo de espera para los threads que no existe en las otras opciones.
Estas otras opciones si que nos ofrecen un resultado mejor y, de igual manera que en la tabla anterior, muy similares. Esto se debe, como ya he comentado antes a que no se crea una espera en las iteraciones, ya que al crearse una variable local en la que se va actualizando el valor de la suma de cada iteración no se crea ningún data race y, por lo tanto, ninguna espera. La similaridad entre los valores de ambos programas se debe, de igual manera que en la tabla anterior, a que ambas hacen lo mismo, aunque, igual que antes, seguimos teniendo que pi_sumlocal usa critical que puede crear algo más de overhead.

	Tiempo de ejecución 4 hilos (s)	Tiempo de ejecución 8 hilos (s)
pi_critical	36,87	33,73
pi_atomic	6,46	6,35
pi_reduction	0,47	0,26
pi_sumlocal	0,48	0,26

Tabla de tiempos de ejecución con 4 y 8 hilos y 100.000.000 iteraciones

En esta parte, la cual ejecutamos el programa pi_omp_parallel con diferentes números de threads, se puede observar como el tiempo de ejecución empieza con casi dos segundos pero a las siguientes ejecuciones cuando aumentamos dichos threads, este tiempo disminuye hasta cierto punto, de ahí hacia delante cuantos más threads se van añadiendo más tiempo necesita la ejecución, la causa de dicha subida de tiempo se debe a la creación, sincronización y destrucción de los diferentes threads empleados en la ejecución. Por otra parte tenemos la gráfica de overheads por cada threads que se crea, en esta se puede apreciar como el tiempo va disminuyendo (al principio más bruscamente) hasta que se estabiliza por el final. Esto se puede interpretar como que existe una penalización por el uso del paralelismo que provoca ese overhead tan alto al principio y a medida que van apareciendo más, este tiempo se distribuye entre los threads creados.



En comparación con la gráfica de arriba está no tiene overhead inicial, suponemos que esto es debido a que no se tiene que realizar un trabajo inicial y todo el overhead que observamos es originado por las tareas que se ejecutan en el único thread del que disponemos. Además podemos ver que el aumento del overhead es lineal, lo que significa que al aumentar el número de tareas no se aumenta el overhead por tarea, es decir, se mantiene constante.