Paralelismo

Lab 2: Brief tutorial on OpenMP programming model

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Jia Long Ji Qiu: par2212
Jiabo Wang: par2220
Primavera 2021-22
24 de marzo de 2022

Índice

Session 1: A very practical introduction to OpenMP (I)

OpenMP questionnaire - Day 1: Parallel regions and implicit tasks

1.hello.c

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

2 veces, ya que la instrucción “pragma omp parallel” crea por defecto 2 threads que se ejecutan en paralelo si no se define la variable de entorno “OMP_NUM_THREADS”.

2. Without changing the program, how to make it to print 4 times the "Hello World!" message?

Utilizando la variable de entorno “OMP_NUM_THREADS=4”

2.hello.c:

1. Is the execution of the program correct? (i.e., prints a sequence of "(Thid) Hello (Thid) world!" being Thid the thread identifier). If not, add a data sharing clause to make it correct.

El resultado esperado de la ejecución de este programa sería que cada thread mostrara por pantalla un “Hello” y un “world!” con su respectivo identificador. Sin embargo, tal y como está escrito el programa, la variable identificador (id) se inicializa antes de crear los threads, por lo que es una variable compartida por todos.

La modificación implementada ha sido poner la variable identificador (id) como privado.

```
#pragma omp parallel num_threads(8) private(id)
{
    id =omp_get_thread_num();
    printf("(%d) Hello ",id);
    printf("(%d) world!\n",id);
}
```

Figura 1. Fragmento de código de 2.hello.c

2. Are the lines always printed in the same order? Why the messages sometimes appear intermixed? (Execute several times in order to see this).

Las líneas que se muestran en la salida no siempre están en el mismo orden. Lo único que se garantiza, una vez arreglada la compartición de datos, es que cada thread haga únicamente un print para “Hello” y otro para “world!”. Por este motivo, a veces los mensajes pueden aparecer cruzados debido a la ejecución en paralelo de los threads.

3.how_many.c:

Assuming the OMP NUM THREADS variable is set to 8 with "OMP NUM THREADS=8 ./3.how many"

1. What does omp get num threads return when invoked outside and inside a parallel region?

La instrucción “omp_get_num_threads()” devuelve el número de threads **en uso** en el momento en el que se invoca. Por lo tanto, en el programa principal, antes de entrar a cualquier región paralela, el número de threads siempre es 1. Dentro de cada región paralela, este depende del número que esté asignado justo antes de entrar:

Primera región: 8 threads. Debido a que la ejecución del programa se ha ejecutado asignando la variable de entorno “OMP_NUM_THREADS=8”, en esta primera región paralela hay 8 threads en uso.

Segunda región: 4 threads. En la segunda región paralela, la ejecución paralela se ha iniciado con la directiva “#pragma omp parallel num_threads(4)”, por lo que el número de threads durante esta región es 4.

Tercera región: 8 threads. A pesar de que en la región anterior se especificaron 4 threads, al haberse indicado como opción en la directiva “#pragma omp parallel”, tras terminar la ejecución dicha región, el número de threads se restaura (vuelve a ser el mismo que había sido especificado antes de la ejecución). Por lo tanto, en esta tercera, el número de threads está determinado por la variable entorno definida desde el principio.

Cuarta región: 2 y 3 threads. La cuarta región paralela está definida dentro de un bucle for el cual consiste en dos iteraciones, donde el valor i = {1, 2}. Justo antes de ejecutarse, se establece el número de threads con la instrucción “omp_set_num_threads(número de threads)” según este valor i, y es por eso que primero indica que hay 2 threads y luego 3.

Quinta región: 4 threads. La quinta región se trata de un caso prácticamente igual al de la segunda.

Sexta región 3 threads. En esta última región el número de threads ha sido determinado por la instrucción “omp_set_num_threads(número de threads)”, la cual fue invocada en la cuarta región. A diferencia de la segunda y quinta región, donde el número de threads fue establecido como opción de la directiva “pragma omp parallel”, esta instrucción establece el número de threads de manera genérica, tal y como lo hace la variable de entorno “OMP_NUM_THREADS”. Concretamente, los 3 threads vienen de la última iteración de la cuarta región, donde se especificó el número de threads con un valor de i = 3.

Finalmente, después de todas las ejecuciones de regiones paralelas, la instrucción vuelve a devolver 1.

2. Indicate the two alternatives to supersede the number of threads that is specified by the OMP NUM THREADS environment variable.

Además de la variable de entorno “OMP_NUM_THREADS”, también se puede especificar el número de threads mediante la instrucción “omp_set_num_threads(número de threads)” o bien como opción en una directiva utilizando “num_threads(número de threads)”.

3. Which is the lifespan for each way of defining the number of threads to be used?

Definir el número de threads a utilizar con la variable de entorno “OMP_NUM_THREADS” y la instrucción “omp_set_num_threads(número de threads)” mantiene el número de threads hasta que se indique otro número con esta última instrucción.

Si se especifica como opción de la directiva “#pragma omp parallel num_threads(número de threads)”, este solo se mantiene dentro de la región paralela en la que se ha definido.

4.data_sharing.c

1. Which is the value of variable x after the execution of each parallel region with different data-sharing attributes (shared, private, firstprivate and reduction)? Is that the value you would expect? (Execute several times if necessary)

Para la opción shared: la mayoría de las veces el valor de la x sale 120, pero no podemos asegurar que siempre sea 120 por la posibilidad de que haya data race.

Para la opción private: la variable x toma el valor de 5 (es determinista), ya que como es privado se pierde al salir de la cláusula.

Para la opción firstprivate: en este caso la variable x también toma el valor de 5 (también determinista), porque la única diferencia que hay respecto al anterior es que ahora cada thread inicializa en local la variable x a 5, pero los cambios se siguen perdiendo una vez se acaba cada ejecución local.

Para la opción reduction: crea variables locales y al final se hace la reducción, es decir acumularlo a la variable global. Y el variable x da 125 y es determinista.

5.datarace.c

1. Should this program always return a correct result? Reason either your positive or negative answer.

Este programa no siempre devuelve un resultado correcto debido al “data race”.

A pesar de que los threads del programa comparten la variable “maxvalue” y los valores que lee cada uno son los mismos en cada ejecución, no se puede garantizar que el resultado sea determinista. Esto es debido a que puede darse el caso en el que varios threads leen un mismo valor en “maxvalue” y, tras comprobar que se cumple la condición para actualizar el valor máximo, “maxvalue” es actualizado primero por el thread que ha leído el valor más grande, pero seguidamente se sobrescribe con un valor más pequeño por otro thread.

2. Propose two alternative solutions to make it correct, without changing the structure of the code (just add directives or clauses). Explain why they make the execution correct.

Una primera solución sería añadir la directiva “#pragma omp critical” antes de la región en la que se hace la comparación y se produce data race, de manera que cuando un thread esté ejecutando esta parte, el resto no podrá interferir hasta que termine. Sin embargo, esta solución es muy ineficiente debido al overhead necesario para la sincronización entre threads.

```
for (i=id; i < N; i+=howmany) {
    #pragma omp critical
    if (vector[i] > maxvalue)
       maxvalue = vector[i];
}
```

Figura 2. Fragmento de código de 5.datarace.c con critical

Otra posibilidad sería utilizar una reducción para la variable “maxvalue”, añadiendo la cláusula “reduction (max: maxvalue)” a la directiva. De esta manera, cada thread calcula el valor de “maxvalue” en local y, al final de la región, se asegura que se aplica correctamente el operador “max” especificado entre las soluciones parciales. Esta opción es preferible a la anterior, ya que requiere menos sincronización y por lo tanto es más eficiente.

```
#pragma omp parallel private(i) reduction(max: maxvalue)
{
int id = omp_get_thread_num();
int howmany = omp_get_num_threads();

for (i=id; i < N; i+=howmany) {
    if (vector[i] > maxvalue)
       maxvalue = vector[i];
}
}
```

Figura 3. Fragmento de código de 5.datarace.c con reduction

3. Write an alternative distribution of iterations to implicit tasks (threads) so that each of them executes only one block of consecutive iterations (i.e. N divided by the number of threads.)

Una posible solución sería la siguiente:

```
#pragma omp parallel private(i) reduction(max: maxvalue)
{
int id = omp_get_thread_num();
int howmany = omp_get_num_threads();
double size = N;
int block_size = size/howmany;
for (i=id * block_size; i < (id + 1) * block_size; i+=1) {
    if (vector[i] > maxvalue)
       maxvalue = vector[i];
}
}
```

Figura 4. Fragmento de código de 5.datarace.c con ejecución por bloques

Cabe destacar que en esta versión se ha asumido un tamaño del problema N múltiplo al número de threads, de manera que la división por bloques es exacta. En el caso de no ser múltiplo, se tendría que tratar con un redondeo superior.

6.datarace.c

1. Should this program always return a correct result? Reason either your positive or negative answer.

Este programa no siempre devuelve un resultado correcto, también debido al data race en la lectura y escritura. Se trata de un caso muy parecido al programa anterior, y esta vez el error está en la manera en la que se actualiza la variable “countmax”, donde varios threads pueden haber leído el mismo valor en “countmax” y lo incrementan sin haber tenido en cuenta los cambios producidos por otros threads.

2. Propose two alternative solutions to make it correct, without changing the structure of the program (just using directives or clauses) and never making use of critical. Explain why they make the execution correct.

Una posible solución sería utilizar la directiva “#pragma omp atomic”, justo antes de la incrementación de la variable “countmax”, de manera que se aprovechan las operaciones atómicas que proporciona hardware, las cuales aseguran que el incremento de “countmax” se realiza de manera indivisible, evitando así el datarace.

```
for (i=id; i < N; i+=howmany) {
    if (vector[i]==maxvalue)
        #pragma omp atomic
        countmax++;
    }
}
```

Figura 5. Fragmento de código de 6.datarace.c con atomic

Otra solución sería hacer uso de una reducción, definiendo el operador suma “+” y aplicado sobre la variable “countmax”, siendo la nueva directiva “#pragma omp parallel private(i) reduction(+: countmax)”. Igual que en el programa anterior (5.datarace), se asegura que cada thread trata “countmax” como una variable local, y al final de la región se calcula la suma total a partir de los respectivos valores que ha calculado cada thread.

```
#pragma omp parallel private(i) reduction(+: countmax)
{
int id = omp_get_thread_num();
int howmany = omp_get_num_threads();

for (i=id; i < N; i+=howmany) {
    if (vector[i]==maxvalue)
        countmax++;
    }
}
```

Figura 6. Fragmento de código de 6.datarace.c con reduction

7.datarace.c

1. Is this program executing correctly? If not, explain why it is not providing the correct result for one or the two variables (countmax and maxvalue)

La ejecución de este programa es incorrecta. El cálculo de la variable “maxvalue” es correcto; el problema reside en la variable “countmax”.

Para calcular “countmax”, se le aplica una reducción de suma de manera que al final de la región paralela se hace un sumatorio de todos los respectivos valores que ha calculado cada thread. Sin embargo, cada valor es correcto únicamente en el contexto en el que se ha calculado, es decir, para el valor máximo parcial de cada thread. Por lo tanto, a la hora de aplicar la reducción, se utilizan todos los valores independientemente de si son válidos o no, cuando solo se debería tener en cuenta aquellos valores que reflejan el número de ocurrencias del valor máximo real.

2. Write a correct way to synchronise the execution of implicit tasks (threads) for this program.

Una solución sería utilizar el constructor “#pragma omp critical” para toda la región en la que se produce data race. Esta no es una buena solución porque tardaría más que una ejecución secuencial, ya que son equivalentes y además se suma el tiempo producido por overheads.

```
#pragma omp parallel private(i)
{
int id = omp_get_thread_num();
int howmany = omp_get_num_threads();

for (i=id; i < N; i+=howmany) {
    #pragma omp critical
    {
    if (vector[i]==maxvalue) countmax++;
    if (vector[i] > maxvalue) {
       maxvalue = vector[i];
       countmax = 1;
    }
    }
}
}
```

Figura 7. Fragmento de código de 7.datarace.c bien sincronizado, con critical

Otra propuesta sería duplicar el bucle y calcular primero el valor máximo para luego poder contarlo. El tamaño del problema pasaría a ser 2N, pero ambas regiones se pueden ejecutar en paralelo sin producirse data race.

```
#pragma omp parallel private(i) reduction(max: maxvalue)
{
int id = omp_get_thread_num();
int howmany = omp_get_num_threads();

for (i=id; i < N; i+=howmany) {
    if (vector[i] > maxvalue) {
       maxvalue = vector[i];
    }
}
}
#pragma omp parallel private(i) reduction(+: countmax)
{
int id = omp_get_thread_num();
int howmany = omp_get_num_threads();

for (i=id; i < N; i+=howmany) {
    if (vector[i]==maxvalue) countmax++;
}
}
```

Figura 8. Fragmento de código de 7.datarace.c bien sincronizado, con dos bucles

Una solución que permite mantener el tamaño del problema a N sería utilizar una implementación de tablas de hash. Dicha tabla sería compartida por todos los threads, y cada vez que se actualice sería necesario utilizar un constructor “atomic” para evitar el data race. El valor máximo se calcularía mediante la reducción, y finalmente, después de la región paralela, el número de ocurrencias del valor máximo se consultaría en la tabla de hash con dicho valor como key.

8.barrier.c

1. Can you predict the sequence of printf in this program? Do threads exit from the #pragma omp barrier construct in any specific order?

La única secuencia de printf predecible es la segunda (donde se indica que un thread despierta y entra a una barrera), ya que el tiempo de suspensión de la ejecución definido para cada thread depende de su respectivo id, y se escala lo suficiente como para que la salida no salga entrecruzada.

Las otras dos secuencias se ejecutan de manera paralela (pero separadas por la segunda), por lo que su salida es indeterminista. La secuencia del primer printf está claro porque la región empieza ejecutándose en paralelo y se invoca antes de que cada thread suspenda su ejecución. En el caso del tercer printf, a pesar de que los threads despiertan de manera ordenada, no se invoca hasta que todos los threads hayan despertado y se encuentren esperando en la barrera creada con “#pragma omp barrier”. El orden en el que se entra a la barrera no influye en el orden de salida, sino que salen todos a la vez. Por lo tanto, esta última secuencia también se ejecuta en paralelo.

Observing overheads

Take a look at the four different versions and make sure you understand them. How many synchronisation operations (critical or atomic) are executed in each version?

pi_omp_critical.c (pi-v4): número de pasos que pasamos como parámetro, se ejecuta una operación critical en cada iteración del bucle for.

pi_omp_atomic.c (pi-v5): número de pasos que pasamos como parámetro, se ejecuta una operación atomic en cada iteración del bucle for.

pi_omp_sumlocal.c (pi-v6): depende del número de threads, cada thread ejecuta una operación critical después del bucle for.

pi_omp_reduction.c (pi-v7): depende del número de threads, cada thread ejecuta una operación atomic para asegurar la correcta reducción en la variable sum.

1. If executed with only 1 thread and 100.000.000 iterations, do you notice any major overhead in the execution time caused by the use of the different synchronisation mechanisms? You can compare with the baseline execution time of the sequential version in pi_sequential.c.

Version	Overhead (us)
pi_omp_critical.c	2474499
pi_omp_atomic.c	4839
pi_omp_sumlocal.c	5383
pi_omp_reduction.c	5682

Tabla 1. Overhead producido según las distintas versiones del programa “pi_omp” con 1 thread

Tiempo de ejecución de pi_sequential.c -> 1.79 s

Como podemos observar, la versión en la que más overhead se produce es la de critical, llegando a ser incluso mayor al tiempo de la ejecución secuencial. Esto se debe a la manera en la que el sistema operativo gestiona los locks y unlocks del programa: aunque solo haya un thread, las instrucciones para bloquear y desbloquear threads se ejecutan igualmente y por lo tanto requiere más tiempo de ejecución.

2. If executed with 4 and 8 threads and the same number of iterations, do the 4 programs benefit from the use of several processors in the same way? Can you guess the reason for this behaviour?

Version	Overhead (us) - 4	Overhead (us) - 8
pi_omp_critical.c	36945255	34656814
pi_omp_atomic.c	5020649	5868909
pi_omp_sumlocal.c	13035	18926
pi_omp_reduction.c	12917	19276

Tabla 2. Overhead producido según las distintas versiones del programa “pi_omp” con 4 y 8 threads

Como se puede observar, a excepción del programa “pi_omp_critical”, el comportamiento general es que cuanto mayor es el número de threads utilizados, más overhead se produce, ya que se requiere más tiempo de sincronización.

También cabe destacar que en los programas “pi_omp_sumlocal” y “pi_omp_reduction” el incremento de overhead es menor respecto a “pi_omp_atomic”, ya que en estos dos primeros se ejecutan menos operaciones de sincronización.

Can you quantify (in microseconds) the cost of each individual synchronisation operation (critical or atomic) that is used?

Dado el tiempo total de overhead y el número de operaciones de sincronización que se ejecutan, el coste individual de cada programa es:

pi_omp_critical: T/steps = 0.02474499 us

pi_omp_atomic: T/steps = 0.00004839 us

pi_omp_sumlocal: T/threads = 5383 us

pi_omp_reduction: T/threads = 5682 us

Session 2: A very practical introduction to OpenMP (Part II)

OpenMP questionnaire - Day 2: explicit tasks

1.single.c

1. What is the nowait clause doing when associated to single?

Cuando una cláusula single tiene una cláusula de nowait, se elimina la barrera implícita que existe al final del single. Y esto permite superponer la ejecución del trabajo no dependiente dentro del single con lo que continúa después.

2. Then, can you explain why all threads contribute to the execution of the multiple instances of single? Why those instances appear to be executed in bursts?

Todas los threads contribuyen a la ejecución debido a que la cláusula nowait elimina la barrera implícita del final de la cláusula single, y por lo tanto, mientras un thread ejecuta una iteración otro thread puede estar ejecutando otra iteración sin la necesidad de que el primero acabe. Las instancias parecen ejecutarse en ráfagas debido al sleep(1) que hay al final de cada iteración, que bloquea la ejecución durante 1 unidad de tiempo. Es decir, como hay 4 threads, entre los 4 se distribuyen las 4 primeras iteraciones y después se bloquean durante una unidad de tiempo y siguen con la ejecución.

2.fibtasks.c

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

El programa no se ejecuta en paralelo debido a que solamente hay una llamada a una constructora, y es #pragma omp task, que su función es la de crear tareas, pero no hay ninguna constructora que cree threads.

2. Modify the code so that tasks are executed in parallel and each iteration of the while loop is executed only once.

```
#pragma omp parallel
#pragma omp single
{
while (p != NULL) {
    printf("Thread %d creating task that will compute %d\n", omp_get_thread_num(), p->data);
    #pragma omp task firstprivate(p)
    processwork(p);
    p = p->next;
}
}
printf("Finished creation of tasks to compute the Fibonacci for numbers in linked list \n");
```

Figura 9: fragmento de código 2.fibtasks.c

La modificación que se ha hecho ha sido añadir la constructora #pragma omp parallel, para la creación de los threads y la constructora #pragma omp single que controla que solo un thread está ejecutando una iteración, por lo que garantizamos que las iteraciones del bucle sólo son ejecutadas una vez.

3. What is the firstprivate(p) clause doing? Comment it and execute again. What is happening with the execution? Why?

Cuando una variable se marca como firstprivate, la variable dentro de la construcción es una nueva variable del mismo tipo, pero se inicializa con el valor de la variable original. Es decir, todos los threads tienen una variable diferente con el mismo valor inicial. Al comentarlo y ejecutarlo de nuevo, todos los threads están usando la misma variable “p”, por lo que se produce datarace.

3.taskloop.c

1. Which iterations of the loops are executed by each thread for each task grainsize or num tasks specified?

En el primer for (#pragma omp taskloop grainsize(VALUE)), las iteraciones del for se dividen por el parámetro VALUE del grainsize. Entonces se crean tareas por las iteraciones consecutivas, por ejemplo, en taskloop.c se hacen N iteraciones (con N = 12) y el VALUE del grainsize es 4, por lo que se crean 12/4 = 3 tareas, cada tarea con 4 iteraciones, por lo que se asignan cada 4 iteraciones consecutivas a un thread. (3 tareas de 4 iteraciones)

En el segundo for (#pragma omp taskloop num_tasks(VALUE)), el número de iteraciones totales del for se dividen por el número de tareas especificadas en la constructora para saber de cuántas iteraciones son cada tarea. Por ejemplo, en taskloop.c el VALUE de num_tasks es 4, por lo que se crean 4 tareas, y para saber cuántas iteraciones tiene cada tarea, hay que dividir el N (que es 12) por 4, que da 3. Por lo que se crean 4 tareas de 3 iteraciones cada una.

2. Change the value for grainsize and num tasks to 5. How many iterations is now each thread executing? How is the number of iterations decided in each case?

En el primer caso (#pragma omp taskloop grainsize(VALUE)), cada thread ejecuta 6 iteraciones, es decir, se crean 2 tareas de 6 iteraciones cada una. Esto ocurre porque la división para obtener el número de tareas no da un número exacto, por lo que grainsize encuentra el primer número (más grande que el actual) tal que el módulo con el número de iteraciones sea 0, en este caso como 12/5 da 2.4, grainsize utilitza el 6, 12/6 = 2.

En cambio en el otro caso (#pragma omp taskloop num_tasks(VALUE)), cada thread ejecuta 2 o 3 iteraciones, esto pasa por que el número de tareas que se crean es fijo, en este caso 5. Y en este caso, al tener que N = 12, 12/5 = 2.4 (no es exacto), por lo que hay threads que ejecutan 2 iteraciones y otros que ejecutan 3 iteraciones.

3. Can grainsize and num tasks be used at the same time in the same loop?

No se puede usar para el mismo bucle for al mismo tiempo num tasks y grainsize porque son contradictorios, ya que dividen las iteraciones de manera diferente.

```
par2220@boada-1:~/lab2/openmp/Day2$ make
icc 3.taskloop.c -Wall -g -O3 -fno-inline -fopenmp -fopenmp -o 3.taskloop
3.taskloop.c(34): error: directive cannot contain both grainsize and num_tasks clauses
       #pragma omp taskloop num_tasks(VALUE) grainsize(VALUE)
                                             ^
```

Figura 10: Captura de la terminal al intentar compilar el código con grainsize y num_tasks

4. What is happening with the execution of tasks if the nogroup clause is uncommented in the first loop? Why?

Si comentamos la cláusula nogroup en el primer bucle for, se elimina el grupo de tareas implícito en la construcción del bucle, por lo que cada thread puede ejecutar la primera tarea encontrada, cada iteración es una tarea.

4.reduction.c

1. Complete the parallelisation of the program so that the correct value for variable sum is returned in each printf statement. Note: in each part of the 3 parts of the program, all tasks generated should potentially execute in parallel.

```
int main() {

    int i;

    for (i=0; i<SIZE; i++) X[i] = i;

    omp_set_num_threads(4);
    #pragma omp parallel
    #pragma omp single
    {
        // Part I
        #pragma omp taskgroup task_reduction(+: sum)
        {
            for (i=0; i< SIZE; i++)
                #pragma omp task firstprivate(i) in_reduction(+: sum)
                sum += X[i];
        }

        printf("Value of sum after reduction in tasks = %d\n", sum);

        // Part II
        #pragma omp taskloop grainsize(BS) firstprivate(sum)
            for (i=0; i< SIZE; i++)
                sum += X[i];
        printf("Value of sum after reduction in taskloop = %d\n", sum);

        // Part III
        #pragma omp taskgroup task_reduction(+: sum)
        {
            #pragma omp taskloop grainsize(BS) firstprivate(sum)
                for (i=0; i< SIZE/2; i++)
                    sum += X[i];

            #pragma omp taskloop grainsize(BS) firstprivate(sum)
                for (i=SIZE/2; i< SIZE; i++)
                    sum += X[i];
            printf("Value of sum after reduction in combined task and taskloop = %d\n", sum);
        }
    }
    return 0;
}
```

Figura 11: Fragmento del código 4.reduction.c con las modificaciones correspondientes

5.synchtasks.c

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

Las tareas a (foo1), b (foo2) y c (foo3) no dependen de ninguna otra tarea, por lo que son los primeros en ejecutarse, la tarea d (foo4) depende de a (foo1) y b (foo2), por lo que empezará su ejecución cuando finalicen las tareas a (foo1) y b (foo2). Finalmente, la tarea e (foo5) depende de c (foo3) y d (foo4), por lo que empezará una vez finalicen c (foo3) y d (foo4).

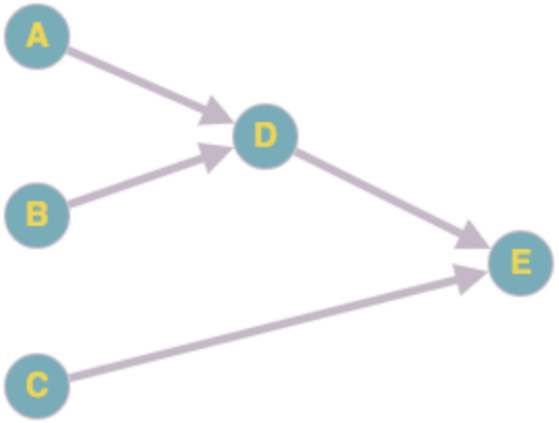

Figura 12: grafo de dependencias de las tareas

2. Rewrite the program using only taskwait as task synchronisation mechanism (no depend clauses allowed), trying to achieve the same potential parallelism that was obtained when using depend.

```
int main(int argc, char *argv[]) {

    #pragma omp parallel
    #pragma omp single
    {
        printf("Creating task foo3\n");
        #pragma omp task
        foo3();
        printf("Creating task foo1\n");
        #pragma omp task
        foo1();
        printf("Creating task foo2\n");
        #pragma omp task
        foo2();
        printf("Creating task foo4\n");
        #pragma omp taskwait
        #pragma omp task
        foo4();
        printf("Creating task foo5\n");
        #pragma omp taskwait
        #pragma omp task
        foo5();
    }
    return 0;
}
```

Figura 13: Fragmento del código 5.synctasks.c con las modificaciones correspondientes

Primero se crean y se ejecutan las tareas a (foo1), b (foo2) y c (foo3), a continuación, hay un taskwait antes de d (foo4), por lo que no se ejecuta si antes no se han terminado a, b y c. Lo mismo ocurre para e (foo5).
En este caso la tarea c (foo3), se podría poner tanto antes de la tarea d (foo4) como después, ya que la tarea d (foo4) depende únicamente de a (foo1) y b (foo2).

3. Rewrite the program using only taskgroup as task synchronisation mechanism (no depend clauses allowed), again trying to achieve the same potential parallelism that was obtained when using depend.

```
int main(int argc, char *argv[]) {
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp taskgroup
        {
            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();
        }
        #pragma omp taskgroup
        {
            printf("Creating task foo4\n");
            #pragma omp task
            foo4();
        }
        printf("Creating task foo5\n");
        #pragma omp task
        foo5();
    }
    return 0;
}
```

Figura 14: Fragmento del código 5.synctasks.c con las modificaciones correspondientes

Al poner taskgroup, garantizamos que no se ejecute un taskgroup si no ha finalizado el anterior. Por lo que se respetan las dependencias. Como ocurre anteriormente. la tarea c (foo3) podría ir tanto en el primer taskgroup con las tareas a (foo1) y b (foo2) como con el taskgroup con la tarea d (foo4) y por la misma razón.

Observing overheads

Thread creation and termination

How does the overhead of creating/terminating threads varies with the number of threads used? Which is the order of magnitude for the overhead of creating/terminating each individual thread in the parallel region?

Se puede observar en los resultados obtenidos a continuación que a medida que se aumentan el número de threads, el overhead también va aumentando, es decir, el tiempo del overhead no se mantiene constante, es proporcional al número de threads. El overhead por thread, en cambio, va disminuyendo con el aumento de threads.

All overheads expressed in microseconds

Nthr	Overhead	Overhead per thread
2	2.2047	1.1023
3	1.7091	0.5697
4	1.7873	0.4468
5	2.0370	0.4074
6	2.1252	0.3542
7	2.0962	0.2995
8	2.3623	0.2953
9	2.3909	0.2657
10	2.5398	0.2540
11	2.4424	0.2220
12	2.7567	0.2297
13	3.0338	0.2334
14	3.4756	0.2483
15	3.0369	0.2025
16	3.7610	0.2351
17	3.2286	0.1899
18	3.3375	0.1854
19	3.2664	0.1719
20	3.8394	0.1920
21	3.2755	0.1560
22	4.2065	0.1912
23	3.3829	0.1471
24	3.7247	0.1552

Tabla 3: Tiempo de los overhead en relación con el número de threads

Task creation and synchronisation

How does the overhead of creating/synchronising tasks varies with the number of tasks created? Which is the order of magnitude for the overhead of creating/synchronising each individual task?

El overhead aumenta a medida que se aumenta el número de tareas y el overhead por tarea se mantiene constante.

Al aumentar el número de tareas, aumenta el número de operaciones de sincronización que provocan overhead, y por lo tanto, el overhead producido es proporcional al número de tareas.

El orden de magnitud de los overheads por tareas es de microsegundos.

All overheads expressed in microseconds

Ntasks	Overhead	Overhead per task
2	0.1974	0.0987
4	0.4849	0.1212
6	0.7172	0.1195
8	0.9521	0.1190
10	1.1838	0.1184
12	1.4166	0.1181
14	1.6497	0.1178
16	1.8827	0.1177
18	2.1216	0.1179
20	2.3498	0.1175
22	2.5900	0.1177
24	2.8195	0.1175
26	3.0408	0.1170
28	3.2905	0.1175
30	3.5139	0.1171
32	3.7633	0.1176
34	3.9862	0.1172
36	4.2100	0.1169
38	4.4456	0.1170
40	4.6936	0.1173
42	4.9224	0.1172
44	5.1653	0.1174
46	5.3811	0.1170
48	5.6289	0.1173
50	5.8603	0.1172
52	6.0789	0.1169
54	6.3277	0.1172
56	6.5527	0.1170
58	6.7773	0.1169
60	7.0300	0.1172
62	7.2987	0.1177
64	7.4733	0.1168

Tabla 4: Tiempo de los overhead en relación con el número de tareas

Conclusiones

En esta segunda práctica se ha hecho una breve introducción a la interfaz de programación OpenMP, y consta de dos sesiones:

La primera sesión ha estado enfocada en las distintas maneras de ejecutar distintos fragmentos de código en paralelo, entrando en detalle sobre las diferentes opciones de ejecución que hay (constructores y directivas) y el efecto que tienen sobre la ejecución del programa, ya sea relacionado con el overhead producido o los errores de data race y data sharing.

La segunda sesión se ha centrado más en el estudio de los overheads causados debido a la creación de tareas y en la sincronización de ellas, junto a las dependencias que hay entre estas para mantener consistencia en el programa.

Gracias a haber puesto en práctica los diversos conceptos teóricos adquiridos, y junto a las aclaraciones del profesor, hemos podido entender mejor las herramientas ofrecidas por la interfaz de programación OpenMP.