UNIVERSITAT POLITÈCNICA DE CATALUNYA

Paralelismo

Laboratorio 2: brief tutorial on OpenMP programming model

24/03/2022

Haopeng Lin Ye

Joan Sales de Marcos

ÍNDICE

1 - OpenMP questionnaire

1.1 - 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"?

El mensaje “Hello world!” es mostrado dos veces ya que esa parte del código se ejecuta con 2 threads.

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

Únicamente es necesario añadir OMP_NUM_THREADS=4 para que ese fragmento se ejecute con 4 threads.

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?

La ejecución del programa no es correcta porque no hay control de flujo dentro de la región paralela, por lo tanto los 8 threads ejecutan lo mismo al mismo tiempo y no se sabe quién está ejecutando qué en cada momento.

```
par2311@boada-1:~/lab2/openmp/Day1$ ./2.hello
(4) Hello (0) world!
(0) Hello (0) world!
(2) Hello (2) world!
(3) Hello (2) world!
(6) Hello (6) world!
(1) Hello (1) world!
(5) Hello (5) world!
(7) Hello (7) world!
```

Figura 1.1:Output original de codigo

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

Figura 1.2: Código modificado

```
par2311@boada-1:~/lab2/openmp/Day1$ ./2.hello
(4) Hello (4) world!
(3) Hello (3) world!
(2) Hello (2) world!
(5) Hello (5) world!
(1) Hello (1) world!
(0) Hello (0) world!
(7) Hello (7) world!
(6) Hello (6) world!
```

Figura 1.3: Output de código modificado

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

Las líneas no se escribirán siempre en el mismo orden. Esto se debe a que los threads no llegan en orden.

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?

```
Starting, I'm alone ... (1 thread)
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!


Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the fourth parallel (2)!
Hello world from the fourth parallel (2)!
Hello world from the fourth parallel (3)!
Hello world from the fourth parallel (3)!
Hello world from the fourth parallel (3)!
Outside parallel, nobody else here ... (1 thread)
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the fifth parallel (4)!
Hello world from the sixth  parallel (3)!
Hello world from the sixth  parallel (3)!
Hello world from the sixth  parallel (3)!
Finishing, I'm alone again ... (1 thread)
```

Figura 1.4 Output del programa 3

Si estamos fuera de la región paralela, nos devolverá 1 ya que estamos en la parte secuencial, y solo habrá un thread ejecutándose.

Hay varias regiones paralelas (4 con #pragma omp parallel y 1 con #pragma omp parallel num_threads(4)).

La primera región y tercera devuelven 8 threads porque lo hemos configurado nosotros en el momento de ejecutar el programa.

En la segunda y quinta región configuramos el número de threads con num_threads(4), de esta manera el valor que muestra será 4.

La cuarta región está dentro de un bucle. Antes de llamar a #pragma omp parallel cambiamos el valor de número de threads por *i,* por tanto el número de threads dependerá de *i*, que en nuestro caso será 2 y 3, correspondiente a los mensajes que muestra el programa.

Finalmente, la última región paralela se ejecutará con 3 porque previamente dentro del bucle cambiamos el valor de num_threads.

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

Podemos definir el número de threads usando la cláusula num_threads() al definir el pragma (especificando entre los paréntesis el número de threads que queremos).

Otra manera sería añadir la función omp_set_num_threads() antes de llamar a la paralelización (también poniendo entre los paréntesis el número de threads).

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

Para la cláusula num_threads() de los pragmas: ésta estará “activa” hasta que se salga de esa región del pragma, es decir, toda la parte que engloba las llaves “{}” que abren esa tarea.

Para la función omp_set_num_threads(): ésta estará “activa” hasta que el programa termine o se vuelvan a definir los threads con otro valor.

4.data_sharing.c

1. Which is the value of variable x after the execution of each parallel region with different data-sharing attribute (shared, private, firstprivate and

reduction)? Is that the value you would expect? (Execute several times if necessary)

```
After first parallel (shared) x is: 120
After second parallel (private) x is: 5
After third  parallel (firstprivate) x is: 5
After fourth parallel (reduction) x is: 125
```

Figura 1.5 Output del programa 4

El primer atributo es **shared.** Este atributo hace que la variable x sea compartida entre todos los threads que están ejecutando esta región del código. Cuando modificamos el valor es posible que dos threads accedan a esta variable al mismo tiempo, provocando que el resultado sea incorrecto.
El segundo atributo es **private(x).** Este atributo especifica que cada thread debe tener su propia instancia de la variable, inicializada con valor 0. Una vez salido de la región paralela el valor de x vuelve a tener el valor de antes de entrar en la región paralela. En nuestro caso x = 5.
El tercer atributo, **firstprivate(x)**, es bastante similar al anterior. La principal diferencia es que la variable de dentro de la región paralela se inicializa con el valor original. Después de salir de la región paralela tendrá el mismo valor que el apartado anterior.
El último atributo es **reduction(+:x).** Especifica que la variable x es privada para cada thread e indica cuál es la operación para el final de la región paralela. En nuestro caso x = ((sumatorio de 0 hasta 15) + 5) = 125.

5.datarace.c

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

No funcionará siempre correctamente ya que la variable que modificamos es compartida. En algún momento puede pasar que dos o más threads accedan al mismo tiempo y nos dará un resultado incorrecto.

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.

>Usar critical: Asegura exclusión mutua en la ejecución del bloque, es decir, no accederán simultáneamente al mismo bloque.

>Usar atomic: Asegura la carga y el almacenamiento de la variable x de forma atómica.

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.

```
int num_bloques = N/ omp_get_num_threads();
int id = omp_get_thread_num();
for (i=id*num_bloques; i < (id+1)*num_bloques; i++) {
     #pragma omp atomic
     if (vector[i] > maxvalue)maxvalue = vector[i];
}
```

Figura 1.6 Código modificado del program 5

6.datarace.c

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

No siempre devolverá un resultado correcto. La variable *countmax* no está protegida, esto causará que cada thread modifique la variable sin tener en cuenta los otros threads, provocando así el error del resultado.

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.

```
#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 1.7 Alternativa 1 del programa 6

```
#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) {
         #pragma omp atomic
         countmax++;
        }
     }
 }
```

Figura 1.8 Alternativa 2 del programa 6

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)

El programa no se ejecuta correctamente debido a que éste tiene en cuenta el valor máximo de cada thread y cada uno de éstos añade uno por lo menos al recuento final.

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

```
#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 1.9 Versión 2 del program 7

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

```
par2311@boada-1:~/lab2/openmp/Day1$ ./8.barrier
(0) going to sleep for 2000 milliseconds ...
(1) going to sleep for 5000 milliseconds ...
(2) going to sleep for 8000 milliseconds ...
(3) going to sleep for 11000 milliseconds ...
(0) wakes up and enters barrier ...
(1) wakes up and enters barrier ...
(2) wakes up and enters barrier ...
(3) wakes up and enters barrier ...
(0) We are all awake!
(2) We are all awake!
(1) We are all awake!
(3) We are all awake!
```

Figura 1.10 Output del programa 8

Después de haber ejecutado 100 veces el programa , no hemos podido observar ningún patrón en específico, excepto que después del omp barrier, el thread 0 siempre acaba el primero ya que es el que tiene menor tiempo de sleep.

Day 2: explicit tasks

1.single.c

Ejecución interactiva:

```
par2319@boada-1:~/lab2/openmp/Day2$ ./1.single
Thread 2 executing instance 1 of single
Thread 0 executing instance 0 of single
Thread 1 executing instance 2 of single
Thread 3 executing instance 3 of single
Thread 0 executing instance 5 of single
Thread 3 executing instance 7 of single
Thread 1 executing instance 6 of single
Thread 2 executing instance 4 of single
Thread 0 executing instance 8 of single
Thread 3 executing instance 11 of single
Thread 1 executing instance 10 of single
Thread 2 executing instance 9 of single
Thread 0 executing instance 12 of single
Thread 3 executing instance 15 of single
Thread 2 executing instance 14 of single
Thread 1 executing instance 13 of single
Thread 1 executing instance 16 of single
Thread 3 executing instance 18 of single
Thread 2 executing instance 17 of single
Thread 0 executing instance 19 of single
```

Figura 2.1: Output del programa 1

Lo que hace el programa es que los diferentes threads ejecuten varias veces el código para ver quién ejecuta qué iteración.

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

Parece ser que el nowait hace que los threads sigan la ejecución sin importar las dependencias, es decir, que todos los threads ejecutan el código simultáneamente (no se esperan a que el que ha empezado antes termine). Se puede apreciar que el código se está ejecutando con 4 threads y también se ve el orden aleatorio de éstos en la terminal.

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?

Cada 4 líneas de output hay una espera de un segundo (Por el sleep(1)) e inmediatamente salen otras cuatro líneas en diferente orden de threads.

Más específicamente: hay un total de 4 threads ejecutando el código en modo “single” y sin esperas (nowait). Si se sigue el código, primero se escriben 4 líneas (una por cada printf) y cuando el thread llega al sleep, se espera un segundo y continúa con la siguiente iteración del for sin esperar al resto de threads. Esto provoca una salida en orden caótico.

2.fibtasks.c

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

El programa no tiene región paralela, por eso solo lo está ejecutando 1 thread. Consecuentemente solo 1 thread creará las tareas y las ejecutará.

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;
      }
```

Figura 2.2: Versión 2 del programa 2.fibtasks.c

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

La clausura firstprivate hace que solo el primer thread cree todas las tareas y que cada una tenga una copia del valor p. Después, todos los threads pueden ejecutar individualmente cada tarea.

3.taskloop.c

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

```
Thread 0 distributing 12 iterations with grainsize(4) ...
Loop 1: (0) gets iteration 8
Loop 1: (1) gets iteration 0
Loop 1: (1) gets iteration 1
Loop 1: (1) gets iteration 2
Loop 1: (1) gets iteration 3
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Loop 1: (1) gets iteration 6
Loop 1: (1) gets iteration 7
Loop 1: (0) gets iteration 9
Loop 1: (0) gets iteration 10
Loop 1: (0) gets iteration 11
Thread 0 distributing 12 iterations with num_tasks(4) ...
Loop 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (1) gets iteration 2
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 4
Loop 2: (1) gets iteration 5
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (1) gets iteration 8
Loop 2: (0) gets iteration 9
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
```

Figura 2.3: Output del programa 3.taskloop.c con gransize y num_task a 4

La cláusula grainsize(grain-size) controla cuántas iteraciones de ciclo se asignan a cada tarea creada. El número de iteraciones del bucle asignadas a cada tarea creada es mayor o igual al valor mínimo del tamaño de gránulo y el número total de iteraciones.

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?

Después de ejecutar el programa varias veces, uno de los resultados ha sido el siguiente:

```
Thread 0 distributing 12 iterations with grainsize(5) ...
Loop 1: (0) gets iteration 6
Loop 1: (0) gets iteration 7
Loop 1: (0) gets iteration 8
Loop 1: (0) gets iteration 9
Loop 1: (0) gets iteration 10
Loop 1: (0) gets iteration 11
Loop 1: (1) gets iteration 0
Loop 1: (1) gets iteration 1
Loop 1: (1) gets iteration 2
Loop 1: (1) gets iteration 3
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Thread 0 distributing 12 iterations with num_tasks(4) ...
Loop 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (1) gets iteration 2
Loop 2: (2) gets iteration 3
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (1) gets iteration 8
Loop 2: (0) gets iteration 9
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
```

Figura 2.4: Output del programa 3.taskloop.c con gransize y num_task a 4

Al cambiar el grainsize a 5, cada thread pasa a ejecutar 6 iteraciones del bucle. Esto ocurre ya que el número de tareas no es divisible por el número de gránulos (12 % 5 ≠ 0), así que el compilador escoge un número divisible superior a éste e inferior al doble. En este caso 6 ya que 12 % 6 = 0; 6 > 5; y 6 < 10.

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

Sí se pueden usar: primero num_tasks dividirá el bucle en el número de tareas especificado y después grainsize dividirá esas tareas en el número de gránulos que se le ponga. Esto solo ocurrirá si num_tasks y grainsize son más pequeños que el número de iteraciones, como podemos deducir de los apartados anteriores.

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

Esta es una de las salidas:

```
Thread 0 distributing 12 iterations with grainsize(5) ...
Thread 0 distributing 12 iterations with num_tasks(4) ...
Loop 2: (0) gets iteration 9
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
Loop 2: (0) gets iteration 6
Loop 2: (0) gets iteration 7
Loop 2: (0) gets iteration 8
Loop 2: (0) gets iteration 3
Loop 2: (0) gets iteration 4
Loop 2: (0) gets iteration 5
Loop 2: (0) gets iteration 0
Loop 2: (0) gets iteration 1
Loop 2: (0) gets iteration 2
Loop 1: (0) gets iteration 6
Loop 1: (0) gets iteration 7
Loop 1: (0) gets iteration 8
Loop 1: (0) gets iteration 9
Loop 1: (0) gets iteration 10
Loop 1: (0) gets iteration 11
Loop 1: (0) gets iteration 0
Loop 1: (0) gets iteration 1
Loop 1: (0) gets iteration 2
Loop 1: (0) gets iteration 3
Loop 1: (0) gets iteration 4
Loop 1: (0) gets iteration 5
```

Figura 2.5: Output del programa 3.taskloop.c con gransize a 5 y num_task a 4 además de descomentar *no group*

Al descomentar la cláusula nogroup, el primer thread que encuentra esa tarea la ejecuta él solo, como se puede ver en la salida. En la salida anterior el thread 0 encuentra primero todas las tareas y las ejecuta (por pura suerte). En otras

ejecuciones varios threads diferentes han encontrado y ejecutado las tasks por separado:

```
Thread 0 distributing 12 iterations with grainsize(5) ...
Thread 0 distributing 12 iterations with num_tasks(4) ...
Loop 1: (1) gets iteration 0
Loop 1: (1) gets iteration 1
Loop 1: (1) gets iteration 2
Loop 1: (1) gets iteration 3
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Loop 2: (0) gets iteration 9
Loop 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (1) gets iteration 2
Loop 1: (2) gets iteration 6
Loop 1: (2) gets iteration 7
Loop 1: (2) gets iteration 8
Loop 1: (2) gets iteration 9
Loop 1: (2) gets iteration 10
Loop 1: (2) gets iteration 11
Loop 2: (1) gets iteration 3
Loop 2: (1) gets iteration 4
Loop 2: (1) gets iteration 5
Loop 2: (2) gets iteration 6
Loop 2: (2) gets iteration 7
Loop 2: (2) gets iteration 8
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
```

Figura 2.6: Otro output del programa 3.taskloop.c con gransize a 5 y num_task a 4 además de descomentar *no group*

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.

Todos los cambios hechos en el programa han sido añadir cláusulas en los pragmas además de reubicar y modificar otros. Ninguna otra parte del documento ha sido modificada. El main queda así:

```
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 2.7: Código modificado del programa 4

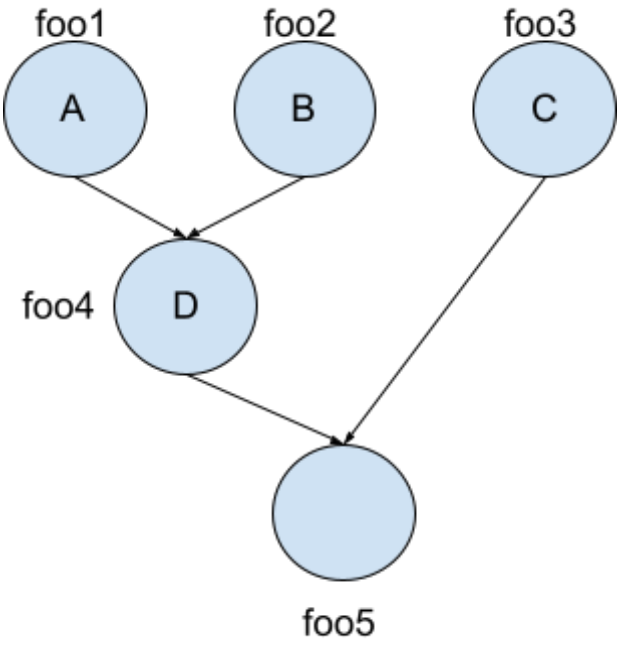
Con este código, la salida al ejecutar el programa es la siguiente:

```
par2319@boada-1:~/lab2/openmp/Day2$ ./4.reduction
Value of sum after reduction in tasks = 33550336
Value of sum after reduction in taskloop = 33550336
Value of sum after reduction in combined task and taskloop =
33550336
```

Figura 2.8: Output del programa 4

5.synchtasks.c

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



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 a, b, c, d;
int main(int argc, char *argv[]) {

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

Figura 2.10: Versión 2 del programa 5

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 a, b, c, d;
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();
          }
               #pragma omp taskgroup
          {
               printf("Creating task foo4\n");
               #pragma omp task
               foo4();
               printf("Creating task foo3\n");
               #pragma omp task
               foo3();
          }

     printf("Creating task foo5\n");
     #pragma omp task
     foo5();
    }
    return 0;
}
```

Figura 2.11: Versión 3 del programa 5

2 - Observing overheads

En esta sección se analizará de dónde salen y cómo afectan los threads a los tiempos de overhead de diferentes versiones del código que calcula el número pi.

Primero de todo, para obtener diferentes datos de overhead con diferentes pragmas ejecutamos los códigos de pi_sequential, pi_omp_atomic con 4 y 8 threads, y pi_omp_critical con 4 y 8 threads. Todos con 100000000 iteraciones.

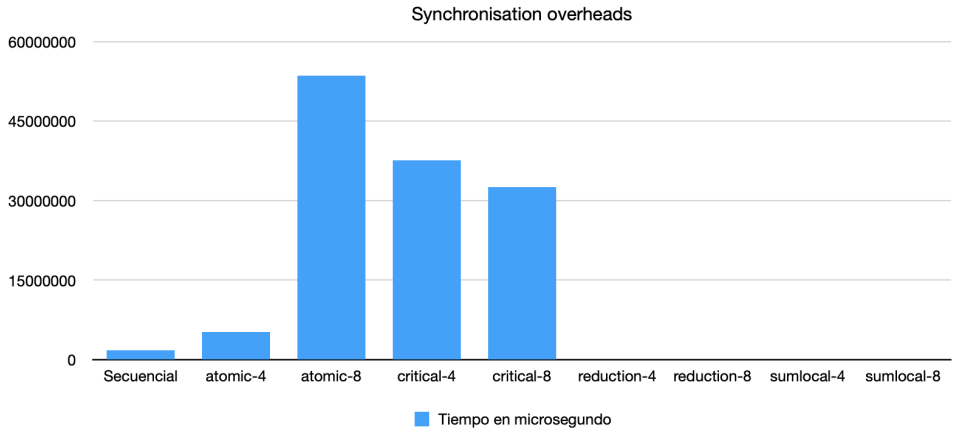


Figura 3.1: Gráfica de los tiempos de overhead de las diferentes ejecuciones de los programas

Synchronisation overheads

	Tiempo en microsegundo
Secuencial	1793315,89
atomic-4	5273915
atomic-8	53567848,75
critical-4	37580443,75
critical-8	32571923,125
reduction-4	12848.75
reduction-8	18927.25
sumlocal-4	13306
sumlocal-8	19425,5

Figura 3.2: Tabla con los tiempos de overhead exactos de las diferentes ejecuciones de los programas

Se puede observar que el programa que usa más tiempo en sincronizarse es pi_omp_atomic con 8 threads. Esto seguramente sea porque en el código se le aplica un pragma *atomic* al resultado de la suma de cada iteración, por tanto todos los threads han de esperar a que acaben los demás antes de comprobar y “fusionar” los resultados. Esto también explica la diferencia abismal entre esta ejecución y la de 4 threads: a menos threads, menos atomicidad y menos sincronización.

Para las versiones de pi_omp_critical, los resultados son similares a la de atomic porque al definir la variable sum como “crítica” con el pragma *critical* estamos diciéndole a los threads que esta información la han de usar exclusivamente por separado para luego juntarla, requiriendo tiempo para sincronizar los resultados.

Nótese que el tiempo de overhead en este caso con 8 threads es menor al de 4 threads, esto se debe a que el pragma *critical* es menos efectivo para paralelizar que el *atomic*.

Para pi_sequential, el tiempo de overhead es mínimo porque es un programa secuencial: no hay que sincronizar resultados.

A continuación observaremos cómo cambia el tiempo de sincronización según el número de threads que se usen en un mismo programa. Para esto usaremos primero el código de pi_omp_parallel:

PI_OMP_PARALLEL

Nthr	Overhead	Overhead per thread
2	2,0060	1,0030
3	1,3941	0,4647
4	1,5768	0,3942
5	1,6657	0,3331
6	1,8158	0,3026
7	1,8609	0,2658
8	2,1255	0,2657
9	2,2412	0,2490
10	2,3314	0,2331
11	2,3412	0,2128
12	2,4923	0,2077
13	2,8210	0,2170
14	3,3356	0,2383
15	2,8614	0,1908
16	3,4790	0,2174
17	3,1132	0,1831
18	3,4804	0,1934
19	3,1210	0,1643
20	3,3183	0,1659
21	3,2153	0,1531
22	3,5358	0,1607
23	3,2243	0,1402
24	3,7088	0,1545

Figura 3.3: Tabla con los tiempos exactos de overhead de pi_omp_parallel hasta 24 threads (la columna de Overhead per thread se obtiene al dividir el tiempo de overhead entre el nº de threads)

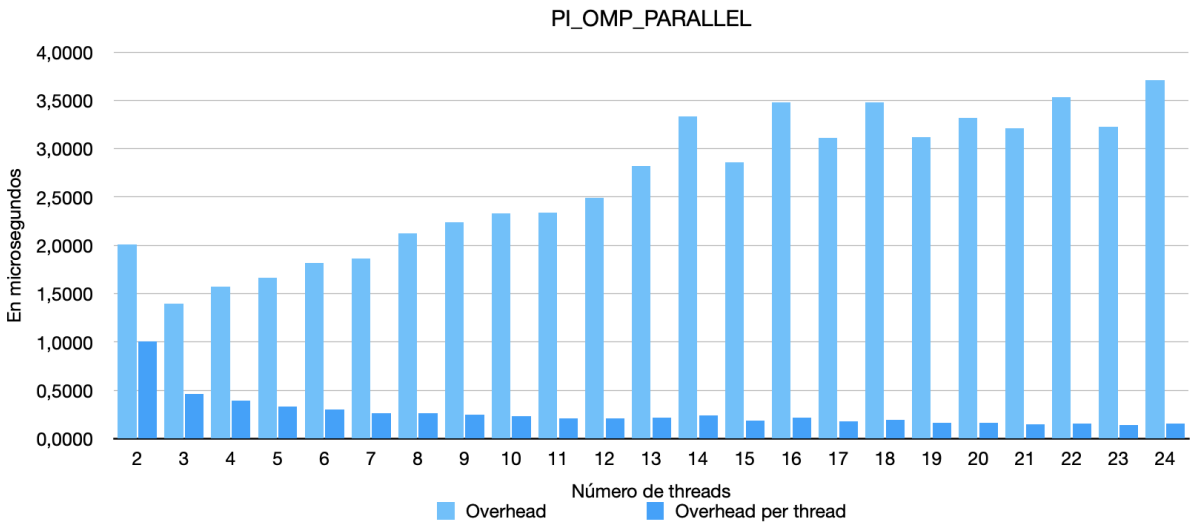


Figura 3.4: Gráfica con los tiempos de overhead de pi_omp_parallel hasta 24 threads

Este código únicamente tiene definido un pragma antes del bucle para calcular la suma: `#pragma omp parallel private(x) firstprivate(sum)`

Se puede observar claramente un incremento en el tiempo de overhead a medida que se van aumentando los threads (No es contínuo y constante pero se puede apreciar el aumento). Esto está acompañado con un decremento de overhead por thread. De estos datos podemos concluir que a mayor número de threads, en general se necesitará más tiempo para sincronizar los resultados pero para cada thread en concreto costará menos tiempo hacerlo.

Al hacer este experimento con el código de pi_omp_tasks (En esta versión del código se crean tareas para hacer el bucle de sum y después se hace un *taskwait* para sincronizar los resultados), hemos obtenido un resultado similar pero con unas diferencias:

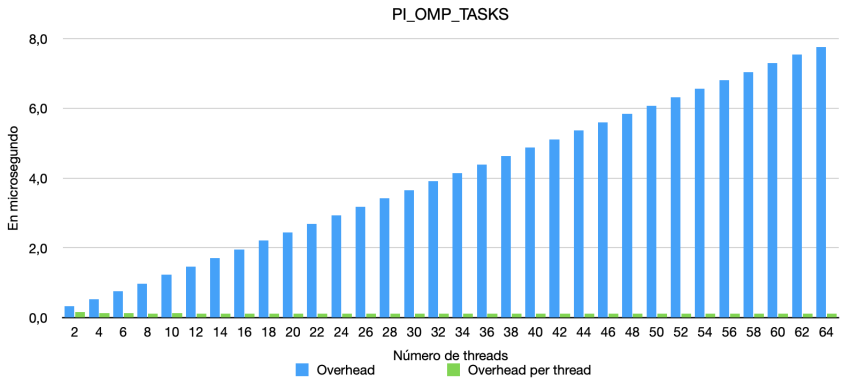


Figura 3.5: Gráfica con los tiempos de overhead de pi_omp_tasks hasta 64 threads

PI_OMP_TASKS

Nthr	Overhead	Overhead per thread
2	0,3299	0,165
4	0,5247	0,1312
6	0,759	0,1265
8	0,9736	0,1217
10	1,2301	0,123
12	1,4698	0,1225
14	1,7122	0,1223
16	1,9547	0,1222
18	2,205	0,1225
20	2,4397	0,122
22	2,6806	0,1218
24	2,93	0,1221
26	3,1708	0,122
28	3,4196	0,1221
30	3,6568	0,1219
32	3,9055	0,122
34	4,1397	0,1218
36	4,3836	0,1218
38	4,6265	0,1217
40	4,8755	0,1219
42	5,106	0,1216
44	5,3695	0,122
46	5,5924	0,1216
48	5,8434	0,1217
50	6,0708	0,1214
52	6,3229	0,1216
54	6,5617	0,1215
56	6,8102	0,1216
58	7,0335	0,1213
60	7,2912	0,1215
62	7,5337	0,1215
64	7,7601	0,1213

Figura 3.6: Tabla con los tiempos exactos de overhead de pi_omp_tasks hasta 64 threads (la columna de Overhead per thread se obtiene al dividir el tiempo de overhead entre el nº de threads)

La primera diferencia que se puede observar es que el tiempo de sincronización a medida que aumentan los threads es mucho más lineal. A parte de esto, también se ve que el overhead por thread casi no varía según el número de threads, esto nos indica que separar las tareas no influye en la sincronización individual de cada thread pero sí en el cómputo global.