Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Lab 2: Brief tutorial on OpenMP programming model

Izan Cordobilla Blanco (par1304)

Alejandro Salvat Navarro (par1120)

Grau en enginyeria informàtica

Quadrimestre de primavera 2021-2022

ÍNDICE

1. INTRODUCCIÓN

En estas dos sesiones de laboratorio aprenderemos las principales estructuras en las extensiones de OpenMP del lenguaje C de programación. Veremos una serie de códigos donde se computa el valor de Pi en paralelo y algunos de éstos no serán correctos. Tendremos que aprender y entender cómo funciona la librería de OpenMP y como necesitamos estructurar el código para conseguir un buen paralelismo. Además, parte de este laboratorio consistirá en observar los overheads producidos.

Nos vamos a introducir a la librería de OpenMP a través de programas de ejemplo y aprenderemos las reglas necesarias que deben seguirse para conseguir un paralelismo real.

2. OpenMP QUESTIONNAIRE

2.1. 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"?

Si ejecutamos el programa con ./1.hello podemos ver dos veces el mensaje de “Hello World!”. Esto es debido a que Boada está restringido, de manera predeterminada, a usar 2 threads de hardware para realizar sus funciones. El programa lo que hace es enviar la función printf(“Hello World!”) a cada thread con el que se ejecuta.

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

Podemos hacer que el programa imprima 4 veces por pantalla el mensaje “Hello World!” sin necesidad de cambiar el código, ya que este paraleliza la función de printf a cada thread.
Por lo tanto, lo único que tendremos que hacer será especificar que queremos ejecutar el archivo con 4 threads, con el comando:

OMP_NUM_THREADS=4 ./1.hello

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 ya que los identificadores de los threads no siempre salen iguales en una misma línea. Esto es debido a que hay *datarace*. Por lo tanto, para conseguir esto, podemos proteger la variable id con un private, donde la línea del pragma quedaría así:

#pragma omp parallel private(id) num_threads(8)

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 siempre aparecen en el mismo orden y los mensajes aparecen mezclados ya que al realizarse las ejecuciones en los distintos threads de forma paralela, no se puede saber con exactitud el orden de finalización de estas. Además, en la misma línea no deberían salir mensajes mezclados de diferentes 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?

```
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 threa
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: resultado de ejecutar 3.how_many

Cuando se invoca fuera de la zona paralela (el mensaje que se ejecuta fuera de la zona paralela es el que empieza por “Outside parallel…”) devuelve 1 thread (ya que está fuera de la zona paralela).

Cuando se invoca dentro de la región paralela devuelve el número de threads que se están utilizando paralelamente.

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

1. Podemos cambiar el número de threads poniendo al inicio del código la línea: #omp_set_num_threads(x).
2. Podemos usar #pragma omp num_threads(x) donde x es el número de threads que queremos usar.

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

La primera definición del número de threads a usar se hace sin modificar el código porque lo hacemos en la terminal cuando ejecutamos el programa. La segunda se hace dentro del programa, en el área que queremos paralelizar.

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)

Al ejecutar ./4.data_sharing obtenemos lo siguiente:

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

Para el caso de shared, todos los threads leen y escriben sobre la misma x. Obtenemos el 120 sumando los números de los threads (0 + 1 + … + 15).

En el caso de private y firstprivate, el resultado es el mismo ya que dentro de la región paralela todos los threads escriben sobre una copia local y, por tanto, el resultado es el que había definido fuera de la región paralela (5).

Por último, en el caso de la reducción, cogemos el valor que había definido fuera de la región paralela (5) y aplicamos la reducción (en total sería 120 + 5 = 125).

5.datarace.c

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

El programa no siempre devuelve el resultado correcto porque depende del número de threads que éste use. Ésto es debido a que el loop se ejecuta de manera diferente cuando los threads varían y por lo tanto el resultado también cambia.

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.

Podríamos usar dos soluciones:

- Meter dentro del loop un #pragma omp critical

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

Figura 2: Solución 1

- Escribiendo antes del if del loop la línea #pragma omp atomic

```
omp_set_num_threads(8);
#pragma omp parallel 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: Solución 2

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.

La solución sería dividir el vector por el número de threads que estemos usando y paralelizando el bucle.

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

Figura 4: Solución propuesta

6.datarace.c

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

Este programa no devolverá siempre el resultado correcto ya que el bucle no es correcto.

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.

Podemos usar dos soluciones para obtener el resultado correcto:

- La primera alternativa es implementar un taskgroup que incluya todo el bucle for, junto a un reduction de la variable countmax, tal y como muestra la siguiente figura:

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

Figura 5: Alternativa 1

- La segunda alternativa es introducir #pragma omp atomic en el interior del bucle tal y como muestra la siguiente figura:

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

Figura 6: Alternativa 2

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)

Este programa no se está ejecutando correctamente ya que hay datarace y algunos valores se están utilizando más de una vez con diferentes valores.

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

Podemos resolverlo añadiendo un vector global que guarde cuantas veces aparece cada valor máximo para todos los threads. Una vez ejecutados los bucles, le asignamos a la variable countmax el valor de dicho vector en la posición maxvalue. Todos estos cambios se pueden visualizar en la siguiente figura:

```
int main()
{
    int i, maxvalue=0;
    int countmax = 0;
    int veccont[N] = {};

    omp_set_num_threads(8);
    #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) {
            #pragma omp atomic
            veccont[maxvalue]++;
        }
        if (vector[i] > maxvalue) {
            maxvalue = vector[i];
            #pragma omp atomic
            veccont[maxvalue]++;
        }
    }

    }
    countmax = veccont[maxvalue];

    if ((maxvalue==15) && (countmax==3))
         printf("Program executed correctly - maxvalue=%d found %d times\n", maxvalue, countmax);
    else printf("Sorry, something went wrong - maxvalue=%d found %d times\n", maxvalue, countmax);

    return 0;
}
```

Figura 7: Modificación del código

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?

No podemos predecir la secuencia de printf's en este programa pero podemos predecir que los mensajes de "going to sleep" van a ser impresos por pantalla antes de los de despertarse. Además, podemos predecir que los mensajes de estar despiertos van a ser impresos antes que los mensajes de cruzar la *barrier*. También, se puede predecir el orden de los threads despertándose (ya que duermen más a medida que su identificador es más grande).

Lo que no podemos predecir es el orden de los mensajes de irse a dormir y el orden de los threads cruzando la barrera, porque éstos no cruzan la *barrier* de una forma específica.

2.2. Day 2: explicit tasks

1.single.c

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

Por defecto, la cláusula single hace que solo 1 thread genere la tarea y el resto espere hasta su terminación. La cláusula nowait asociada a single elimina esa espera, permitiendo a los demás threads trabajar mientras se lleva a cabo la tarea generada por el single paralelamente.

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?

Esto se debe a la eliminación de la espera de la cláusula single. Ahora diferentes threads estarán ejecutando diferentes tareas de forma paralela, por lo que acabarán todas prácticamente en el mismo instante de tiempo, pero sin ningún orden garantizado.

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?

No se usa pragma omp parallel, por lo tanto, el programa es ejecutado todo por un mismo thread

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

Lo que deberemos hacer para esto será, dentro del bucle del while, antes de ejecutar el cuerpo del mismo, añadir las líneas:

#pragma omp parallel

#pragma omp single

De manera que el codigo quedaria como en la figura:

```
p = init_list(N);
head = p;


while (p != NULL) {
  printf("Thread %d creating task that will compute %d\n", omp_get_thread_num(), p->data);
  #pragma omp parallel
  #pragma omp single
  #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");
```

De esta manera, paralelizamos las diferentes iteraciones de dicho bucle, como podemos ver en consola al ejecutar el programa con el cambio realizado, donde los números de fibonacci tienen diversos threads asociados.

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

firstprivate(p) hace que todos los threads contengan su propia copia de la variable. Por lo tanto, al borrar el *firstprivate*, el programa se ejecutará solo con 1 thread, aquel que contiene la variable con la que trabajamos.

3.taskloop.c

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

En la siguiente figura podemos ver la ejecución de las distintas iteraciones de cada bucle, junto al número de thread que la ejecuta (número entre paréntesis).

```
par1120@boada-1:~/lab2/openmp/Day2$ ./3.taskloop
Thread 0 distributing 12 iterations with grainsize(4) ...
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: (2) gets iteration 4
Loop 1: (2) gets iteration 5
Loop 1: (2) gets iteration 6
Loop 1: (2) gets iteration 7
Thread 0 distributing 12 iterations with num_tasks(4) ...
Loop 2: (2) gets iteration 0
Loop 2: (2) gets iteration 1
Loop 2: (2) gets iteration 2
Loop 2: (0) gets iteration 9
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
Loop 2: (2) gets iteration 3
Loop 2: (2) gets iteration 4
Loop 2: (2) gets iteration 5
Loop 2: (1) gets iteration 8
```

Figura 8: Resultado de ejecutar 3.taskloop

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?

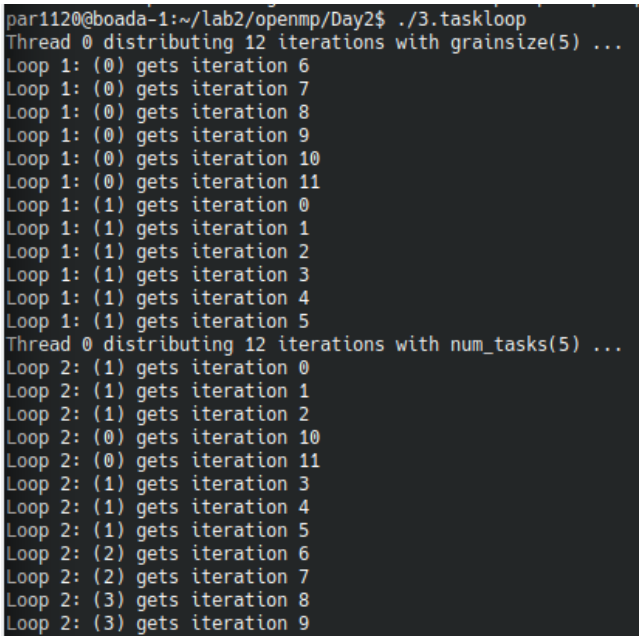

Figura 9: Resultado de ejecutar 3.taskloop con grainsize y numtasks = 5

Cada thread ejecuta 5 iteraciones, después, se ejecutan las restantes. Al cambiar el grainsize a 5 se observa que todas las iteraciones se ejecutan de manera consecutiva, cada grupo por un thread concreto, ya que grainsize influye en el número de iteraciones seguidas por cada tarea ejecutada. Por otra parte, vemos que al modificar num_tasks, las ejecuciones de los threads aparecen desordenadas en consola. Esto es porque, como su nombre indica, num_tasks aumenta el número de tareas, por lo que, al ser paralelizables, no se garantiza ningun orden de terminación

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

No se puede, ya que en caso de hacerlo, ocurre un error.

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

nogroup provoca la eliminación de la definición implícita de región del taskloop, por lo que provoca un override en el taskloop, haciendo que grainsize y num_tasks se ejecuten a la vez, como se puede ver en la imagen

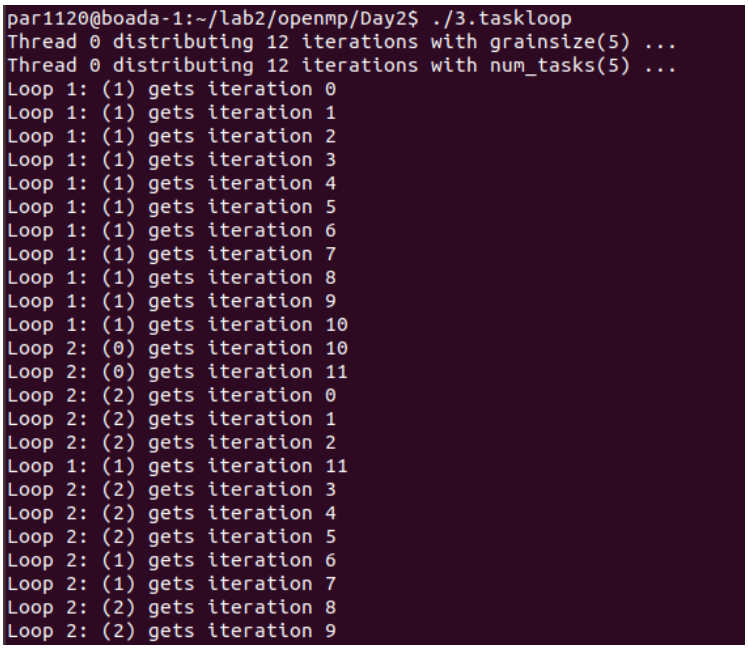

Figura 10: Ejecución 3.taskloop

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.

Para retornar el valor correcto de la variable sum en cada parte, usaremos:

#pragma omp taskgroup task_reduction(+: sum)

y

#pragma omp taskgroup task_reduction(+: sum)

En diferentes partes del código, de la siguiente manera:

```
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)
    #pragma omp task 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)
        {
    for (i=0; i< SIZE/2; i++)
        #pragma omp task firstprivate(i)
        sum += X[i];

    #pragma omp taskloop grainsize(BS)
    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: Programa modificado de 4.reduction.c

Una vez modificado el código como se muestra en la imagen, se obtendrá el mismo valor de sum en las tres diferentes partes (como se puede ver en la siguiente imagen), indicándonos así que hemos implementado correctamente la paralelización.

```
par1120@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 12: Resultado de la ejecución

5.synchtasks.c

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

```
par1120@boada-1:~/lab2/openmp/Day2$ ./5.synchtasks
Creating task foo1
Creating task foo2
Creating task foo3
Creating task foo4
Starting function foo1
Creating task foo5
Starting function foo3
Terminating function foo1
Starting function foo2
Terminating function foo2
Starting function foo4
Terminating function foo3
Terminating function foo4
Starting function foo5
Terminating function foo5
```

Figura 13: Resultado ejecución programa.

El TDG de programa quedaría como el de la siguiente imagen:

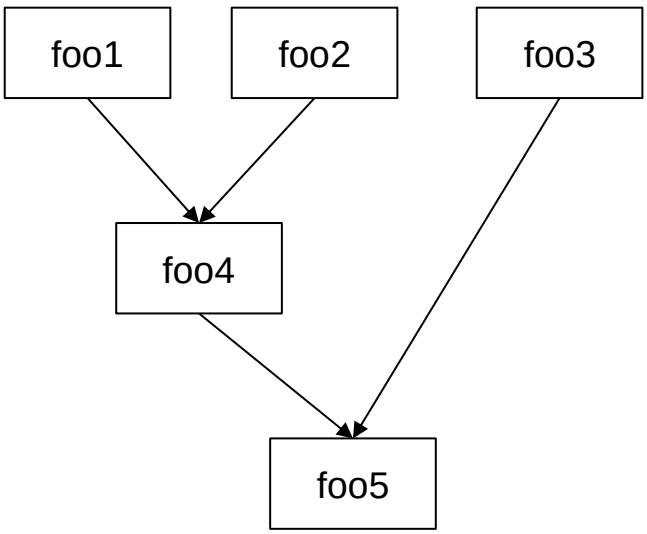

Figura 14: TDG de las tareas del programa

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.

Lo único que deberemos hacer será añadir taskwait en foo4 y foo5, de manera que se vean obligadas a esperar a que las anteriores tareas acaben, como se puede ver en la figura.

La paralelización no es exactamente igual que la original, ya que en este caso foo4 depende de foo3 y foo5 de foo4. Pero en la práctica, el rendimiento es similar al paralelismo original.

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

Figura 15: Modificación programa usando taskwait

3. Rewrite the program using only taskgroup as task synchronisation mechanism (no depend clauses allowed), again try

La manera de paralelizar el programa con un paralelismo potencialmente parecido mediante taskgroups es la siguiente:

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

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

Figura 16: Modificación programa usando taskgroup

Mediante el primer taskgroup, agrupamos las 3 primeras tareas en un mismo conjunto, de manera que tengan que terminar para poder ejecutarse foo4 y foo5. Una vez acabado, comenzará el segundo taskloop, que solo contendrá la foo4, de manera que foo5 se vea obligada a esperar a su terminación para ejecutarse.

3. OBSERVING OVERHEADS

En esta sección estudiaremos y veremos el comportamiento de los overheads a medida que cambiamos el número de threads, cambiamos cláusulas de OpenMP. etc.

Para empezar, vamos a hacer un breve resumen de los programas que vamos a utilizar:

- pi omp critical.c: es éste programa una región crítica es usada para proteger todos los accesos a la variable *sum*, asegurando acceso exclusivo.
- pi omp atomic.c: usa atomic para asegurar un acceso atómico (indivisible) a la memory location donde está almacenada la variable *sum*.
- pi omp sumlocal.c: para cada thread se crea una copia privada llamada *sumLocal* y después, ésta se suma a la variable global *sum* que se encuentra dentro de una región crítica.
- pi omp reduction.c: utiliza la cláusula *reduction* aplicada a la variable global *sum.*

Cabe destacar que la versión secuencial del programa “pi sequential.c” tiene un tiempo de ejecución de **1.7942 segundos** ejecutándose con 100000000 iteraciones.

Compilaremos los 4 programas resumidos anteriormente y los ejecutaremos usando el script **submit-omp-sh** con 1, 4 y 8 threads. Además, usaremos siempre 100000000 iteraciones y ejecutaremos el programa 3 veces para obtener un resultado mucho más preciso. A continuación se muestran las tablas y los gráficos de los tiempos de ejecución además de una pequeña reflexión de los resultados obtenidos.

1 Thread

	1rst try	2nd try	3rd try	Total
Critical	2,724	2,474	2,474	2,557
Atomic	0,005	0,006	0,005	0,005
SumLocal	0,004	0,004	0,004	0,005
Reduction	0,004	0,002	0,004	0,003

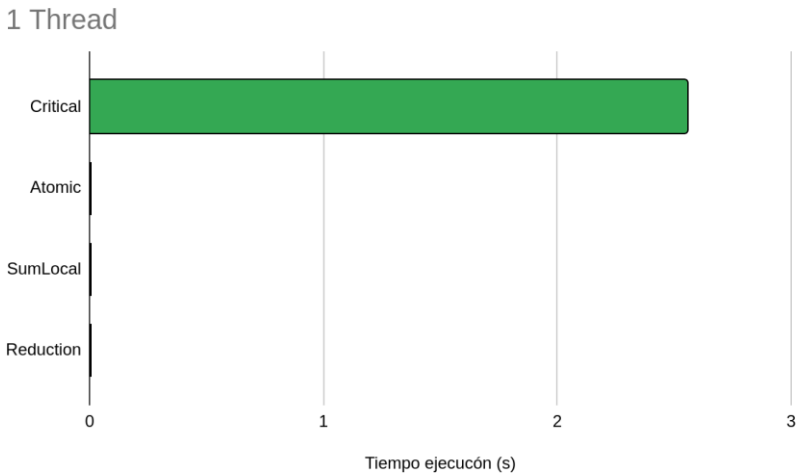

Figura 17: Gráfico de tiempo de los diferentes programas ejecutados con 1 thread

4 threads

	1rst try	2nd try	3rd try	Total
Critical	37,38	36,16	37,64	37,06
Atomic	5,23	4,96	5,31	5,17
SumLocal	0,01	0,01	0,01	0,01
Reduction	0,01	0,01	0,01	0,01

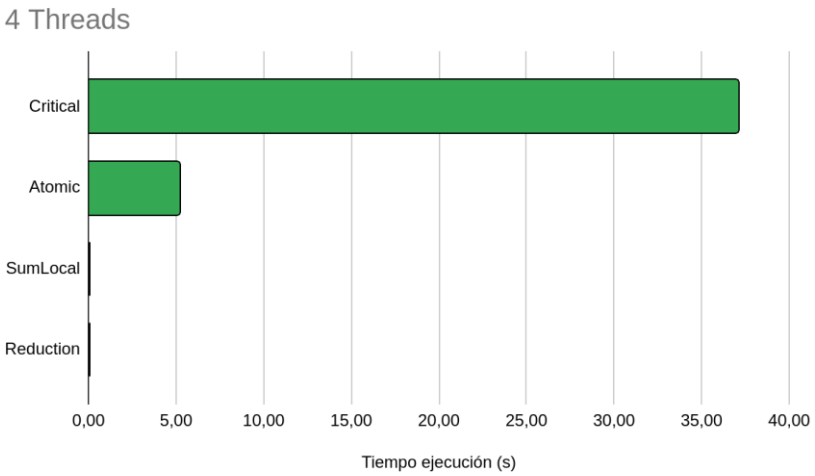

Figura 18: Gráfico de tiempo de los diferentes programas ejecutados con 4 threads.

8 Threads

	1rst try	2nd try	3rd try	Total
Critical	34,009	33,853	32,439	33,434
Atomic	5,417	5,792	5,762	5,657
SumLocal	0,020	0,020	0,018	0,019
Reduction	0,020	0,020	0,020	0,020

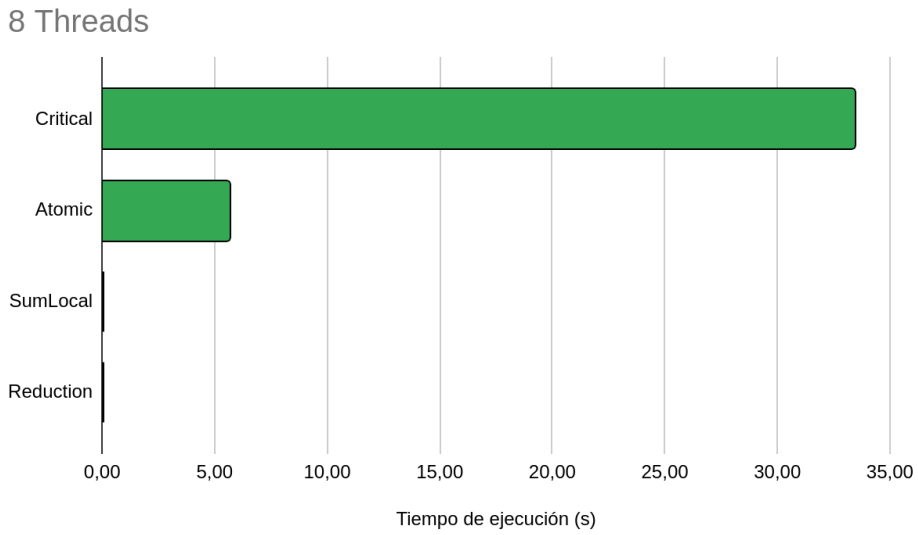

Figura 19: Gráfico de tiempo de los diferentes programas ejecutados con 8 threads.

Observando todas las tablas y gráficos, nos damos cuenta que la versión que tarda más en ejecutarse siempre es la que utiliza *critical.* También podemos ver que con 4 y 8 threads, la versión *atomic* tarda bastante en comparación las versiones de *sumLocal* y *Reduction.* Esto es así ya que las versiones de *critical* y *atomic* garantizan un acceso seguro a las posiciones de memoria correspondientes mediante hacer que otros threads esperen su turno. Además, luego éstos se tienen que sincronizar.

3.1. Thread creation and termination

En este gráfico podemos observar el crecimiento de los overheads y de los overheads/threads del programa pi_omp_parallel.c. Para obtenerlo, hemos tenido que ejecutar el comando "# sbatch ./submit-omp.sh pi omp parallel 1 24". es decir, utilizamos 24 threads para ejecutar una iteración.

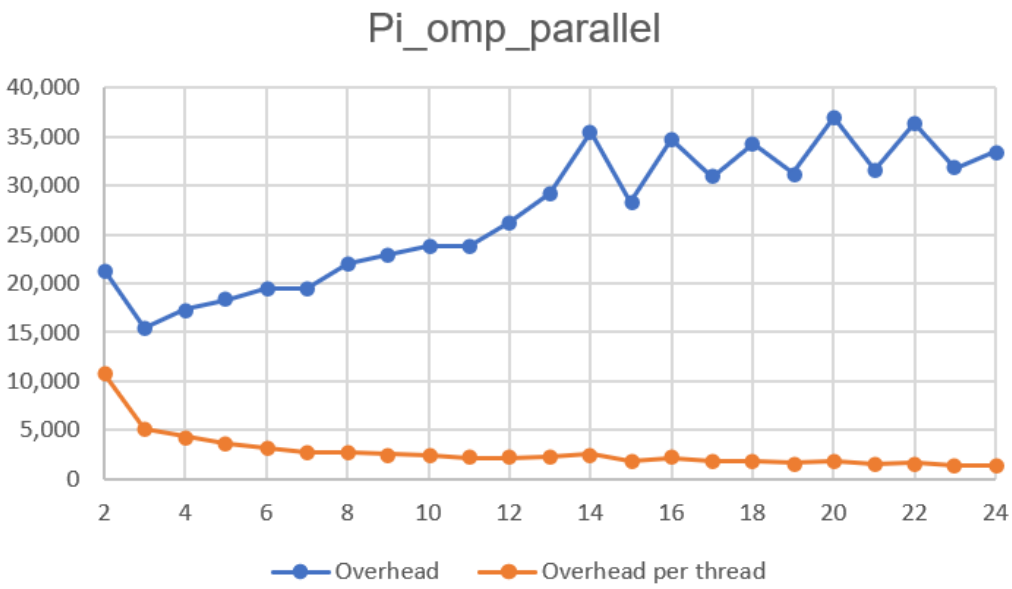

Figura 20: Gráfico de tiempo (en ns) de overheads y overheads/thread con diferentes threads.

En él se representa el crecimiento de los overheads con la línea azul. Fácilmente podemos ver que ésta crece a medida que los threads crecen también, aunque el crecimiento no es lineal. Era de esperar ya que es normal que a medida que aumentan los threads también lo hagan los overheads.

Por otra parte, la línea naranja simboliza los overheads/thread. Ésta, en contra, tiene una tendencia a decrecer a medida que crecen los threads. Podemos ver que el overhead/thread más grande se encuentra donde el programa se ejecuta con menos threads.

Para resumir este gráfico podríamos decir que aunque a medida que los threads crecen el overhead crece también, los overheads/thread decrecen.

3.2. Task creation and synchronisation

En este gráfico podemos observar el crecimiento de los overheads y de los overheads/threads del programa pi_omp_tasks.c. Para obtenerlo, hemos tenido que ejecutar el comando "# sbatch ./submit-omp.sh pi omp tasks 10 1". Es decir, utilizamos un único threads para para ejecutar 10 iteraciones

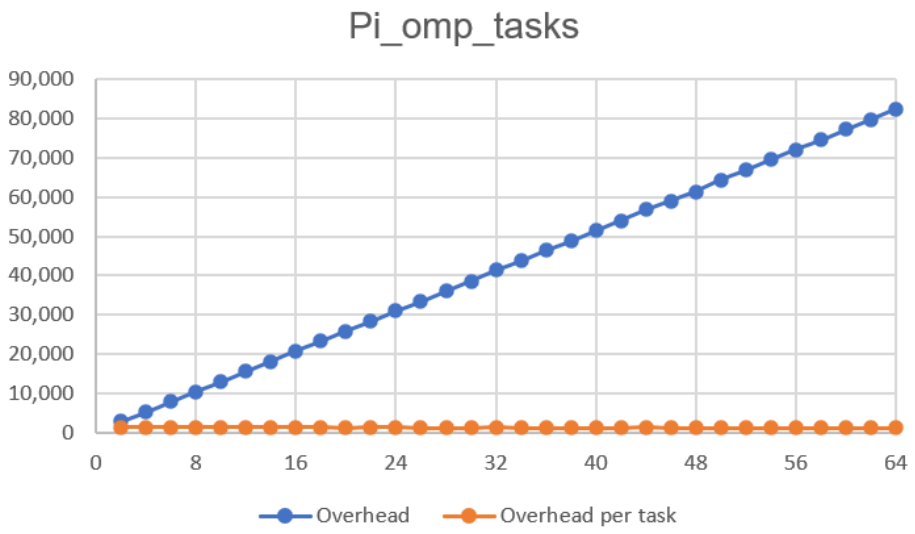

Figura 21: Gráfico de tiempo (en ns) de overheads y overheads/task con diferentes tasks.

En él se representan los overheads a través de la línea azul. Podemos ver que los overheads tienen un crecimiento prácticamente lineal a medida que aumentamos las iteraciones. El overhead inicial con una única iteración es prácticamente 0 y escala hasta 8, en el máximo de iteraciones (10).

Por otra parte, los overheads/thread están representados con la línea naranja y son prácticamente iguales entre ellos.

4. CONCLUSIONES

En este laboratorio hemos aprendido a usar diferentes pragmas de OpenMP para la paralelización de procesos, además, también hemos descubierto diferentes problemas con los que nos podemos encontrar a la hora de intentar implementar dichos pragmas de paralelización.

En la sesión 1 nos hemos enfocado en el funcionamiento de OpenMP y todas sus posibilidades, estudiando las características de cada uno de los pragmas disponibles en la librería y sus funcionalidades

En la sesión 2 nos hemos centrado más en diversos posibles problemas que pueden surgir al paralelizar un programa, viendo que el paso de la teoría al mundo real es una tarea que puede complicarse si no se va con cuidado con características tales como los overheads, la sincronización de los procesos, la generación de tareas, entre otras.

En resumen, hemos aprendido a usar OpenMP y a solucionar posibles problemas que nos encontremos más adelante, de manera que podamos estar preparados para crear nuestros propios programas paralelos