Lab2: Brief tutorial on OpenMP programming model
Q2 2021-22

Integrantes:
Pol Pérez Castillo - par2116
Cristian Sánchez Estapé - par2118

Fecha de entrega: 22-03-2022

Índice

Introducción

En este laboratorio se trabaja un primer acercamiento al modelo de programación de OpenMP, en concreto la declaración de regiones paralelas y tareas implícitas y explícitas. También se estudian los overheads de sincronización, creación y finalización de tareas que se producen al insertar estas funcionalidades de OpenMP.

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"?**
   El mensaje se mostrará 2 veces, puesto que la directriz #pragma omp parallel, si no se le especifica el número de threads con que ejecutar un programa, por defecto lo hará con 2.

2. **Without changing the program, how to make it to print 4 times the "Hello World!" message?**
   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 dado que no se fuerza ningún orden. La cláusula necesaria para garantizar la ejecución en orden correcto es #pragma omp critical {...}.
2. **Are the lines always printed in the same order? Why the messages sometimes appear intermixed? (Execute several times in order to see this).**
   No, dado que esto ya depende de los recursos que le son asignados a cada thread por parte del sistema operativo, y por lo tanto, cada thread llegará en momentos diferentes en diferentes ejecuciones a la cláusula *critical*.

3.how many.c

1. **What does omp get num threads return when invoked outside and inside a parallel region?**
   Fuera de la región paralela, siempre devuelve 1; dentro de una región paralela, devuelve el número de threads especificado en la región (8 si es por defecto, o el número que venga definido en *#pragma … num_threads(n)*). Cabe destacar que, cuando se hace uso de omp_set_num_threads(i), se modifica el número de threads con que se ejecuta por defecto.
2. **Indicate the two alternatives to supersede the number of threads that is specified by the OMP NUM THREADS environment variable.**
   Una es la instrucción *omp_set_num_threads(i)*, que cambia el número de threads a aquél especificado por la instrucción, y la otra es la que viene

indicada en la cláusula *#pragma omp parallel num_threads(i)*, que tiene el mismo efecto pero solo para la región afectada por el pragma.

3. **Which is the life span for each way of defining the number of threads to be used?**
   Con la cláusula *#pragma omp parallel num_threads(i)* solo afecta a la región del pragma. Con la instrucción *OMP NUM THREADS* y *omp_set_num_threads(i)* se define por defecto el número de threads que se aplica a las regiones paralelas del programa para toda su ejecución a no ser que se especifique lo contrario con la cláusula pragma.

4.data sharing.c

1. **Which is the value of variable x after the execution of each parallel region with different datasharing attribute (shared, private, firstprivate and reduction)? Is that the value you would expect? (Execute several times if necessary)**
   En *shared* el valor de la x és 120. El valor es incorrectamente correcto, puesto que la cláusula shared implica que dos (o más) threads pueden leer o escribir de X en un mismo instante de tiempo, con lo cual la suma se realizaría de manera errónea y este mismo error se arrastraría durante la ejecución.
   En *private* el valor de la x és 5. Es correcto dado que el valor asignado previo a la región paralela se debe mantener, puesto que dichas regiones, al tratar la variable *x* de manera privada, se limitan a hacer copias de *x* para trabajar con ellas.
   En *firstprivate* el valor de la x és 5. También es correcto, dado que, a efectos prácticos, aplica lo mismo que se ha comentado para la *private*.
   Tanto en el caso *private* como en *firstprivate* el resultado es 5 porque el printf se ejecuta fuera de la región paralela.
   En *reduction* el valor de la x és 125. Es correcto dado que *reduction* crea variables privadas temporales que acumulan los resultados de las sumas y, posteriormente, al llegar al fin de la región paralela se terminan sumando todas en la variable indicada (en este caso *x*).

5.datarace.c

1. **Should this program always return a correct result? Reason either your positive or negative answer.**
   No siempre devolverá el resultado correcto dado que, como *maxvalue* es compartida, es posible que se dé aquella situación en que dos threads escriben en *maxvalue* al mismo tiempo y, por lo tanto, no se llegue necesariamente al valor deseado.
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.**
   Por un lado, una alternativa sería incorporar la directriz *reduction(max:maxvalue)* en la cláusula *#pragma omp parallel* inicial. Con esto, se harían uso de variables locales que almacenarían el máximo local y, posteriormente, seleccionaría la más grande entre ellas.
   Por otro lado, otra alternativa sería hacer uso de una variable local *maxvaluelocal* que, siendo única para cada región paralela, encontraría el

máximo de dicha región y, posteriormente, se compararía con las demás, permitiendo así su correcta selección.

```c
int main()
{
    int i, maxvalue=0;

    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)
           maxvalue = vector[i];
    }
    }

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

    return 0;
}
```

Figura 1: primera modificación de 5.datarace.c (añadir la directriz reduction)

```c
int main()
{
    int i, maxvalue=0;

    omp_set_num_threads(8);
    #pragma omp parallel private(i)
    {
    int id = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int maxvaluelocal = 0;

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

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

    return 0;
}
```

Figura 2: segunda modificación de 5.datarace.c (uso de una variable local para encontrar atómica y manualmente el máximo)

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 distribución alternativa aquí planteada es que cada thread recorra el vector de 3 en 3 posiciones consecutivas. Cabe destacar que esta distribución debe tener en cuenta, pues, que en el caso del thread 7 se

pueda tratar de acceder a una posición fuera del vector, error que queda solventado si se incluye dicha casuística en el *for*.

```
int main()
{
    int i, maxvalue=0;

    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=3*id; i < 3*(id+1) && i < N; i++) {
        if (vector[i] > maxvalue)
           maxvalue = vector[i];
    }
    }

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

    return 0;
}
```

Figura 3: tercera modificación de 5.datarace.c

6.datarace.c

1. **Should this program always return a correct result? Reason either your positive or negative answer.**
   No siempre devolverá el resultado correcto, dado que *countmax* sigue siendo compartida, con lo cual se puede dar aquella situación en que dos threads escriban en *countmax* al mismo tiempo y, por lo tanto, no se llegue necesariamente al valor deseado.
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.**
   Por un lado, podemos hacer uso de la directriz *reduction(+:countmax)* en la cláusula #pragma omp parallel …, con lo cual se realizarán las sumas pertinentes privadas, y que finalmente serán acumuladas en la variable *countmax*.
   Por otro lado, podemos hacer uso de la cláusula *#pragma omp atomic*, la cual garantiza que las sumas *countmax*++ no se solapen y, por lo tanto, nos salven de leer y escribir a la vez en la misma variable.

```c
int main()
{
    int i, countmax = 0;
    int maxvalue = 15;

    omp_set_num_threads(8);
    #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++;
        }
    }

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

    return 0;
}
```

Figura 4: primera modificación de 6.datarace.c (añadir la directriz reduction)

```c
int main()
{
    int i, countmax = 0;
    int maxvalue = 15;

    omp_set_num_threads(8);
    #pragma omp parallel private(i)
    {
    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++;
        }
    }

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

    return 0;
}
```

Figura 5: segunda modificación de 6.datarace.c (añadir la cláusula atomic)

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)**

   No, dado lo siguiente: al hacer uso de la directriz *reduction(max:maxvalue)*, la variable *maxvalue* se vuelve privada, con lo cual los threads, al recorrer el vector, van encontrando máximos *locales*, con lo cual, cuando uno de ellos encuentra el máximo global, los demás threads no son informados de ello, lo cual lleva a que, al final de la región paralela, se elija correctamente el 15 como máximo, pero el *countmax* sea la suma de las apariciones de los máximos *locales* según las regiones recorridas por los threads.

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

```
int main()
{
    int i, maxvalue=0;
    int countmax = 0;

    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) 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++;
    }
    }

    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 6: modificación de 7.datarace.c

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 se puede predecir en qué orden aparecerán las impresiones: el primero se realizará en un orden cualquiera, pero se realizará conjuntamente; posteriormente, se entrará en un sleep, después del cual se realizará una segunda impresión (también en un orden indeterminado), y finalmente los threads se esperaran en el *#pragma omp barrier,* imprimiendo así el último mensaje de manera desordenada.

- Day 2: explicit tasks

1.single.c

1. **What is the nowait clause doing when associated to single?**
   Evitar que los demás threads se esperen al final de la cláusula.
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?**
   La presencia del sleep garantiza que todos los threads realicen parte de la ejecución a la vez. Lo mismo aplica a esta *ejecución en ráfagas*.

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 hace uso de la cláusula *#pragma omp parallel*, con lo cual nunca se llega a paralelizar el programa (motivo por el cual el posterior *#pragma omp task* no aplique).
2. **Modify the code so that tasks are executed in parallel and each iteration of the while loop is executed only once.**

```
int main(int argc, char *argv[]) {
     struct node *temp, *head;

     omp_set_num_threads(6);
     printf("Staring computation of Fibonacci for numbers in linked list \n");

     p = init_list(N);
     head = p;

     {
     while (p != NULL) {
        #pragma omp parallel
        #pragma omp single
          {
            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");

     printf("Finished computation of Fibonacci for numbers in linked list \n");
     p = head;
     while (p != NULL) {
        printf("%d: %d computed by thread %d \n", p->data, p->fibdata, p->threadnum);
        temp = p->next;
        free (p);
        p = temp;
     }
     free (p);

     return 0;
}
```

Figura 7: modificación de 2.fibtasks.c

3. **What is the firstprivate(p) clause doing? Comment it and execute again. What is happening with the execution? Why?**
   Firstprivate asegura que, por cada thread que (dentro de la región paralela) ejecute las instrucciones contenidas en la cláusula *firsprivate*, se tendrá una copia de *p* y esta se encontrará inicializada según el valor que tuviera antes de llegar a dicha cláusula. En nuestro caso, no sucede nada ni se dan

problemas durante la ejecución, dado que el *single* garantiza que la región paralelizada solo se ejecutará por un único thread, con lo cual, por mucho que *p* no sea privada, no habrá conflictos entre threads.

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 bucle, a cada thread le corresponden 4 iteraciones (dado que, debido a la cláusula *grainsize*, cada thread hará *VALUE*, es decir, 4 iteraciones); en cambio, en el segundo bucle, cada thread le corresponden 3 iteraciones, dado que se crearán *VALUE* tareas, con lo cual a cada thread le corresponderan 12/4 = 3 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?**
   Con la modificación, cada thread ejecuta 6 iteraciones. Esto se debe a que la cláusula *#pragma omp taskloop grainsize(VALUE)* establece un rango que va desde [VALUE, 2*VALUE), con lo cual trata de encontrar la manera más óptima de repartir las tareas de dentro de la región entre los threads existentes. Dado que trabajamos con más de 2 threads, la paralelización se realiza de la manera más óptima, hecho que se consigue tratando de distribuir equitativamente entre los threads existentes.
   En cuanto a la manera de distribuir las iteraciones entre los distintos threads, hacemos referencia a la descripción realizada en el apartado anterior: *grainsize* distribuye *VALUE* iteraciones por cada thread (teniendo en cuenta la existencia implícita del rango que acabamos de mencionar).
3. **Can grainsize and num tasks be used at the same time in the same loop?**
   La cláusula *grainsize* y la cláusula *num_tasks* se excluyen mutuamente y no es posible que aparezcan en la misma directiva de taskloop.
4. **What is happening with the execution of tasks if the nogroup clause is uncommented in the first loop? Why?**
   Las ejecuciones de los dos bucles se solapan y, por lo tanto, aparecen entremezcladas. Sin la directiva *nogroup*, la cláusula *taskloop grainsize* se encargaría de garantizar que, si bien la ejecución de dentro de dicha región fuera paralela, el programa se esperase hasta que dicha región hubiera terminado de ejecutar todas sus tareas antes de continuar con el mismo.

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
        sum = 0;
        #pragma omp taskloop grainsize(BS)
        for (i=0; i< SIZE; i++)
        #pragma omp atomic
            sum += X[i];

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

        // Part III
        sum = 0;
        #pragma omp task private(i)
        for (i=0; i< SIZE/2; i++)
        #pragma omp atomic
            sum += X[i];

        #pragma omp taskloop grainsize(BS)
        for (i=SIZE/2; i< SIZE; i++)
        #pragma omp atomic
            sum += X[i];

        printf("Value of sum after reduction in combined task and taskloop = %d\n", sum);
    }

    return 0;
}
```

Figura 8: modificación de 4.reduction.c

5.synchtasks.c

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

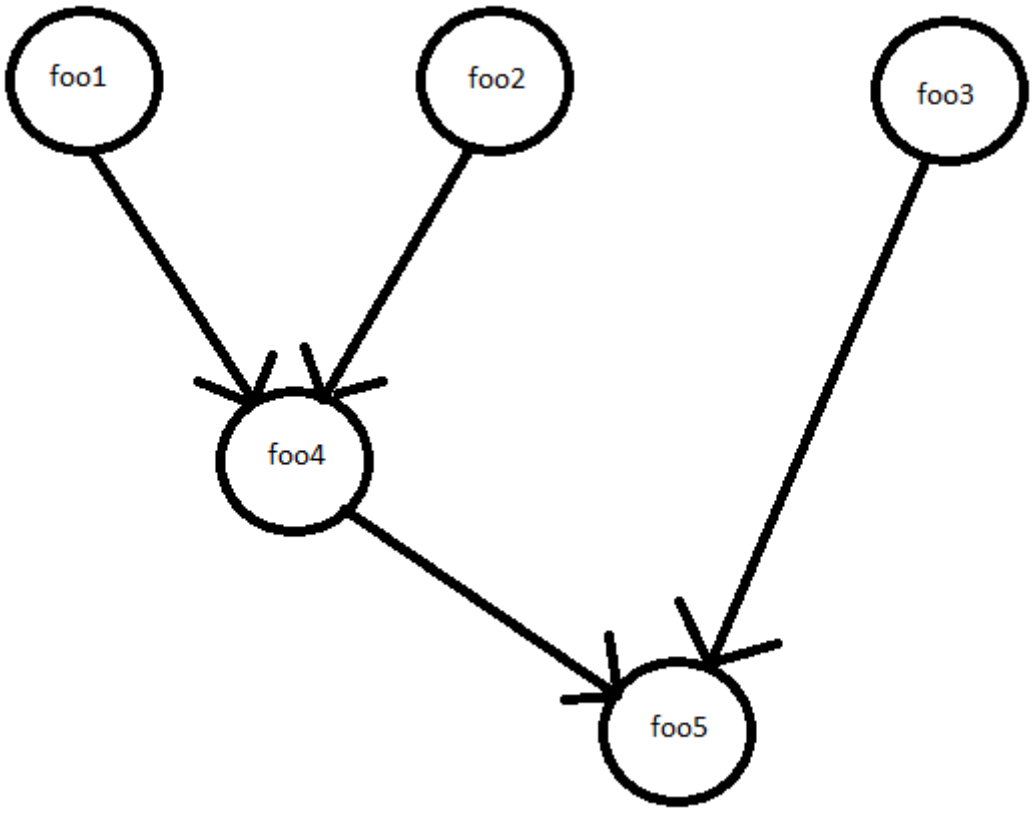
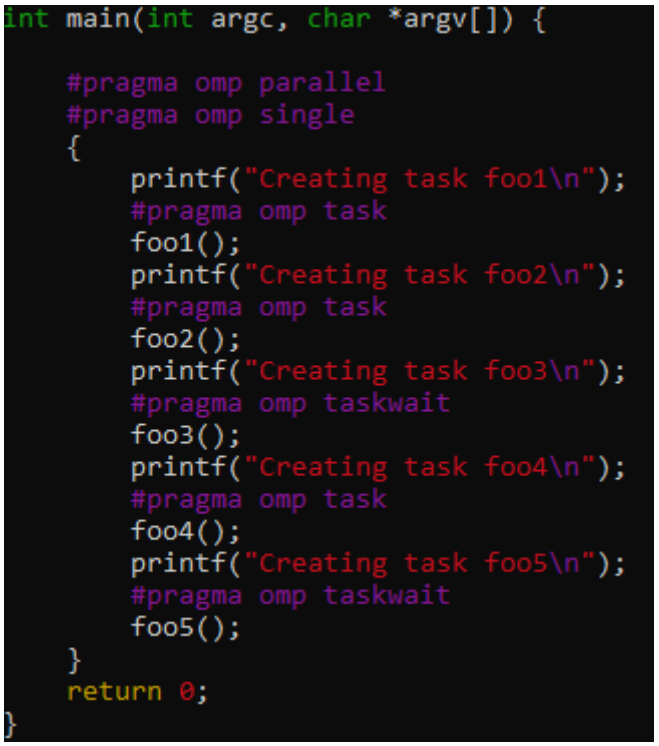


Figura 9: grafo de dependencias del programa 5.synchtasks

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

Figura 10: modificaciones con la directiva "taskwait" del programa 5.synchtasks

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

Figura 11: modificaciones con la cláusula "taskgroup" del programa 5.synchtasks

Observing overheads

- Day 1: synchronisation 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?

Critical : 100 millones
Atomic: 100 millones
Critical y *atomic* crean este número de operaciones de sincronización debido a que su respectiva cláusula está definida dentro del bucle interno el cual se ejecuta tantas veces como tamaño de le definamos con el *sumbit-omp*, en el caso de este ejercicio 100 millones de veces.
Sumlocal: Numero de threads
Reduction: Numero de threads
Sumlocal y *reduction* crean exclusivamente número de threads operaciones de sincronización porque son el número de variables temporales que se crean entre todos los threads para almacenar la suma temporal y luego poder agruparla en la variable principal *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.**
   La única versión que se presenta con un overhead considerable es *pi_omp_critical*, lo cual encaja con el hecho de que, al usar la directriz *#pragma omp critical*, se asegura que dicha región se ejecute de manera secuencial y, además, en ello vienen implícitos los overheads de creación y cierre de dicha región. Además, dado que el programa realiza 100 millones de iteraciones, a cada iteración creará y cerrará dicha región, lo cual lleva a que el overhead mismo sea mayor incluso que el tiempo de ejecución secuencial del programa.
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?**
   No, hay diferencias notables entre las versiones y el “beneficio” que saca cada una de ellas de usar un mayor número de threads y, de hecho, se puede afirmar que no existe beneficio alguno. La razón por la cual esto sucedería varía entre conjuntos de versiones: por un lado, *critical* y *atomic* fuerzan a que una región determinada sea ejecutada exclusivamente por un thread al mismo tiempo, con lo cual, al aumentar el número de threads, aumenta la probabilidad de que varios de ellos quieran realizar dicha operación y deban esperar a que otro acabe; por otro lado, en *sumlocal* y *reduction* la operación se realiza una vez terminadas las iteraciones, con lo cual, con 1 thread, se realizará una vez, pero al aumentar dicho número, la cantidad de operaciones aumenta, lo cual implica también la aparición de sus correspondientes overheads.

Take note of all the results that you obtain and reach your conclusions about the overheads associated with these OpenMP constructs. Can you quantify (in microseconds) the cost of each individual synchronisation operation (critical or atomic) that is used?

Versión	1 thread	4 threads	8 threads
Secuencial	$1,79 * 10^6 \mu s$		
Critical	$2800341\ \mu s$	$36475366,5\ \mu s$	$35696355\ \mu s$
Atomic	$3010\ \mu s$	$5052470,25\ \mu s$	$5880830\ \mu s$
Sumlocal	$5213\ \mu s$	$13004,5\ \mu s$	$19969,875\ \mu s$
Reduction	$4405\ \mu s$	$11933,75\ \mu s$	$19882\ \mu s$

Tabla 1: tiempo de overheads según la versión ejecutada[1]

Versión	4 threads	8 threads
Critical	$0,365\ \mu s$	$0,357\ \mu s$
Atomic	$0,051\ \mu s$	$0,059\ \mu s$
Sumlocal	$3251,13\ \mu s$	$2496,23\ \mu s$
Reduction	$2983,44\ \mu s$	$2485,25\ \mu s$

Tabla 2: tiempo por operación de sincronización según versión y número de threads

[1] En el caso de la versión secuencial, se trata de su tiempo de ejecución total.

- Day 2: thread creation/termination, task creation and synchronisation

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?
Globalmente, a mayor número de threads, menor es el overhead de creación/terminación de cada thread. Eso sí: a medida que se aumenta dicho número, el overhead por thread cada vez es más inestable, con lo cual se tienen más probabilidades de incurrir en ineficiencia durante la ejecución del programa.
En cuanto al orden de magnitud del overhead creación/terminación de cada thread, a excepción del primero (el cual tiene un orden de magnitud de 10^{-6}), los threads tienen overheads de un orden de magnitud de 10^{-7}.

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?
En este caso, el overhead de creación/sincronización de las tareas incrementa de manera lineal según el número de tareas creadas, mientras que el overhead por tarea se mantiene prácticamente estancado (ni aumenta ni disminuye de manera significativa). Además, salvo para el primer caso (en el cual el orden de magnitud es 10^{-8}), los threads tienen overheads de un orden de magnitud de 10^{-7}.

Conclusiones

Este laboratorio ha resultado útil para aprender a manejar las cláusulas *#pragma omp task* en relación a directivas tales como *taskwait*, *dependent(in/out: …)*, *critical/atomic*, etc. Además, ha sido necesario para tratar la relación existente entre este tipo de directivas (usadas para sincronización de tareas y para establecer correctamente un flujo de ejecución convenientemente deseado) y los *overheads* (o “penalizaciones” temporales) inherentes a estas.