PAR Laboratory Assignment 2

ÍNDICE

Lab 2: Brief tutorial on OpenMP programming model

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

El objetivo de esta sesión de laboratorio es familiarizarnos con la librería OpenMP, ver cómo funciona y ver que necesita el código para generar paralelismo. Este laboratorio se compone de 2 sesiones: en la primera nos familiarizaremos con las regiones paralelas y las tareas implícitas.

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"?
   Imprime "Hello world!" 2 veces, porque imprime tantos outputs como threads existan, en este caso 2, puesto que *#pragma omp parallel* reparte la ejecución del programa entre todos los threads disponibles.
2. Without changing the program, how to make it print 4 times the "Hello World!" message?
   Para que se ejecute 4 veces el mensaje tenemos que cambiar el número de threads para la ejecución. Lo haremos introduciendo el siguiente 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.
   No, la ejecución no es correcta ya que los threads están cambiando el valor de la variable *id* entre sus ejecuciones. La manera de solucionar este problema es añadiendo *private(id)* a *#pragma omp parallel*.
2. Are the lines always printed in the same order? Why do the messages sometimes appear intermixed? (Execute several times in order to see this).
   No. Como los threads no siguen un orden específico de ejecución, no se respeta el orden de los threads y en algunos casos un thread puede intercalar su mensaje entre los de otro. Esto es debido a que el *private(id)* provoca que ejecute un "Hello" y un "world!" y puede que tengamos situaciones donde después de un "Hello" se imprima otro "Hello", ya que no controlamos en qué orden ejecutan los threads. A continuación mostramos un output con esta situación:

```
(4) Hello (2) Hello (2) world!
(2) world!
(0) Hello (0) world!
(1) Hello (1) world!
(3) Hello (3) world!
(5) Hello (5) world!
(6) Hello (6) world!
(7) Hello (7) world!
```

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?

   Dentro de la parte paralela retorna el siguiente output:

```
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)!
```

   Y en la parte no paralela del código le pertenece la siguiente parte del output:

```
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)
```

   Empieza la ejecución con 1 thread ("*Starting, I'm alone …*") y seguidamente con *OMP_NUM_THREADS=8* cambia a 8 threads, comando que imprime 8 veces el mensaje del *first parallel*. El output *second parallel* son 4 mensajes ya que la instrucción va precedida de *#pragma omp parallel num_threads(4)* y el *third parallel*, por la misma razón que el primero, imprime su mensaje 8 veces también. En cambio, en el *fourth parallel*, el número de threads va determinado por la variable *i*

(*omp_set_num_threads(i)*) dentro de un de 2 hasta 3, es decir el mensaje se imprime 5 veces, 2 para 2 threads y 3 para 3 threads.
Saliendo de la parte paralela ("*Outside parallel, nobody else here …*"), el *fifth parallel* se imprime 4 veces porque va precedido de *#pragma omp parallel num_threads(4)* (donde solo queremos paralelizar para esta instrucción) y, finalmente, el *sixth parallel* donde, como no usamos *#pragma omp parallel*, ya se ejecuta de forma secuencial 3 veces, puesto que la variable *i* del for anterior termina con el valor 3.

2. Indicate the two alternatives to supersede the number of threads that is specified by the OMP NUM THREADS environment variable
   Podemos cambiar el número de threads con las funciones omp_set_num_threads(*num_threads*) o num_threads(*num_threads*), donde *num_threads* serían el número de procesadores que queremos.

3. Which is the lifespan for each way of defining the number of threads to be used?
   La duración de cada una de estas funciones es la siguiente:
   - omp_set_num_threads(*num_threads*) → Durante toda la ejecución o hasta que se vuelva a modificar el número de threads.
   - num_threads(*num_threads*) → Solo se aplica a la siguiente línea de código.

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)
   El siguiente output es un ejemplo de una de las ejecuciones que hemos realizado:

```
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
```

   Este muestra el valor de la variable *x* después de cada uno de los *parallels*. El primer resultado viene provocado por la suma de *x* por todos los threads que están siendo ejecutados, es decir 16, por lo tanto x es 120 (16*(16-1)/2 = 120). En el segundo *parallel*, la *x* es 5, ya que al poner la variable como privada, cada uno de los threads genera la variable local x (sin inicializar) y le suma su número id. Al acabar la operación las variables locales se destruyen y se imprime el valor de la x, que no ha sido modificada. El tercer resultado la x también es 5, por la misma razón que la anterior pero esta vez la variable viene inicializada con el primer valor de *x*. Finalmente, el cuarto *parallel* la *x* es 125 ya que anteriormente era 5 más los threads añaden su id a la variable *x* (16*(16-1)/2 = 120).
   Nos esperábamos este valor para el segundo y el tercer *parallel* (*private* y *firstprivate*), pero no del primero y el cuarto, ya que el valor va variando dependiendo del thread.

5.datarace.c

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

La ejecución no siempre es correcta porque como varios thread se ejecutan de forma simultánea no sabemos si se realizan todas las comparaciones necesarias entre los números de esta primera tanda, esto puede dar lugar a que quede guardado como maxvalue un valor que no sea el mayor y es posible que dejemos atrás el máximo de la secuencia y por tanto fallemos en la ejecución.

Para comprobar nuestra hipótesis ejecutamos repetidamente el código con el comando: "`for i in {1..100}; do ./script.sh; done | grep wrong`" y efectivamente algunas de las ejecuciones fallaron.

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.
   #pragma omp critical
   Protege cada acceso a la variable haciendo este acceso exclusivo y evitando escrituras indeseadas.
   #pragma omp barrier
   Identifica un punto de sincronización en el que los subprocesos de la región paralela esperarán hasta que todos los demás subprocesos de esa sección alcancen el mismo punto.

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 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/howmany; i+=howmany) {
    if (vector[i] > maxvalue)
      maxvalue = vector[i];
  }
  }
[...]
```

6.datarace.c

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

No, realizando múltiples ejecuciones vemos que a veces retorna un resultado erróneo. El valor de "countmax" no siempre es el mismo, ya que en bucle *for* dónde se incrementa el "countmax" varía el número de iteraciones según la id del thread.

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.

SOLUCIÓN 1

```
[...]

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

[...]
```

En este caso evitamos que los threads ejecuten el if a la vez con la directiva *barrier* para evitar escrituras indeseadas.

SOLUCIÓN 2

```
[...]

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

[...]
```

Con *atomic* impedimos que varios threads puedan modificar el valor de "countmax" a la vez este.

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, todas las ejecuciones que realizamos están mal (Sorry, something went wrong - maxvalue=15 found 9 times). El resultado correcto debería ser countmax = 3 y maxvalue = 15. Nos sale erróneamente countmax = 9 porque al usarse 8 threads, suma los 8 threads + 1.

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

```
#include <stdio.h>
#include <omp.h>
/* Execute several times before answering the questions    */
/* with ./3.datarace                                       */
/* Q1: Is the program executing correctly? If not, explain */
/*    why it is not providing the correct result for one   */
/*    or the two variables (countmax and maxvalue)         */
/* Q2: Write a correct way to synchronize the execution    */
/*    of implicit tasks (threads) for this program.        */

#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8, 15, 15};

int main()
{
  int i, maxvalue=0;
  int countmax = 0;
  int howmany;
  omp_set_num_threads(8);
  #pragma omp parallel private(i) reduction(max: maxvalue)
  {
          int id = omp_get_thread_num();
          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();
          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;
}
```

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 la salida. En el caso del primer print no sabemos qué thread lo ejecutará antes. En el caso del segundo print, aunque el tiempo de espera de los threads con un id mayor será más largo, no podemos asegurar que se impriman en orden de id porque desconocemos el tiempo que emplea cada thread para ejecutar ese fragmento de código. En el último print, la instrucción barrier no influye en el orden de ejecución de los threads.

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

Una vez vistos cómo funcionan los códigos con la librería OpenMP y ver cómo trabajan las regiones paralelas, en la siguiente sesión nos centraremos en las tareas explícitas y nos habituaremos todavía más a modificar los códigos para crear paralelismo real.

Day 2: Explicit tasks

1.single.c

1. What is the nowait clause doing when associated to single?
Ejecuta de uno en uno sin esperar al thread anterior, es decir mientras el thread 0 procesa la primera iteración, el 1 va ejecutando la segunda etc.
2. Then, can you explain why all threads contribute to the execution of the multiple instances of single? Why do those instances appear to be executed in bursts?
Contribuyen porque el nowait rompe la barrera implícita de *single*, gracias a esta instrucción se reparten las iteraciones entre los 4 threads.
En este loop primero se ejecutan los printf, y después los 4 threads esperan un segundo en la función *sleep(1)*, una vez ha pasado este tiempo el loop pasa a la siguiente iteración y se vuelven a imprimir los *printf*.
Es por ese tiempo que pasan los threads en el sleep que el output sale a 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?
Porque no hay ninguna instrucción para que se ejecute en paralelo. Simplemente se está ejecutando de forma predeterminada.

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;
     #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;
      }
```

[…]

3. What is the firstprivate(p) clause doing? Comment it and execute again. What is happening with the execution? Why?
Define la variable p como privada para cada uno de los threads y la inicializa con el valor de la variable original, es decir el nodo apuntado en esa iteración del bucle.

3.taskloop.c

1. Which iterations of the loops are executed by each thread for each task grainsize or num tasks specified?
Para el *Loop1* las iteraciones se ejecutan por 3 *threads* como máximo por cada *grainsize*.
Para el *Loop2*, en cambio, las iteraciones se ejecutan por 4 *threads* como máximo, puesto que tenemos que generar 4 tareas y estas serán de 3 iteraciones cada una. No podemos predecir con exactitud qué thread ejecutará qué iteraciones, pero sí podemos asegurar que cada thread ejecutará *grainsize* iteraciones consecutivas en el caso del loop 1 y N/tasknum en el caso del loop 2.

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 este caso se generarán N/grainsize tasks, estás tendrán un tamaño mínimo de 5 y un tamaño máximo de 7, estos tamaños vienen determinados por la directiva grainsize, que permite crear tasks de [grainsize,grainsize*2-1].
Hemos visto que por norma general se crean tasks de 6 iteraciones, aunque, partiendo de la teoría, creemos que también existe la posibilidad de que se generen tasks de 7 y 5 iteraciones respectivamente.

3. Can grainsize and num tasks be used at the same time in the same loop?
Además de que el compilador no lo permite, no tiene sentido utilizar estas dos directivas juntas porque en el caso de que se pudieran ejecutar, la última en ejecutarse sobreescribiría la configuración de la primera.

4. What is happening with the execution of tasks if the nogroup clause is uncommented in the first loop? Why?
Al descomentar la directiva nogroup, los dos loops se intercalan en la ejecución.Esto es porque no se genera la región implícita del 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.

```
[...]

int main()
{
    int i;

    //inicializa el bucle
    for (i=0; i<SIZE; i++)
      X[i] = i;

    omp_set_num_threads(4);
    #pragma omp parallel
    #pragma omp single
    {
      // las tareas generadas dentro de este scope forman parte del mismo
      //grupo. Genera un sum para cada task de este grupo después los suma.
      #pragma omp taskgroup task_reduction(+: sum)
      {
          for (i=0; i< SIZE; i++)
            // A cada una de las tareas
            #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);0



     // Part III FUNCIONA BIEN
      #pragma omp taskgroup task_reduction(+: sum)
      {
          for (i=0; i< SIZE/2; i++)
            #pragma omp task firstprivate(i) in_reduction(+: sum)
              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;
```

```
}
```

5.synchtasks.c

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

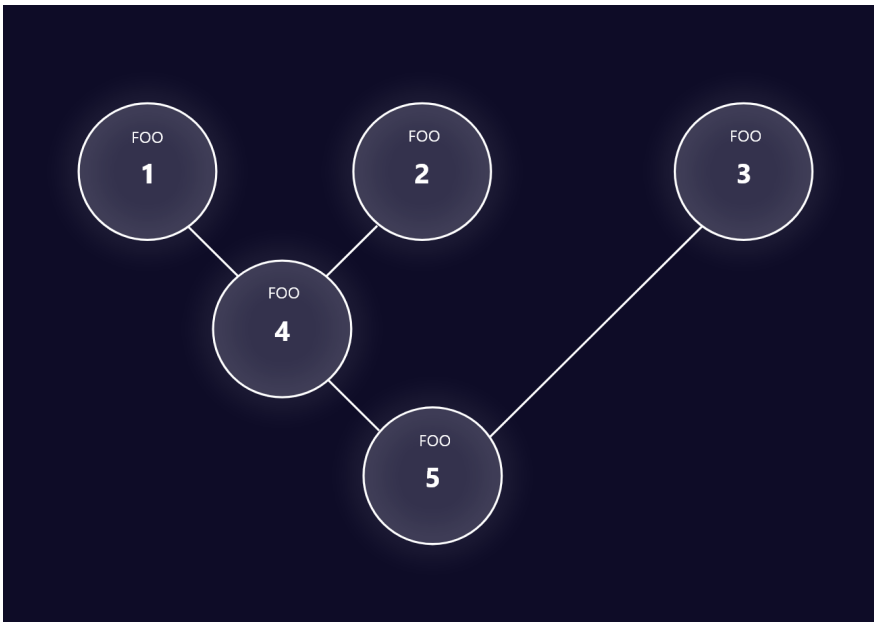

fig. 1: Grafo de dependencias de *5.synchtasks.c*

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.

[...]

```
void foo1() {
      printf("Starting function foo1\n");
        sleep(1);
      printf("Terminating function foo1\n");
}

void foo2() {
      printf("Starting function foo2\n");
        sleep(1);
      printf("Terminating function foo2\n");
}

void foo3() {
      printf("Starting function foo3\n");
        sleep(3);
      printf("Terminating function foo3\n");
}

void foo4() {
      printf("Starting function foo4\n");
        sleep(1);
      printf("Terminating function foo4\n");
```

```
}

void foo5() {
       printf("Starting function foo5\n");
         sleep(1);
       printf("Terminating function foo5\n");
}

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
      //#pragma omp task depend(out:a)
      foo1();
      printf("Creating task foo2\n");
      #pragma omp task
      //#pragma omp task depend(out:b)
      foo2();
      printf("Creating task foo3\n");
      #pragma omp task
      //#pragma omp task depend(out:c)
      foo3();
      printf("Creating task foo4\n");
      #pragma omp taskwait
      //#pragma omp task depend(in: a, b) depend(out:d)
      foo4();
      printf("Creating task foo5\n");
      #pragma omp taskwait
      //#pragma omp task depend(in: c, d)
      foo5();
    }
    return 0;
}
```

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.
[...]

```
void foo1() {
       printf("Starting function foo1\n");
          sleep(1);
       printf("Terminating function foo1\n");
}

void foo2() {
```

```
        printf("Starting function foo2\n");
          sleep(1);
        printf("Terminating function foo2\n");
}

void foo3() {
        printf("Starting function foo3\n");
          sleep(3);
        printf("Terminating function foo3\n");
}

void foo4() {
        printf("Starting function foo4\n");
          sleep(1);
        printf("Terminating function foo4\n");
}

void foo5() {
        printf("Starting function foo5\n");
          sleep(1);
        printf("Terminating function foo5\n");
}

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
      //#pragma omp task depend(out:a)
      foo1();
      printf("Creating task foo2\n");
      #pragma omp task
      //#pragma omp task depend(out:b)
      foo2();
      printf("Creating task foo3\n");
      #pragma omp task
      //#pragma omp task depend(out:c)
      foo3();
      printf("Creating task foo4\n");
      #pragma omp taskgroup
      //#pragma omp task depend(in: a, b) depend(out:d)
      foo4();
      printf("Creating task foo5\n");
      #pragma omp taskgroup
      //#pragma omp task depend(in: c, d)
      foo5();
    }
    return 0;
}
```

Observing overheads

Synchronisation overheads

Para entender los códigos primero establecimos qué son cada una de las operaciones de cada código:

- **Critical**: Protege cada acceso a la variable *suma* haciendo este acceso exclusivo
- **Reduction**: Usa las propiedades de reducción, es decir, todos los threads acumulan valores parciales en copias privadas de *suma* y al terminar la ejecución de la región afectada, el compilador se encarga de actualizar la variable global de forma segura.
- **Atomic**: Garantiza acceso indivisible a la localización de memoria donde está guardada.
- **Sumlocal**: Es similar a *reduction* pero la actualización final se hace en una región *critical*.

En cada versión se ejecutan diferentes operaciones de sincronización, ya sea *critical* o *atomic*:
pi_omp_critical: La operación *critical* se ejecuta *num_steps-myid* veces, donde *num_steps* es el número de iteraciones introducidas por el usuario y *myid* el identificador del *thread*
pi_omp_atomic.c: Igual que el anterior, la operación *atomic* se ejecuta *num_steps-myid* veces, ya que el código no cambia en absoluto excepto la operación *critical* ahora es *atomic*.
pi_omp_sumlocal.c: En este caso la operación *critical* se ejecuta NUMITERS veces, donde este va definido al principio de código.
pi_omp_reduction.c: En este código no encontramos ni operaciones *critical* ni *atomic*.

Primero ejecutamos la versión secuencial del código (*pi_sequential.c*) para comparar los resultados de las ejecuciones de las otras cuatro versiones. Lo que obtenemos del secuencial es lo siguiente:
Wall clock execution time = 1.793040037 seconds = 1793040.037 microseconds
Value of pi = 3.1415926536

Si ejecutamos las 4 versiones y ponemos en cola su ejecución usando *submit-omp.sh* obtenemos los siguientes resultados (teniendo en cuenta que todas se ejecutan con 100.000.000 de iteraciones):

1. Si ejecutamos con 1 thread
   Obtenemos los siguientes *outputs* por cada versión de código:
   pi_omp_critical.c:
   Total overhead when executed with 100000000 iterations on 1 threads: 2687059.0000 microseconds
   pi_omp_atomic.c:
   Total overhead when executed with 100000000 iterations on 1 threads: 12103.0000 microseconds

pi_omp_sumlocal.c:
Total overhead when executed with 100000000 iterations on 1 threads: 3934.0000 microseconds

pi_omp_reduction.c:
Total overhead when executed with 100000000 iterations on 1 threads: 11472.0000 microseconds

Si examinamos los resultados obtenidos en los outputs podemos ver un gran overhead en la versión *pi_omp_critical.c*, ya que el tiempo de ejecución sube considerablemente comparado con la versión secuencial.

1 thread

pi_omp_critical.c:	pi_omp_atomic.c:	pi_omp_sumlocal.c:	pi_omp_reduction.c:
2.687.059,00	12.103,00	3.934,00	11.472,00

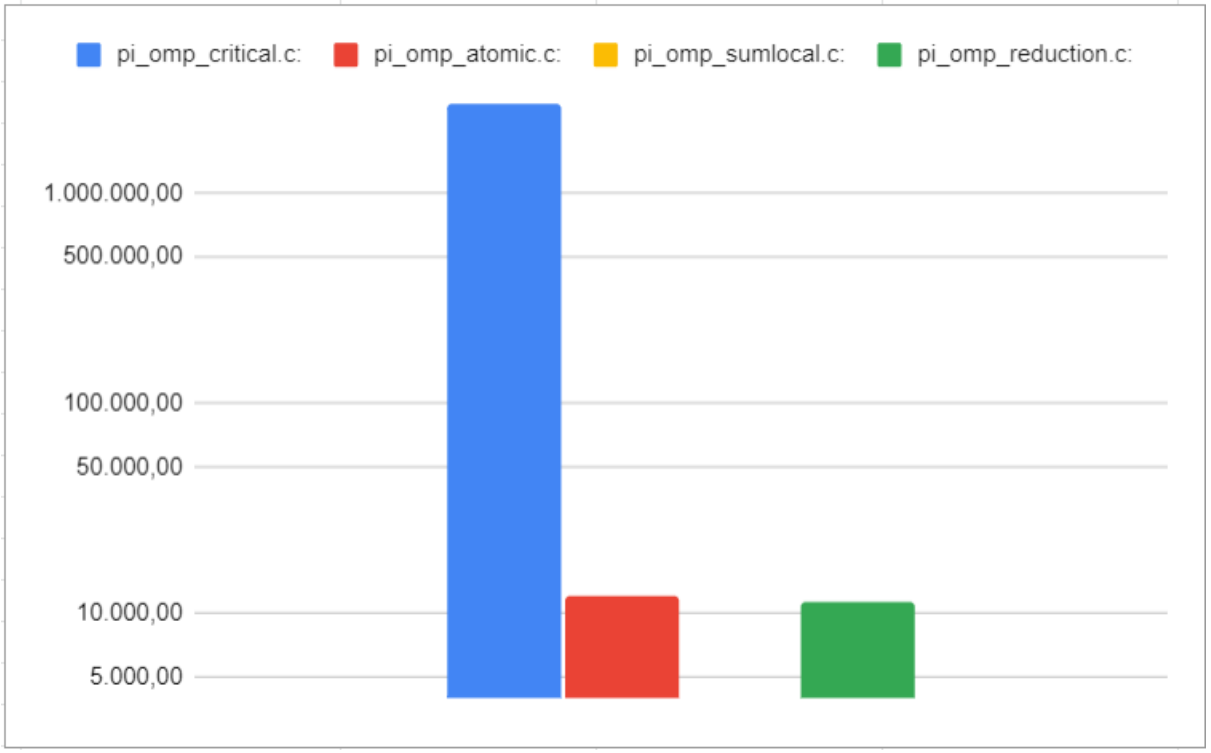

fig. 2: Tiempo de ejecución por programa con 1 thread

2. Si ejecutamos con 4 threads
   pi_omp_critical.c:
   Total overhead when executed with 100000000 iterations on 4 threads: 2977436.2500 microseconds
   pi_omp_atomic.c:
   Total overhead when executed with 100000000 iterations on 4 threads: 5263887.2500 microseconds

pi_omp_sumlocal.c:
Total overhead when executed with 100000000 iterations on 4 threads: 7623.2500 microseconds
pi_omp_reduction.c:
Total overhead when executed with 100000000 iterations on 4 threads: 10223.7500 microseconds

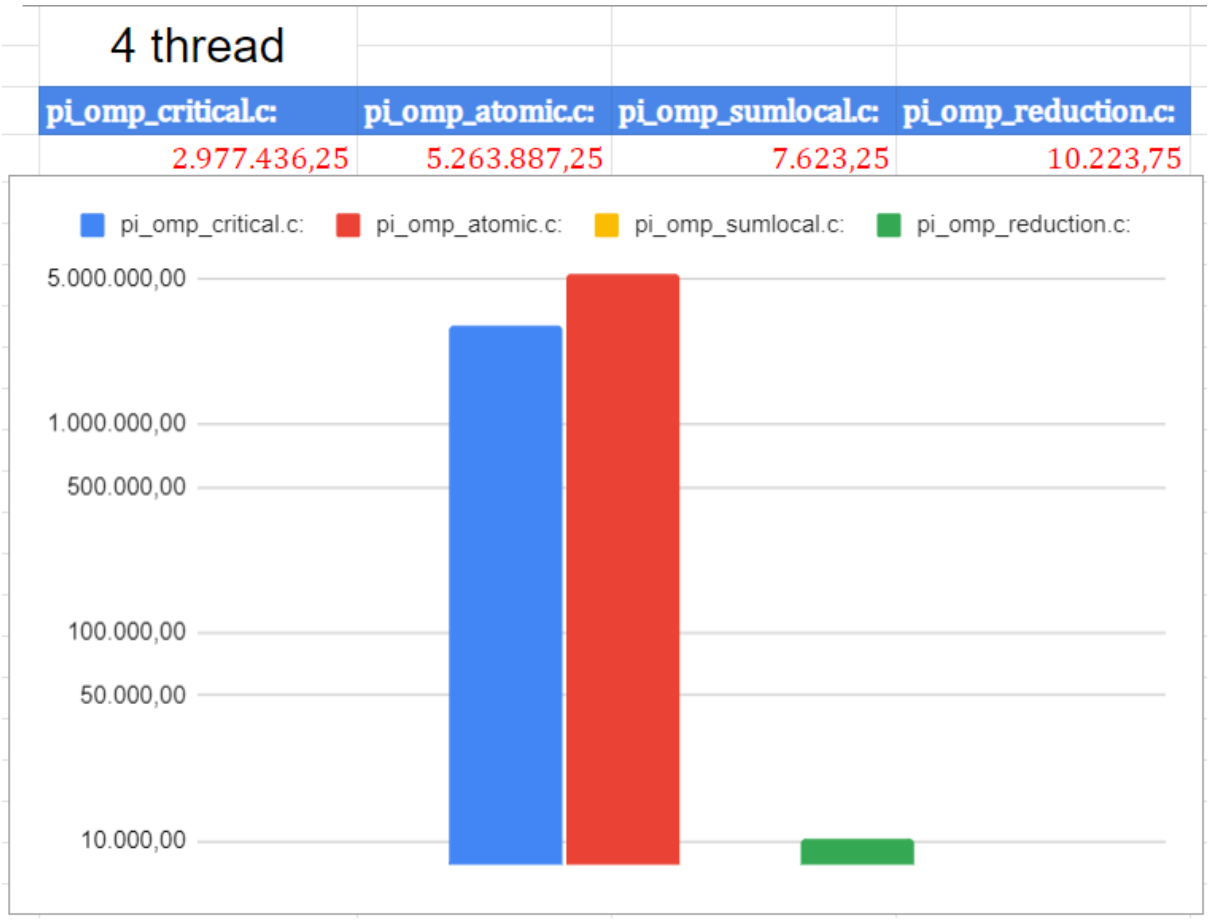

fig. 3: Tiempo de ejecución por programa con 4 thread

3. Si ejecutamos con 8 threads
pi_omp_critical.c:
Total overhead when executed with 100000000 iterations on 8 threads: 32888562.5000 microseconds
pi_omp_atomic.c:
Total overhead when executed with 100000000 iterations on 8 threads: 5773798.3750 microseconds
pi_omp_sumlocal.c:
Total overhead when executed with 100000000 iterations on 8 threads: 20516.8750 microseconds
pi_omp_reduction.c:
Total overhead when executed with 100000000 iterations on 8 threads: 19975.6250 microseconds

Como vemos en los resultados de las ejecuciones con 4 y 8 threads, podemos comprobar con los tiempos que las versiones que más se benefician del uso de varios procesadores son el *pi_omp_sumlocal* y el *pi_omp_reduction*. Esto se ve porque el tiempo de ejecución está muy por debajo de la versión secuencial, en cambio las versiones *pi_omp_critical* y *pi_omp_atomic* los tiempos son superiores a *pi_secuencial*. Este comportamiento de estas últimas versiones se debe a que dentro del for interno encontramos el *#pragma omp critical*, lo que significa que se ejecuta de forma secuencial cada iteración y el tiempo de ejecución es mucho más alto.

8 thread

pi_omp_critical.c:	pi_omp_atomic.c:	pi_omp_sumlocal.c:	pi_omp_reduction.c:
32.888.562,50	5.773.798,38	20.516,88	19.975,63

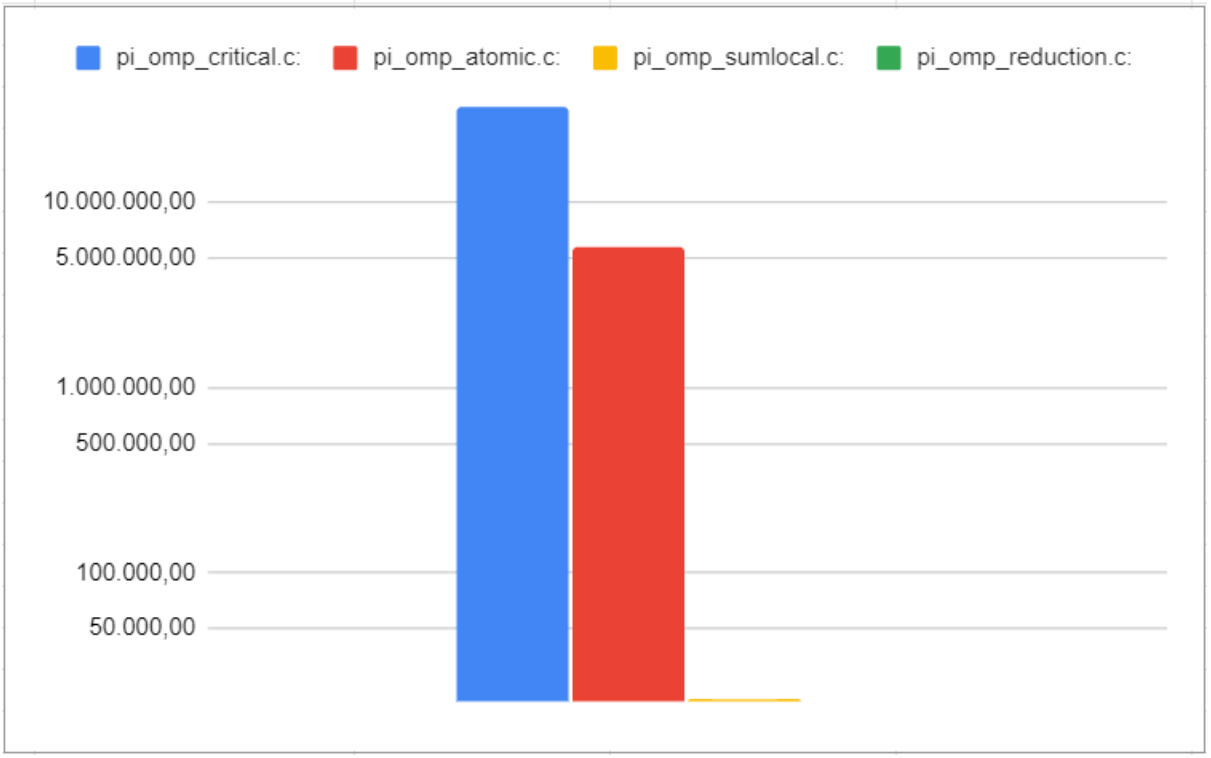

fig. 4: Tiempo de ejecución por programa con 8 thread

Thread creation and termination

En esta sección medimos los overheads relacionados con la creación de regiones paralelas. En este caso trabajamos con el código de *pi_omp_parallel.c*. Esto si lo ejecutamos (en la cola, con 1 iteración y 24 threads) con el sript *submit-omp.sh* nos muestra la siguiente tabla que vemos en el output de a continuación. Los overheads y los overheads per thread están expresados todos en microsegundos.

Si observamos los datos de las ejecuciones vemos que el tiempo de los overheads no es constante sino que va aumentando a medida que se sube el número de threads (aunque vemos casos que disminuye un poco, la tendencia es que sube cuando aumenta el número de procesadores). En los overheads por thread, en cambio, tiende a disminuir el tiempo cuando se sube el número de threads, puesto que no calcula tanto a medida que hay más threads y, por lo tanto, en tiempo se reduce.

All overheads expressed in microseconds

Nthr	Overhead	Overhead per thread
2	1.9523	0.9761
3	1.5079	0.5026
4	1.7219	0.4305
5	1.8238	0.3648
6	1.9244	0.3207
7	1.9555	0.2794
8	2.1730	0.2716
9	2.2152	0.2461
10	2.3150	0.2315
11	2.3121	0.2102
12	2.4632	0.2053
13	2.8140	0.2165
14	3.2441	0.2317
15	2.8837	0.1922
16	3.7531	0.2346
17	3.0394	0.1788
18	3.8909	0.2162
19	3.0634	0.1612
20	3.1091	0.1555
21	3.1331	0.1492
22	3.5944	0.1634
23	3.1550	0.1372
24	3.4014	0.1417

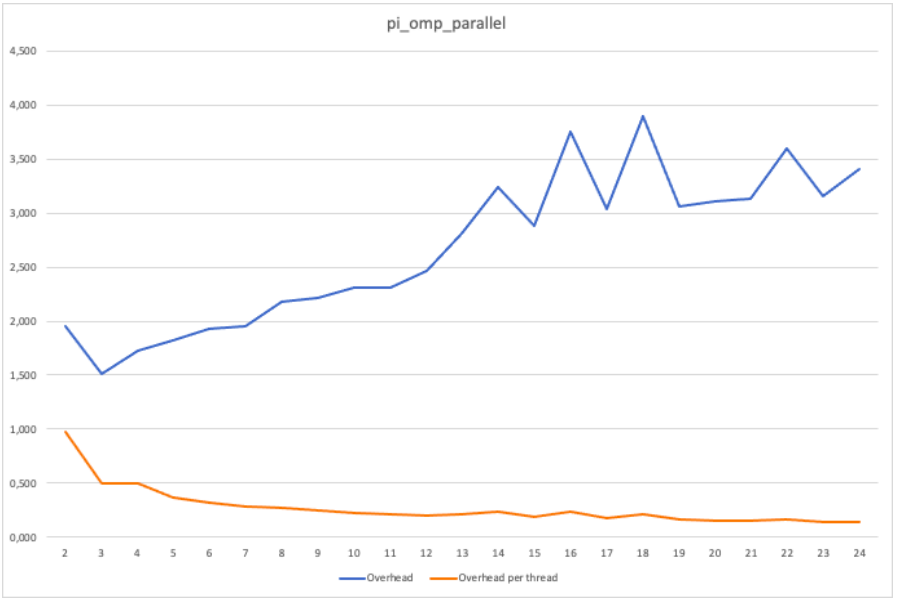

fig. 5: Comparación tiempo de ejecución en pi_omp_parallel

Task creation and synchronisation

En esta sección medimos los overheads relacionados con la creación de tareas. En este caso trabajamos con el código de *pi_omp_tasks.c* y lo ejecutamos en cola con 10 iteraciones y un solo thread. Esto lo hacemos, como en la sección anterior, con el script *submit-omp.sh* y nos da el siguiente output de a continuación. En este vemos que el tiempo de los overheads (expresado en microsegundos, al igual que los overheads por tarea) tiene una tendencia ascendente, más pronunciada que el de regiones paralelas anterior, a medida que subimos en número de tareas.
En cambio, el tiempo de los overheads por tarea (a parte del primero, donde hay 2 tareas), se mantiene casi constante, y esto se debe a la inestabilidad de la máquina, aunque la diferencia con la implementación en threads no es muy significativa para el balanceo.

All overheads expressed in microseconds

Ntasks	Overhead	Overhead per task
2	0.1461	0.0731
4	0.4774	0.1193
6	0.7178	0.1196
8	0.9485	0.1186
10	1.1822	0.1182
12	1.4156	0.1180
14	1.6470	0.1176
16	1.8781	0.1174
18	2.1138	0.1174
20	2.3388	0.1169
22	2.5834	0.1174
24	2.8114	0.1171
26	3.0438	0.1171
28	3.2888	0.1175
30	3.5139	0.1171
32	3.7512	0.1172
34	4.0107	0.1180
36	4.2140	0.1171
38	4.4609	0.1174
40	4.6832	0.1171
42	4.9284	0.1173
44	5.1540	0.1171
46	5.3852	0.1171
48	5.6019	0.1167
50	5.8502	0.1170
52	6.0892	0.1171
54	6.3318	0.1173
56	6.5457	0.1169
58	6.7790	0.1169
60	7.0299	0.1172
62	7.2534	0.1170
64	7.4742	0.1168

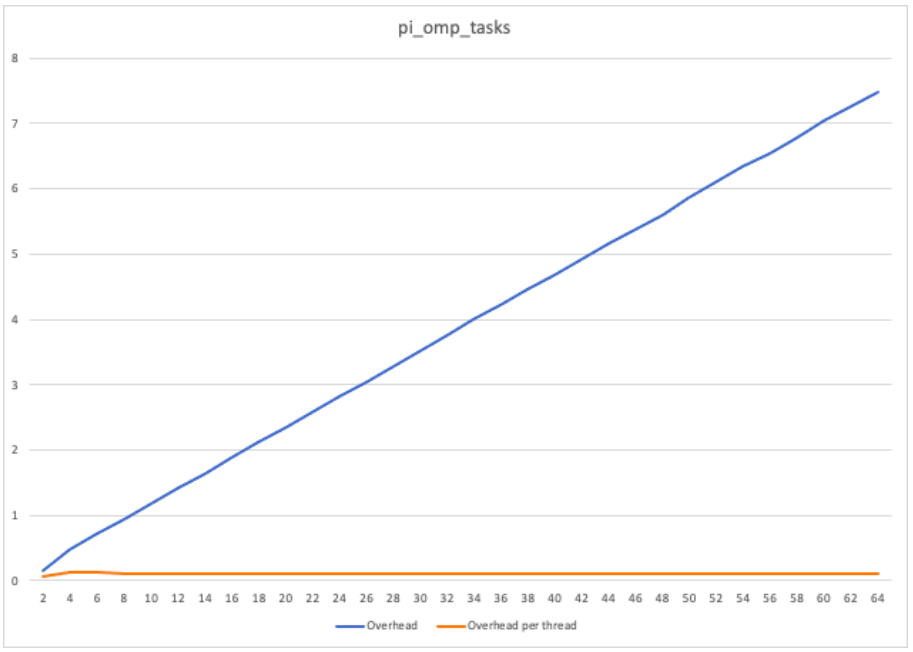

fig. 6: Comparación tiempo de ejecución en pi_omp_task

Conclusiones

En la primera parte de esta entrega, tratamos con OpenMP, una interfaz de programación que soporta programación multiproceso y experimentamos con las tareas implícitas. Empleando sus directivas pudimos organizar, sincronizar y controlar la ejecución de los diferentes threads con el fin de mejorar la eficiencia o corregir el funcionamiento, en el caso de que fuera incorrecto.
En esta primera sesión, además del lifespan de las diferentes directivas, hemos observado la importancia de controlar las escrituras en códigos de ejecución paralela y cómo el sistema operativo distribuye, entre los threads, la carga de trabajo.

En la segunda sesión seguimos trabajando en el estudio de las directivas OpenMP pero esta vez centrándonos en las tareas explícitas. En esta sección seguimos paralelizando y corrigiendo código, al igual que en la primera parte y aprendimos nuevas directivas que nos permitieron gestionar la creación de tareas y su granularidad..

Anexo

En esta sección recopilamos algunos comandos o reflexiones que hicimos mientras realizamos el deliverable, además de los códigos corregidos de la práctica.

Lab1 s1 (Anexo)

```
-------------------------Glossario
Comandos, respuestas, anotaciones → Azul
Outputs → Rojo
Comentarios código → Verde
Código referente a omp → Rosa
-------------------------Recordatorio
sbatch -p execution submit-omp.sh 3dfft_omp 1

ls -ltr
-------------------------------------

cp /scratch/nas/1/par0/sessions/lab2.tar.gz .

tar -zxvf lab2.tar.gz

cd ./lab2/pi/

make pi-vx-debug

./pi-vx-debug num-threads

make pi-vx-omp

---------------------------Truco para ejecutar múltiples veces

for i in {1..100}; do ./nombre_ejecutable; done > prueba.txt

gedit prueba.txt

----------------------------Deliverable
Day 1:
1.hello.c
  1-veces que imprime el hello world
    cd ./lab2/openmp/Day1/
    make 1.hello
    ./1.hello
```

Hello world!
Hello world!

2-commando para que imprima 4 veces hello world

OMP_NUM_THREADS=4 ./1.hello
Hello world!
Hello world!
Hello world!
Hello world!

2.hello.c

1-Is the execution correct?

make 2.hello
./2.hello
(4) Hello (2) world!
(2) Hello (2) world!
(6) Hello (6) world!
(1) Hello (1) world!
(3) Hello (5) world!
(5) Hello (5) world!
(0) Hello (0) world!
(7) Hello (7) world!

---------[Código 2.hello.c corregido]----------

```
/* Execute with ./2.hello                                    */
/* Q1: Is the execution of the program correct? Add a        */
/*   data sharing clause to make it correct                  */
/* Q2: Are the lines always printed in the same order?       */
/*   Why the messages sometimes appear intermixed?           */

int main ()
{
     int id;
     #pragma omp parallel num_threads(8) private(id)
     {
    id =omp_get_thread_num();
    printf("(%d) Hello ",id);
    printf("(%d) world!\n",id);
     }
     return 0;
}
```

Nueva ejecución. Output:

make 2.hello

./2.hello
(4) Hello (4) world!
(0) Hello (0) world!
(2) Hello (2) world!
(3) Hello (3) world!
(6) Hello (6) world!
(1) Hello (1) world!
(5) Hello (5) world!
(7) Hello (7) world!

3.how_many.c

1-return de omp_get_num_threads

- En la parte paralelizada

make 3.how_many
OMP_NUM_THREADS=8 ./how_many

#Empieza la ejecución con un thread
Starting, I'm alone ... (1 thread)

#con el comando "OMP_NUM_THREADS=8" se cambia el numero de threads a 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 first parallel (8)!

#con la instrucción "#pragma omp parallel num_threads(4)" se cambia el numero de threads a 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 szecond parallel (4)!

#con el comando "OMP_NUM_THREADS=8" se cambia el numero de threads a 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 third parallel (8)!
#En la primera iteración del bucle, al ejecutar omp_set_num_threads(i);,
como i ==2 el número de threads pasa a ser 2

Código bucle:
for (int i=2; i<4; i++) {
        omp_set_num_threads(i);
        #pragma omp parallel
        printf("Hello world from the fourth parallel (%d)!\n",
omp_get_num_threads());
}
Hello world from the fourth parallel (2)!
Hello world from the fourth parallel (2)!

#En la segunda iteración del bucle, al ejecutar omp_set_num_threads(i);,
como i ==3 el número de threads pasa a ser 3
Hello world from the fourth parallel (3)!
Hello world from the fourth parallel (3)!
Hello world from the fourth parallel (3)!

#cómo no usamos la instrucción "#pragma omp parallel" el código se ejecuta
de forma secuencial
Outside parallel, nobody else here ... (1 thread)

#con la instrucción "#pragma omp parallel num_threads(4)" se cambia el
numero de threads a 4 pero solo para esta instrucción.
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)!

#cómo no usamos la instrucción "#pragma omp parallel" el código se ejecuta
de forma secuencial. La i==3 se ejecuta de forma global
Hello world from the sixth  parallel (3)!
Hello world from the sixth  parallel (3)!
Hello world from the sixth  parallel (3)!

#cómo no usamos la instrucción "#pragma omp parallel" el código se ejecuta
de forma secuencial
Finishing, I'm alone again ... (1 thread)
```

- En la parte secuencial

```
make 3.how_many
OMP_NUM_THREADS=8 ./how_many
```

---------[Código 3.how_many.c corregido]------------------

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

/* If the OMP_NUM_THREADS variable is set to 8 at execution
time with   */
/* OMP_NUM_THREADS=8 ./3.how_many
     */
/* Q1: What does omp_get_num_threads return when invoked
outside and   */
/*   inside a parallel region?
     */
/* Q2: Indicate the two alternatives to superseed the
number of threads */
/*   that is specified by the OMP_NUM_THREADS environment
variable.   */
/* Q3: Which is the life span for each way of defining the
number of */
/*   threads to be used?
     */

int main ()
{
     printf("Starting, I'm alone ... (%d thread)\n",
omp_get_num_threads());
     #pragma omp parallel
     printf("Hello world from the first parallel (%d)!\n",
omp_get_num_threads());

     #pragma omp parallel num_threads(4)
     printf("Hello world from the szecond parallel
(%d)!\n", omp_get_num_threads());

     #pragma omp parallel
     printf("Hello world from the third parallel (%d)!\n",
omp_get_num_threads());
```

```
    for (int i=2; i<4; i++) {
    omp_set_num_threads(i);
    #pragma omp parallel
    printf("Hello world from the fourth parallel (%d)!\n",
omp_get_num_threads());
    }

    printf("Outside parallel, nobody else here ... (%d
thread)\n", omp_get_num_threads());

    #pragma omp parallel num_threads(4)
    printf("Hello world from the fifth parallel (%d)!\n",
omp_get_num_threads());

    #pragma omp parallel
    printf("Hello world from the sixth  parallel (%d)!\n",
omp_get_num_threads());

    printf("Finishing, I'm alone again ... (%d thread)\n",
omp_get_num_threads());
    return 0;
}
```

4.data_sharing.c

#Los threads añaden su id a la variable x (1+2+3+4+5… → 16*(16-1)/2 = 120)
After first parallel (shared) x is: 120

#Al poner la variable como privada, cada uno de los threads genera una variable local x (sin inicializar) y le suma su número id, al acabar la operación las variables locales se destruyen y se imprime el valor de la x (no ha sido modificada).
After second parallel (private) x is: 5

#Al poner la variable como privada, cada uno de los threads genera una variable local x (inicializada con el valor de x) y le suma su número id, al acabar la operación las variables locales se destruyen y se imprime el valor de la x (no ha sido modificada).
After third parallel (firstprivate) x is: 5

#Los threads añaden su id a la variable x (1+2+3+4+5… → 16*(16-1)/2 = 120) sobre la variable x. Cómo x==5 y la suma de los id == 120 la suma total es 125.
After fourth parallel (reduction) x is: 125

5.datarace.c
Vamos probando a hacer varias ejecuciones para ver con que valores falla y en que posiciones detecta el máximo

```
for i in {1..100}; do ./script.sh; done > prueba.txt
for i in {1..100}; do ./script.sh; done | grep wrong
```

---------[Código 5.datarace.c corregido]-------------------

```
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11,
10, 9, 8, 15, 15};

#include <stdio.h>
#include <omp.h>
/* Q1: Is the program executing correctly? Why?                */
/* Q2: 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.                                      */
/* Q3: 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.                       */

#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11,
10, 9, 8, 15, 15};

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/howmany; 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;
}
```

```
#-------------------------ejemplo ejecución 16 threads

{0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 0, 0, 0}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 3}

max value está tomando a la vez todos estos valores: {0, 0, 0, 1, 2, 3, 4, 5,
6, 7, 15, 14, 13, 12, 11, 10, 9,
```

6.datarace.c

---------[Código 6.datarace.c corregido]--------------------

```
#include <stdio.h>
#include <omp.h>
/* Q1: Is the program executing correctly? Why?                */
/* Q2: 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. */

#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11,
10, 9, 8, 15, 15};

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) {
```

```
         //#pragma omp barrier
         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;
}


#-------------------------
```

---------[Código 7.datarace.c corregido]-------------------

```
#include <stdio.h>
#include <omp.h>
/* Execute several times before answering the questions    */
/* with ./3.datarace                                       */
/* Q1: Is the program executing correctly? If not, explain */
/*     why it is not providing the correct result for one  */
/*     or the two variables (countmax and maxvalue)        */
/* Q2: Write a correct way to synchronize the execution    */
/*     of implicit tasks (threads) for this program.       */

#define N 1 << 20
int vector[N]={0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11,
10, 9, 8, 15, 15};

int main()
{
     int i, maxvalue=0;
     int countmax = 0;
     int howmany;
     omp_set_num_threads(8);
     #pragma omp parallel private(i) reduction(max: maxvalue)
     {
          int id = omp_get_thread_num();
          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();
          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;
}

#--------------------------
```

---------[Código 8.barrier.c]--------------------

```
#include <stdio.h>
#include <unistd.h>
#include <omp.h>

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

int main ()
{
    int myid;
    #pragma omp parallel private(myid) num_threads(4)
    {
        int sleeptime;

      myid=omp_get_thread_num();
         sleeptime=(2+myid*3)*1000;

      printf("(%d) going to sleep for %d milliseconds
...\n",myid,sleeptime);
      usleep(sleeptime);
```

```
      printf("(%d) wakes up and enters barrier ...\n",myid);
      #pragma omp barrier
      printf("(%d) We are all awake!\n",myid);
    }
    return 0;
}


#---------------------------
    #pragma omp parallel private(i) reduction(max: maxvalue)

    #pragma omp parallel private(i)  reduction(+: countmax)
```

Lab2 s2 (Anexo)

Formatear codigo: http://hilite.me/
Documentación_OMP:https://www.openmp.org/spec-html/5.1/openmpsu55.html
Presentación info OMP:
https://blog.rwth-aachen.de/hpc_import_20210107/attachments/44007431/44204044.pdf
Glossario operaciones OMP:
https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-160

----------Glossario
Comandos, respuestas, anotaciones → Azul
Outputs → Rojo
Comentarios código → Verde
Código referente a omp → Rosa
----------Recordatorio
#añade a la cola de ejecución un programa (ejecución paralela)
sbatch -p execution submit-omp.sh 3dfft_omp 1

#muestra los últimos documentos modificados
ls -ltr

#contador de líneas
wc -l
ej: output | wc -l

-CRITICAL: Protegeix cada acce´s a la variable **sum** fent aquest acce´s exclusiu

-ATOMIC: Garantitza acce´s indivisible a la localitzacio´ de memo`ria on **sum** esta´ guardada.

-REDUCTION: Usa les propietats de reduction, e´s a dir, tots els threads acumulen valors parcials en co`pies privades de **sum** i a l'acabar l'execucio´ de la regio´ afectada, el compilador s'encarrrega d'actualitzar la variable global de forma segura.

-SUMLOCAL: E´s similar a reduction pero` l'actualitzacio´ final es fa a una regio´ cri´tical

---

```
cp /scratch/nas/1/par0/sessions/lab2.tar.gz .

tar -zxvf lab2.tar.gz

cd ./lab2/pi/

make pi-vx-debug

./pi-vx-debug num-threads

make pi-vx-omp
```

----------Truco para ejecutar múltiples veces

```
for i in {1..100}; do ./nombre_ejecutable; done > prueba.txt

gedit prueba.txt
```

---------[Código 2.fibtasks.c corregido]-------------------

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"
#define N 25

/* Q1: Why all tasks are created and executed by the same thread?     */
/*     In other words, why the program is not executing in parallel?  */
/* Q2: Modify the code so that tasks are executed in parallel and     */
/*     each iteration of the while loop is executed only once         */
/* Q3: What is the firstprivate(p) clause doing? Comment it and       */
/*     execute again. What is happening with the execution? Why?      */

struct node {
   int data;
   int fibdata;
   int threadnum;
   struct node* next;
};

int fib(int n) {
   int x, y;
   if (n < 3) {
        return(1);
   } else {
      x = fib(n - 1);
      y = fib(n - 2);
      return (x + y);
   }
}
```

```
void processwork(struct node* p)
{
   int n;
   n = p->data;

   p->fibdata += fib(n);
   p->threadnum = omp_get_thread_num();
}

struct node* init_list(int nelems) {
    int i;
    struct node *head, *p1, *p2;

    p1 = malloc(sizeof(struct node));
    head = p1;
    p1->data = 1;
    p1->fibdata = 0;
    p1->threadnum = 0;
    for (i=2; i<=nelems; i++) {
       p2  = malloc(sizeof(struct node));
       p1->next = p2;
       p2->data = i;
       p2->fibdata = 0;
       p2->threadnum = 0;
       p1 = p2;
    }
    p1->next = NULL;
    return head;
}

struct node *p;

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;
     #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");

     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;
}
```

---------[Código 3.taskloop.c corregido]-------------------

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h> /* OpenMP */
#define N 12

/* Execute multiple times before answering the questions below   */
/* Q1: Which iterations of the loops are executed by each thread */
/*     for each task grainsize or num_tasks specified?           */
/* Q2: 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?                */
/* Q3: Can grainsize and num_tasks be used at the same time in   */
/*     the same loop?                                            */
/* Q4: What is happening with the execution of tasks if the      */
/*     nogroup clause is uncommented in the first loop? Why?     */

#define VALUE 4

int main()
{
    int i;

    omp_set_num_threads(4);
    #pragma omp parallel
    #pragma omp single
    {
      printf("Thread %d distributing %d iterations with grainsize(%d)
...\n", omp_get_thread_num(), N, VALUE);
      #pragma omp taskloop grainsize(VALUE) // nogroup
      for (i=0; i < N; i++) {
            printf("Loop 1: (%d) gets iteration %d\n",
omp_get_thread_num(), i);
      }
```

```
      printf("Thread %d distributing %d iterations with num_tasks(%d)
...\n", omp_get_thread_num(), N, VALUE);
      #pragma omp taskloop num_tasks(VALUE)
      for (i=0; i < N; i++) {
            printf("Loop 2: (%d) gets iteration %d\n",
omp_get_thread_num(), i);
      }
    }

    return 0;
}
```

```
Outputs ordenados para entender las tareas que se generan:
Thread 0 distributing 12 iterations with grainsize(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 1: (1) gets iteration 6
Loop 1: (1) 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

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


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(5) ...
Loop 2: (2) gets iteration 0
Loop 2: (2) gets iteration 1
Loop 2: (2) 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: (0) gets iteration 8
Loop 2: (0) gets iteration 9
#---------------------------
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
```

"If a `grainsize` clause is present, the number of logical iterations assigned to each generated task is greater than or equal to the minimum of the value of the grain-size expression and the number of logical iterations, but less than two times the value of the grain-size expression." → De la documentación de omp

---------[Código 4.reduction.c corregido]--------------------

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>   /* OpenMP */

/* Q1: Complete the parallelization 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.                                    */

#define SIZE 8192
#define BS   16
int X[SIZE], sum;
```

```
int main()
{
    int i;

    //inicializa el bucle
    for (i=0; i<SIZE; i++)
      X[i] = i;

    omp_set_num_threads(4);
    #pragma omp parallel
    #pragma omp single
    {
      // las tareas generadas dentro de este scope forman parte del mismo
      //grupo. Suponemos que genera un sum para cada task de este grupo
      //después los suma.
      #pragma omp taskgroup task_reduction(+: sum)
      {
          for (i=0; i< SIZE; i++)
            // A cada una de las tareas
            #pragma omp task firstprivate(i) in_reduction(+: sum)
              sum += X[i];
      }

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

        // Part II B CAREFUL
      #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);0

      sum=0;

     // Part III FUNCIONA BIEN
      #pragma omp taskgroup task_reduction(+: sum)
      {
          for (i=0; i< SIZE/2; i++)
            #pragma omp task firstprivate(i) in_reduction(+: sum)
              sum += X[i];
      }

      printf("xd = %d\n", sum);
      int sum2 = sum;
      #pragma omp taskloop grainsize(BS)
      for (i=SIZE/2; i< SIZE; i++)
          sum += X[i];

printf("VALOR A SUMAR %d\n VALOR SUM %d\n\n", X[i],sum);
  printf("Value of sum after reduction in combined task and taskloop = %d\n",
sum);
    }

    return 0;
}
```

```
Output 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 = 33550635
```

---------[Código 5.synchtasks.c corregido]----------

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "omp.h"

/* Q1: Draw the task dependence graph that is specified in this program  */
/* Q2: Rewrite the program only using taskwait as task synchronization   */
/*     mechanism (no depend clauses allowed)                             */
/* Q3: Rewrite the program only using taskgroup as task synchronization  */
/*     mechanism (no depend clauses allowed)                             */

void foo1() {
      printf("Starting function foo1\n");
        sleep(1);
      printf("Terminating function foo1\n");
}

void foo2() {
      printf("Starting function foo2\n");
        sleep(1);
      printf("Terminating function foo2\n");
}

void foo3() {
      printf("Starting function foo3\n");
        sleep(3);
      printf("Terminating function foo3\n");
}

void foo4() {
      printf("Starting function foo4\n");
        sleep(1);
      printf("Terminating function foo4\n");
}

void foo5() {
      printf("Starting function foo5\n");
        sleep(1);
      printf("Terminating function foo5\n");
}

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
```

```
      //#pragma omp task depend(out:a)
      foo1();
      printf("Creating task foo2\n");
      #pragma omp task
      //#pragma omp task depend(out:b)
      foo2();
      printf("Creating task foo3\n");
      #pragma omp task
      //#pragma omp task depend(out:c)
      foo3();
      printf("Creating task foo4\n");
      #pragma omp taskwait
      //#pragma omp task depend(in: a, b) depend(out:d)
      foo4();
      printf("Creating task foo5\n");
      #pragma omp taskwait
      //#pragma omp task depend(in: c, d)
      foo5();
    }
    return 0;
}
```

```
-----------------------------------------------------------------
Creating task foo1
Creating task foo2
Creating task foo3
Creating task foo4
Starting function foo3
Starting function foo1
Terminating function foo1
Starting function foo2
Terminating function foo2
Terminating function foo3
Starting function foo4
Terminating function foo4
Creating task foo5
Starting function foo5
Terminating function foo5
```

---------[Código 5.synchtasks.c corregido]----------------

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "omp.h"

/* Q1: Draw the task dependence graph that is specified in this program
*/
/* Q2: Rewrite the program only using taskwait as task synchronization
*/
/*     mechanism (no depend clauses allowed)
*/
```

```
/* Q3: Rewrite the program only using taskgroup as task synchronization
*/
/*     mechanism (no depend clauses allowed)
*/

void foo1() {
      printf("Starting function foo1\n");
        sleep(1);
      printf("Terminating function foo1\n");
}

void foo2() {
      printf("Starting function foo2\n");
        sleep(1);
      printf("Terminating function foo2\n");
}

void foo3() {
      printf("Starting function foo3\n");
        sleep(3);
      printf("Terminating function foo3\n");
}

void foo4() {
      printf("Starting function foo4\n");
        sleep(1);
      printf("Terminating function foo4\n");
}

void foo5() {
      printf("Starting function foo5\n");
        sleep(1);
      printf("Terminating function foo5\n");
}

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
      //#pragma omp task depend(out:a)
      foo1();
      printf("Creating task foo2\n");
      #pragma omp task
      //#pragma omp task depend(out:b)
      foo2();
      printf("Creating task foo3\n");
      #pragma omp task
```

```
      //#pragma omp task depend(out:c)
      foo3();
      printf("Creating task foo4\n");
      #pragma omp taskgroup
      //#pragma omp task depend(in: a, b) depend(out:d)
      foo4();
      printf("Creating task foo5\n");
      #pragma omp taskgroup
      //#pragma omp task depend(in: c, d)
      foo5();
    }
    return 0;
}
```

```
Creating task foo1
Creating task foo2
Creating task foo3
Creating task foo4
Starting function foo4
Starting function foo1
Terminating function foo4
Creating task foo5
Terminating function foo1
Starting function foo5
Starting function foo2
Terminating function foo5
Starting function foo3
Terminating function foo2
Terminating function foo3
```