PAR Laboratory Assignment

Lab2: Brief tutorial on OpenMP programming model

Sergio Araguás Villas - par4107
David Pérez Barroso - par4121
11-10-2022
22-23 Q1

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Índice

Day 1: Parallel regions and implicit tasks

- **1.hello.c**

1. ¿Cuántas veces verás el mensaje "¡Hola mundo!" si el programa se ejecuta con "./1.hello"?
El mensaje Hola Mundo se mostrará por pantalla 2 veces, ya que la instrucción *#pragma omp parallel* reparte el trabajo entre los threads disponibles donde en este caso són dos threads.

2. Sin cambiar el programa ¿Cómo hacer que se imprima 4 veces el mensaje "Hello World!"?
Escribiendo por consola "OMP NUM THREADS=4 *./1.hello*", lograremos que el programa se ejecute en 4 threads, obteniendo en pantalla el mensaje Hello World! cuatro veces sin haberlo modificado.

- **2.hello.c**

1. ¿Es correcta la ejecución del programa? (es decir, imprime una secuencia de "(Thid) Hello (Thid)world!" siendo Thid el identificador del hilo). Si no, agregar una cláusula de intercambio de datos para que sea correcto.
No es correcta, ya que los *threads* modifican el valor de la variable id entre ellos. Hemos agregado *private(id)* para hacer la variable id privada en cada *thread* para así lograr la correcta ejecución.

2. ¿Las líneas siempre se imprimen en el mismo orden? ¿Por qué los mensajes a veces aparecen entremezclados? (Ejecuta varias veces para ver esto).
No, los mensajes en cada ejecución no se imprimen en el mismo orden, ya que los threads no tienen un orden específico en su ejecución, dando como consecuencia que los printf de distintos threads se intercalan y tengamos como resultado una salida como la siguiente:

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

- **3.how_many.c:**

Suponiendo que la variable OMP_NUM_THREADS está establecida en 8 con "OMP_NUM_THREADS=8 ./3.how_many"

1. ¿Qué devuelve omp_get_num_threads cuando se invoca fuera y dentro de una región paralela?
La ejecución comienza con solo 1 thread, puesto que la primera parte no se encuentra en la parte a paralelizar, por lo que se ejecutará secuencialmente devolviendo la función omp_get_num_threads() un 1, esto pasará también con los mensajes de "*Outside parallel..*" y "*Finishing..*".

Con *OMP_NUM_THREADS*=8 le decimos al programa que en la parte a paralelizar se ejecute con 8 threads, por lo que tanto el mensaje de "*first parallel*" como el de "*third parallel*" se escribirá por pantalla 8 veces puesto que son ejecutados por 8 threads. La instrucción para estos casos devolverà un 8.

Dado que para los mensajes "*second parallel*" y "*fifth parallel*" se especifica antes que se quiere ejecutar con 4 threads mediante la instrucción *#pragma omp parallel num_threads*(4), estos saldrán 4 veces por pantalla y la instrucción nos devolverà un 4.

El mensaje "*fourth parallel*" nos lo encontramos dentro de un bucle al que se le asigna un número de threads utilizando la función *omp_set_num_threads*(*i*) con la variable *i*, esta tomarà como valores 2 y 3, por lo que impirmirà dos veces por pantalla el mensaje devolviendo la función omp_get_num_threads() un 2 y tres veces devolviendo un 3.

Aquí la instrucción *omp_set_num_threads*(*i*), nos ha cambiado el valor del número de threads con los que se ejecute el programa, por lo tanto dejará de ser 8 que son los que habíamos asignado manualmente, y pasarán a ser tantos como nos diga el último valor de *i* que en este caso será 3. Esto nos lo encontraremos en el mensaje "sixth parallel", donde se mostrará por pantalla 3 veces y la función devolverá un 3.

2. Indique las dos alternativas para reemplazar el número de subprocesos especificado por la variable de entorno OMP_NUM_THREADS.

- Utilizando la función *omp_set_num_threads(x)* donde x será el número de threads deseados, para establecer el valor de threads por defecto en todas las regiones paralelas del programa.
- Especificando después de *#pragma omp parallel* con *num_threads(x)* donde x será el número de threads deseados.

3. ¿Cuál es el tiempo de vida para cada forma de definir el número de hilos a utilizar?

Si se define después de *#pragma omp parallel con num_threads(x)*, solo afectará a la instrucción que tenga debajo.
Si se define con la función *omp_set_num_threads(x)*, establecerá el número de threads por defecto en todas las regiones paralelas del programa si no se especifica lo contrario.

- **4.data_sharing.c**

1. ¿Cuál es el valor de la variable x después de la ejecución de cada región paralela con diferente atributo de intercambio de datos (shared, private, firstprivate y reduction)? ¿Es ese el valor que esperaría? (Ejecutar varias veces si es necesario)

En el *shared* podrá tomar los valores de entre 0 a 31, ya que solo permanecerá el valor del último thread que ejecute la línea *x= tmp + omp_get_thread_num()*;
Esto sucede porque todos los threads tendrán la variable tmp inicializada a 0, y debido a la existencia de una instrucción sleep(1), a los threads no les dará tiempo a modificar la variable *tmp* con el valor de *x* de otros threads, sobreescribiendo cada thread la x con su valor.

Si quitamos la instrucción sleep(1) se sumarían los valores de 0 a 31 dando 496

Con *private* devolverá 5, puesto que cada thread tendrá su propia variable privada (local) sin inicializar, a la que se sumará el valor de su id, pero al acabar la ejecución el valor de esta variable "local" se destruye dejando el valor original de 5 que se le había asignado antes a x.

Con *firstprivate* ocurre lo mismo que en el caso de *private* por lo que devuelve 5,, con la diferencia que aquí los threads inicializan la variable privada con el valor que tiene inicialmente x.

Con *reduction(+:x)* nos dará el resultado de sumar todos los valores de los ids del 0 al 31 sumándole los 5 iniciales de la variable x.

- **5.datarace.c**

1. ¿Debe este programa devolver siempre un resultado correcto?

No, la mayoría de las veces devuelve un resultado incorrecto, ya que como varios thread se ejecutan de forma simultánea no podemos asegurar que se realizan todas las comparaciones necesarias provocando en la mayoría de veces encuentren un valor erróneo en el *maxvalue*.

Además, y debido a la función sleep(), puede darse el caso de que se sobreescriba la variable *maxvalue* con un valor que no sea el valor máximo del vector, pero que para el thread que lo ha ejecutado si lo sea.

2. Proponer dos alternativas de solución para hacerlo correcto, sin cambiar la estructura del código (solo agregar directivas o cláusulas). Explica por qué hacen la ejecución correcta.

#pragma omp barrier

Añadiendo *#pragma omp barrier* logramos sincronizar los threads al final de cada iteración evitando así el data racing que se produce en la otra versión.

```
int main()
{
    int i, maxvalue=3;

    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)
          {
             sleep(1); // this is just to force problems
             maxvalue = vector[i];
         }
         #pragma omp barrier
       }

    }
```

Fig.1. Código con #pragma omp barrier

#pragma omp critical

Con esta instrucción, protegemos el acceso a las variables haciendo que cada thread escriba sin interrupciones de otro.

```
int main()
{
    int i, maxvalue=3;

    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 critical
          if (vector[i] > maxvalue)
          {
             sleep(1); // this is just to force problems
             maxvalue = vector[i];
         }

       }

    }
```

Fig.2. Código con #pragma omp critical

3. Escriba una distribución alternativa de iteraciones para tareas implícitas (hilos) de modo que cada uno de ellos ejecute solo un bloque de iteraciones consecutivas (es decir, N dividido por el número de hilos).

```
int main()
{
    int i, maxvalue=3;

    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)
          {
             sleep(1); // this is just to force problems
             maxvalue = vector[i];
         }

       }

    }
```

Fig.3. Código con reduction

- **6.datarace.c**

1. ¿Debe este programa devolver siempre un resultado correcto?

No, en ocasiones devuelve un resultado erróneo debido a que *countmax* puede actualizarse de una manera errónea a causa de la ejecución simultánea de los threads.

2. Proponer dos alternativas de solución para hacerlo correcto, sin cambiar la estructura del programa (sólo usando directivas o cláusulas) y nunca haciendo uso de críticas. Explica por qué hacen la ejecución correcta.

Con *#pragma omp barrier*

```
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)
            countmax++;
        }
```

Fig.4. Código con #pragma omp barrier

De esta manera sincronizamos los threads en cada iteración evitando así escrituras indeseadas.

Con *#pragma omp atomic*

```
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++;
        }
    }
```

Fig.5. Código con #pragma omp atomic

De esta manera hacemos que la suma de *countmax* se haga solo por un thread simultáneamente, asegurando el valor correcto de *countmax*.

Synchronization Overheads

Se nos pide estudiar los overheads para diferentes códigos con distintos mecanismos de sincronización para actualizar la variable sum, los programas a observar se detallan a continuación con las instrucciones correspondientes:

- **pi_omp_critical:** Se utiliza la instrucción ***critical*** para proteger cada acceso a *sum*, asegurando el acceso exclusivo a la variable para cada *thread*.

- **pi_omp_atomic:** Utiliza la instrucción ***atomic*** para garantizar a cada *thread* un acceso indivisible a la zona de memoria donde se encuentra la variable *sum* almacenada.

- **pi_omp_sumlocal:** Utiliza una ***copia privada de sumlocal por thread***, seguido de una actualización global usando solo una región crítica.

- **pi_omp_reduction**: Utiliza la instrucción ***reduction*** aplicada a la variable sum, por lo que cada thread acumulará una versión privada y parcial de la variable y al final el compilador se encargará de actualizar la variable global de una forma segura.

Empezaremos ejecutando ***pi_sequential.c***, esta es la versión secuencial del código, nos servirá para comparar los resultados con las otras versiones. El resultado obtenido es el siguiente:

Wall clock execution time = 1.246712923 seconds = 1246712.923 microseconds
Value of pi = 3.1415926536

A continuación, se ejecutan las diferentes versiones con 100000000 iteraciones y 1, 4 y 8 threads, el resultado devuelto de las diferentes ejecuciones es la diferencia que hay entre el tiempo de ejecución real y el tiempo de ejecución ideal (tiempo_seq / #threds).

- **pi_omp_critical:**
  Total overhead when executed with 100000000 iterations on 1 threads:
  1861117.0000 microseconds
  Total overhead when executed with 100000000 iterations on 4 threads:
  47625101.0000 microseconds
  Total overhead when executed with 100000000 iterations on 8 threads:
  42885696.2500 microseconds

- **pi_omp_atomic:**
  Total overhead when executed with 100000000 iterations on 1 threads:
  -3949.0000 microseconds
  Total overhead when executed with 100000000 iterations on 4 threads:
  8434521.7500 microseconds
  Total overhead when executed with 100000000 iterations on 8 threads:
  9817535.0000 microseconds

- **pi_omp_sumlocal:**
  Total overhead when executed with 100000000 iterations on 1 threads:
  -2484.0000 microseconds
  Total overhead when executed with 100000000 iterations on 4 threads:
  3741.0000 microseconds
  Total overhead when executed with 100000000 iterations on 8 threads:
  17677.8750 microseconds

- **pi_omp_reduction**:
  Total overhead when executed with 100000000 iterations on 1 threads:
  -2484.0000 microseconds
  Total overhead when executed with 100000000 iterations on 4 threads:
  3957.7500 microseconds
  Total overhead when executed with 100000000 iterations on 8 threads:
  15924.5000 microseconds

Observamos que usando *critical* se emplea más tiempo en todas las ocasiones frente a las expectativas de tiempo ideales paralelas. Esto se debe al tiempo de overhead que se suma al tiempo de cada ejecución. La diferencia entre el tiempo real e ideal paralelo aumentará conforme aumenten los threads ya que a más paralelización más necesidad de sincronización y por ende más tiempo de overheads.

En las demás versiones a la hora de ejecutarse con un thread, obtenemos un tiempo negativo, creemos que esto se debe a alguna optimización que hace el compilador o por localidad, ya que al comparar el tiempo ideal con el tiempo real nunca podría dar inferior que 0.

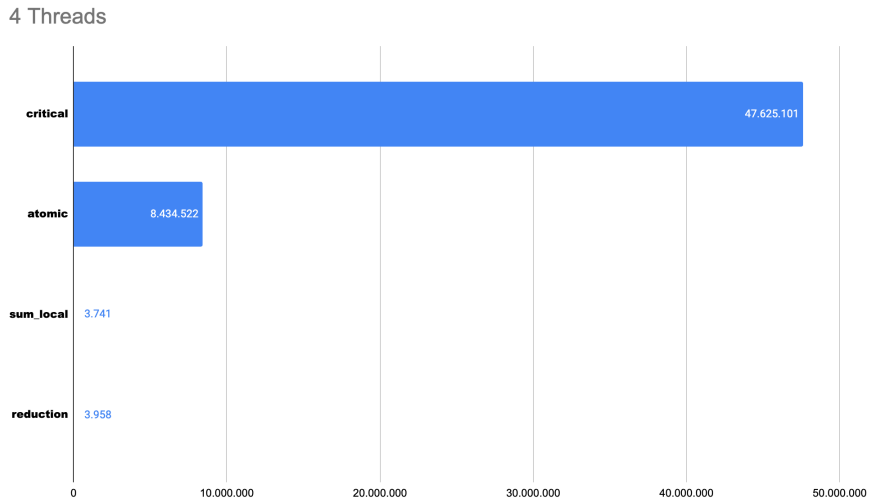

Fig.6. Gráfico para los tiempos de ejecución con 4 threads

Encontramos que las diferencias de tiempo entre tiempo real y tiempo ideal paralelo no son las mismas para todos los códigos. Como podemos ver en el gráfico (Fig.6.) *critical* es el que más tiempo de overhead añade a sus ejecuciones, seguido de *atomic*, acabando con *sumlocal* y *reduction* cuyos tiempos de overhead son mínimos en comparación con critical y atomic.

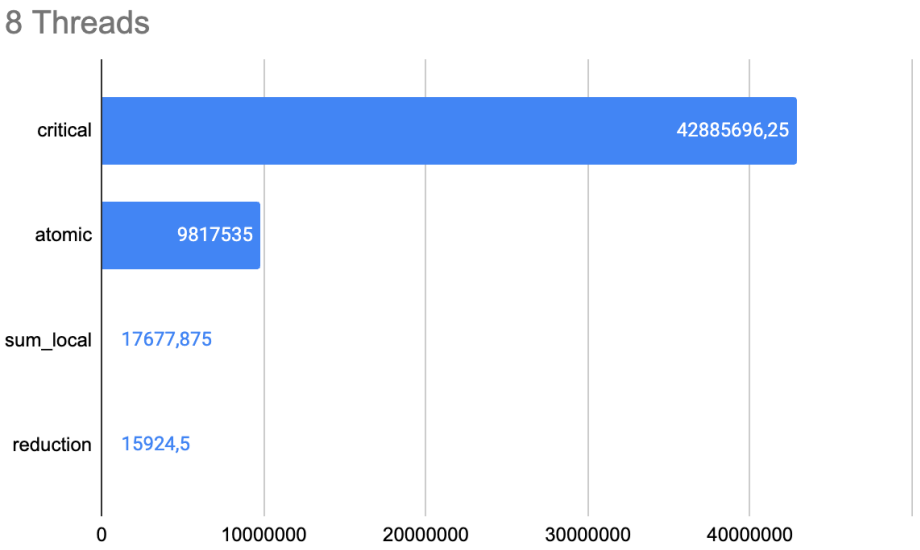

Fig.7. Gráfico para los tiempos de ejecución con 8 threads

Como esperábamos con ocho threads (Fig.7.) obtenemos las mismas conclusiones que con cuatro, además de ver que la diferencia entre tiempos aumenta respecto a la ejecución con cuatro threads, debido al aumento de paralelización del código y la necesidad de sincronización que genera más overheads.

Day 2: explicit tasks

- **1.single.c**

  1. ¿Qué hace la cláusula *nowait* cuando se asocia a single?
  Cuando se utiliza la cláusula *nowait*, se salta el *barrier* implícito que hay al final de *single*, haciendo ahora que los threads no esperen a que acabe y ejecutando ahora el resto de iteraciones.

  2. ¿Puede explicar por qué todos los subprocesos contribuyen a la ejecución de las múltiples instancias de single? ¿Por qué esas instancias parece que se ejecuta en ráfagas?
  La contribución de los demás threads se debe a la cláusula *nowait*, que permite a estos ejecutar otras iteraciones a la vez que el primero. Las ráfagas son debidas a la instrucción *sleep(1)*, los *threads* ejecutan una iteración, y se esperan un segundo antes de ejecutar la siguiente debido a esta instrucción.

- **2.fibtasks.c**

  1. ¿Por qué todas las tareas son creadas y ejecutadas por el mismo hilo? En otras palabras, ¿por qué el programa no se ejecuta en paralelo?
  El código no dispone de ninguna instrucción para poder ejecutarse en paralelo, por lo que se ejecutará con un solo thread.

  2. Modifique el código para que las tareas se ejecuten en paralelo y cada iteración del ciclo while sea ejecutado una sola vez.

```
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;
}
```

Fig.8. Código pregunta 2.2

3. ¿Qué hace la cláusula firstprivate(p)? Comentarlo y ejecutar de nuevo. ¿Qué está pasando con la ejecución? ¿Por qué?

firstprivate hace que cada tarea tenga su propia variable privada p que se inicializa con el valor de p original. Si comentamos el *firstprivate(p)*, no se pasa la p actualizada y todos los threads acceden simultáneamente a ella creando un segmentation fault al acceder a p->next.

- **3.taskloop.c**

1. ¿Qué iteraciones de los bucles ejecuta cada subproceso para cada tarea *grainsize* o *num_tasks*?

Debido a que *la* cláusula *grainsize* especifica el número de iteraciones del *Loop1* que habrá en cada tarea creada, el número mínimo de iteraciones que tendrá cada tarea será de 4, sin superar el doble del valor de *grainsize* que es 7.

En el *Loop* 2 el número de iteraciones por tarea estará entre 3 y 12, ya que el mínimo de tareas será el especificado por *num_task* que en este caso es 4 y el máximo de iteraciones será el número de iteraciones del bucle, por lo tanto, el mínimo será (12 iteraciones / 4 tareas = 3 iteraciones por tarea) y el máximo será (12 iteraciones / 12 tareas = 1 iteraciones por tarea)

2. Cambie el valor de grainsize y num tareas a 5. ¿Cuántas iteraciones tiene ahora cada subproceso ejecutando? ¿Cómo se decide el número de iteraciones en cada caso?

Loop 1 entre 5 y 9 iteraciones.
Loop 2 entre (12/5 = 2,4) = 3 y 12 iteraciones.
Como hemos explicado en la pregunta anterior, al utilizar *grainsize* el número de iteraciones por tarea queda definido entre el valor del *grainsize* y el doble de este (excluido).
Mientras que para *num_task* se define por el número de iteraciones del bucle entre valor de *num_tasks* y el número de iteraciones del bucle.

3. ¿Se pueden usar las tareas grainsize y num al mismo tiempo en el mismo bucle?

No, ya que estas dos cláusulas son mutuamente exclusivas, además el compilador no lo permite.

4. ¿Qué sucede con la ejecución de tareas si la cláusula nogroup no está comentada en el primer bucle? ¿Por qué?

Al no estar comentada, los dos loops se intercalan debido a que no se genera la región implícita del taskloop

- **4.reduction.c**

1. Completa la paralelización del programa para que el valor correcto para la variable sum sea devuelto en cada instrucción printf. Nota: en cada parte de las 3 partes del programa, todas las tareas generadas debería potencialmente ejecutarse en paralelo.

```
int main()
{
    int i;

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

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

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

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

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

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

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

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

Fig.9. Código completo paralelizado

```
par4121@boada-6:~/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
```

Fig.10. Resultado de la ejecución del código correcto

- **5.synchtask.c**

1. Dibuje el gráfico de dependencia de tareas que se especifica en este programa

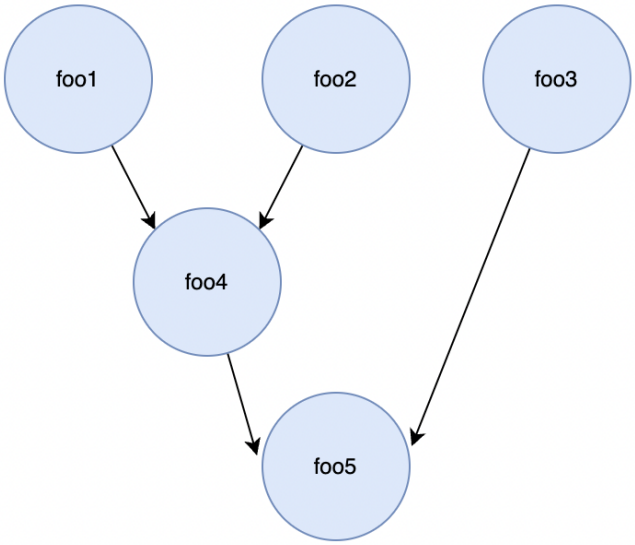

Fig.11. Grafo de dependencias del programa 5.synchtask.c

2. Vuelva a escribir el programa usando solo *taskwait* como mecanismo de sincronización de tareas (sin dependencia). cláusulas permitidas), tratando de lograr el mismo paralelismo potencial que se obtuvo utilizando *depend*.

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

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

Fig.12. Código con taskwait del programa 5.synchtask.c

Como se ve en el grafo de dependencias (Fig.11.), para obtener el mismo resultado que con depend, las únicas tareas que deberán esperar serán *foo4* y *foo5* puesto que para ejecutar *foo4*, deben haber acabado *foo1* y *foo2*, y para poder ejecutar *foo5* ,deben haber acabado *foo3* y *foo4*. Hemos cambiado de orden *foo4* y *foo3* para que estas se ejecuten paralelamente, pero en materia de tiempo, el resultado hubiera sido el mismo si *foo3* se hubiera ejecutado con *foo1* y *foo2*.

3. Vuelva a escribir el programa usando solo el grupo de tareas como mecanismo de sincronización de tareas (sin dependientes). cláusulas permitidas), nuevamente tratando de lograr el mismo paralelismo potencial que se obtuvo cuando se usa *depend*.

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

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

Fig.13. Código con taskgroup del programa 5.synchtask.c

Creando los *tasksgroup* que se observan en el código (Fig.13.) obtenemos el mismo comportamiento que en el apartado anterior.

Thread creation and termination

En este apartado se nos pide observar los overheads relacionados con la creación de las regiones paralelas. Para ello estudiaremos el overheads introducido a la hora de ir añadiendo threads a la ejecución paralela del programa ***pi_omp_parallel.c***.

Con la ejecución de este utilizando una iteración y hasta veinte threads, obtenemos los siguientes resultados:

All overheads expressed in microseconds

Nthr	*Overhead*	*Overhead per thread*
2	2,8987	1,4493
3	2,3187	0,7729
4	2,5447	0,6362
5	2,6657	0,5331
6	2,9751	0,4958
7	3,1079	0,4440
8	3,2758	0,4095
9	3,6374	0,4042
10	3,7024	0,3702
11	3,7973	0,3452
12	4,0110	0,3342
13	3,8876	0,2990
14	4,0142	0,2867
15	4,1656	0,2777
16	4,.2079	0,2630
17	4,6774	0,2751
18	4,5016	0,2501
19	4,6785	0,2462
20	4,8352	0,2418

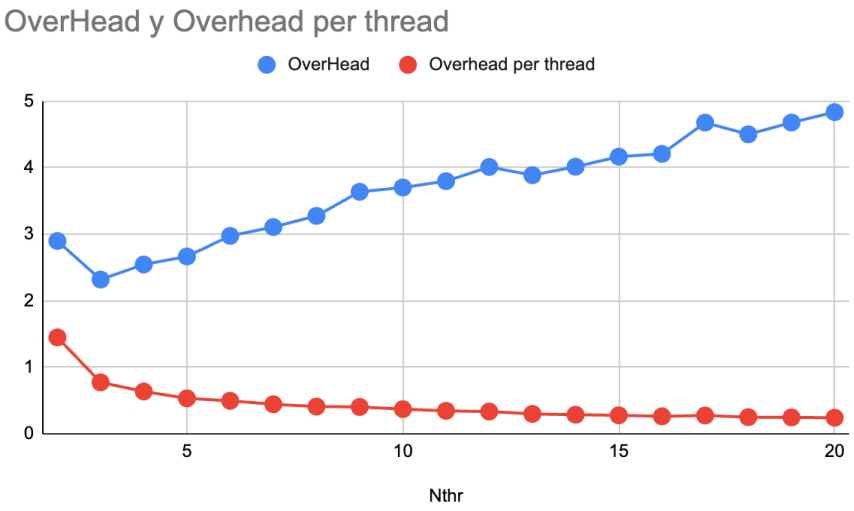

Fig.14. Gráfico de Overheads per thread y Overheads

Observamos en la gráfica (Fig.14.) que el tiempo total por overhead aumenta conforme se van añadiendo threads, esto se debe a que existe más paralelización y por lo tanto más necesidad de sincronización. Por otra parte, los overheads por thread disminuyen conforme se añaden más threads debido a que el trabajo por thread se reduce.

Task creation and synchronization

En este punto estudiaremos los overheads que tienen relación con la creación de tareas. Para ello estudiaremos el comportamiento del programa ***pi_omp_tasks.c***, ejecutándolo con diez iteraciones y con un thread. Esto nos ayudará a observar la evolución del overhead total y el overhead por tarea a medida que se van creando estas. Los datos obtenidos con dicha ejecución son los siguientes:

All overheads expressed in microseconds

Ntasks	*Overhead*	*Overhead per task*
2	0.2305	0.1153
4	0.4248	0.1062
6	0.6581	0.1097
8	0.8627	0.1078
10	1.0739	0.1074
12	1.2780	0.1065
14	1.4905	0.1065
16	1.6973	0.1061
18	1.9155	0.1064
20	2.1302	0.1065
22	2.3436	0.1065
24	2.5510	0.1063
26	2.7795	0.1069
28	2.9817	0.1065
30	3.1958	0.1065
32	3.4108	0.1066
34	3.6295	0.1068
36	3.8318	0.1064
38	4.0487	0.1065
40	4.2730	0.1068
42	4.4794	0.1067
44	4.6885	0.1066
46	4.9026	0.1066
48	5.1176	0.1066
50	5.3385	0.1068
52	5.5459	0.1067
54	5.7767	0.1070
56	5.9853	0.1069
58	6.1946	0.1068
60	6.4008	0.1067
62	6.6263	0.1069
64	6.8291	0.1067

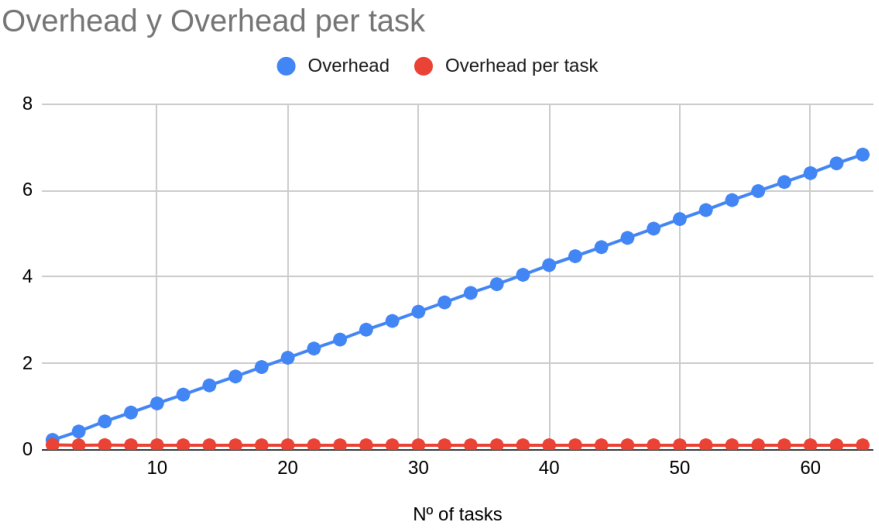

Fig.15. Gráfico de Overheads per task y Overheads

Como se deduce con los datos obtenidos y apreciando el gràfico (Fig.15.), se observa que a medida que se van creando tareas, el overhead total aumenta, que a diferencia con el apartado anterior, este lo hace de una manera más pronunciada, por otra parte como se puede ver claramente en el gràfico, el overhead por tarea se mantiene casi constante independientemente del número de tareas. La pequeña variación de este es debido a que la máquina puede estar ejecutando otros procesos que afecten sensiblemente a este tiempo.