Informe

Laboratori 2 de PAR

Fall 2021 - 2022

Berta Fitó Casas (*par2206*)

Marc Heras Villacampa (*par2210*)

07/10/2021 - 21/10/2021

Índex

1. A very practical introduction to OpenMP (I)

Data de compleció: 07/10/2021

Introducció:

Ambdues sessions del segon Laboratori de PAR, tal i com els seus títols (I i II) indiquen, tenien el mateix objectiu: introduir l'alumnat a la comprensió i ús de les extensions de la interfície *OpenMP* al llenguatge de programació C. Centrades en la lectura i correcció de codis del còmput del nombre Pi en paral·lel, presentaven un seguit d'exemples pràctics.

Procediment:

Abans que res, vam extreure del directori

> */scratch/nas/1/par0/sessions/**lab2.tar.gz***

tots els fitxers que necessitaríem al llarg de les dues següent sessions del Laboratori de PAR. Ho vam copiar al nostre directori i descomprimir mitjançant les comandes

> ***cp** ../par0/sessions/lab2.tar.gz **.***
>
> *tar -zxvf **lab2.tar.gz***

El procediment previ de preparació de la pràctica (accedir a *boada*, aplicar la comanda d'*environment*) es mantenen inalterats respecte les primeres tres sessions del Laboratori.

1.1. Computing number Pi

> En el directori proveït hi havia múltiples versions del codi del càlcul de Pi, implementant pas a pas *OpenMP* per a poder fer el càlcul en paral·lel. Els fitxers s'anomenaven
>
> > ***pi-vx**.c*
>
> Se'ns oferien dues opcions per executar-los:
>
> > *make pi-vx-**debug***
> >
> > *./run-**debug**.sh **pi-vx*** [n iteracions] [n *threads*]
>
> Aquesta primera consistia en una execució interactiva que mostrava les iteracions executades per cada *thread* i el càlcul resultant de Pi. Tenia un nombre d'iteracions per defecte de **32**, però admetia d'altres valors (petits). La segona opció era encuar:
>
> > *make pi-vx-**omp***
> >
> > *sbatch ./submit-**omp**.sh **pi-vx*** [n iteracions] [n *threads*]
>
> Aquest script permetia mesurar el temps d'execució del codi. Funcionava amb un nombre d'iteracions més elevat (per defecte, **100000000**), que també podia modificar-se. Ambdues opcions tenien adjudicades **4** *threads* com a paràmetre estàndard, també amb la possibilitat de ser reescrit.

Per tal de visualitzar els resultats amb *Paraver*, com ja vam fer en el Laboratori anterior, ens calia seguir el mateix procediment (més detalladament explicat al primer **Informe**) amb un script que adjudicava un nombre d'iteracions una mica menor a l'anterior (**100000**):

*sbatch ./submit-**extrae**.sh **pi-vx***

wxparaver

1.2. Parallelisation with OpenMP

S'explica com crear una regió paral·lelitzada i un conjunt de *threads* a executar les tasques implícites que s'hi generin (replicacions de les instruccions dins el codi tractat).

Els fitxers explicats i estudiats a continuació es trobaven al directori

*lab2/**pi***

1.2.1. Defining the parallel region and using implicit tasks

Al codi de **pi-v0**.c (versió inicial no paral·lelitzada del codi de càlcul de Pi) s'introdueix l'ús de la crida de mesura de temps d'execució

omp_get_wtime()

En les pròximes taules, obtindríem els temps d'execució després d'usar les comandes

*sbatch ./submit-**omp**.sh **pi-vx***

***more** submit-**omp.**sh.o-*[job_id]

Per de les execucions de **run-debug**.sh i **submit-omp.**sh vam obtenir:

script	**run-debug.sh**	**submit-omp.sh**
Resultat	3,1416740338	3,1415926536
Temps d'execució	*No es mostra*	0,397339106 s

Imatge 1. Taula de mètriques obtingudes amb l'execució de pi-v0.c

En visualitzar l'execució amb *Paraver* amb el procediment anterior i extreure'n el perfil d'estats vam veure que durant el 100% de l'execució només treballava la *thread* **1** i les altres esperaven a ser creades:

Imatge 2. Línia temporal de l'execució pi-v0

	Running	Not created	I/O	Others
THREAD 1.1.1	100.00 %	-	0.00 %	0.00 %
THREAD 1.1.2	-	100.00 %	0.00 %	-
THREAD 1.1.3	-	100.00 %	0.00 %	-
THREAD 1.1.4	-	100.00 %	0.00 %	-
Total	100.00 %	300.00 %	0.00 %	0.00 %
Average	100.00 %	100.00 %	0.00 %	0.00 %
Maximum	100.00 %	100.00 %	0.00 %	0.00 %
Minimum	100.00 %	100.00 %	0.00 %	0.00 %
StDev	0 %	0.00 %	0.00 %	0 %
Avg/Max	1	1.00	0.30	1

Imatge 3. Perfil de l'estat de les threads *en l'execució de pi-v0*

Amb aquesta informació podem veure que tot i demanar els recursos per a executar en 4 *threads* el codi, només s'executa en una perquè no s'ha programat la paral·lelització de cap regió del programa.

En repetir l'execució, els temps oscilaven entre els 0,25 i 0,45 segons, així que considerem que el temps correcte i acceptable és al voltant dels 0,3 segons.

La versió **pi-v1**.c introdueix

```
#pragma omp parallel
```

Amb aquest constructe s'adjudicaran a totes les *threads* que el programa utilitzi una tasca implícita executant la part del programa que segueixi la instrucció entre *{ }*. Tanmateix, com que *OpenMP* fa de totes les variables compartides per defecte, el codi estarà mal fet (la **i** del *loop* i **x** estan compartides) a no ser que es declari

```
#pragma omp private(llista de variables a privatitzar)
```

dins la regió paral·lela, o definint la pròpia variable dins la regió.

A **pi-v2**.c vam veure que aplicar el constructe private tant a **i** com a **x** no era l'estratègia a seguir: així, totes les *threads* assignades a l'execució (**run-debug**.sh) recorrien tot el *for*. La tercera versió del codi (**pi-v3**.c) cridava

```
omp_get_num_threads()
```

per poder distribuir les iteracions del *loop* entre les *threads* executant les tasques implícites:

```
int id = omp_get_thread_num();
int num_threads = omp_get_num_threads();

for (i=id; i < num_steps; i=i+num_threads) {
   [codi]
```

```
}
```

En executar el codi amb **run-debug**.sh, vam veure que la feina es dividia entre les *threads* d'aquesta manera:

Thread	Iteracions
0	0, 4, 8, 12, 16, 20, 24, 28
1	1, 5, 9, 13, 17, 21, 25, 29
2	2, 6, 10, 14, 18, 22, 26, 30
3	3, 7, 11, 15, 19, 23, 27, 31

Imatge 4. Repartiment d'iteracions per thread *a pi-v3.c*

El resultat, però, encara no era correcte; l'accés a la variable **id** provocava *data race* (més d'una *thread* accedeix a la mateixa dada, almenys una de les quals escrivint, sense estar regulats els accessos; depenent de l'ordre dels accessos, els resultats varien), cosa que es feia més aparent en executar el codi amb **submit-omp**.sh.

1.2.2. Using synchronisation to avoid data races

La versió **pi-v4**.c fa servir el constructe

```
#pragma omp critical
```

per crear una regió d'exclusió mútua (que només una *thread* pugui executar alhora). Ara ja es repartien correctament les 32 iteracions entre les *threads* **0**, **1**, **2** i **3** (cosa que es veu en el resultat de **run-debug**.sh), però amb un mètode (com es pot comprovar en l'execució de **submit-omp**.sh) terriblement ineficient, introduïnt grans *overheads* de sincronització.

El codi de **pi-v5**.c, en canvi, incorpora la línia

```
#pragma omp atomic
```

que manté l'execució en paral·lel a través del *hardware* de manera *atòmica* (és a dir, les operacions de *lectura-computació-escriptura* es tornen indivisibles). L'estratègia és encara *molt* ineficient, però millor que la de **pi-v4**.c (comparant l'execució de **submit-omp**.sh en ambdós):

Versió	Temps d'execució
pi-v4.c	37,447938204 s
pi-v5.c	5,927720070 s

Imatge 5. Taula comparativa de l'eficiència de les versions 4 i 5 del codi de Pi

La versió 6 (**pi-v6**.c) es considera més eficient, ja que manté els *atomics* o *criticals* fora el bucle. Defineix una variable privada (declarada dins la zona paral·lelitzada)

```
sumlocal = 0;
```

per a cada *thread*, que acumularà els seus resultats en acabar (fent ús dels constructes **atomic** o **critical**), contribuint al valor de la variable global **sum** (que guarda la suma dels valors parcials de la fórmula d'on es pot obtenir Pi al llarg de les iteracions del programa). En mesurar l'escalabilitat d'aquest codi respecte el seqüencial (**pi-v0**.c) vam comprovar que el seu temps d'execució era aproximadament un quart de l'original ($S_P = T_0/T_6 =$ **3,660706507**).

Versió	Temps d'execució
pi-v0.c	0,395701544 s
pi-v6.c	0,108094931 s

Imatge 6. Taula comparativa de l'eficiència de les versions seqüencial (0) i 6 del codi de Pi

A **pi-v7**.c s'incorpora

```
#pragma omp parallel private(i, x) reduction(+:sum)
```

cosa que permet fer desaparèixer el **critical** (o **atomic**) del final del bucle, ja que ho implementa d'una manera similar a un **for** en comparació a un **while**. Crea una còpia privada de la variable de reducció **sum**, que inicialitza al valor neutral especificat per l'operador (**0** sent en aquest cas l'operador **+**); també indica quina és l'operació a aplicar en tenir tots els valors parcials de **sum** computats (suma; **+**).

Versió	Temps d'execució
pi-v5.c	5,927720070 s
pi-v6.c	0,108094931 s
pi-v7.c	0,109068155 s

Imatge 7. Taula comparativa de l'eficiència de les versions 5, 6 i 7 del codi de Pi

La comanda de sincronització

```
#pragma omp barrier
```

afegida a la versió 8 (**pi-v8**.c) del codi serveix per fer que cap *thread* continuï avançant fins que totes les altres arribin a aquesta línia de codi. En executar **submit-omp**.sh vam veure que cada *thread*, després de la **barrier**, imprimia els valors parcials de Pi. Així, l'operació de reducció especificada al constructe **parallel** es donava:

```
#pragma omp parallel private(i, x) reduction(+:sum)
  {
    int id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();

    for (i=id; i < num_steps; i=i+num_threads) {
      x = (i+0.5)*step;
      sum += 4.0/(1.0+x*x);
```

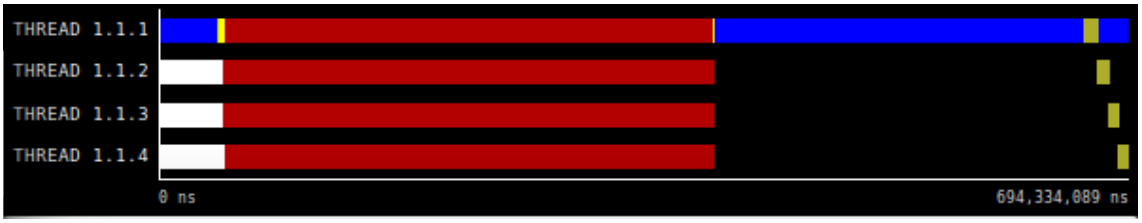
```
            #if _DEBUG_
            printf("thread id:%d it:%d\n",id,i);
            #endif
        }
        #pragma omp barrier
            printf ("Thread %d finished with the computation, partial value for
            pi=%12.10f\n", id, step*sum);
    }
```

Com es pot veure a l'anterior fragment del codi de **pi-v8**.c, en crear la regió paral·lelitzada ja s'aplicava l'operació de reducció. Això és perquè es paral·lelitza el càlcul de valors parcials de Pi, i després l'operació de suma establerta amb el signe **+** sobre la variable **sum** a

```
reduction(+:sum)
```

és executada pel constructe. Així, el resultat ja estarà computat quan arribi a l'últim estadi del càlcul de Pi, fora la regió paral·lelitzada.

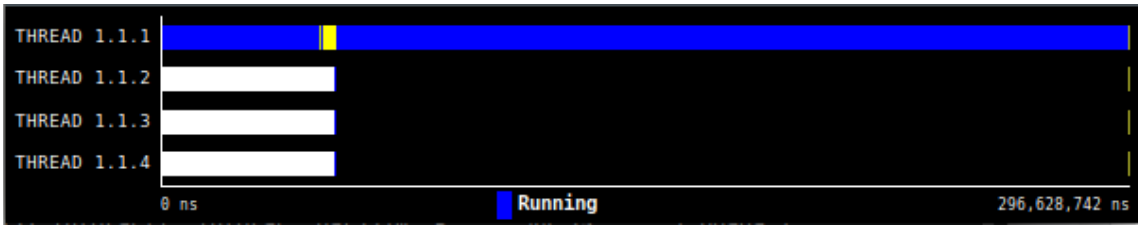
Per a una comparació més visual de la seva millora de rendiment, segueixen les execucions extretes de *Paraver* de les cinc versions de codi correctes.



Imatge 8. Línia temporal de l'execució de pi-v4.c

Imatge 9. Línia temporal de l'execució de pi-v5.c

Imatge 10. Línia temporal de l'execució de pi-v6.c



Imatge 11. Línia temporal de l'execució de pi-v7.c

Imatge 12. Línia temporal de l'execució de pi-v8.c

Es fa patent, observant l'evolució en els temps d'execució i divisió de tasques, que la versió 4 (**pi-v4**.c) és la menys eficient de les estudiades. A partir de **pi-v6**.c, però, la millora de rendiment fins la versió 8 (**pi-v8**.c) es fa molt menys significativa.

1.2.3. Summary of code versions

Version	Description of changes	Correct?
v0	Sequential code. Makes use of `omp_get_wtime` to measure execution time	yes
v1	Added `parallel` construct and `omp_get_thread_num()`	no
v2	Added `private` for variables `x` and `i`	no
v3	Manual distribution of iterations using `omp_get_num_threads()`	no
v4	Added `critical` construct to protect `sum`	yes
v5	Added `atomic` construct to protect `sum`	yes
v6	Private variable `sumlocal` and final accumulation on `sum`	yes
v7	Use of `reduction` clause on `sum`	yes
v8	Use of `barrier` construct	yes

Imatge 13. Llistat de canvis (acumulatius) en les diverses versions del codi de Pi

1.3. Test your understanding

Per accedir als fitxers l'estudi dels quals es demanava a continuació vam accedir al directori

*lab2/openmp/**Day1***

Tots els codis d'exemple s'havien de compilar fent ús del Makefile facilitat. Sempre s'executarien interactivament, afegint quan calgués una especificació del nombre de *threads* a emprar.

***OMP_NUM_THREADS=**X ./x.programa*

El procediment seguit per a cada pregunta de l'"Entregable" s'especifica a les respostes pertinents.

1.4. Observing overheads

1.4.1. Synchronisation overheads

Per últim, se'ns demanava estudiar quatre programes, accessibles al directori

*lab2/**overheads***

Les dades extretes per a la comparació dels quatre codis es recullen a continuació:

Codi	Característiques	Nombre d'operacions de sincronització (critical o atomic)
pi_omp_critical.c	Equivalent a **pi-v4**.c	1 critical
pi_omp_atomic.c	Equivalent a **pi-v5**.c	1 atomic
pi_omp_sumlocal.c	Equivalent a **pi-v6**.c	1 critical
pi_omp_reduction.c	Equivalent a **pi-v7**.c	0

Imatge 14. Taula informativa sobre les operacions de sincronització aplicades als codis

Compliant-los amb el Makefile facilitat, havíem de, primerament, comparar els *overheads* generats per encuar-los amb una *thread*, quatre i vuit i 100000000 iteracions

*sbatch ./submit-**omp**.sh **pi_omp**_**[codi]*** [n iteracions] [n *threads*]

amb l'execució de la versió seqüencial del mateix codi:

Codi	Overhead		
Nombre de threads	**1**	**4**	**8**
pi_omp_critical.c	2 479 969 us	36 115 549 us	34 571 980 us
pi_omp_atomic.c	10 458 us	5 039 188 us	5 754 777 us
pi_omp_sumlocal.c	10 566 us	11 389 us	22 709 us
pi_omp_reduction.c	4 825 us	11 404 us	20 110 us

*Imatge 15. Taula d'*overheads *per a cada codi amb 1* thread *i 100000000 iteracions*

L'*overhead* s'obtenia de la diferència resultant entre els T_{IDEAL} (T_{SEQ}/N *threads*) i T_{REAL}, facilitada per la funció **difference** dels codis.

```
double difference (long int num_steps, int n_threads) {
        double x, sum=0.0;
        double pi1, pi2;
        double step = 1.0/(double) num_steps;

        double stamp1=getusec_();
        for (int iter=0; iter<NUMITERS ; iter++) {
                sum = 0.0;
                for (long int i=0; i<num_steps; ++i) {
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
                }
```

```
                pi1 += num_steps * sum;
        }
        pi1 = step * sum;
        stamp1=getusec_()-stamp1;

        omp_set_num_threads(n_threads);
        double stamp2=getusec_();
        for (int iter=0; iter<NUMITERS ; iter++) {
                sum = 0.0;
                #pragma omp parallel private(x)
                {
                        int myid = omp_get_thread_num();
                        int howmany = omp_get_num_threads();
                        for (long int i=myid; i<num_steps; i+=howmany) {
                                x = (i+0.5)*step;
                                sum += 4.0/(1.0+x*x);
                        }
                }
        pi2 += step * sum;
        }
        stamp2=getusec_()-stamp2;
        return((stamp2-(stamp1/n_threads))/NUMITERS);
}
```

Al codi, **stamp1** es correspon al temps transcorregut durant l'execució seqüencial (no presenta cap regió paral·lelitzada) del primer *loop*, calculat amb la crida

```
getusec_()
```

al principi i el final del **for**, que desa en format *double* el temps transcorregut fins aleshores. Després, els resta per obtenir la diferència (i, per tant, la duració del bucle). El mateix procés se segueix amb **stamp2**, que computa el temps d'execució del bucle paral·lelitzat per a **n_threads** *threads* (cal mencionar que se li han extret les variances característiques de cada versió, que s'explicaran per a cadascuna). Finalment, s'opera l'equivalent a

$$\text{Overhead} = T_{REAL} - T_{SEQ}/(1 \text{ o } 4 \text{ o } 8)$$

fent una mitjana de tots els resultats per a les **NUMITERS** (definit a 100000000) iteracions executades. Així, s'obté l'*overhead* causat per la paral·lelització del segon *loop*.

En base a la taula a la ***Imatge 15***, vam poder observar un augment ineludible en l'*overhead* dels codis, a mesura que s'executaven amb més processadors. Això es deu al fet que les *threads* requereixen més temps de sincronització com més n'hi ha, entre d'altres factors (com ara una partició de tasques imperfecta) que poden allargar el temps d'execució d'un fragment de programa paral·lelitzat.

Tanmateix, es feia aparent amb les diferents implementacions l'*overhead* variava. En **pi_omp_critical**.c l'*overhead* era dramàticament superior, per a 1, però sobretot per

a 4 i 8 *threads*, respecte als altres codis. Afegia una sola modificació respecte l'anterior representació de la funció **difference**:

```
[...]
x = (i+0.5)*step;
#pragma omp critical
sum += 4.0/(1.0+x*x);
[...]
```

El constructe **critical** ja s'ha explicat que és considerat el mètode més lent per evitar el còmput de valors incorrectes en les variables compartides entre *threads*. Crea una zona d'exclusió mútua, expulsant de l'execució del fragment de codi que guarda a totes les *threads* menys una. Així, aquesta zona concreta del codi, en lloc d'optimitzar-la, la reconvertia a una execució seqüencial: una a una, les *threads* havien d'executar la suma iterativa de la variable **sum**, intentant totes alhora calcular el seu total.

Sabent que l'operació que resolien els quatre codis per calcular els respectius *overheads* era

$(T_{REAL} - T_{SEQ}/\textbf{n\_threads})/\textbf{NUMITERS}$

i sabent que

NUMITERS = **100 000 000**

El cost de l'operació **critical** vam determinar, per al mètode (que s'assumirà ídem per als següents codis)

2 479 969 / 100 000 000 = **0,025 us per cada iteració** en una *thread*

36 115 549 / 100 000 000 = **0,36 us per cada iteració** en 4 *threads*

34 571 980 / 100 000 000 = **0,35 us per cada iteració** en 8 *threads*

que era incremental en afegir múltiples *threads* a l'execució.

La regió paral·lelitzada de **pi_omp_atomic**.c era ja més eficient, tot i que encara, sobretot amb l'ús de 4 i 8 *threads*, veia un *overhead* molt alt. Afegia a la funció **difference**:

```
[...]
x = (i+0.5)*step;
#pragma omp atomic
sum += 4.0/(1.0+x*x);
[...]
```

Tal i com havíem après durant la sessió, era un mètode altament ineficient per assegurar que les *threads* no barregin els valors de les variables: evita que les operacions *lectura-computació-escriptura* es divideixin. Per a 1 *thread* encara presentava una millora substancial respecte **pi_omp_critical**.c, perquè no li calia esperar a que acabés cap altra conjunt atòmic d'instruccions; ara, en un entorn cada

cop paral·lelitzat amb més *threads* (4, 8), aquesta indivisibilitat aplicada al *hardware* feia que el temps d'execució de les tasques implícites anés en augment, generant quasi una seqüència d'operacions a fer, lenta i mal distribuida (forçant temps d'espera entre *threads*, tot i que no tant com en l'anterior versió, perquè encara podien anar alternant en acabar les operacions atòmiques), mentre es calculava el total de **sum**.

El cost de l'operació **atomic** el teoritzàrem a quasi negligible en una sola *thread* (aproximadament 0,1 **nano**segons per iteració), amb cost de

> 5 039 188 / 100 000 000 = **0,05 us per cada iteració** en 4 *threads*
>
> 5 754 777 / 100 000 000 = **0,06 us per cada iteració** en 8 *threads*

En canvi, **pi_omp_sumlocal**.c ja presenta un *overhead* notablement menor; sobretot per a 4 i 8 *threads*. El seu codi conté dues modificacions respecte la funció **difference** explicada a les pàgines 9 i 10:

```
[...]
#pragma omp parallel private(x) firstprivate(sumlocal)
[...]
        x = (i+0.5)*step;
        sumlocal += 4.0/(1.0+x*x);
}
#pragma omp critical
sum += sumlocal;
[...]
```

Ara, abans de comentar l'efecte del nou constructe, cal apuntar que

> #pragma omp critical

no es troba dins el bucle (tot i que encara dins al regió paral·lelitzada) que calculava el valor de **sum** (ara **sumlocal**): el *loop* ja no conté operacions de sincronització. Això es deu a l'aplicació de

> firstprivate(**sumlocal**)

que fa que cada *thread* generi una instància pròpia de **sumlocal**, una variable declarada fora la regió paral·lelitzada, inicialitzada al mateix valor que guardi l'original (ja la vam veure l'anterior Laboratori). Per tant, dins el *loop* cada *thread* anava calculant la suma parcial, partint del **0.0** a què estava inicialitzada la variable.

Seria en acabar el bucle que les *threads* editarien de manera bloquejant per evitar edicions en paral·lel de la variable **sum**, per tal de no perdre cap part de la suma del valor de Pi.

El funcionament d'aquesta part del codi es basava en com fèiem que es repartissin la feina del càlcul les diferents *threads*: en començar el *loop* **for** es determina des de quin valor **i** ha de començar a calcular cadascuna, amb un increment igual al nombre de *threads* en funcionament, processant, cadascuna d'elles, una porció del valor de

Pi calculat. En acabar, doncs, cadascuna de les *threads* haurà de sumar a la variable **sum** la seva aportació al càlcul.

Això sí, donat que s'obligava les *threads* a esperar la compleció de moltes menys zones d'exclusió mútues, els *overheads* disminuïen dràsticament.

El cost de l'operació critical, en aquest cas, el vàrem poder quantificar (en nanosegons, per mostrar els resultats de manera més clara) a

10 566 000 (ns) / 100 000 000 = **0,11 ns per iteració** en 1 *thread*

11 389 000 (ns) / 100 000 000 = **0,11 ns per iteració** en 4 *threads*

22 709 000 (ns) / 100 000 000 = **0,23 ns per iteració** en 8 *threads*

L'última optimització del codi es trobava a **pi_omp_reduction**.c, que ja no contenia cap operació de sincronització (ni **critical** ni **atomic**) i, tanmateix, presentava el menor dels *overheads* per a 1 *thread*, i uns resultats negligiblement divergents als obtinguts amb **pi_omp_sumlocal**.c. L'única modificació a la funció **difference** que contenia era:

```
[...]
#pragma omp parallel private(x) reduction(+:sum)
[...]
```

Com vam estudiar en aquesta mateixa sessió, aquest constructe permetia privatitzar la variable **sum** (de valor inicial **0.0**) i a través del seu identificador de reducció (**+**, en aquest cas), procedimentalment afegir a la variable el valor final de la privatització de **sum**. D'aquesta manera, s'assoleix el mateix que feia ja el codi antecessor a aquest (**pi_omp_sumlocal**.c) amb l'ús de **sumlocal**, però estalviant-se l'operació extra per passar el valor de la variable auxiliar a **sum**. D'aquí la davallada de l'*overhead* en l'execució amb 1 *thread* i la silimitud en 4 i 8 *threads*.

Com que no hi havia operacions de sincronització en **pi_omp_reduction**.c, no hi havia cap cost a quantificar.

Entregable:

Parallel regions and implicit tasks

1. *hello.c*

    a. *How many times will you see the "**Hello World!**" message if the program is executed with "**./1.hello**"?*

    El programa no té cap mena d'instrucció per a que no es dupliqui la tasca en executar-ho en múltiples *threads*, així que cal esperar que el missatge aparegui dos cops (per les dues *threads* del processador on s'executi).

```
int main ()
{
        #pragma omp parallel
```

```
            printf("Hello world!\n");

            return 0;
        }
```

Paral·lelitzant la regió de codi sense especificar l'ús d'una única thread per executar l'única instrucció escrita, es generen tantes tasques implícites (execució replicada de la regió paral·lelitzada per a cada *thread*) com *threads* se li adjudiquen al programa. En aquest cas, donat que s'adjudicaven dues *threads* a l'execució (interactiva), s'imprimiria dues vegades.

En fer córrer el programa, efectivament, el missatge apareixia dos cops.

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

A la terminal introduir la comanda

```
OMP_NUM_THREADS=4 ./1.hello
```

canvia el nombre de *threads* usades pel programa, imprimint el missatge 4 vegades.

2. *hello.c*

a. *Is the execution of the program correct? Add a data-sharing clause to make it correct.*

No. El programa paral·lelitza una regió de codi que hauria d'imprimir un cert nombre de "Hello World!"s especificant el número adjudicat a la *thread* (**id**) que l'ha executat entre parèntesis davant de cada mot (que s'escriuen per separat).

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

Com que no s'especifica la necessitat d'una sincronització per imprimir el missatge ordenadament, l'*output* no serà el correcte. Totes les *threads* llegiran el valor d'**id** alhora, que anirà canviant a mesura que cada tasca implícita s'executi, i escriuran el valor que trobin a cada moment. Per exemple, un dels *outputs* va ser:

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

Afegint la crida a

#pragma omp **private(id)**

es forçava un accés exclusiu per a cada *thread* de la variable **id** que li correspongués, de manera que, tot i seguir escrivint de manera desordenada els resultats (el llistat anava canviant, mai enumerant els missatges de l'1 al 7), cada tasca imprimiria estrictament el que se li demanés amb la **id** que li pertanyés.

b. *Are the lines always printed in the same order? Why do the messages sometimes appear intermixed?*

No. La llista va canviant a cada execució, mai seguint un ordre de 0 a 7.

(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!

Això es deu al fet que les vuit *threads* s'executen paral·lelament, i acabaran la tasca replicada (implícita) aproximadament alhora. L'ordre d'impressió, doncs, no serà necessàriament l'ordre correcte segons les **id** de les *threads*.

3. *how_many.c*

OMP_NUM_THREADS = 8 (***OMP_NUM_THREADS=8*** *./3.how_many*)

a. *What does **omp_get_num_threads** return when invoked outside and inside a parallel region?*

Retorna el nombre de *threads* que estan en execució dins la regió paral·lela, retornant 1 si la regió de codi és seqüencial (en una regió seqüencial tan sols hi ha una única *thread* en execució).

b. *Indicate the two alternatives to supersede the number of threads that is specified by the **OMP_NUM_THREADS** environment variable.*

La primera opció consisteix en especificar a la capçalera de les zones paral·lelitzades creades el nombre de *threads* que l'executaran.

```
#pragma omp parallel num_threads(X)
```

Ara, totes aquelles regions paral·lelitzades sense especificar la quantitat de *threads* a adjudicar-hi s'executaran amb les que es donin al codi en cridar el programa amb la consola

```
OMP_NUM_THREADS=Y ./3.how_many
```

Però aquelles amb el constructe **num_threads(**X**)** faràn ús d'**X** *threads*, fent cas omís d'**OMP_NUM_THREADS**.

La segona opció fa ús de la funció

```
omp_set_num_threads(Z);
```

que, precedint immediatament la capçalera d'una regió paral·lelitzada, determina quantes *threads* executaran tasques implícites mentre duri, ignorant també **OMP_NUM_THREADS**.

El resultat serà el següent:

```
Starting, I'm alone ... (1 thread)
Hello world from the first parallel (Y)!
Hello world from the first parallel (Y)!
Hello world from the first parallel (Y)!
Hello world from the szecond parallel (X)!
Hello world from the szecond parallel (X)!
Hello world from the szecond parallel (X)!
Hello world from the szecond parallel (X)!
Hello world from the third parallel (Y)!
Hello world from the third parallel (Y)!
Hello world from the third parallel (Y)!
Hello world from the fourth parallel (Z)!
Hello world from the fourth parallel (Z)!
Hello world from the fourth parallel (Z)!
Hello world from the fourth parallel (Z)!
Hello world from the fourth parallel (Z)!
Outside parallel, nobody else here ... (1 thread)
Hello world from the fifth parallel (X)!
Hello world from the fifth parallel (X)!
Hello world from the fifth parallel (X)!
Hello world from the fifth parallel (X)!
Hello world from the sixth  parallel (Y)!
Hello world from the sixth  parallel (Y)!
Hello world from the sixth  parallel (Y)!
Finishing, I'm alone again ... (1 thread)
```

On els **1** es corresponen a crides de omp_get_num_threads() fora d'una zona paral·lelitzada; les **Y** a crides de la funció des d'una zona paral·lelitzada sense especificar-ne les threads a usar (les determinades, doncs, per **OMP_NUM_THREADS**; el seu valor per defecte era 2); les **X**, valors marcats per **num_threads(**X**)**, i les **Z**, el resultat d'**omp_set_num_threads(**Z**)**.

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

num_threads(X**)** només afecta la regió paral·lela a la qual s'adjudica; **omp_set_num_threads(**Z**)** determinarà el nombre de *threads* usades en totes les regions paral·leles que la segueixin. Cal recalcar que:

1) **num_threads(**X**)** és un constructe adjunt a

```
#pragma omp parallel
```

mentre que **omp_set_num_threads(**Z**)** precedeix les capçalera, sent, doncs, una operació independent del constructe (i que en pot afectar més d'un).

2) En cas que entressin en conflicte, **num_threads(**X**)** prendria preferència, sobreescrivint la quantitat de *threads* que podria haver determinat **omp_set_num_threads(**Z**)**.

4. *data_sharing.c*

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

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**

b. *Is that the value you would expect?*

Al principi del codi es determina que totes les subseqüents regions paral·lelitzades s'executaran fent ús de 16 *threads*.

```
omp_set_num_threads(16);
```

Per a **shared**, el valor de **x** és equivalent al sumatori de les 16 etiquetes numèriques de les 16 *threads* executant el codi, perquè la variable és compartida entre totes les *threads* i hi aniran sumant el seu identificador:

```
0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15
```

Com que la segona regió paral·lelitzada fa d'**x** una variable privada per a cada *thread* (que en tindrà la seva pròpia instància), el valor que prengui dins la regió paral·lelitzada resultarà irrellevant (serien equivalents a les seves etiquetes). Fora la regió paral·lelitzada, el valor d'**x** serà el que s'havia establert prèviament, obviant les instàncies específiques de les *threads*. Per

a referència, aquest seria l'*output* del codi si es forcés una impressió de la progressió dels valors de **x** dins la regió paral·lelitzada:

After second (**PAR**) parallel (private) x is: **0**
After second (**PAR**) parallel (private) x is: **12**
After second (**PAR**) parallel (private) x is: **15**
After second (**PAR**) parallel (private) x is: **3**
After second (**PAR**) parallel (private) x is: **1**
After second (**PAR**) parallel (private) x is: **8**
After second (**PAR**) parallel (private) x is: **10**
After second (**PAR**) parallel (private) x is: **4**
After second (**PAR**) parallel (private) x is: **14**
After second (**PAR**) parallel (private) x is: **9**
After second (**PAR**) parallel (private) x is: **2**
After second (**PAR**) parallel (private) x is: **13**
After second (**PAR**) parallel (private) x is: **5**
After second (**PAR**) parallel (private) x is: **7**
After second (**PAR**) parallel (private) x is: **11**
After second (**PAR**) parallel (private) x is: **6**

After second parallel (private) x is: **5**

El constructe **firstprivate** fa que cada *thread* obtingui una instància pròpia d'**x**, i força que s'inicialitzi al valor que tenia fora la regió paral·lelitzada (**5**). Per tant, el valor que oferirà fora la regió paral·lelitzada serà el mateix que en **private** (**5**), però els valors que prendria per a cada iteració serien:

After third (**PAR**) parallel (private) x is: **5**
After third (**PAR**) parallel (private) x is: **18**
After third (**PAR**) parallel (private) x is: **6**
After third (**PAR**) parallel (private) x is: **7**
After third (**PAR**) parallel (private) x is: **19**
After third (**PAR**) parallel (private) x is: **8**
After third (**PAR**) parallel (private) x is: **9**
After third (**PAR**) parallel (private) x is: **15**
After third (**PAR**) parallel (private) x is: **10**
After third (**PAR**) parallel (private) x is: **14**
After third (**PAR**) parallel (private) x is: **12**
After third (**PAR**) parallel (private) x is: **13**
After third (**PAR**) parallel (private) x is: **11**
After third (**PAR**) parallel (private) x is: **16**
After third (**PAR**) parallel (private) x is: **17**
After third (**PAR**) parallel (private) x is: **20**

After third parallel (firstprivate) x is: **5**

És a dir, el resultat de

5 + [número identificador de la thread]

Finalment, el valor de la regió paral·lelitzada amb **reduction(+:x)** s'explica entenent que aquest constructe privatitzarà **x** per a cada *thread*, per finalment sumar al final totes les seves instàncies al valor original d'**x** (**5**). Així, s'obté el **125**:

```
120 + 5
```

5. *datarace.c*

   a. *Is the program executing correctly? Why?*

   El programa escrivia

   Program executed correctly - maxvalue=15 found

   cada vegada que l'executàvem, perquè havia aconseguit trobar el valor màxim del vector que estudiava. La variable **maxvalue** no és privada a la regió paral·lelitzada, i per tant totes les *threads* podran editar-la quan sigui adient.

   Amb la variable **i** privatitzada, però, només la *thread* **0** recorrerà tot el vector: la resta completaran un *loop* des d'una posició més avançada (la *thread* **1** obviarà el primer element i començarà la execució al segon; la **2**, començarà al tercer, i així fins la *thread* amb id **7**, ja que se'n definixen només 8).

   Tot i així, l'execució no és correcta, ja que **maxvalue** no es privatitza per editar-lo quan una *thread* pretén reescriure la variable amb el valor que està analitzant. El codi sembla executar-se correctament perquè el valor més alt que conté el vector (**15**) està intercalat tres vegades, fent que la majoria de les *threads* el tracti i, per tant, que sigui altament improvable que la solució acabi donant-se malament.

   b. *Propose two alternative solutions to make it correct, whithout changing the structure of the code (just add directives or clauses). Explain why they make the execution correct.*

   Una solució seria afegir la clàusula **critical** per bloquejar la variable **maxvalue** quan se'n vol editar el valor. Una altra opció seria afegir el mètode **reduction** amb l'identificador de reducció **max** per a la variable **maxvalue**, de manera que en igualar la instància privada de la variable **maxvalue** a **vector[i]**, si el privat és superior al públic, se substitueixi.

```
#pragma omp parallel private(i)
   {
      int id = omp_get_thread_num();
      int howmany = omp_get_num_threads();

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

```
        }
    }
```

o bé

```
#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];
      }
  }
```

La raó per la qual aquests canvis fan que ara l'execució sigui correcta és que, sense fer-los, el màxim real podria perdre's si hi ha una edició paral·lela de la variable i es queda com a valor final el que no és correcte. No vam observar tal cas perquè el valor màxim dins el vector està repetit tres vegades i és improbable que passi.

c. *Write an alternative distribution of iterations to implicit tasks (threads) so that each one of them executes only one block of consecutive iterations (i.e.: N divided by the number of threads).*

L'alternativa suggerida es pot aconseguir editant la condició del ***for*** de manera que quedi així:

```
for(i = id; i < (i+(N/howmany)); ++i)
```

6. *datarace.c*

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

No, ja que, com passava a l'apartat anterior, no hi ha cap mètode que protegeixi la variable **countmax** cada cop que una *thread* intenta editar-la.

b. *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.*

Les mateixes que en l'apartat *5.b* servirien:

Afegir la clàusula critical per protegir la variable **countmax** quan vol ser editada, o bé fer ús de la clàusula **reduction**, amb una implementació probablement confusa que sumi a 1 la variable **countmax** privada (inicialitzada prèviament a 0) per tal que es sumi automàticament en acabar la part paral·lela.

7. *datarace.c*

   a. *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**).*

      No, la variable **countmax** acaba l'execució amb valor **9** quan hauria de ser **3**. El problema en aquesta execució és que el recompte de **countmax** i la cerca de **maxvalue** no es poden fer alhora, ja que el valor màxim no és definitiu fins haver recorregut tot el vector i, per tant, se sumen altres valors que no són el major com si ho fossin.

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

      La manera de fer-ho correctament és separant la cerca del valor màxim i el recompte de vegades que surt:

```
[...]
#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];
      }
   }
}

#programa omp barrier

#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++;
   }
}
[...]
```

      És important posar la **barrier** entre ambdues parts de l'execució perquè no hi hagi cap *thread* que comenci a sumar a **countmax** sense seguretat que el valor màxim s'hagi trobat.

8. *barrier.c*

    a. *Can you predict the sequence of printf in this program? Do threads exit from the* ***#pragma omp barrier*** *construct in any specific order?*

    Teòricament no, ja que no hi ha un ordre preestablert de quina serà la *thread* que comenci primer, però sí que es pot saber que primer apareixeran tots els missatges de les *threads* “anant a dormir,” després els missatges de quan es “desperten” i entren a la barrera i, finalment, el de com acaben.

    La qüestió és que es podria donar el cas en que una *thread* es despertés abans que una altra imprimeixi que va a dormir, ja que no hi ha cap barrera que no ho permeti. Només sembla que aquest sigui un ordre obligat, ja que hi ha una diferència temporal substancial entre que s’imprimeix el missatge d’anar a dormir i el de despertar-se. En canvi, de segur que el missatge final serà escrit en bloc més o menys alhora per totes les *threads*, ja que així ho dictamina la **barrera**.

    Les línies individuals és impossible predir quina de les *threads* és la que la imprimirà primer, ja que depèn de la distribució dels recursos del sistema.

2. A very practical introduction to OpenMP (II)

Data de compleció: 14/10/2021

Introducció:

La segona sessió d'aquest Laboratori era una continuació de la primera. Usant un model de tasques, encara s'analitzava l'execució d'un seguit de versions del codi del càlcul de Pi. Per acabar, s'estudiaven els *overheads* relacionats amb la creació de regions paral·leles i tasques a *OpenMP*.

Procediment:

2.1. Parallelisation with OpenMP

Les **tasques explícites** són un mètode molt més versàtil d'expressar el paral·lelisme a *OpenMP*. Es generen seguint els passos:

1) Seleccionar una de les *threads* executant una **tasca implícita**, que serà l'encarregada de generar les **tasques explícites** que executaran altres *threads*.
2) Sincronitzar mitjançant *task barriers* les **tasques explícites**.

El procediment per executar les versions del codi de Pi és ídem al de la sessió anterior: mitjançant el Makefile, calia compilar

*make pi-vx-**debug***

*make pi-vx-**omp***

I executar els binaris corresponents (modificant-ne els valors preestablerts si calia)

*./run-**debug**.sh **pi-vx*** [n iteracions] [n threads]

*sbatch ./submit-**omp**.sh **pi-vx*** [n iteracions] [n threads]

Per visualitzar-ne els resultats amb Paraver, l'ús d'*Extrae* era el mateix:

*sbatch ./submit-**extrae**.sh **pi-vx***

wxparaver

Al codi s'establia **32** com el valor per defecte d'iteracions en **debug** i **100000000** en **omp**; i **4** per al nombre de processadors usats en ambdós. Per a **extrae** les iteracions es reduïen a **100000**.

2.1.1. Tasking execution model and use of explicit tasks

La primera versió del codi a estudiar era **pi-v9**.c, que divideix el codi en dos *loops* iterant cadascun la meitat de les iteracions. Per cada *loop* es defineix una **tasca explícita**, utilitzant el constructe

#pragma omp **task**

que genera una tasca del fragment especificat (en aquest cas, un dels *loops*) i l'afegeix a una unitat de computació diferida, un conjunt d'on qualsevol *thread* pot extreure una tasca a executar. Cal notar que al codi apareixen

```
#pragma omp task private(i, x)
```

fent de les variables **i** i **x** privades per a cada *thread*. Així, en executar-lo amb **submit-omp**.sh quedava patent que cada iteració s'executava quatre vegades i que el valor de Pi no era correcte:

```
Wall clock execution time = 3.585220098 seconds
Value of pi = 0.0001396456
```

El que passa és que no es limita que la regió paral·lelitzada de la tasca l'executi una sola *thread* i per tant, les 4 threads l'anuncien com a tasca explícita per a que alguna altra la processi. Conseqüentment, tant la primera meitat dels càlculs com la segona es processen **quatre** cops cadascuna.

Així, la següent versió del codi (**pi-v10**.c) soluciona aquest problema amb l'ús del constructe

```
#pragma omp simple
```

Limitant d'aquesta manera el nombre de *threads* que generen tasques a una; mentre s'esperin a que acabi el constructe, les altres pararan atenció al conjunt de tasques a executar. El resultat d'executar **run-debug**.sh

```
thread id:0 it:0
thread id:0 it:1
thread id:0 it:2
thread id:0 it:3
thread id:0 it:4
thread id:0 it:5
thread id:0 it:6
thread id:0 it:7
thread id:0 it:8
thread id:0 it:9
thread id:0 it:10
thread id:0 it:11
thread id:0 it:12
thread id:0 it:13
thread id:0 it:14
thread id:0 it:15
thread id:2 it:16
thread id:2 it:17
thread id:2 it:18
thread id:2 it:19
thread id:2 it:20
thread id:2 it:21
thread id:2 it:22
```

```
thread id:2 it:23
thread id:2 it:24
thread id:2 it:25
thread id:2 it:26
thread id:2 it:27
thread id:2 it:28
thread id:2 it:29
thread id:2 it:30
thread id:2 it:31
Value of pi = 0.0000000000
```

era incorrecte, perquè tot i distribuir correctament les iteracions entre les *threads* disponibles, el valor de Pi no quedava modificat corresponentment.

El codi a **pi-v11**.c dues modificacions:

1) Per tal d'obtenir el valor correcte de Pi cal assegurar-se que les tasques generades per la *thread* acaben abans d'arribar a l'operació

   ```
   pi = step * sum;
   ```

   del càlcul de **pi**; sense tenir-ne el resultat final, el còmput de Pi no podrà ser fiable. Això requerirà sincronització de tasques.

2) Caldrà incloure, també, una clàusula com les usades en les versions 4, 5 o 7 per tal de poder fer les contribucions pertinents per part de cadascuna de les *threads* a la variable **sum**. Caldrà, doncs, escollir entre *critical, atomic* o *reduction*, la última de les quals era la més eficient, i per tant, la destriada d'entre les tres. **pi-v11.c** també inclou les instruccions:

   ```
   #pragma omp taskgroup

   #pragma omp task_reduction

   #pragma omp in_reduction
   ```

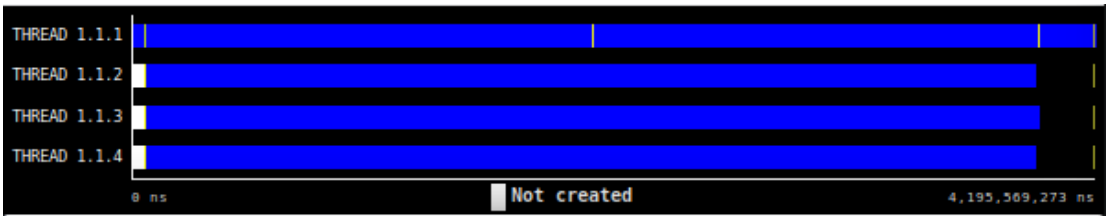
   La primera serveix com a barrera al final de la tasca per a que s'esperi a que les altres *threads* acabin la execució de la mateixa regió de codi abans de continuar, així com la implementació de la suma parcial d'ambdues tasques generades gràcies a **task_reduction**.

Obtenim els resultats de les execucions:

script	**run-debug.sh**	**submit-omp.sh**
Resultat	3,1415926536	3,1415925436
Temps d'execució	*No es mostra*	0,20547587 s

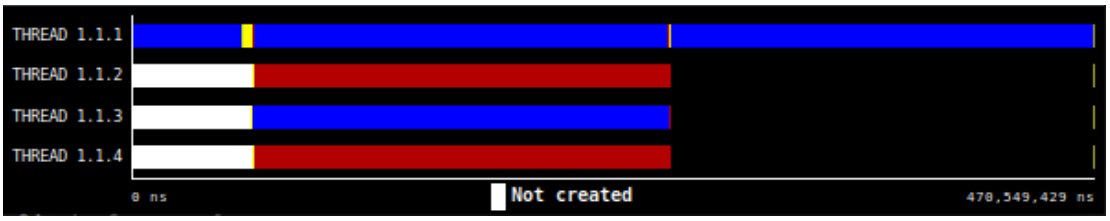
Imatge 16. Taula de mètriques obtingudes amb l'execució de pi-v11.c

El resultat coincideix, de moment, amb el considerat correcte per Pi amb només els decimals mostrats en pantalla, **3,1415926536**. El temps d'execució de la versió 11 és molt baix comparat amb altres que veurem més endavant; ja és bastant eficient *per se*, però hi ha una millor configuració.



Imatge 17. Línia temporal de l'execució de pi-9.c

Imatge 18. Línia temporal de l'execució de pi-v10.c



Imatge 19. Línia temporal de l'execució de pi-v11.c

La versió **pi-v12**.c ofereix una alternativa basada en l'ús del constructe

```
#pragma omp atomic
```

per protegir l'actualització de la variable compartida sum. Per tal d'esperar la terminació de les tasques inclou

```
#pragma omp taskwait
```

que força la *thread* dins la regió **single** (la que ha creat les dues tasques) a esperar la seva terminació. Només aleshores usarà la variable global **sum** (que les tasques hauran computat).

script	**run-debug.sh**	**submit-omp.sh**
Resultat	3,1415926536	3,1415926536
Temps d'execució	*No es mostra*	7,936388969 s

Imatge 20. Taula de mètriques obtingudes amb l'execució de pi-v12.c

Com es pot veure, les operacions de sincronització afegides encara generaven un *overhead* considerable: a l'anterior sessió ja ens havien avisat que **atomic** era altament ineficient (tot i que **critical** hauria suposat un pitjor temps).

El codi de **pi-v13** introdueix el constructe

```
#pragma omp task depend(in o out o inout: llista)
```

usat per especificar si les variables estan:

1) **in**: la tasca dependrà de tota la resta de tasques que referenciïn almenys un dels elements a la llista dins la clàusula **out** o **inout**.
2) **out**: la tasca dependrà de tota la resta de tasques que referenciïn almenys un dels elements a la llista dins la clàusula **in** o **out** o **inout**.
3) **inout**: ídem a **out**.

Així, OpenMP pot aproximar el temps d'execució de totes les tasques, per evitar que cap s'executi quan les seves dependències estiguin completes. En aquest cas, la tercera tasca

```
#pragma omp task depend(in: sum1, sum2)
sum += sum1 + sum2;
```

espera que acabin les altres dues, reemplaçant

```
#pragma omp taskgroup

#pragma omp task_reduction

#pragma omp in_reduction
```

així com

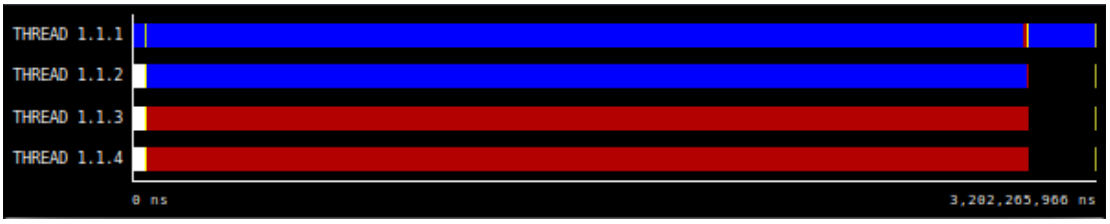
```
#pragma omp atomic

#pragma omp taskwait
```

respectivament, de **pi-v11**.c i **pi-v12**.c.

script	**run-debug.sh**	**submit-omp.sh**
Resultat	3,1415926536	3,1415926536
Temps d'execució	*No es mostra*	2,753843069 s

Imatge 21. Taula de mètriques obtingudes amb l'execució de pi-v13.c

L'overhead d'aquesta versió ja és molt menor que en la anterior, però molt considerable encara. De fet, el temps d'execució encara és més de 10 vegades superior al de la versió 11. Calen encara algunes modificacions per rebaixar-lo significativament.



Imatge 22. Línia temporal de l'execució de pi-v13.c

La versió **pi-v14**.c és l'última versió. Pren el constructe

```
#pragma omp taskloop
```

per generar una tasca per a un cert nombre d'iteracions consecutives, controlades amb una de les següents clàusules:

```
#pragma omp taskloop num_tasks(4)
```

que especifica el nombre de tasques a generar o

```
#pragma omp taskloop grainsize(X)
```

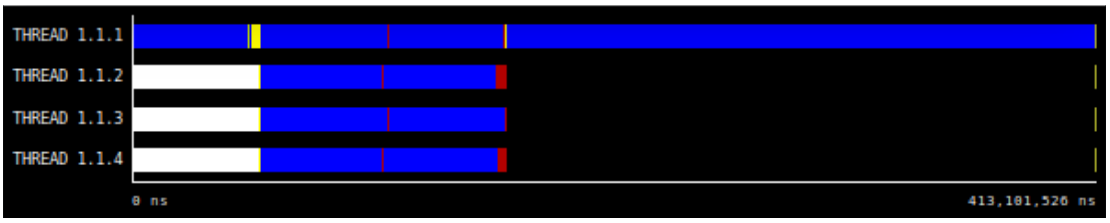
que controla el nombre d'iteracions consecutives per tasca.

El comportament d'ambdós constructes s'il·lustra amb més profunditat a l'apartat "Entregable" (a resultes de l'estudi del codi de **3.taskloop.c**).

script	**run-debug.sh**	**submit-omp.sh**
Resultat	3,1415926536	3,1415659236
Temps d'execució	*No es mostra*	0,110300779 s

Imatge 23. Taula de mètriques obtingudes amb l'execució de pi-v14.c

Mesurant l'escalabilitat amb **submit-omp**.sh vam poder comprovar que era el codi més eficient, gairebé un 50% més ràpid que el segon més eficient i quasi 4 vegades més veloç que el seqüencial.



Imatge 24. Línia temporal de l'execució de pi-v14.c

Entre les versions 11 i 13 veiem una diferència substancial en les gràfiques que ens proporciona Paraver: **tot i que** sembla que la versió 13 paral·lelitza més porció del programa, això és fals, ja que tan sols passa que la part paral·lela té un *overhead* molt més elevat que la versió 11 i en conseqüència, sembla que la part seqüencial sigui menor.

En el que s'assemblen, però, és que ambdós tenen dues *threads* que semblen esperar a sincronitzar dades durant molta estona, que comparant amb la versió 14, no es dona. Són aquestes regions les que van en contra de l'eficiència del programa, i la raó per la que a la versió 14 es veu una millora tan important.

2.1.2. Summary of code versions

Version	Description of changes	Correct?
v9	Use of `task` construct	no
v10	Use of `single` construct to have a single task generator	no
v11	Use of `taskgroup` and reductions (`task_reduction` and `in_reduction`)	yes
v12	Use of `taskwait` and `atomic`	yes
v13	Use of `task` with dependences (`depend` clause)	yes
v14	Use of `taskloop` to generate tasks from loop iterations	yes

Imatge 25. Llistat de canvis (acumulatius) en les diverses versions del codi de Pi

2.2. Test your understanding

Per accedir als fitxers a estudiar vam accedir a

*/lab2/openmp/**Day2***

Tots els codis d'exemple s'havien de compilar fent ús del Makefile facilitat. Sempre s'executarien interactivament. El procediment seguit per a cada qüestió de l'"Entregable" s'especifica a les respostes pertinents.

2.3. Observing overheads

Per mesurar els *overheads* relacionats amb la creació de regions paral·leles i sincronització s'analitzaren els resultats dels codis disponibles a

*.../lab2/**overheads***

2.3.1. Thread creation and termination

El codi **pi_omp_parallel**.c permetia calcular la diferència entre el temps d'execució seqüencial i l'execució paral·lela usant un cert nombre de *threads*, efectivament quantificant l'*overhead* generat pel constructe

#pragma omp **parallel**

S'aconseguia amb la funció **difference** (idèntica a la de l'anterior sessió) que s'executa **NUMITERS** vegades per fer una mitjana d'ambdós temps d'execució (seqüencial i paral·lel, aconseguits canviant el nombre de threads assignades al codi amb la funció intrínseca marcada al fragment anterior). Al *main* s'itera sobre un nombre de threads (**max_threads**) determinat per l'usuari com a argument.

Compilant el codi amb el Makefile facilitat i executant iterativament el binari generat

make **pi_omp_parallel**

./submit-omp.sh **pi_omp_parallel 1 24**

per a una (**1**) iteració i vint-i-quatre (**24**) threads màximes s'obtenien els resultats:

codi	**pi_omp_parallel.c**
Overhead	3,5408 us
Overhead per tasca	0,1475 us

Imatge 26. Resultats de l'execució de pi_omp_parallel.c per a 24 threads

És destacable com per una sola iteració ja és observable un overhead de més de 3 microsegons. Suposa un temps d'espera substancial, per això és important tenir algoritmes molt eficient si pretenem executar programes en molts *threads* alhora.

Executant mateix programa per a un límit de 64 *threads* obtenim el següent:

Nombre de *threads*	**Overhead**
2	1,6265 us
4	1,7652 us
8	2,3717 us
16	3,1810 us
32	12,3863 us
64	21,9070 us

Imatge 27. Overheads de l'execució de pi_omp_parallel.c per a N threads

L'ordre de magnitud per l'*overhead* de crear i terminar una *thread* individual era d'un valor proper a **0,31 us** (la mitjana dels valors obtinguts en el bolcat d'*overheads* per tasca de 1 a 64 *threads*).

2.3.2. Task creation and synchronisation

El fitxer **pi_omp_tasks**.c creava tasques dins la funció **difference** mitjançant una sola *thread*. Tal i com feia el codi explicat a l'anterior subapartat, mesurava la diferència entre els temps d'execució seqüencial i paral·lelitzat per a un cert nombre de threads, iterant **NUMITERS** vegades per trobar una mitjana en els valors.

Al main hi ha un *loop* que itera entre el número mínim (**MINTASKS**) i màxim (**MAXTASKS**) de tasques que es volguessin generar.

La diferència que s'obtenia del càlcul era l'overhead introduït pels constructes

```
#pragma omp task
```

```
#pragma omp taskwait
```

Usant la target apropiada del Makefile vam encuar el binari generat

```
make pi_omp_tasks
```

```
sbatch ./submit-omp.sh pi_omp_tasks 10 1
```

per a deu (**10**) iteracions i (**1**) *thread*. Altre cop, el valor de Pi no era correcte.

codi	**pi_omp_tasks.c**
Overhead	1,1828 us
Overhead per thread	0,1183 us

Imatge 28. Resultats de l'execució de pi_omp_tasks.c per a 1 thread

La fluctuació observada en l'*overhead* generat per la creació i sincronització de tasques fina a 64 *threads* fou:

Nombre de tasques	**Overhead**
2	0,3065 us
4	0,4977 us
8	0,9501 us
16	1,8998 us
32	3,7755 us
64	7,4786 us

Imatge 29. Overheads de l'execució de pi_omp_parallel.c per a N threads

Així, l'ordre de magnitud per l'*overhead* de crear i sincronitzar cada tasca individual individual era, en mitjana, **0,1083 us**.

Entregable:

Explicit tasks

9. *single.c*

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

    La clàsula

    ```
    #pragma omp single nowait
    ```

    permet que les *threads* se saltin la barrera implícita al final d'una regió paral·lelitzada amb **single** que ha especificat que la regió delimitada només l'executi com a tasca implícita una de les *threads* (no necessàriament la principal).

    b. *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?*

    Amb **nowait**, la *thread* individual executa la primera iteració del bucle. Les altres (en aquest cas, en total són **4**)

```
omp_set_num_threads(4);
```

se la saltaran (donat que ja no tenen la restricció de la barrera que les feien esperar a que acabés l'execució la *thread* individual) i arribaran immediatament a la segona iteració, a la qual, en ser una nova ocurrència del bloc que estan tractant, serà adjudicat a alguna de les 3 *threads*. Es repartiran així la feina de manera pràcticament concurrent; les *threads* aniran executant iteracions alternatives del *loop* fins que acabi.

```
Thread 0 executing instance 0 of single
Thread 2 executing instance 2 of single
Thread 1 executing instance 1 of single
Thread 3 executing instance 3 of single
Thread 0 executing instance 5 of single
Thread 1 executing instance 4 of single
Thread 3 executing instance 6 of single
Thread 2 executing instance 7 of single
Thread 0 executing instance 8 of single
Thread 1 executing instance 9 of single
Thread 3 executing instance 10 of single
Thread 2 executing instance 11 of single
Thread 0 executing instance 13 of single
Thread 2 executing instance 15 of single
Thread 3 executing instance 14 of single
Thread 1 executing instance 12 of single
Thread 2 executing instance 13 of single
Thread 1 executing instance 18 of single
Thread 0 executing instance 17 of single
Thread 3 executing instance 19 of single
```

Aquest comportament queda il·lustrat per aquest resultat de l'execució de codi.

En executar el codi, sembla que vagi a batzegades perquè cada iteració del bucle conté la instrucció

```
sleep(1);
```

que pausarà l'execució de la *thread* almenys durant el temps especificat (**1** segon); tanmateix, la CPU pot seguir enviant *threads* a fer tasques implícites, de manera que al final es mostrarà la impressió de les quatre iteracions que ocupin les quatre *threads* quasi alhora (sempre en grups de 4).

10. fibtasks.c

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

Tot i que el codi especifica el nombre de *threads* a assignar a l'execució del codi

```
omp_set_num_threads(6);
```

i conté el constructe

```
#pragma omp task firstprivate(p)
```

per crear les tasques a executar al llarg de l'execució i privatitzar **p** amb el valor que tenia fora la regió paral·lelitzada, l'operació per realment *crear* la regió paral·lelitzada per a la qual les anteriors línies es preparen brilla en la seva absència.

Donat que en cap moment es paral·lelitza cap part del codi, **5** de les **6** *threads* (aquelles a les quals no s'ha adjudicat el paper principal o, pel mateix preu, cap funció) no faran cap de les tasques creades, perquè l'execució és seqüencial.

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

Paral·lelitzar un **while** no és una acció que OpenMP recolzi, *a priori*. A diferència dels **for**, que tenen un espai iteratiu preestablert, els **while** (així com els **do** amb sortides condicionals) poden tenir problemes de dependències. Per exemple, en paral·lelitzar les tasques podrien iterar sobre més valors dels que recorrerien en una execució seqüencial, perquè la condició a assolir seria l'única a comprovar-se per a cada iteració, en lloc d'estar-ne comprovant múltiples alhora, que aleshores ja serien inservibles.

En el nostre cas, però, el **while** que el codi presentava era paral·lelitzable: contenia tots els elements necessaris abans del primer valor **NULL** (que marcava el final de la llista) i, per tant, no podia fer cap iteració innecessària.

Només es mostra el fragment modificat del codi

```
#pragma omp parallel
#pragma omp single nowait
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;
 }
```

que genera l'*output*

```
Staring computation of Fibonacci for numbers in linked list
Thread 0 creating task that will compute 1
Thread 0 creating task that will compute 2
Thread 0 creating task that will compute 3
Thread 0 creating task that will compute 4
Thread 0 creating task that will compute 5
```

Thread **0** creating task that will compute 6
Thread **0** creating task that will compute 7
Thread **0** creating task that will compute 8
Thread **0** creating task that will compute 9
Thread **0** creating task that will compute 10
Thread **0** creating task that will compute 11
Thread **0** creating task that will compute 12
Thread **0** creating task that will compute 13
Thread **0** creating task that will compute 14
Thread **0** creating task that will compute 15
Thread **0** creating task that will compute 16
Thread **0** creating task that will compute 17
Thread **0** creating task that will compute 18
Thread **0** creating task that will compute 19
Thread **0** creating task that will compute 20
Thread **0** creating task that will compute 21
Thread **0** creating task that will compute 22
Thread **0** creating task that will compute 23
Thread **0** creating task that will compute 24
Thread **0** creating task that will compute 25
Finished creation of tasks to compute the Fibonacci for numbers in linked list
Finished computation of Fibonacci for numbers in linked list
1: 1 computed by thread **3**
2: 1 computed by thread **1**
3: 2 computed by thread **4**
4: 3 computed by thread **3**
5: 5 computed by thread **1**
6: 8 computed by thread **1**
7: 13 computed by thread **2**
8: 21 computed by thread **2**
9: 34 computed by thread **4**
10: 55 computed by thread **2**
11: 89 computed by thread **4**
12: 144 computed by thread **1**
13: 233 computed by thread **2**
14: 377 computed by thread **4**
15: 610 computed by thread **1**
16: 987 computed by thread **3**
17: 1597 computed by thread **2**
18: 2584 computed by thread **4**
19: 4181 computed by thread **1**
20: 6765 computed by thread **3**
21: 10946 computed by thread **2**
22: 17711 computed by thread **4**
23: 28657 computed by thread **1**
24: 46368 computed by thread **0**
25: 75025 computed by thread **0**

És a dir: la *thread* **0** genera totes les tasques, que després són executades paral·lelament per les *threads* **0**, **1**, **2**, **3** i **4**.

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

La clàusula **firstprivate(p)** fa que cada *thread* obtingui una instància privada de la variable **p**, inicialitzada per al valor que tenia abans d'obrir la zona paral·lelitzada. Comentant-lo, s'obté l'*output*:

```
Staring computation of Fibonacci for numbers in linked list
Thread 4 creating task that will compute 1
Thread 4 creating task that will compute 2
Thread 4 creating task that will compute 3
Thread 4 creating task that will compute 4
Thread 4 creating task that will compute 5
Thread 4 creating task that will compute 6
Thread 4 creating task that will compute 7
Thread 4 creating task that will compute 8
Thread 4 creating task that will compute 9
Thread 4 creating task that will compute 10
Thread 4 creating task that will compute 11
Thread 4 creating task that will compute 12
Thread 4 creating task that will compute 13
Thread 4 creating task that will compute 14
Thread 4 creating task that will compute 15
Thread 4 creating task that will compute 16
Thread 4 creating task that will compute 17
Thread 4 creating task that will compute 18
Thread 4 creating task that will compute 19
Thread 4 creating task that will compute 20
Thread 4 creating task that will compute 21
Thread 4 creating task that will compute 22
Thread 4 creating task that will compute 23
Thread 4 creating task that will compute 24
Thread 4 creating task that will compute 25
Segmentation fault
```

Com que sense el **firstprivate(p)** valor de **p** no esdevé una instància específica per a cada *thread* executant una tasca, l'emmagatzematge de la variable queda compartida. Quan passa això, s'hauria d'assegurar que hi ha una sincronització adequada entre les tasques, per evitar que les dades emmagatzemades no acabin abans no s'arribi al final de l'execució de la tasca explícita, cosa que aquí passa.

11. taskloop.c

a. *Which iterations of the* loops *are executed by each* thread *for each task **grainsize** or **num_tasks** specified?*

El constructe

```
#pragma omp taskloop grainsize(4)
```

determina que el nombre d'iteracions del bucle assignades a cada tasca generada és major i igual al valor mínim introduït a **grainsize** i el d'iteracions lògiques del *loop* (aquí **12**), però menys que el doble del valor dins el constructe (**8**).

Per tant, en aquest cas (tal i com es pot veure al següent output obtingut de l'execució del codi), a la *thread* **0** se li han adjudicat **8** iteracions (el màxim possible) i a la **1**, **4**.

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

En el constructe

```
#pragma omp taskloop num_tasks(4)
```

en canvi, es creen tantes tasques com el mínim entre l'especificat a la clàusula (**4**) i el nombre d'iteracions lògiques del *loop* (**12**). Cada tasca, com a mínim, ha de tenir una iteració lògica.

La segona part de l'anterior output així ho il·lustra:

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

Les **12** iteracions es divideixen en **4** tasques a executar per les **3** de les **4** *threads* adjudicades al codi

```
omp_set_num_threads(4);
```

Així, la *thread* **0** rep **1** tasca (9, 10, 11), la *thread* **1**, **1** més (6, 7, 8) i la *thread* **2**, les **2** restants (0, 1, 2 i 3, 4, 5).

En cap dels dos constructes s'especifica l'ordre d'execució de les tasques.

b. *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?*

Per al constructe

```
#pragma omp taskloop grainsize(5)
```

el mínim d'iteracions lògiques a assignar per *thread* queda alterat (ara és **5** i no **4**). En conseqüència, la divisió d'iteracions per *thread* canvia; en aquest cas, a **6** iteracions atorgades a, respectivament a **0** i **1**.

```
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
```

En canvi, per a

```
#pragma omp taskloop num_tasks(5)
```

caldrà que les iteracions es divideixin en **5** tasques. En aquest cas:

```
Loop 2: (1) gets iteration 0
Loop 2: (1) gets iteration 1
Loop 2: (1) gets iteration 2
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (2) gets iteration 3
Loop 2: (2) gets iteration 4
```

Loop 2: (**2**) gets iteration 5
Loop 2: (**0**) gets iteration 8
Loop 2: (**0**) gets iteration 9

A la *thread* **0** se li assignen **2** tasques (10, 11 i 8, 9), a la **1** se n'hi donen **2** més (0, 1, 2 i 6, 7) i a la *thread* **2** n'hi pertoca **1** (3, 4, 5). Cal recalcar que es podran dividir les iteracions de qualsevol manera i en qualsevol ordre, sempre que totes les tasques almenys continguin una iteració lògica en la seva execució.

c. *Can **grainsize** and **num_tasks** be used at the same time in the same* loop*?*

No, les clàusules **grainsize** i **num_tasks** són mútuament excloents i, per tant, no es poden utilitzar al mateix *loop*.

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

Es dona el següent *output*:

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

La clàusula **nogroup** evita que es creï la regió de **taskgroup** implícita que altrament s'establiria en cridar el constructe

```
#pragma omp taskloop
```

Per tant, les tasques explícites creades ja no estaran incloses en una regió específica, i sense la barrera entre la seva execució i la de qualsevol altra part del codi la presentació dels seus resultats esdevé imprevisible. Tot i que en aquest exemple les dues capçaleres s'han escrit abans que res, aquest no serà sempre el cas: qualsevol fragment del **Loop 1** (el que ara està marcat pel **nogroup**) podrà aparèixer abans que les impressions.

12. reduction.c

a. *Complete the parallelisation of the program so that the correct value for variable sum is returned to each printf statement.*

Fent el càlcul demanat a mà, a fi de determinar quin valor calia que **sum** presentés al final de l'execució, vam trobar que la primera part ja oferia el resultat correcte:

$$\sum_{0}^{8191} i = 33\,550\,336$$

Per tant, per al primer fragment de codi no vam proposar cap modificació: a la variable ja se li havia aplicat el constructe

```
#pragma omp task firstprivate(i) in_reduction(+: sum)
```

Per tant, **i** ja estava privatitzada i inicialitzada al valor que tenia fora la regió paral·lelitzada i **sum** també estava protegida per **in_reduction**, que determina que la tasca específica és part del **taskloop** (prèviament establert) i també s'encarregaria de fer el sumatori final.

La segona part la vam canviar a

```
#pragma omp taskloop grainsize(BS) reduction(+:sum)
```

El valor original que imprimia era 67 009 468, clarament incorrecte, però després de la modificació, el valor passa a ser el correcte (33 550 336).

A més, vam afegir un

```
sum = 0;
```

Per a que es reiniciés el valor de **sum** i així la Part 2 no acabés mostrant el doble del resultat final correcte.

Finalment, la tercera part la vam deixar com es mostra a continuació:

```
sum = 0;
// Part III
#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) reduction(+: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);
```

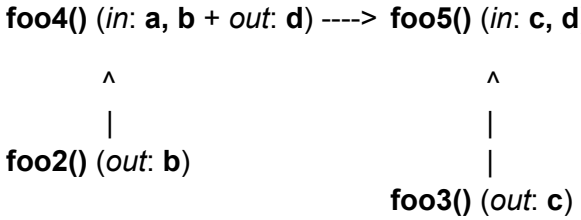
El valor inicial de 100 394 187 era clarament incorrecte. Part del problema és que no es reiniciava el valor inicial de sum. Amb la línia de

```
sum = 0;
```

s'evita la acumulació del resultat. Només caldria afegir el mateix tipus de **reduction** que a la *Part 1* per a que el primer for funcioni, i la mateixa solució aplicada a la *Part 2* per a que funcioni correctament el segon for, protegint a cadascun d'ells la variable **sum** amb la clàusula **reduction** corresponent.

13. synchtasks.c

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



Recordem que:

1) Totes les tasques amb una variable marcada com a **in** seran dependents de tota tasca prèvia amb la mateixa variable com a **out** o **inout**.
2) Totes les tasques amb una variable marcada com a **out** o **inout** seran dependents de tota tasca prèvia.

b. *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**.*

Amb el constructe

```
#pragma omp taskwait
```

es poden especificar esperes a que les tasques filles de l'actual (i l'actual) acabin abans no es comenci la següent. Tenint en compte les dependències anteriors, doncs, vam proposar la variació del codi:

```
[...]
printf("Creating task foo1\n");
#pragma omp task
foo1();
printf("Creating task foo2\n");
#pragma omp task
foo2();
#pragma omp taskwait
printf("Creating task foo4\n");
#pragma omp task
foo4();
printf("Creating task foo3\n");
#pragma omp task
foo3();
#pragma omp taskwait
printf("Creating task foo5\n");
#pragma omp task
foo5();
[...]
```

Així, les tasques corresponents a **foo4()** i **foo5()** havien d'esperar a la compleció de, respectivament, **foo1()** i **foo2()**, i **foo3()** i **foo4()**. Donat que la tasca de **foo3()** no depenia de res però calia haver-se completat abans no s'executés **foo5()**, i no es podia permetre que s'arribés a **foo5()** sense haver executat **foo4()**, vam considerar que era acceptable moure-la perquè s'executés després del **taskwait** de **foo4()**. Això, en aparença, la feia dependent de **foo1()** i **foo2()**, però vam crure que era irrellevant, perquè es podia executar en qualsevol moment i així reduïem (teòricament) el temps d'espera de **foo4()**.

Donat que **foo3()** i **foo4()** s'executarien paral·lelament, el canvi d'ordre en el codi no és significatiu; és un recurs estètic per visualitzar millor l'ordre de dependències en funció del diagrama de l'anterior apartat.

c. *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 using **depend**.*

El constructe

```
#pragma omp taskgroup
```

especifica una espera per a la compleció d'una tasca i la de les seves tasques filles i descendents abans de passar a la següent. Altre cop emulant les dependències originals, el codi obtingut va ser:

```
[...]
#pragma omp taskgroup
{
```

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

Per mantenir les dependències originals, doncs, havíem de crear un **taskgroup** per a **foo1()** i **foo2()**, que forçosament s'havien d'executar abans de **foo4()**, i **foo3()** i **foo4()**, que havien d'haver acabat abans de **foo5()**. Altre cop, l'aparent dependència de **foo3()** a **foo1()** i **foo2()** no ens va semblar significativa: no afectava al codi i la seva falta de dependències permetia la seva execució en qualsevol punt (abans de **foo5()**) del codi.

Eccolo!