Exercicis Pràctics OpenMP (Part I)

L'objectiu principal d'aquests exercicis és que experimenteu amb les capacitats d'OpenMP fent servir casos simples, facilitant d'aquesta manera la transició entre els continguts discutits en la class de teoria i la seva aplicació al cas pràctic (més complex) treballat en el laboratori.

El plantejament dels exercicis i la mecànica de treball per resoldre'ls consisteixen en:

1) Per cada apartat, es proporcionen un conjunt de fragments de codi que caldrà executar i analitzar (els programes corresponents els trobareu a /home/alumnos/pp/alumnos/Avaluats-problemes/OpenMP/sessio1/, copieu els arxius al vostre compte per fer els problemes).

2) Per cada apartat, es fa un conjunt de preguntes sobre cada fragment de codi presentat. Heu de respondre cada pregunta, **justificant sempre la vostra resposta**. En alguns casos la justificació serà molt curta, en altres més complexa i, en altres, potser consistirà en un nou fragment de codi.

3) La solució als exercicis s'inclourà en aquest document que s'haurà de lliurar al CV en la data establerta.

NOTA: En molts apartats trobareu l'enunciat "*Expliqueu els valors que...*", en aquests casos no estem gaire interessats en els *valors* com a tal sinó en explicar *per què* es produeixen.

Exercicis

a) Compartició de variables

Donat el codi en c de l'arxiu apartat-a.c, responeu a les següents preguntes :

i) Expliqueu per a que serveixen les següents clàusules d'openmp utilitzades al codi: *num_threads*, *private*, *firstprivate*, *lastprivate*, *master*.

num_threads: Permet especificar el nombre de fils (threads) que s'han d'utilitzar per a executar una regió paral·lela. Per exemple, num_threads(4) indicaria que es volen utilitzar quatre fils per a l'execució paral·lela de la regió.

private: Indica que cada fil (thread) ha de tenir una còpia privada de les variables especificades. Això significa que cada fil tindrà la seva pròpia instància de les variables assenyalades i no es compartiran entre els fils.

firstprivate: És semblant a private, però mentres private inicialitza les variables amb valors *randoms*, firstprivate les inicialitza amb els valors de les variables compartides fora de la regió paral·lela.

lastprivate: Aquesta clàusula s'utilitza per assegurar que una variable sigui assignada amb el valor de la darrera iteració del bucle paral·lel quan s'ha acabat la seva execució. Això és útil quan es vol conservar el valor d'una variable després d'una regió paral·lela.

master: Indica que només el fil principal (el que ha creat la regió paral·lela (0)) executarà el bloc de codi especificat. Els altres fils esperaran a la fi de la regió abans de continuar amb la seva execució. Permet que un fil concret s'encarregui d'una tasca específica.

ii) Quines parts del codi s'executen en paral·lel ? Hi ha alguna part que tot i ser dins d'una regió paral·lela es executada per un sol thread ?

```
    #pragma omp parallel num_threads(2) private(b)
firstprivate(a) PART PARAL·LELITZADA:
    { inici
        printf("1) FIRST: a = %d | b = %d [thread:%d]\n", a,b,
omp_get_thread_num());
        a = 10;
        #pragma omp master PART SEQUENCIAL (THREAD 0):
        { inici
        a = 25;
        } fi
         printf("2) SECOND : a = %d | b = %d [thread:%d] \n",
a, b, omp_get_thread_num());


        c = omp_get_thread_num();
        sleep(1);
```

```
                    printf("3)   c   =   %d   [thread:%d]\n",   c,
omp_get_thread_num());
    } fi
```

```
     #pragma  omp  parallel  for  lastprivate(c)  num_threads(4)
PART PARAL·LELITZADA:
    for(int i = 0; i < 4; i++)
    {
        c = i;
    }
```

iii) Raoneu com canvia cadascun dels valors de les variables que s'imprimeix en pantalla al llarg de l'execució.

Instrucció	Thread 0	Thread 1

`int main(){`
`  int a,b,c;`
 	
`  a = 5;`
`  b = 10;`
`  c = 0;`
	a = 5
	b = 10
	c = 0
		a = 5
		b = 10
		c = 0

`  #pragma omp parallel num_threads(2)`
`private(b) firstprivate(a)`
`  {`
`  [...]`
`  }`
	a = 5
	b = ø
	c = 0
		a = 5
		b = ø
		c = 0

`a = 10;`	a = 10 b = ø c = 0	a = 10 b = ø c = 0

`  #pragma omp master`
`  {`
`  a = 25;`
	a = 25
	b = ø
	c = 0
		a = 10
		b = ø
		c = 0

`}` `[...]`		
`c = omp_get_thread_num();`	a = 25 b = ø c = 2	a =10 b = ø c = 2
`printf("4) a = %d, b = %d, c = %d\n", a,b,c);`	a = 5 b = 10 c = 2	
`c = -1;`	a = 5 b = 10 c = -1	

Instrucció:	Thread 0	Thread 1	Thread 2	Thread 3

`#pragma omp parallel for lastprivate(c) num_threads(4)`
`for(int i = 0; i < 4; i++)`
`{`
`c = i;`
`}`
	a = 5
	b = 10
	c = 0
		a = 5
		b = 10
		c = 1
			a = 5
			b = 10
			c = 2
				a = 5
				b = 10
				c = 3

`printf("5) Value of c outside the second parallel region : %d\n", c);`	a = 5 b = 10 c = 3			

b) Bucles niats

Donats els següents bucles niats del arxiu apartat-b.c :

```
#pragma omp parallel for num_threads(4) private(j)
for ( i = 0; i < NUM_ITER ; i++ ) {
     for ( j = 0; j < NUM_ITER ; j++ ) {
             sum[i][j] =  a[i][j] + b[i][j];
     }
}
```

iv) Modifiqueu el codi proporcionat afegint **un sol pragma** per tal de repartir el treball del **primer bucle for** entre **quatre** threads. Expliqueu per a què serveixen les clàusules que heu utilitzat.

Considereu que el resultat final que s'imprimeix per pantalla ha de ser el mateix que el que es mostra per a la versió seqüencial. Es proporciona un print per tal de veure el nombre de threads i la iteració que esta fent a cada moment si fos necessari.

```
#pragma omp parallel for num_threads(4) private(j)
```

Les clàusules utilitzades han estat:

omp parallel for: Aquest pragma indica que el bloc de codi següent s'executarà en paral·lel. Cada iteració del bucle serà assignada a un fil diferent. Els fils s'encarreguen d'executar les iteracions del bucle en paral·lel.

num_threads(4): Aquesta clàusula especifica que volem utilitzar quatre fils per a l'execució paral·lela del bucle. Això assegura que les iteracions del bucle s'assignin als quatre fils especificats.

private(j): Amb la clàusula private(j), cada fil tindrà una còpia privada de la variable j, assegurant que no hi hagi conflictes d'accés ni sobrecàrregues de dades en la variable j mentre s'executa el bucle paral·lel. Donat que cada fil s'encarrega d'una iteració del primer bucle, la j es compartiria entre els 4 threads a la vegada i el resultat seria incorrecte.

v) Expliqueu com es reparteixen la feina els threads que s'han creat, es a dir, quina part de cadascun dels bucles faran els threads.

Amb NUM_ITER igual a 4, els bucles for iteraran quatre vegades, de 0 a 3, designant cada valor de i per cada thread (thread 0 → i = 0, etc). Cada fil executarà una part del treball, amb les iteracions dels bucles repartides equitativament entre els fils. En aquest cas, cada fil agafaria una

de les quatre iteracions del primer bucle for, i dins d'aquestes, executaria les quatre iteracions del segon bucle for. Per cada thread, la j és privada, per a que no afecti a la j del bucle interior d'un altre thread.

vi) Volem transformar aquests dos bucles niats en un de sol utilitzant una única clàusula. Quina seria ? Quin(s) paràmetre(s) li hem de passar ? Com canvia la feina que fa cadascun dels threads respecte a l'apartat anterior ? Mostreu el pragma a afegir per combinar els bucles.

La clàusula a utilitzar seria collapse(n), on *n* seria el nombre de bucles a combinar, en aquest cas 2. L'execució dels dos for combinats es reparteix entre quatre fils (threads) com en l'exemple anterior. Cada fil rebrà una part del treball, però ara serà una part de les combinacions de i i j enlloc de simplement les iteracions de i com en l'exemple anterior.

c) Regions paral·leles

//Arxiu apartat-c.c

```
void par_func(){
        int id = omp_get_thread_num();
        #pragma omp parallel
        {
           printf("PAR_FUNC: Thread %d fill de %d des de la
secció paral·lela\n", omp_get_thread_num(), id);
        }
}


int main(){
      #pragma omp parallel num_threads(2)
        {
           printf("MAIN: thread %d des de la secció
paral·lela\n", omp_get_thread_num());
           par_func();
        }
}
```

i) Quants threads executen la secció paral·lela del *main* ? Quants threads es creen per executar la secció paral·lela de la funció *par_func* ? Explica el resultat que s'imprimeix per pantalla al executar aquest codi.

Informació impresa per pantalla:

```
MAIN: thread 0 des de la secció paral·lela

MAIN: thread 1 des de la secció paral·lela

PAR_FUNC: Thread 0 fill de 0 des de la secció paral·lela

PAR_FUNC: Thread 0 fill de 1 des de la secció paral·lela
```

Degut a que en el main es crida `#pragma omp parallel num_threads(2)`, es crida la funció **par_func()** per cada thread, la cual crida a `#pragma omp parallel`, la qual intenta executar codi en paralel. Però com que aquesta última crida és executada per cada un dels fils, no es pot dividir aquest mateix fil en cap més. Per aquesta raó no es creen més fils a la funció **par_func()**. Com a més fils especificats a `num_threads(2)`, més vegades es cridarà la funció **par_func()**, que en cap cas es podrà paral·lelitzar.

ii) Per a que serveix el pragma **omp_set_nested** ? Modifiqueu el codi per tal de permetre la creació d'una regió paral·lela niada a *par_funct*. Aquesta regió paral·lela haurà de generar dos threads que executaran cadascun el codi de la funció *par_func*.

El pragma **omp_set_nested** és una funció de la llibreria OpenMP que es fa servir per activar o desactivar la capacitat de tenir bucles paral·lels aniuats. Quan aquesta capacitat està activada, es permet que un bloc de codi paral·lel contengui altres blocs de codi paral·lel dins seu.

La modificació del main seria la següent:

```
int main(){
      omp_set_nested(1);
      #pragma omp parallel num_threads(2)
      {
         #pragma omp parallel num_threads(1)
         {
            printf("MAIN: thread %d des de la secció
paral·lela\n", omp_get_thread_num());
            par_func();
         }
      }
}
```