

Programmation OpenMP — la directive «task»

La directive «task»

- ☐ la directive «task» crée une tâche de manière **explicite** ;
- ☐ toutes les threads se **partagent le travail disponible** parmi l'ensemble des tâches créées ;
- ☐ la directive «taskwait» **garantie** que toutes les tâches enfants créées dans la même tâche courante **ont fini**.

1 – Comparez les deux programmes suivants :

```
1 #include <omp.h>
2 #include <stdio.h>
3
4 main()
5 {
6     int i;
7     #pragma omp parallel private(i)
8     { for(i=0; i<10; i++)
9         { printf("Task %d: %i\n",
10             omp_get_thread_num(), i);
11         }
12     }
13 }
```

```
1 #include <omp.h>
2 #include <stdio.h>
3
4 main()
5 {
6     int i;
7     #pragma omp parallel private(i)
8     { for(i=0; i<10; i++)
9         #pragma omp task
10         { printf("Task %d: %i\n",
11             omp_get_thread_num(), i);
12         }
13     }
14 }
```

Combien de threads vont être utilisées et quel va être leur travail ?

Il se passe la même chose :

- ▷ la région parallèle de la ligne 8 à la ligne 12-13 est exécutée par chaque cœur présent dans la machine ;
 - ▷ chaque cœur exécute la boucle *for* :
 - ◇ dans le programme de gauche, chaque cœur affiche les 10 sorties ;
 - ◇ dans le programme de droite, chaque cœur lance une «task» et ne dispose que du seul cœur qu'elle a à sa disposition ce qui ne permet qu'une seule task par cœur.
 - ▷ sur une machine avec 4 cœurs, cela va conduire à l'affichage de 40 lignes pour les deux programmes.
- Si on veut répartir les occurrences de la boucle «for» entre différents cœurs, il faut :
- forcer l'exécution par un seul cœur du corps de la boucle avec une directive «omp single» ;
 - lancer une task pour traiter chaque occurrence :

```
1 #include <omp.h>
2 #include <stdio.h>
3 main()
4 {
5     int i;
6     #pragma omp parallel private(i)
7     {
8         #pragma omp single
9         for(i=0; i<10; i++)
10             #pragma omp task
11             { printf("Task %d: %i\n",
12                 omp_get_thread_num(), i);
13             }
14     }
15 }
```

```
pef@darkstar:~/PARALLELISMEII$ ./td3_exo1_2
Task 1: 0
Task 2: 1
Task 3: 2
Task 0: 3
Task 1: 4
Task 2: 5
Task 3: 6
Task 0: 7
Task 1: 8
Task 2: 9
```

2 – Que va produire le code suivant :

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <omp.h>
4
5 int main()
6 {
7     #pragma omp parallel sections
8     {
9         #pragma omp section
10        {
11            int i;
12            for (i=0;i<4;i++)
13            {
14                #pragma omp task
15                printf("Task %d\n",
16                    omp_get_thread_num());
17            }
18        }
19        #pragma omp section
20        {
21            int i;
22            for (i=0;i<4;i++)
23            {
24                #pragma omp task
25                printf("Task %d\n",
26                    omp_get_thread_num());
27            }
28        }
29    }
```

Cela va dépendre du nombre de cœur disponible ou de threads définies par OMP_NUM_THREADS :

- ☐ la région parallèle est définie en ligne 7 : elle répartie deux sections entre deux cœurs/threads ;
- ☐ dans chacune de ces sections :
 - ♦ on essaye de lancer une tâche dans une boucle for ;
 - ♦ s'il reste un cœur/thread disponible, la tâche peut s'exécuter, sinon le travail est assuré par la thread courante.

L'exécution suivante est obtenue avec 4 cœurs :

```
pef@darkstar:~/PARALLELISMEII$ ./td3_exo2
Task 0
Task 3
Task 1
Task 2
Task 0
Task 3
Task 1
Task 2
```

3 – Décrivez le fonctionnement du programme suivant :

```
1 #include <stdio.h>
2 int work( int i )
3 {
4     if ( i > 0 )
5     {
6         #pragma omp parallel
7         {
8             #pragma omp task
9             {
10                work( i-1 );
11            }
12            #pragma omp task
13            {
14                work( i-1 );
15            }
16            #pragma omp taskwait
17            printf( "Completed %i\n", i );
18        }
19    }
20 }
21 void main()
22 {
23     work( 3 );
24 }
```

La région parallèle des lignes 6 à 19 est exécuté par tous les cœurs disponibles et l'utilisation des tâches ne change rien : on peut créer autant de tâches que l'on veut mais elle ne seront exécutées que si un cœur est disponible.

Pour 4 cœurs, on a le résultat suivant :

```
pef@darkstar:~/PARALLELISMEII$ ./td3_exo3 | sort
Completed 1
Completed 1
Completed 1
Completed 1
Completed 1
Completed 1
Completed 1
Completed 1
Completed 2
Completed 2
Completed 2
Completed 2
Completed 3
Completed 3
```

On note qu'il y a 2 affichages de la valeur 3, puis 4 de la valeur 2 et enfin 8 de la valeur 1.

Le code suivant est conforme à l'idée de répartir le code entre différents cœurs :

```
1 #include <stdio.h>
2 int work( int i )
3 {
4     if ( i > 0 )
5     {
6         #pragma omp task
7         {
8             work( i-1 );
9         }
10        #pragma omp task
11        {
12            work( i-1 );
13        }
14        #pragma omp taskwait
15        printf( "Completed %i\n", i );
16    }
17 }
18 void main()
19 {
20     #pragma omp parallel
21     {
22         #pragma omp single
23         work( 3 );
24     }
25 }
```

Ce qui donne l'exécution suivante avec 4 cœurs :

```
pef@darkstar:~/PARALLELISMEII$ ./td3_exo3_1
Completed 1 by task 1
Completed 1 by task 0
Completed 1 by task 2
Completed 1 by task 3
Completed 2 by task 1
Completed 2 by task 0
Completed 3 by task 1
```

Ce qui est conforme à l'attente du programmeur.

4 – Essayez de paralléliser ces morceaux de code ou expliquez pourquoi ce n'est pas possible :

a.

```
1 for (i = 0; i < sqrt(x); i++)
2 {
3     a[i] = 2.3 * i;
4     if (i < 10)
5         b[i] = a[i];
6 }
```

```
1 #include <omp.h>
2 #include <math.h>
3
4 void main (void)
5 {
6     int a[10], b[10];
7     int x=100;
8     int i;
9
10    #pragma omp parallel for
11    for (i = 0; i < sqrt(x); i++)
12    {
13        a[i] = 2.3 * i;
14        if (i < 10)
15            b[i] = a[i];
16    }
17 }
```

La parallélisation pourrait utiliser une directive de «worksharing for», mais la borne max de la variable *i* n'est pas déterminée lors de la compilation.

```
pef@darkstar:~/PARALLELISMEII$ gcc -fopenmp -o td3_exo4 td3_exo4.c
td3_exo4.c: In function 'main':
td3_exo4.c:11:14: error: invalid controlling predicate
    for (i = 0; i < sqrt(x); i++)
```

b.

```
1 flag = 0;
2 for (i = 0; i < n && !flag; i++)
3 {
4     a[i] = 2.3 * i;
5     if (a[i] < b[i])
6         flag = 1;
7 }
```

```
1 #include <omp.h>
2 #include <math.h>
3
4 void main (void)
5 {
6     int a[10], b[10];
7     int x=100;
8     int i;
9     int flag = 0;
10    int n = 100;
11
12    #pragma omp parallel for
13    for (i=0; i<n && !flag; i++)
14    {
15        a[i] = 2.3 * i;
16        if (a[i] < b[i])
17            flag = 1;
18    }
19 }
```

Là également la condition de continuation de la boucle for ne permet pas sa parallélisation :

```
pef@darkstar:~/PARALLELISMEII$ gcc -fopenmp -o td3_exo4_1 td3_exo4_1.c
td3_exo4_1.c: In function 'main':
td3_exo4_1.c:13:13: error: invalid controlling predicate
    for (i = 0; i < n && !flag; i++)
```