

Programmation OpenMP

■ ■ ■ Les différentes formes de parallélisme

```

1- 1 #include <omp.h>
    2 #include <stdio.h>
    3 #include <stdlib.h>
    4
    5 int main ()
    6 { int nthreads, tid;
    7
    8 #pragma omp parallel private(nthreads, tid)
    9 {
   10     tid = omp_get_thread_num();
   11     printf("Hello World from thread = %d\n", tid);
   12
   13     if (tid == 0)
   14     { nthreads = omp_get_num_threads();
   15       printf("Number of threads = %d\n", nthreads);
   16     }
   17 }
   18 }

```

- ▷ les lignes 8 à 17 définissent une région parallèle : tous les cœurs présents sur la machine exécutent cette région parallèle simultanément ;
- ▷ chaque thread obtient en ligne 10 son numéro qu'elle range dans la variable privée *tid* ;
- ▷ seule la thread de numéro 0 en ligne 13 consulte et affiche le nombre total de threads ;
- ▷ après la ligne 17, à la sortie de la région parallèle, il n'y a plus qu'une seule thread.

2 – Quel va être le résultat de ce programme, `omp_workshare1.c` :

```

1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #define CHUNKSIZE 10
5 #define N 100
6
7 int main ()
8 { int nthreads, tid, i, chunk;
9   float a[N], b[N], c[N];
10
11   for (i=0; i < N; i++)
12       a[i] = b[i] = i * 1.0;
13   chunk = CHUNKSIZE;
14
15   #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
16   {
17       tid = omp_get_thread_num();
18       if (tid == 0)
19       { nthreads = omp_get_num_threads();
20         printf("Number of threads = %d\n", nthreads);
21       }
22       printf("Thread %d starting...\n",tid);
23
24       #pragma omp for schedule(dynamic,chunk)
25       for (i=0; i<N; i++)
26       { c[i] = a[i] + b[i];
27         printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
28       }
29   } /* end of parallel section */
30 }

```

- ▷ région parallèle entre les lignes 16 et 29 ;
- ▷ ligne 24 : « Work Sharing », répartition et partage de travail :
 - ◇ les différentes occurrences de la boucle `for` de la ligne 25 sont réparties entre chacune des threads de la région parallèle de manière dynamique : dès qu'une thread est libre, elle obtient du travail et si le travail n'est pas suffisant, une thread peut ne rien faire.
 - ◇ Le découpage se fait suivant des « chunks », morceaux, de taille fixe ;
 - ◇ en ligne 28, toutes les threads ont fini de travailler et le travail de la boucle `for` a été entièrement réalisé (toutes les occurrences ont été faites).

4 – Quel est le résultat de ce programme, `omp_workshare2.c` :

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #define N 50
5
6 int main ()
7 { int i, nthreads, tid;
8   float a[N], b[N], c[N], d[N];
9
10  for (i=0; i<N; i++) {
11    a[i] = i * 1.5; b[i] = i + 22.35; c[i] = d[i] = 0.0;
12  }
13  #pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
14  {
15    tid = omp_get_thread_num();
16    if (tid == 0)
17      { nthreads = omp_get_num_threads();
18        printf("Number of threads = %d\n", nthreads);
19      }
20    printf("Thread %d starting...\n",tid);
21
22    #pragma omp sections nowait
23    {
24      #pragma omp section
25      {
26        printf("Thread %d doing section 1\n",tid);
27        for (i=0; i<N; i++)
28          { c[i] = a[i] + b[i];
29            printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
30          }
31      }
32      #pragma omp section
33      {
34        printf("Thread %d doing section 2\n",tid);
35        for (i=0; i<N; i++)
36          { d[i] = a[i] * b[i];
37            printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
38          }
39      }
40    } /* end of sections */
41    printf("Thread %d done.\n",tid);
42  } /* end of parallel section */
43 }
```

- ▷ région parallèle définie de la ligne 13 à 42 ;
- ▷ tous les cœurs exécutent les lignes 16 et 20 ;
- ▷ seule la thread de numéro 0 exécute les lignes de 16 à 19 ;
- ▷ en ligne 22 se définit des « sections » :
 - ◇ la section de la ligne 24 à 31 est exécutée par une seule thread ;
 - ◇ la section de la ligne 32 à 39 est exécutée par une autre thread en parallèle de la première section ;
 - ◇ s'il y a plus de deux threads, elles ne font rien (pas de section disponible) ;
 - ◇ à la ligne 40, il y a une « barrière de synchronisation » : toutes les sections ont fini de s'exécuter.
- ▷ la ligne 41 est exécutée par toutes les threads.

5 – Et de celui-ci, `omp_reduction.c` :

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 int main ()
5 {
6   int i, n;
7   float a[100], b[100], sum;
8
9   /* Some initializations */
10  n = 100;
11  for (i=0; i < n; i++)
12    a[i] = b[i] = i * 1.0;
13  sum = 0.0;
14
15  #pragma omp parallel for reduction(+:sum)
16    for (i=0; i < n; i++)
17      sum = sum + (a[i] * b[i]);
18  printf(" Sum = %f\n",sum);
19 }
```

- ▷ ligne 16 : une région parallèle combinée à une directive de « work sharing » : le travail de la boucle `for`, de la ligne 16 à 17, va être réparti entre les différentes threads ;
- ▷ ce travail réalise une somme dans la variable `sum` désignée pour faire une opération de réduction :
 - ◇ chaque thread travaille sur ses occurrences de la boucle `for` avec une copie privée de la variable `sum` en réalisant l'opération « + » ;
 - ◇ à la fin du « work sharing » toutes les valeurs obtenues par chacune des threads vont être combinées automatiquement pour obtenir un résultat global avec l'opération « + ».

La réduction porte sur des opérations commutatives et associatives. Elle permet de tirer parti des optimisations matérielles des processeurs si disponible : protection des accès concurrents, combinaison des résultats efficaces, ce qui n'est pas possible en programmation non parallèle.

6 – Comment vont s’organiser les différentes threads dans le programme, `omp_orphan.c` :

```

1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #define VECLLEN 100
5 float a[VECLLEN], b[VECLLEN], sum;
6
7 void dotprod ()
8 { int i,tid;
9
10     tid = omp_get_thread_num();
11     #pragma omp for reduction(+:sum)
12     for (i=0; i < VECLLEN; i++)
13         { sum = sum + (a[i]*b[i]);
14           printf(" tid= %d i=%d\n",tid,i);
15         }
16 }
17 int main ()
18 { int i;
19
20     for (i=0; i<VECLLEN; i++) a[i]=b[i]=1.0*i;
21     sum = 0.0;
22     #pragma omp parallel
23         dotprod();
24     printf("Sum = %f\n",sum);
25 }

```

- ▷ la région parallèle est définie sur la ligne 23 ;
- ▷ une directive de « work sharing » est définie dans la fonction « dotprod » :
 - ◊ si la fonction est appelée au sein d’une région parallèle, comme c’est le cas ici, la directive va répartir le travail entre les différentes threads déclenchées par la région parallèle ;
 - ◊ si la fonction n’est pas appelée dans une région parallèle, alors la directive de « work sharing » est ignorée et le travail est réalisé par une seule thread en séquentiel.
- ▷ on parle de « directive orpheline » : cela peut être utile dans le cas de l’écriture de bibliothèques de fonctions parallèles.

7 – Ce programme affiche l’ensemble des informations du contexte parallèle, `omp_getEnvInfo.c` :
Comme sa description l’indique...

8 – Décrivez l’organisation des threads pour le programme suivant, `omp_mm.c` :

- ▷ région parallèle définie de la ligne 18 à 52 ;
- ▷ la ligne 20 est exécutée par toutes les threads sur une variable privée `tid` ;
- ▷ seule la thread de numéro 0 exécute les lignes 21 à 26 ;
- ▷ « work sharing » de la ligne 28 à 31 exécutées par toutes les threads de la région parallèle ;
- ▷ « work sharing » de la ligne 32 à 35 exécutées par toutes les threads de la région parallèle ;
- ▷ « work sharing » de la ligne 36 à 39 exécutées par toutes les threads de la région parallèle ;
- ▷ « work sharing » de la ligne 44 à 52 exécutées par toutes les threads de la région parallèle :
 - ◊ seule la boucle `for` utilisant l’indice `i` en ligne 45 est répartie entre les différentes threads de manière régulière, « static » et suivant des morceaux, « chunks », prédéfinis ;
 - ◊ chaque thread réalise un certain ensemble d’occurrences de `ide` la boucle de la ligne 45 :
 - * [0..9] * [20..29] * [40..49] * [60..61]
 - * [10..19] * [30..39] * [50..59]
 - ◊ chaque thread réalise la boucle `for` de la ligne 48 complètement, `c-à-d` pour toutes les occurrences de la variable `j`.
- ▷ à partir de la ligne 54 : la région parallèle est finie et une seule thread réalise le travail qui correspond à l’affichage du résultat.

Un programme Python, « `verif_mul_matrices.py` », pour vérifier l’exactitude des résultats :

```

#!/usr/bin/python3
import numpy as np

NRA = 62 # nombre de lignes de A
NCA = 15 # nombre de colonnes de A
NCB = 7 # nombre de colonnes de B

matA = np.zeros((NRA, NCA))
matB = np.zeros((NCA, NCB))

for i in range(NRA):
    for j in range(NCA):
        matA[i][j] = i+j

for i in range(NCA):
    for j in range(NCB):
        matB[i][j] = i*j

# print(matA)
# print(matB)
print(np.matmul(matA, matB))

```

```

1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define NRA 62          /* number of rows in matrix A */
6 #define NCA 15         /* number of columns in matrix A */
7 #define NCB 7          /* number of columns in matrix B */
8
9 int main (int argc, char *argv[])
10 { int tid, nthreads, i, j, k, chunk;
11   double a[NRA][NCA], /* matrix A to be multiplied */
12   b[NCA][NCB], /* matrix B to be multiplied */
13   c[NRA][NCB]; /* result matrix C */
14
15   chunk = 10;          /* set loop iteration chunk size */
16
17   /*** Spawn a parallel region explicitly scoping all variables ***/
18   #pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
19   {
20     tid = omp_get_thread_num();
21     if (tid == 0)
22     {
23       nthreads = omp_get_num_threads();
24       printf("Starting matrix multiple example with %d threads\n",nthreads);
25       printf("Initializing matrices...\n");
26     }
27     /*** Initialize matrices ***/
28     #pragma omp for schedule (static, chunk)
29     for (i=0; i<NRA; i++)
30       for (j=0; j<NCA; j++)
31         a[i][j]= i+j;
32     #pragma omp for schedule (static, chunk)
33     for (i=0; i<NCA; i++)
34       for (j=0; j<NCB; j++)
35         b[i][j]= i*j;
36     #pragma omp for schedule (static, chunk)
37     for (i=0; i<NRA; i++)
38       for (j=0; j<NCB; j++)
39         c[i][j]= 0;
40
41     /*** Do matrix multiply sharing iterations on outer loop ***/
42     /*** Display who does which iterations for demonstration purposes ***/
43     printf("Thread %d starting matrix multiply...\n",tid);
44     #pragma omp for schedule (static, chunk)
45     for (i=0; i<NRA; i++)
46     {
47       printf("Thread=%d did row=%d\n",tid,i);
48       for(j=0; j<NCB; j++)
49         for (k=0; k<NCA; k++)
50           c[i][j] += a[i][k] * b[k][j];
51     }
52   } /*** End of parallel region ***/
53
54   /*** Print results ***/
55   printf("*****\n");
56   printf("Result Matrix:\n");
57   for (i=0; i<NRA; i++)
58   {
59     for (j=0; j<NCB; j++)
60       printf("%6.2f  ", c[i][j]);
61     printf("\n");
62   }
63   printf("*****\n");
64   printf ("Done.\n");
65
66 }

```