Compte rendu TP4 :

Initiation OpenMP

Réalisé par :

Mhiri Yasmine

Mtibaa Amal

Riahi Mohamed Wassim

GL4/groupe1

2018-2019

Exercice 1

1.

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int nthreads, tid; //nthreads=nombre de thread , tid= thread id
/* Fork un ensemble de threads ayant chacun sa propre copie des variables */
#pragma omp parallel private(nthreads, tid)
  {


  tid = omp_get_thread_num(); //obtenir l'id du thread
  printf("Hello World du thread num= %d\n", tid); //Imprimer l'Id de Thread


  if (tid == 0) //Seule le thread maitre fait ces instructions
    {
    nthreads = omp_get_num_threads();
    printf("Nombre des threads = %d\n", nthreads);
    }

  }//Tous les threads rejoignent le thread principal et se terminent

}
```

Le programme omp_hello_world.c permet de créer des threads qui affichent leurs ids, le thread maitre (tid=0) affiche le nombre total des threads Voici un affichage de l'exécution de ce programme

```
C:\Users\asus\Desktop\Mtibaa Amal\semestre2\Architecture Paralélle\TP4>gcc -fopenmp -o hello omp_hello_world.c

C:\Users\asus\Desktop\Mtibaa Amal\semestre2\Architecture Paralélle\TP4>hello
Hello World du thread num= 1
Hello World du thread num= 3
Hello World du thread num= 2
Hello World du thread num= 0
Nombre des threads = 4
```

2. On ajoute **omp_set_num_threads(7);**

```c
int main (int argc, char *argv[])
{
int nthreads, tid;
omp_set_num_threads(7);
/* Fork un ensemble de threads ayant chacun sa propre copie des variables */
#pragma omp parallel private(nthreads, tid)
  {


  tid = omp_get_thread_num(); //obtenir l'id du thread
  printf("Hello World du thread num= %d\n", tid); //Imprimer l'Id de Thread


  if (tid == 0) //Seule le thread maitre fait ces instructions
    {
    nthreads = omp_get_num_threads();
    printf("Nombre des threads = %d\n", nthreads);
    }

  }//Tous les threads rejoignent le thread principal et se terminent

}
```

Exemple d'affichage :

```
C:\Users\asus\Desktop\Mtibaa Amal\semestre2\Architecture Paralélle\TP4>gcc -fopenmp -o hello omp_hello_world.c

C:\Users\asus\Desktop\Mtibaa Amal\semestre2\Architecture Paralélle\TP4>hello
Hello World du thread num= 1
Hello World du thread num= 3
Hello World du thread num= 4
Hello World du thread num= 0
Nombre des threads = 7
Hello World du thread num= 5
Hello World du thread num= 6
Hello World du thread num= 2
```

3. En exécutant plusieurs fois le programme, on observe que l'ordre des instructions d'affichage n'est pas le même et change à chaque fois.

```
C:\Users\asus\Desktop\Mtibaa Amal\semestre2\Architecture Paralélle\TP4>hello
Hello World du thread num= 4
Hello World du thread num= 5
Hello World du thread num= 3
Hello World du thread num= 0
Nombre des threads = 7
Hello World du thread num= 6
Hello World du thread num= 1
Hello World du thread num= 2

C:\Users\asus\Desktop\Mtibaa Amal\semestre2\Architecture Paralélle\TP4>hello
Hello World du thread num= 2
Hello World du thread num= 6
Hello World du thread num= 1
Hello World du thread num= 4
Hello World du thread num= 3
Hello World du thread num= 0
Nombre des threads = 7
Hello World du thread num= 5
```

Exercice 2 :

1.

```
int main (int argc, char *argv[])
{
int nthreads, tid;
omp_set_num_threads(4);
/* Fork un ensemble de threads ayant chacun sa propre copie des variables */
#pragma omp parallel private(nthreads, tid)
  {


  tid = omp_get_thread_num(); //obtenir l'id du thread
  printf("Thread num= %d a commence\n", tid); //Imprimer l'Id de Thread

  printf("Thread num= %d a termine\n",tid);

  }//Tous les threads rejoignent le thread principal et se terminent

}
```

Exemple d'affichage

```
Thread num= 2 a commence
Thread num= 1 a commence
Thread num= 1 a termine
Thread num= 3 a commence
Thread num= 3 a termine
Thread num= 0 a commence
Thread num= 0 a termine
Thread num= 2 a termine
```

2.

```c
int main (int argc, char *argv[])
{
int nthreads, tid;
int valeur1=1000;
int valeur2=2000;
omp_set_num_threads(4);
/* Fork un ensemble de threads ayant chacun sa propre copie des variables */
#pragma omp parallel private(nthreads, tid,valeur2)
  {

  tid = omp_get_thread_num(); //obtenir l'id du thread
  printf("Thread num= %d a valeur1= %d et valeur 2=% d \n", tid, valeur1,valeur2);
  valeur2++;


  }//Tous les threads rejoignent le thread principal et se terminent

}
```

```
Thread num= 1 a valeur1= 1000 et valeur 2= 15138620
Thread num= 3 a valeur1= 1000 et valeur 2= 19332924
Thread num= 0 a valeur1= 1000 et valeur 2= 0
Thread num= 2 a valeur1= 1000 et valeur 2= 17235772
```

4. **Private** : Toutes les références à l'objet original sont remplacées par des références aux nouveaux objets.

Valeur 2 affiche la valeur stockée dans la case mémoire du nouvel objet du thread +1,

5. **Firstprivate** : La clause FIRSTPRIVATE combine le comportement de la clause PRIVATE avec l'initialisation automatique des variables dans sa liste

• Les variables listées sont initialisées en fonction de la valeur de leurs objets d'origine avant l'entrée dans la zone partagée.

```c
int main (int argc, char *argv[])
{
int nthreads, tid;
int valeur1=1000;
int valeur2=2000;
omp_set_num_threads(4);
/* Fork un ensemble de threads ayant chacun sa propre copie des variables */
#pragma omp parallel firstprivate(nthreads, tid,valeur2)
  {

  tid = omp_get_thread_num(); //obtenir l'id du thread
  valeur2++;
  printf("Thread num= %d a valeur1= %d et valeur 2=% d \n", tid, valeur1, valeur2);


  }//Tous les threads rejoignent le thread principal et se terminent
```

```
Thread num= 1 a valeur1= 1000 et valeur 2= 2001
Thread num= 2 a valeur1= 1000 et valeur 2= 2001
Thread num= 3 a valeur1= 1000 et valeur 2= 2001
Thread num= 0 a valeur1= 1000 et valeur 2= 2001
```

Valeur 2 est 2001 pour tous les threads.

Exercice 3 :

1.

```
Debut Thread 2 ...
Nombre de threads = 4
Thread 2: c[0]= 0.000000
Debut Thread 3 ...
Debut Thread 0 ...
Thread 0: c[20]= 40.000000
Thread 0: c[21]= 42.000000
Thread 0: c[22]= 44.000000
Thread 0: c[23]= 46.000000
Thread 0: c[24]= 48.000000
Thread 0: c[25]= 50.000000
Thread 0: c[26]= 52.000000
Thread 0: c[27]= 54.000000
Thread 0: c[28]= 56.000000
Thread 0: c[29]= 58.000000
Thread 0: c[30]= 60.000000
Thread 0: c[31]= 62.000000
```

Le Programme lance 4 threads et affiche la valeur de c[i] (avec c[i]=2*i). L'affichage n'est pas forcément ordonné.

2.

```
int main (int argc, char *argv[])
{
int nthreads, tid, i, chunk;
float a[N], b[N], c[N];


for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Nombre de threads = %d\n", nthreads);
    }
  printf("Debut Thread %d ...\n",tid);

  #pragma omp for schedule(static,chunk)
  for (i=0; i<N; i++)
    {
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }
  }
```

```
Debut Thread 2 ...
Debut Thread 1 ...
Debut Thread 3 ...
Thread 2: c[20]= 40.000000
Thread 3: c[30]= 60.000000
Thread 3: c[31]= 62.000000
Thread 1: c[10]= 20.000000
Thread 3: c[32]= 64.000000
Thread 1: c[11]= 22.000000
Thread 3: c[33]= 66.000000
Thread 3: c[34]= 68.000000
Thread 3: c[35]= 70.000000
Thread 3: c[36]= 72.000000
Thread 3: c[37]= 74.000000
Thread 3: c[38]= 76.000000
Nombre de threads = 4
Debut Thread 0 ...
Thread 0: c[0]= 0.000000
Thread 1: c[12]= 24.000000
Thread 2: c[21]= 42.000000
Thread 2: c[22]= 44.000000
Thread 0: c[1]= 2.000000
Thread 1: c[13]= 26.000000
Thread 1: c[14]= 28.000000
Thread 2: c[23]= 46.000000
Thread 2: c[24]= 48.000000
```

3. Différence :

- Dynamic ordonnancement des travaux selon le principe du « premier arrivé, premier servi ».
- Static signifie que les itérations blocs sont cartographiées de manière statique à l'exécution de threads dans un « round-robin ».

5. Au début du programme, on a une seul Thread :

- « parallel » : divise le thread actuel en une nouvelle équipe de threads pour la durée du prochain bloc / instruction, après quoi l'équipe est fusionnée en un seul thread.
- « for » : divise le travail de la boucle for entre les threads de l'équipe actuelle. Il ne crée pas de threads, il divise simplement le travail entre les threads de l'équipe en cours d'exécution.
- « parallel for » : est un raccourci pour les deux commandes à la fois: parallel et for.

Parallèle crée une nouvelle équipe et, pour les divisions, cette équipe gère différentes parties de la boucle. Si votre programme ne contient jamais de construction parallèle, il n'y a jamais plus d'un thread; le thread principal qui démarre le programme et l'exécute, comme dans les programmes sans thread.

6. Collapse : Spécifie le nombre de boucles associées à une directive for (1 par défaut). Si l'expression entière vaut plus que 1, les itérations de toutes les boucles associées sont groupées pour former un unique espace d'itération qui sera réparti entre les threads. L'ordre des itérations de la boucle groupée correspond à l'ordre des itérations des boucles originales

Exemple :

```
#pragma omp fo r collapse ( 2 )
for ( i = 0; i < 10; i ++)
    for ( j = 0; j < 10; j ++)
        f ( i , j ) ;
```