Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Parallélisme & Applications

TP n°1

OpenMP

Compilation avec OpenMP

Pour pouvoir utiliser OpenMP, il est nécessaire de :

- inclure le fichier d'en-tête « `omp.h` » :

```
1 #include <omp.h>
```

- compiler avec l'option « `-fopenmp` » :

```
xterm
$ gcc -Wall -fopenmp -o Mon_prog mon_source.c
```

- Pour mesurer le temps d'exécution :

```
1 double temps_initial = omp_get_wtime();
2 /* Travail */
3 printf("Temps pris :%f\n", omp_get_wtime() - temps_initial);
```

Créations de threads multi-cœur

Dans les exercices suivants, vous pourrez fixer le nombre de threads au-delà du nombre réel de cœurs disponibles dans votre ordinateur à l'aide de la commande :

```
xterm
$ export OMP_NUM_THREADS=8
```

Tapez le programme suivant :

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

#define OCCUPATION 1

int main()
{
   int i = -1;

   #pragma omp parallel
   {
      int j;
      i = omp_get_thread_num();
      for(j=0;j<OCCUPATION;j++);
      printf("La valeur parallele est %d\n", i);
   }
   printf("La valeur sequentielle est %d\n", i);
}
```

1 – a. compilez et exécutez le programme :

```
xterm
$ ./mon_programme | sort
```

qu'est-ce vous observez ?

b. faites variez le paramètre « `OCCUPATION` » à la valeur 10, 100, 1000, 10000, 100000…
Qu'est-ce que vous observez ?

c. Comment « corriger » le problème ?

2 – Tapez le programme suivant :

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

int i;
#pragma omp threadprivate(i)

int main()
{
   i = -1;
   #pragma omp parallel
   {
      printf("La valeur parallele est %d\n", i);
   }
   printf("La valeur sequentielle est %d\n", i);
}
```

a. Qu'est-ce que vous observez à l'exécution ?

b. Modifiez votre code en ajoutant la clause « `copyin` » : `#pragma omp parallel copyin(i)`
Qu'est-ce que vous observez ?

Essayez maintenant le code suivant :

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

int main()
{
   int i = -1;
   #pragma omp parallel private(i)
   {
      printf("La valeur parallele est %d\n", i);
   }
   printf("La valeur sequentielle est %d\n", i);
}
```

Qu'elles sont les différences ?

3 – Tapez le programme suivant :

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

int i;
#pragma omp threadprivate(i)

int main()
{
   i = -1;
   #pragma omp parallel
   {
         #pragma omp single copyprivate(i)
         {
            i = 2;
         }
      printf("La valeur parallele est %d\n", i);
   }
   printf("La valeur sequentielle est %d\n", i);
}
```

Décrivez le fonctionnement de ce programme et ce que font les threads.

Barrière de synchronisation

4 – Tapez le code suivant :

```
#include <omp.h>
#include <stdio.h>
int main()
{

   #pragma omp parallel
   {
      #pragma omp master
      {
         printf( "thread master\n" );
      }
      #pragma omp barrier
      #pragma omp single
      {
         printf( "Thread single\n" );
      }
   }
}
```

Qu'est-ce que vous observez en testant le code précédent et en faisant varier le nombre de threads (vous pourrez également afficher le numéro de la thread) ?

Parallélisme de boucle

5 – On considère un tableau de n cases contenant chacune une valeur aléatoire :

a. Écrire un programme C qui lit la taille du tableau sur la ligne de commande, le crée et le remplit de valeurs aléatoires.

b. Écrire une méthode `carre(...)` prenant le tableau en paramètre et élevant la valeur de chacune des cases au carré.

c. Modifiez ce programme pour que les boucles « for » soient parallélisées par OpenMP.

d. Faites afficher par votre programme le nombre de *threads* qui sont crées par OpenMP. Que remarquez vous ?

e. Modifiez votre programme pour fixer le nombre de *threads* de la boucle for à 1, 5 puis 10.
Dans chaque cas, mesurez le temps pris par votre programme, que constatez vous ?

f. Cherchez comment à l'aide de la variable OMP_NUM_THREADS il est possible de modifier le nombre de threads sans modifier votre code.

6 – On veut profiter de la boucle de la méthode `carre()` pour faire la somme de tous les éléments du tableau.

a. Dans la méthode `carre(...)` précédente, déclarez une variable `int total` avant la boucle « for » et utilisez la pour faire la somme de toutes les cases du tableau dans la boucle.
Que vaut `total` en sortie de boucle? pourquoi ?

b. Indiquez maintenant que `total` est une variable privée.
Que vaut-elle en sortie de boucle? pourquoi ?

c. À l'aide d'une section critique, implémentez correctement la somme des éléments. Que constatez vous en terme de performances?

d. À l'aide d'une opération atomique, modifiez votre implémentation. Que constatez vous par rapport à la version précédente ?

e. Utilisez la « clause de réduction » pour arriver au même résultat et discutez les mérites de cette solution comparée à la précédente.

7 – Soit le programme suivant :

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

omp_lock_t lock;

double calcul( double* array, int length )
{
   double total = 0.0;
   #pragma omp parallel for
   for ( int i=0; i<length; i++ )
   {
      omp_set_lock( &lock );
      total += array[i];
      omp_unset_lock( &lock );
   }
   return total;
}
int main()
{
   double array[1024];
   omp_init_lock( &lock );
   calcul( array, 1024 );
   omp_destroy_lock( &lock );
}
```

Comprez le résultat au travail précédent.

Parallélisme de blocs

8 – Programmez une version OpenMP de l'algorithme QuickSort, dont la version séquentielle est donnée sur http://p-fb.net/.

```
xterm
./quicksort 5000000 100
```

9 – Parallélisation du tri-fusion :

a. Écrivez une version C « mono-thread » du tri fusion ;

b. Parallélisez cette version de de telle sorte que :
   - le tableau initial est divisé en 2 sous tableaux ;
   - ces 2 sous tableaux sont triés en parallèle ;
   - le résultat est fusionné pour donner le tableau final.