Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Parallélisme & Applications

Corrections – TP n°1

OpenMP

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 :

```
$ export OMP_NUM_THREADS=8
```

1 – Tapez le programme suivant :

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 #define OCCUPATION 1
 5
 6 int main()
 7 {
 8    int i = -1;

 9    #pragma omp parallel
10    {
11       int j;
12       i = omp_get_thread_num();
13       for(j=0;j<OCCUPATION;j++);
14       printf("La valeur parallele est %d\n", i);
15    }
16    printf("La valeur sequentielle est %d\n", i);
17 }
```

a. Qu'est-ce que vous observez ? *On observe que la valeur affichée par chacun des threads est parfois correct (une valant 0 et l'autre 1) ou bien toutes les deux valent 1.*

```
pef@solaris:~/ParallelismeII/TP1$ ./exo1 | sort
La valeur parallele est 0
La valeur parallele est 0
La valeur parallele est 4
La valeur parallele est 4
La valeur parallele est 6
La valeur parallele est 6
La valeur parallele est 7
La valeur parallele est 7
La valeur sequentielle est 7
```

b. Comment « corriger » le problème ?

Il faut rendre la variable i ***locale*** *à chaque thread.*

```
 9    ...
10   #pragma omp parallel private(i)
11   {
12      int j;
13      int i;
14      i = omp_get_thread_num();
15      for(j=0;j<OCCUPATION;j++);
16      printf("La valeur parallele est %d\n", i);
17    ...
```

```
La valeur parallele est 0
La valeur parallele est 1
La valeur parallele est 2
La valeur parallele est 3
La valeur parallele est 4
La valeur parallele est 5
La valeur parallele est 6
La valeur parallele est 7
La valeur sequentielle est -1
```

On remarque qu'après la région parallèle, `i` *reprend la valeur* `-1`*.*

2 – Tapez le programme suivant :

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 int i;
 5 #pragma omp threadprivate(i)
 6
 7 int main()
 8 {
 9    i = -1;
10    #pragma omp parallel
11    {
12       printf("La valeur parallele est %d\n", i);
13    }
14    printf("La valeur sequentielle est %d\n", i);
15 }
```

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

```
pef@solaris:~/ParallelismeII/TP1$ ./exo2
La valeur parallele est 0
La valeur parallele est 0
La valeur parallele est 0
La valeur parallele est -1
La valeur parallele est 0
La valeur parallele est 0
La valeur parallele est 0
La valeur parallele est 0
La valeur sequentielle est -1
```

Chaque thread possède une nouvelle variable `i`, la thread correspondant à la fonction `main` possède la variable i orginale et affiche la valeur `-1`. À la fin, la seule variable `i` est celle de `main`.

b. Avec le `#pragma omp parallel copyin(i)` Qu'est-ce que vous observez ?

```
pef@solaris:~/ParallelismeII/TP1$ ./exo2
La valeur parallele est -1
La valeur parallele est -1
La valeur parallele est -1
La valeur parallele est -1
La valeur parallele est -1
La valeur parallele est -1
La valeur parallele est -1
La valeur parallele est -1
La valeur sequentielle est -1
```

*La valeur de la variable i est **copiée** dans chaque thread.*

Essayez maintenant le code suivant :

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3 int main()
 4 {  int i = -1;
 5    #pragma omp parallel private(i)
 6    {
 7       printf("La valeur parallele est %d\n", i);
 8    }
 9    printf("La valeur sequentielle est %d\n", i);
10 }
```

Qu'elles sont les différences ?

```
La valeur parallele est 159875080
La valeur parallele est 11643966
La valeur parallele est 11643966
La valeur parallele est 11643966
La valeur parallele est 11643966
La valeur parallele est 11643966
La valeur parallele est 11643966
La valeur parallele est 11643966
La valeur sequentielle est -1
```

Chaque thread obtient sa variable `i` qui, étant non initialisée, contient n'importe quelle valeur. À la fin de la région parallèle, il ne reste plus que la variable `i` de `main`.

3 – Tapez le programme suivant :

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int i;
5  #pragma omp threadprivate(i)
6
7  int main()
8  {
9     i = -1;
10    #pragma omp parallel
11    {
12       #pragma omp single copyprivate(i)
13       {
14           i = 2;
15       }
16       printf("La valeur parallele est %d\n", i);
17    }
18    printf("La valeur sequentielle est %d\n", i);
19 }
```

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

```
pef@solaris:~/ParallelismeII/TP1$ ./exo3
La valeur parallele est 2
La valeur parallele est 2
La valeur parallele est 2
La valeur parallele est 2
La valeur parallele est 2
La valeur parallele est 2
La valeur parallele est 2
La valeur parallele est 2
La valeur sequentielle est 2
```

*Une des threads effectue un `copyprivate` et **diffuse**, broadcast, sa valeur de `i` aux autres threads à la fin de la directive `omp single`.*
*Une **barrière implicite** existe à la fin de cette directive `omp single`, ce qui veut dire que toutes les threads se synchronisent sur la fin de cette directive ⟹ toutes les threads obtiennent `2`.*
À la fin de la région parallèle, la variable de `i` de `main` conserve la valeur 2.

Barrière de synchronisation

4 – Tapez le code suivant :

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main()
4  {
5
6    #pragma omp parallel
7    {
8      #pragma omp master
9      {
10       printf( "thread master\n" );
11     }
12     #pragma omp barrier
13     #pragma omp single
14     {
15       printf( "Thread single\n" );
16     }
17   }
18 }
```

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) ?

On a toujours `master` puis `single` : seule la thread master travaille, mais la barrière de synchronisation impose que toutes les threads attendent même si elles ne travaillent pas.

```
pef@solaris:~/ParallelismeII/TP1$ ./barriere
thread master
Thread single
pef@solaris:~/ParallelismeII/TP1$ ./barriere
thread master
Thread single
```

Si on supprime le `barrier`, alors une thread qui ne travaillait pas peut faire faire la partie « single ».

```
pef@solaris:~/ParallelismeII/TP1$ ./barriere
thread master
Thread single
pef@solaris:~/ParallelismeII/TP1$ ./barriere
Thread single
thread master
```

▬▬▬ 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
#include <malloc.h>
#include <time.h>
#include <stdio.h>

void affiche(long int *tab, int taille)
{
   int i=0;

   printf("[");
   for(i=0; i < taille; i++)
      printf("%ld ",tab[i]);
   printf("\b]\n");
}

void carre(long int *tab, int taille)
{
   int i=0;

   for(i=0; i < taille; i++)
      tab[i] *= tab[i];
}
```

```c
int main()
{
   long int *tableau = NULL;
   int i,n;

   printf("Entrer la dimension : ");
   scanf("%d", &n);
   tableau = malloc( sizeof(long int) * n);

   srand( time(NULL) );

   for(i=0; i<n; ++i)
      tableau[i] = (rand() % 100);

   affiche(tableau,n);

   carre(tableau,n);

   affiche(tableau,n);
}
```

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

```c
#include <malloc.h>
#include <time.h>
#include <stdio.h>

void affiche(long int *tab, int taille)
{
   int i=0;

   printf("[");
   for(i=0; i < taille; i++)
      printf("%ld ",tab[i]);
   printf("\b]\n");
}

void carre(long int *tab, int taille)
{
   int i=0;

   #pragma omp parallel for
   for(i=0; i < taille; i++)
      tab[i] *= tab[i];
}
```

```c
int main()
{
   long int *tableau = NULL;
   int i,n;

   printf("Entrer la dimension : ");
   scanf("%d", &n);
   tableau = malloc( sizeof(long int) * n);

   srand( time(NULL) );

   #pragma omp parallel for
   for(i=0; i<n; ++i)
      tableau[i] = (rand() % 100);

   affiche(tableau,n);

   carre(tableau,n);

   affiche(tableau,n);
}
```

d. Faites afficher par votre programme le nombre de *threads* qui sont crées par OpenMP.
Que remarquez vous ? *Il y a autant de threads que de cœurs dans le processeur de l'ordinateur.*

e. Modifiez votre programme pour fixer le nombre de *threads* de la boucle for à 1, 5 puis 10.

```
$ echo 10000000 > taille_probleme
$ time ./exo5_omp < taille_probleme
Nombre de threads 1
Entrer la dimension :
real  0m0.355s
user  0m0.236s
sys   0m0.112s
```

```
$ echo 10000000 > taille_probleme
$ time ./exo5_omp < taille_probleme
Nombre de threads 5
Entrer la dimension :
real  0m1.332s
user  0m1.396s
sys   0m0.940s
```

```
$ echo 10000000 > taille_probleme
$ time ./exo5_omp < taille_probleme
Nombre de threads 10
Entrer la dimension :
real  0m1.475s
user  0m1.440s
sys   0m1.164s
```

On remarque que :

- ◇ *quand le nombre de threads est supérieur au nombre de cœurs : le temps n'est pas meilleur car les threads supplémentaires sont simulées ;*
- ◇ *le temps avec une seul thread peut être **très inférieur** au temps parallèle : il faut que la quantité de travail soit suffisante, sinon le surcoût du parallélisme est trop important.*

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.
On peut utiliser :
- ◇ *la commande shell* `export OMP_NUM_THREADS=7`
- ◇ *l'instruction* `omp_set_num_threads(NOMBRE_THREADS);`

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 ?

```
36 void carre(long int *tab, int taille)
37 {
38    int i=0;
39    long int total = 0;

40    #pragma omp parallel for
41    for(i=0; i < taille; i++)
42    {
43       tab[i] *= tab[i];
44       total += tab[i];
45    }
46    printf("Total = %ld\n",total);
47 }
```

On modifie la génération du tableau pour mettre dans chaque case la valeur de son index (éviter des valeurs aléatoires).
En exécution séquentielle :

```
pef@solaris:~/ParallelismeII/TP1$ ./exo6
Entrer la dimension : 1000
Total = 332833500
```

En exécution parallèle :

```
pef@solaris:~/ParallelismeII/TP1$ ./exo6
Entrer la dimension : 1000
Total = 276597487
```

On constate que la valeur de `total` est différente, erronée dans le cas de l'exécution parallèle.
***La variable `total` est partagée** entre les différentes threads d'où sa corruption possible à chaque modification concurrente.*

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

```
36 void carre(long int *tab, int taille)
37 {  int i=0;
38    long int total = 0;

39    #pragma omp parallel for private(total)
40    for(i=0; i < taille; i++)
41    {  tab[i] *= tab[i];
42       total += tab[i];
43    }
44    printf("Total = %ld\n",total);
45 }
```

```
pef@solaris:~/ParallelismeII/TP1$ ./exo6
Entrer la dimension : 1000
Total = 0
```

En déclarant la variable `total` comme privée pour chaque thread, à la sortie de la région parallèle on perd la valeur calculée.

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

```
36 void carre(long int *tab, int taille)
37 {  int i=0;
38    long int total = 0;

39    #pragma omp parallel for
40    for(i=0; i < taille; i++)
41    {  tab[i] *= tab[i];
42       #pragma omp critical
43       total += tab[i];
44    }
45    printf("Total = %ld\n",total);
46 }
```

```
pef@solaris:~/ParallelismeII/TP1$ ./exo6_c
Entrer la dimension : 1000
Total = 332833500
```

Le résultat est correct, mais les performances sont moins bonnes, car la section critique annule la possibilité de traiter en parallèle les calculs sur la variable partagée `total`.

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

```
36 void carre(long int *tab, int taille)
37 {  int i=0;
38    long int total = 0;

39    #pragma omp parallel for
40    for(i=0; i < taille; i++)
41    {  tab[i] *= tab[i];
42       #pragma omp atomic
43       total += tab[i];
44    }
45    printf("Total = %ld\n",total);
46 }
```

```
pef@solaris:~/ParallelismeII/TP1$ ./exo6_d
Entrer la dimension : 1000
Total = 332833500
```

Le résultat est correct et l'exécution est identique à celle de la version avec la section critique, ce qui est normal, la directive `critical` réalisant le même travail à l'échelle d'une seule instruction d'affectation/calcul.

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

```
36 void carre(long int *tab, int taille)
37 {  int i=0;
38    long int total = 0;

39    #pragma omp parallel for reduction(+:total)
40    for(i=0; i < taille; i++)
41    {  tab[i] *= tab[i];
42       total += tab[i];
43    }
44    printf("Total = %ld\n",total);
45 }
```

```
pef@solaris:~/ParallelismeII/TP1$ ./exo6_e
Entrer la dimension : 1000
Total = 332833500
```

Le résultat est correct, et la clause de réduction permet de combiner de manière efficace les différentes valeurs de `total` *à la fin de chaque thread suivant l'opérateur +.*
Cette version est la plus efficace et performante.

7 – Soit le programme suivant :

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 omp_lock_t lock;
 5
 6 double calcul( double* array, int length )
 7 {
 8   double total = 0.0;
 9   #pragma omp parallel for
10   for ( int i=0; i<length; i++ )
11   {
12     omp_set_lock( &lock );
13     total += array[i];
14     omp_unset_lock( &lock );
15   }
16   return total;
17 }
18 int main()
19 {
20   double array[1024];
21   omp_init_lock( &lock );
22   calcul( array, 1024 );
23   omp_destroy_lock( &lock );
24 }
```

Dans cette version, on crée une section critique explicitement à l'aide de loquet.
Les performances sont similaires à la version de l'exercice 6.c) utilisant une section critique.

Parallélisme de blocs

8 – Programmez une version OpenMP de l'algorithme QuickSort.

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3 #include <time.h>
 4
 5 #define N 10
 6
 7 void swap(int *a, int *b)
 8 {
 9    int bulle = *a;
10    *a = *b;
11    *b = bulle;
12 }
13
14 void quicksort(int *v, int deb, int fin)
15 {
16   if ( deb < fin )
17   {
18     int pivot = v[deb];
19     int separ = deb;
20     int i;
21
22    printf("Thread %d de %d a %d\n", omp_get_thread_num(),
23    deb, fin);
24     for ( i = deb + 1; i <= fin; ++i )
25     {
26       if ( v[i] < pivot )
27       {
28         ++separ;
29         swap(&v[i], &v[separ]);
30       }
31     }
32     swap(&v[deb], &v[separ]);
33    #pragma omp parallel sections shared(v)
34     {
35       #pragma omp section
36       {
37          quicksort(v, deb, separ - 1);
38       }
39       #pragma omp section
40       {
41          quicksort(v, separ + 1, fin);
42       }
43    }
44   }
45 }
```

```
46 void affiche(int *tab, int taille)
47 {
48    int i;
49    printf("[");
50    for(i = 0; i<taille; i++)
51       printf("%d, ",tab[i]);
52    printf("]\n");
53 }
54 int main()
55 {
56    int i;
57    int v[N];
58    srand(time(NULL));
59
60    for(i=0 ; i<N; i++)
61    {
62       v[i] = (rand() % 1000);
63    }
64
65    printf("Tableau original\n");
66    affiche(v, N);
67
68    quicksort(v, 0, N-1);
69    printf("\nTableau trie\n");
70    affiche(v, N);
71
72 }
```

*On peut également autoriser le **parallèlisme imbriqué** avec :*

```
$ export OMP_NESTED=TRUE
```

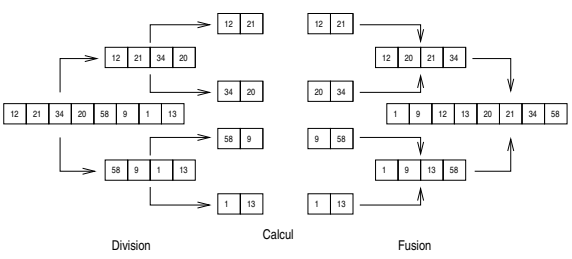
Ce qui permettra d'autoriser la création de nouvelles threads lors de la rencontre d'une nouvelle région parallèle dans une région parallèle.

Une version utilisant les « tasks *» permettrait d'autoriser la création de plus de travail parallèle par rapport à l'utilisation des sections et sans avoire recours aux «* nested *».*

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.



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

typedef struct {
  float val;
  int index;
} THEFIT;

THEFIT *work;
THEFIT *a;
#pragma omp threadprivate (work,a)

void RecMergeSort(int left, int right);
void Sort(THEFIT *Ain, int n);
void Merge(int s, int n, int m);
void merge2(THEFIT *d1,int n,THEFIT *d2,int m,
    THEFIT *out);

THEFIT *vector(int nl, int nh);
void free_vector(THEFIT *v, int nl);

void main() {
    THEFIT *data,*output;
    int i,j,k,k1,k2,k3,k4;
    printf("sort in c\n");
    i=32;
    data=vector(1,i);
    for(j=1;j<=i;j++) {
        data[j].index=j;
        data[j].val=(float)rand()/(float)RAND_MAX;
        printf("%d %g\n",data[j].index,data[j].val);
    }
    printf("\n\n");
    k=i/2;
    k1=k+1;
    k2=(i-k1)+1;
  #pragma omp sections
    {
  #pragma omp section
    Sort(&data[1],k);
  #pragma omp section
    Sort(&data[k1],k2);
    }
    for(j=1;j<=k;j++) {
        printf("%d %g\n",data[j].index,data[j].val);
    }
    printf("\n\n");
    printf("\n\n");
    for(j=k1;j<=i;j++) {
        printf("%d %g\n",data[j].index,data[j].val);
    }
    printf("\n\n");
    printf("\n\n");
    output=vector(1,i);
    merge2(&data[1],k,&data[k1],k2,&output[1]);
    for(j=1;j<=i;j++) {
        printf("%2d %10.7f\n",output[j].index,
            output[j].val);
    }
}
 void Sort(THEFIT *Ain, int n){
    work=vector(1,n);
    a=Ain-1;
    RecMergeSort(1,n);
    free_vector(work,1);
 }

  void RecMergeSort(int left, int right) {
    int  middle;
    if (left < right) {
        middle = (left + right) / 2;
        RecMergeSort(left,middle);
        RecMergeSort(middle+1,right);
        Merge(left,middle-left+1,right-middle);
    }
  }
THEFIT *vector(int nl, int nh)
{     THEFIT *v;

        v=(THEFIT *)malloc((unsigned)
          (nh-nl+1)*sizeof(THEFIT));
        if (!v) {
        printf("allocation failure in ivector()\n");
                  exit(1); }
        return v-nl;
}

void free_vector(THEFIT *v, int nl)
{   free((char*) (v+nl));
}
```

```c
  void Merge(int s, int n, int m) {
    int i,  j, k, t, u;
    k = 1;
    t = s + n;
    u = t + m;
    i = s;
    j = t;
    if ((i < t) && (j < u)){
        while ((i < t) && (j < u)){
            if (a[i].val >= a[j].val){
                work[k] = a[i];
                i = i + 1;
                k = k + 1;
            } else {
                work[k] = a[j];
                j = j + 1;
                k = k + 1;
            }
         }
    }
    if(i < t ){
        while (i < t ) {
            work[k] = a[i];
            i = i + 1;
            k = k + 1;
        }
    }
    if(j < u){
        while (j < u ) {
            work[k] = a[j];
            j = j + 1;
            k = k + 1;
        }
    }
    i = s;
    k=k-1;
    for( j = 1; j<= k; j++) {
        a[i] = work[j];
        i = i + 1;
    }
  }

void merge2(THEFIT *d1,int n,THEFIT *d2,int m,
      THEFIT *out)
{
  int i,j,k;
  i=1;
  j=1;
  d1--; d2--; out--;
  for( k=1; k<=n+m;k++) {
    if(i > n){
      out[k]=d2[j];
      j=j+1;
    }
    else if(j > m){
      out[k]=d1[i];
      i=i+1;
    } else {
      if(d1[i].val > d2[j].val){
        out[k]=d1[i];
        i=i+1;
      } else {
        out[k]=d2[j];
        j=j+1;
      }
    }
   }
  }
```