

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
10    #pragma omp parallel
11    {
12        int j;
13        i = omp_get_thread_num();
14        for(j=0; j<OCCUPATION; j++);
15        printf("La valeur parallele est %d\n", i);
16    }
17    printf("La valeur sequentielle est %d\n", i);
18 }
```

- 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$ ./ex01 | 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.

<pre>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 ...</pre>	<pre>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 parallele est 7 La valeur sequentielle est -1</pre>
---	--

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 ?

```
péf@solaris:~/ParallélismeII/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 sequentielle est -1
```

Chaque thread possède une nouvelle variable *i*, la thread correspondant à la fonction *main* possède la variable *i* originale 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 ?

```
péf@solaris:~/ParallélismeII/TP1$ ./exo2
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 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 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 \Rightarrow 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 :

- Écrire un programme C qui lit la taille du tableau sur la ligne de commande, le crée et le remplit de valeurs aléatoires.
- Écrire une méthode `carre(...)` prenant le tableau en paramètre et élevant la valeur de chacune des cases au carré.

```
1 #include <malloc.h>
2 #include <time.h>
3 #include <stdio.h>
4
5 void affiche(long int *tab, int taille)
6 {
7     int i=0;
8
9     printf("[");
10    for(i=0; i < taille; i++)
11        printf("%ld ", tab[i]);
12    printf("\b\n");
13 }
14 void carre(long int *tab, int taille)
15 {
16     int i=0;
17
18    for(i=0; i < taille; i++)
19        tab[i] *= tab[i];
20 }
```

```
20 int main()
21 {
22     long int *tableau = NULL;
23     int i,n;
24
25     printf("Entrer la dimension : ");
26     scanf("%d", &n);
27     tableau = malloc( sizeof(long int) * n);
28
29     srand( time(NULL) );
30
31     for(i=0; i<n; ++i)
32         tableau[i] = (rand() % 100);
33
34     affiche(tableau,n);
35     carre(tableau,n);
36     affiche(tableau,n);
37 }
```

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

```
1 #include <malloc.h>
2 #include <time.h>
3 #include <stdio.h>
4
5 void affiche(long int *tab, int taille)
6 {
7     int i=0;
8
9     printf("[");
10    for(i=0; i < taille; i++)
11        printf("%ld ", tab[i]);
12    printf("\b\n");
13 }
14 void carre(long int *tab, int taille)
15 {
16     int i=0;
17
18    #pragma omp parallel for
19    for(i=0; i < taille; i++)
20        tab[i] *= tab[i];
21 }
```

```
21 int main()
22 {
23     long int *tableau = NULL;
24     int i,n;
25
26     printf("Entrer la dimension : ");
27     scanf("%d", &n);
28     tableau = malloc( sizeof(long int) * n);
29
30     srand( time(NULL) );
31
32     #pragma omp parallel for
33     for(i=0; i<n; ++i)
34         tableau[i] = (rand() % 100);
35
36     affiche(tableau,n);
37     carre(tableau,n);
38     affiche(tableau,n);
39 }
```

- Faites afficher par votre programme le nombre de *threads* qui sont créés par OpenMP. Que remarquez vous ? Il y a autant de *threads* que de cœurs dans le processeur de l'ordinateur.
- 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 simulés ;
- ◇ 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:~/ParallélismeII/TP1$ ./exo6
Entrer la dimension : 1000
Total = 332833500

```

En exécution parallèle :

```

pef@solaris:~/ParallélismeII/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:~/ParallélismeII/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:~/ParallélismeII/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:~/ParallélismeII/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:~/ParallélismeII/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 **parallélisme imbriqué** avec :

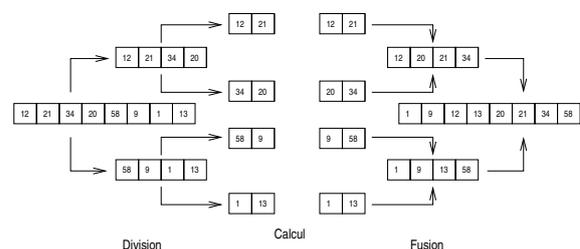
```
$ export OMP_NESTED=TRUE
```

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 avoir recours aux « nested ».

9 – Parallélisation du tri-fusion :

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



```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4
5 typedef struct {
6     float val;
7     int index;
8 } THEFIT;
9
10 THEFIT *work;
11 THEFIT *a;
12 #pragma omp threadprivate (work,a)
13
14 void RecMergeSort(int left, int right);
15 void Sort(THEFIT *Ain, int n);
16 void Merge(int s, int n, int m);
17 void merge2(THEFIT *d1, int n, THEFIT *d2, int m,
18     THEFIT *out);
19
20 THEFIT *vector(int nl, int nh);
21 void free_vector(THEFIT *v, int nl);
22
23 void main() {
24     THEFIT *data,*output;
25     int i,j,k,k1,k2,k3,k4;
26     printf("sort in c\n");
27     i=32;
28     data=vector(1,i);
29     for(j=1;j<=i;j++) {
30         data[j].index=j;
31         data[j].val=(float)rand()/(float)RAND_MAX;
32         printf("%d %g\n",data[j].index,data[j].val);
33     }
34     printf("\n\n");
35     k=i/2;
36     k1=k+1;
37     k2=(i-k1)+1;
38     #pragma omp sections
39     {
40     #pragma omp section
41     Sort(&data[1],k);
42     #pragma omp section
43     Sort(&data[k1],k2);
44     }
45     for(j=1;j<=k;j++) {
46         printf("%d %g\n",data[j].index,data[j].val);
47     }
48     printf("\n\n");
49     printf("\n\n");
50     for(j=k1;j<=i;j++) {
51         printf("%d %g\n",data[j].index,data[j].val);
52     }
53     printf("\n\n");
54     printf("\n\n");
55     output=vector(1,i);
56     merge2(&data[1],k,&data[k1],k2,&output[1]);
57     for(j=1;j<=i;j++) {
58         printf("%2d %10.7f\n",output[j].index,
59             output[j].val);
60     }
61 }
62 void Sort(THEFIT *Ain, int n){
63     work=vector(1,n);
64     a=Ain-1;
65     RecMergeSort(1,n);
66     free_vector(work,1);
67 }
68
69 void RecMergeSort(int left, int right) {
70     int middle;
71     if (left < right) {
72         middle = (left + right) / 2;
73         RecMergeSort(left,middle);
74         RecMergeSort(middle+1,right);
75         Merge(left,middle-left+1,right-middle);
76     }
77 }
78 THEFIT *vector(int nl, int nh)
79 {
80     THEFIT *v;
81
82     v=(THEFIT *)malloc((unsigned)
83         (nh-nl+1)*sizeof(THEFIT));
84     if (!v) {
85         printf("allocation failure in ivector()\n");
86         exit(1); }
87     return v-nl;
88 }
89 void free_vector(THEFIT *v, int nl)
90 {
91     free((char*) (v+nl));
92 }

```

```

1 void Merge(int s, int n, int m) {
2     int i, j, k, t, u;
3     k = 1;
4     t = s + n;
5     u = t + m;
6     i = s;
7     j = t;
8     if ((i < t) && (j < u)){
9         while ((i < t) && (j < u)){
10             if (a[i].val >= a[j].val){
11                 work[k] = a[i];
12                 i = i + 1;
13                 k = k + 1;
14             } else {
15                 work[k] = a[j];
16                 j = j + 1;
17                 k = k + 1;
18             }
19         }
20     }
21     if(i < t){
22         while (i < t) {
23             work[k] = a[i];
24             i = i + 1;
25             k = k + 1;
26         }
27     }
28     if(j < u){
29         while (j < u) {
30             work[k] = a[j];
31             j = j + 1;
32             k = k + 1;
33         }
34     }
35     i = s;
36     k=k-1;
37     for( j = 1; j<= k; j++) {
38         a[i] = work[j];
39         i = i + 1;
40     }
41 }
42
43 void merge2(THEFIT *d1, int n, THEFIT *d2, int m,
44     THEFIT *out)
45 {
46     int i,j,k;
47     i=1;
48     j=1;
49     d1--; d2--; out--;
50     for( k=1; k<=n+m;k++) {
51         if(i > n){
52             out[k]=d2[j];
53             j=j+1;
54         }
55         else if(j > m){
56             out[k]=d1[i];
57             i=i+1;
58         } else {
59             if(d1[i].val > d2[j].val){
60                 out[k]=d1[i];
61                 i=i+1;
62             } else {
63                 out[k]=d2[j];
64                 j=j+1;
65             }
66         }
67     }
68 }

```