

## Introduction à OpenMP

### 1 Création d'une équipe de threads

La directive `#pragma omp parallel` indique que l'instruction ou le bloc d'instructions suivant cette directive doit être réalisé par une *équipe* de threads. On appelle *section parallèle* l'instruction ou le bloc concerné par la directive.

```
1 #include <stdio.h>
2
3 int main()
4 {
5
6     printf("Bonjour_\n");
7     printf("Au_revoir_\n");
8
9     return 0;
10 }
```

1. Modifiez le programme `hello.c` en ajoutant une directive `parallel` en ligne 5.
2. Compilez puis exécutez le programme comme-ci de rien n'était : `gcc hello.c ; ./a.out`  
.
3. Recompilez en donnant cette fois l'option `-fopenmp` à `gcc`. Exécutez le programme plusieurs fois. Combien de threads sont utilisés ? Comparez ce nombre au nombre de coeurs (logiques) de votre machine. À quel moment s'affiche le message « Au revoir ! » ?
4. La variable d'environnement `OMP_NUM_THREADS` permet de contrôler le nombre de threads utilisés par défaut. Lancez l'exécution en entrant la commande `OMP_NUM_THREADS=13 ./a.out`  
.
5. La fonction `omp_get_thread_num()` retourne le numéro d'équipier (l'identité) du thread qui l'appelle. Sa mise en œuvre nécessite l'inclusion du fichier `<omp.h>`. Modifiez les deux appels à `printf()` de sorte à afficher également le numéro du thread appelant. Compilez puis exécutez plusieurs fois le programme en faisant varier le nombre de threads. L'ordre d'exécution est-il toujours le même ?
6. Faites en sorte que chaque thread affiche également le message « Au revoir » en englobant dans une seule section parallèle les deux appels à `printf`. Testez le programme avec une bonne douzaine de threads.
7. La directive `#pragma omp barrier` permet à une équipe de threads de se fixer un rendez-vous (de s'attendre) à une ligne de code donnée. Insérer une directive `barrier` entre les deux appels à `printf`. Compilez et testez le programme.

### 2 Section critique

La directive `#pragma omp critical` impose que chaque thread exécute l'instruction ou le bloc d'instructions suivant en solitaire. Le code gardé par la directive `critical` est appelé *section*

*critique* et on dit que la section critique est exécutée en *exclusion mutuelle*. Utiliser la directive `critical` pour faire en sorte que les affichages des différents threads ne s'entremêlent pas (NB. on supprimera la barrière placée lors de l'exercice précédent).

### 3 Exécution par un seul thread

Remplacer la directive `critical` pour la directive `master` et observer le comportement du programme sur plusieurs exécutions. Faire de même avec la directive `single`.

### 4 Distribution d'une boucle for

On cherche à calculer en parallèle la somme des éléments d'un vecteur :

```
1 int sum = 0;
2
3 for(int i = 0; i < N; i++)
4     sum+=t[i];
```

Une solution est d'utiliser une distribution cyclique (on obtient le nombre de threads engagés via un appel à `omp_get_num_thread()`):

```
1 int sum = 0;
2
3 #pragma omp parallel
4 {
5     int mon_i;
6     int ma_somme = 0;
7     int nb_threads = omp_get_num_thread();
8
9     for(mon_i = omp_get_thread_num(); mon_i < N ; mon_i += nb_threads)
10         ma_somme += t[mon_i];
11
12     #pragma omp critical
13         sum += ma_somme ;
14 }
```

La directive `#pragma omp for` permet de coder cela plus simplement :

```
1 int sum = 0;
2
3 #pragma omp parallel
4 {
5     int i;
6     int ma_somme = 0;
7
8     #pragma omp for schedule(static,1)
9         for(i=0; i < N; i++)
10             ma_somme += t[i];
11
12     #pragma omp critical
13         sum += ma_somme ;
14 }
```

La directive `omp for` indique que les indices de la boucle qui suit sont à répartir entre les threads et la clause `schedule` précise l'algorithme de distribution à utiliser. Il y a quatre principaux types de distribution :

- `schedule(static)` indique que les indices sont à distribués équitablement en autant de blocs d'indices consécutifs qu'il y a de threads;
- `schedule(static, k)` indique que les indices doivent être distribués cycliquement par blocs de `k` indices.
- `schedule(dynamic, k)` indique que les indices doivent être distribués à la demande par blocs de `k` indices. Notons que la version 2 du code peut être obtenue en utilisant la clause `schedule(dynamic, 1)`.
- `schedule(guided, k)` pour les curieux.

Notons enfin que la directive `pragma omp parallel for` est la contraction de l'enchaînement des directives `parallel` puis `for`.

Étudions ces différentes politiques à l'aide du programme `boucle-for.c` :

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 #include <omp.h>
5
6 int
7 main()
8 {
9     int i;
10
11     for(i=0; i < 40; i++)
12         printf("%d_traite_%i\n", omp_get_thread_num(), i);
13
14     return 0;
15 }
```

1. Tout d'abord placez une directive `parallel for` en ligne 10. Compilez et testez plusieurs fois.
2. Modifiez le programme pour tester différentes politiques de distribution. Utilisez la commande `./a.out | sort -n -k 3` pour visualiser plus facilement le travail réalisé.

## 5 Calcul sur les images

Utiliser OpenMP pour coder les noyaux `blur` et `mandel`.