Première série d'exercices sur OpenMP
Prise en main, Ordonnancement dynamique et ordonnancement statique

1. Examiner/compiler/exécuter le code de l'exemple Bonjour tout le monde

Vérifier que la bibliothèque openmp est installée sur votre machine, sinon vous la téléchargez,

su (entrez le password root)
apt-get install libgomp1

En fonction de votre compilateur c/c++, utiliser l'une des commandes suivantes pour compiler votre programme:

icc -openmp omp_bonjour.c -o bonjour
pgcc -mp omp_bonjour.c -o bonjour
gcc -fopenmp omp_bonjour.c -o bonjour

Pour exécuter le code, il suffit de taper la commande "**./bonjour"** et le programme doit s'exécuter.
Combien de threads on été créer ?
Pourquoi?

Varier le nombre de threads et relancer le programme "bonjour"

1. Définissez le nombre de thread en utilisant la variable d'environnement OMP_NUM_THREADS.
   setenv OMP_NUM_THREADS 4
   export OMP_NUM_THREADS=4
2. Ré-exécutez l'exemple de code et noter la sortie.
   ./bonjour
3. Votre sortie devrait ressembler à ci-dessous. L'ordre réel des chaînes de sortie peut varier d'une exécution à l'autre.

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 1
Hello World from thread = 2
```

2. Examiner/compiler/exécuter le code de l'exemple workshare1 (partage de boucle)

Cet exemple illustre l'utilisation de la boucle de partage de travail d'OpenMP. Remarquez qu'il spécifie l'ordonnancement dynamique des threads et leurs attribue le nombre d'itérations que doit faire chaque thread.

Nous supposons que la variable d'environnement **OMP_NUM_THREADS** est toujours à 4

gcc -fopenmp omp_workshare1.c -o workshare1
./workshare1 | sort

1. Examinez la sortie, noter qu'elle est pipée avec la commande sort(tri). Ainsi, il sera plus facile de voir comment la boucle d'itérations est effectivement ordonnancée sur le pool de threads.

2. Exécutez la double commande plusieurs fois et passer en revue la sortie. Que voyez-vous? En général l'ordonnancement dynamique n'est pas déterministe. Chaque fois que vous lancer le programme, différents threads peuvent exécuter des morceaux (blocs) de travail différents. Il est même possible qu'un thread peut ne pas faire aucun travail parce qu'un autre est plus rapide et demande plus de travail. En effet il pourrait être possible pour un thread d'effectuer l'ensemble des travaux.
3. Editez le fichier source et remplacer l'ordonnancement dynamique par l'ordonnancement statique (remplacez **dynamic scheduling** par **static scheduling)**
4. Recompilez et exécutez le programme ainsi modifié. Notez la différence de sortie comparée à celle produite par un ordonnancement dynamique. En particulier, notez que le thread 0 prend le premier morceau, le thread 1 le second et ainsi de suite
5. Exécutez le couple de programme encore plusieurs fois. Est-ce que la sortie change d'une exécution à une autre? Avec l'ordonnancement statique, la répartition du travail est déterministe, et ne doit pas y avoir de différence entre les différentes exécutions, et chaque thread obtient un travail à faire.
6. Réfléchir sur les différences de performances possibles entre la programmation dynamique et statique.

3. Examiner/compiler/exécuter le code de l'exemple workshare2e (partage de sections)

Cet exemple illustre l'utilisation des sections d'OpenMP pour la construction du travail partagé. Noter comment la région parallèle est divisée en différentes sections, dont chacune sera exécutée par un thread.

gcc -fopenmp omp_workshare2.c -o workshare2

./workshare2

Exécuter le programme plusieurs fois et observer toues les différences en sortie. Comme vous avez que 2 section, vous remarquerez que certains threads n'exécutent aucune tâche. Vous remarquerez ou vous ne remarquerez pas que les threads exécutant les tâches peuvent varier. Par exemple, la première fois le thread 0 et le thread 1 travaillent alors que lors d'une autre exécution, ce sont le thread 0 et le thread 3 qui travaillent. Il est même possible que le même thread peut effectuer tout le travail. Le choix du thread qui va faire le travail n'est pas déterministe dans ce cas

4. Examiner/compiler/exécuter le code de l'exemple sumVector (reduction et single)