Tâches OpenMP

1 Deux tâches

Modifiez le programme `deux-taches.c` pour qu'il exécute deux tâches en parallèle : chaque tâche écrira un mot et le numéro du thread qui l'exécute.

2 For en tâches

Modifiez le programme `for-en-taches.c` afin que les indices soient distribués au moyen de tâches tout en conservant un comportement globalement similaire.

3 Taskwait vs Barrier

Lancez plusieurs fois le programme `task-wait.c` avec 4 threads. Utilisez les trace produites pour analyser finement le comportement du programme. Remplacer la directive `taskwait` par une directive `barrier`. Observez les traces produites. Conclure.

4 Tâches et durée de vie des variables locales

Lancez plusieurs fois le programme `task.c` et analysez le comportement du programme. Observez ensuite le comportement du programme lorsque la directive `taskwait` est commentée.

Pour l'annecdote, observez que gcc sérialise l'exécution des tâches lorsque la clause `shared(chaine)` est remplacée par `firstprivate(chaine)`.

5 Parallélisation du TSP à l'aide de tâches OpenMP

Dupliquer le répertoire source initial pour paralléliser l'application à l'aide de tâches. Au niveau du main() il s'agit de créer une équipe de threads et de faire en sorte qu'un seul thread démarre l'analyse. Au niveau de la fonction tsp lancer l'analyse en faisant en sorte de ne créer des tâches parallèles que jusqu'au niveau `grain`. Deux techniques d'allocation mémoire sont à comparer :

1. allocation dynamique : un tableau est alloué dynamiquement et initialisé avant la création de la tâche - ce tableau sera libéré à la fin de la tâche ;
2. allocation automatique : le tableau est une variable locale allouée et initialisée dans la tâche - il est alors nécessaire d'utiliser la directive taskwait après avoir créé toutes les tâches filles.

Comparer les performance obtenues par les deux approches sur le cas 15 villes et seed 1234 pour des grains variant de 1 à 9. Comparer à celles obtenues à l'aide des techniques *imbriquées* et *collapse*.

Relever ensuite le(s) meilleur(s) grain(s) pour 12 et 24 threads. Calculer les accélérations obtenues.

6 Dépendances entre tâches

Dans le programme suivant les tâches peuvent être exécutées dans un ordre aléatoire. Il s'agit de faire en sorte qu'une tâche traitant le couple d'indices `(i,j)` doivent attendre que les tâches traitant les couples `(i-1,j)` et `(i,j-1)` soient terminées pour pouvoir être exécutée.

```
int main (int argc, char **argv)
{
  int A[T][T];;
  int i,j;

  int k = 0;

  for (i=0; i < T; i++ )
     for (j=0; j < 10; j++ )
#pragma omp task firstprivate(i,j)
#pragma omp atomic capture
        A[i][j] = k++;

 for (i=0; i < T; i++ )
    {
      puts("");
     for (j=0; j < T; j++ )
       printf(" %2d ",A[i][j]) ;
    }
}
```