TP4: TSP en OpenMP

Nous allons ici paralléliser TSP à l'aide d'OpenMP.

1 Expériences à réaliser

Dans les sections suivantes, nous allons réaliser quelques parallélisations OpenMP assez similaires à celles que nous avons faites avec pthread.

Rappel : `./tsp 14 1234` est censé retourner un chemin de longueur 245.

Dans chaque cas, comparez vos résultats avec ceux obtenus d'une manière similaire avec pthread. Notamment, faites varier le nombre de threads utilisés avec la variable d'environnement `OMP_NUM_THREADS` (pensez à essayer avec un seul thread pour comparer à la version purement séquentielle), ainsi que la valeur NUM.

Essayez aussi d'enlever l'option `-fopenmp` du Makefile, comparez le temps obtenu à celui de la version séquentielle. Pensez éventuellement à ajuster votre code pour optimiser ce cas (en utilisant des `#ifdef _OPENMP` au besoin). En déduire un intérêt d'OpenMP par rapport à utiliser les pthreads.

2 En utilisant notre gestion de tâches

Repartez de la version 2 de l'implémentation que vous avez faite lors des TPs précédents, c'est-à-dire la version séquentielle avec des tâches (n'utilisez pas les fichiers fournis avec le TP4, ils servent pour les sections suivantes seulement, car le type `Path_t` y est différent). Plutôt qu'utiliser une boucle de création de threads, utilisez simplement une boucle de la forme :

```
#pragma omp parallel
    while (!empty_queue(&q)) {
        Path_t path; int hops, len, cuts;
        if (get_job(&q, path, &hops, &len)
            tsp(hops, len, path, &cuts);
    }
```

Il faudra bien sûr ajouter quelques sections critiques aux mêmes endroits que la version pthread, sinon ce n'est que par chance si votre programme donne le bon résultat.

3 En créant de nombreux threads

Repartez de la version fournie avec le TP4, qui est la séquentielle originelle. Il "suffit" ici d'ajouter un
`#pragma omp parallel for if (hops < NUM)`
juste avant la boucle
`for (i=0; i < NrTowns; i++)`
de la fonction `tsp`.

Faites attention cependant à bien préciser quelles variables doivent être privées. Attention, on a changé le type de `Path_t` pour bien expliciter le fait que la variable `path` n'est ici qu'un pointeur, et non un tableau. Notamment, il faudra recopier `path` dans un nouveau tableau, sinon les threads vont se partager le même tableau pointé ! (dans la version pthread, on avait implémenté cette copie à votre place dans `tsp-job.c`).

Observez les performances, oui, ce n'est pas terrible du tout. C'est parce que l'implémentation d'OpenMP que l'on utilise (celle de gcc) n'optimise pas encore bien le cas où le `if` désactive le parallélisme, or ce surcoût est payé à chaque parcours de ville ! Pour obtenir des performances, il faut donc utiliser plutôt une structure de la forme :

```
if (hops < NUM) {
#pragma omp parallel for
    for (i=0; i < NrTowns; i++) {
        ...
    }
} else {
    for (i=0; i < NrTowns; i++) {
        ...
    }
}
```

Attention, pour qu'OpenMP effectue bien un parallélisme récursif pour NUM > 1, il faut positionner la variable d'environnement `OMP_NESTED` à `true` , car ce n'est pas activé par défaut.

4 En utilisant les tâches d'OpenMP

À partir de la version séquentielle fournie, parallélisez-la à l'aide de tâches. Pensez à utiliser un malloc (Path_t est désormais un pointeur plutôt qu'un tableau), à protéger l'accès à `minimum`, à éviter de créer des tâches à tous les niveaux, à éviter d'allouer et copier à tous les niveaux, et à désallouer aux bons moments.

Est-ce intéressant d'utiliser les tâches OpenMP plutôt que les faire soi-même ?