TP 2 : Le problème du voyageur de commerce

Résumé

Il s'agit de mettre en œuvre une version parallèle d'un programme de résolution du problème du Voyageur de Commerce à l'aide de threads. Afin de ne pas avoir à réécrire des parties de code sans rapport avec le sujet qui nous concerne, la version séquentielle vous est donnée. Comme d'habitude en parallélisme, on s'intéressera d'abord au découpage du problème, puis à sa répartition. Nous essaierons également de déterminer les phénomènes à l'origine du gain en performance d'une exécution parallèle par rapport à une exécution séquentielle.

À toutes fins utiles, on rappelle le problème à résoudre : un voyageur de commerce doit parcourir un ensemble de villes en passant par chacune des villes. L'ensemble des villes forme un graphe complet (i.e. on peut aller de n'importe quelle ville à n'importe quelle ville). Quel sera le chemin optimal, c'est-à-dire celui qui minimise la distance à parcourir ?

1 Exposition du problème

1.1 Fichiers de départ

Vous trouverez dans `~sathibau/PAP/TP2` un ensemble de fichiers qui va vous permettre de résoudre le problème posé sans avoir à partir *ex nihilo*.

- le fichier `tsp-types.h` contient un ensemble de structures de données qui seront utiles pour le problème ;
- le fichier `tsp-job.c` contient un ensemble de primitives permettant de gérer une liste de jobs ;
- le fichier `tsp-job.h` contient les prototypes de fonctions implémentées dans le fichier précédent ;
- le fichier `tsp-main.c` contient les fonctions d'initialisation des diverses structures ainsi que la boucle principale du programme ;
- le fichier `Makefile` permet de construire le tout.

1.2 Explications des types de tsp-types.h

- Une ville est désignée par un indice, dans l'intervalle $[\![0,$`NrTowns` $-1]\!]$, où `NrTowns` est limité à `MAXE`.
- Les distances entre villes sont stockées dans une matrice : `tab[ville1][ville2]` donne la distance entre la ville $ville1$ et la ville $ville2$.
- Une tâche (ou *job*) sera représentée par un chemin `path`, son nombre de villes `hops` et sa longueur `len` ;
- Vous disposez d'un module `tsp-job` qui permet de gérer une liste de tâches, il suffit de passer `&q` aux fonctions.

1.3 Squelette du programme : `main`

Dans le fichier `tsp-main.c`, vous trouverez un ensemble de variables globales. Celles dont vous aurez vraiment besoin (i.e. que vous allez manipuler fréquemment) sont :

- la liste de tâches : `TSPqueue q;`
- le tableau de distances : `DTab_t distance`.

Ne vous occupez pas de la fonction `genmap` : elle est responsable de la création d'une "carte" des villes, et de l'initialisation du tableau des distances. Vous n'aurez aucune modification à apporter à cette fonction.

En ce qui concerne le programme, vous pouvez constater qu'il prend deux arguments, qui sont le nombre réel de villes à parcourir et un élément de départ (*seed*) qui servira pour initialiser les générateurs de nombres aléatoires utilisés dans les premières fonctions. Le point important est que vous devez considérer qu'après un appel à la fonction `genmap`, vous disposez d'un jeu de données parfaitement valides et utilisables. De plus, comme ce jeu de données est généré de façon pseudo-aléatoire, la même *seed* donnera les mêmes résultats. Cela sera utile pour vérifier vos résultats.

2 Ce qu'il faut faire

Le but du jeu est de produire un programme qui affiche à l'écran la route optimale (d'un point de vue distance parcourue) ; c'est-à-dire qui génère un tableau d'indices correspondant à ladite route. Pour simplifier, nous considérons que la ville de départ est celle de numéro 0 (i.e. `path[0] = 0`).

2.1 Version séquentielle

Vous disposez déjà d'une version séquentielle. Étudiez son implémentation (basée sur une fonction récursive). Essayez-la pour vérifier qu'elle fonctionne (avec 14 villes et une `seed` 1234, on trouve un chemin minimal 245).

2.2 Version séquentielle (2)

Pour simplifier, avant de passer à une version parallèle nous allons d'abord effectuer le découpage du calcul, qui sera ensuite tout de même effectué de manière séquentielle :

- tout d'abord il faut constituer une liste de tâches à l'aide des fonctions fournies par `tsp-job.h` : nous allons considérer qu'une tâche est un début de chemin, de profondeur fixée `NUM`. Par exemple, si nous restreignons la profondeur à `NUM=3`, l'ensemble des tâches sera en fait constitué des chemins de 3 villes parcourues à partir de la ville 0 (notre point de départ) et dont tous les éléments sont distincts (on ne parcourt la ville qu'une fois). On ajoute ces tâches à l'aide de `add_job`.
- une fois tous les jobs ajoutés, on termine la liste de tâches en appelant `no_more_jobs` .
- ensuite, pour calculer la solution, il faut parcourir la liste des tâches, séquentiellement, à l'aide de `get_job`, et pour chacune d'entre elles, terminer le travail en appelant simplement `tsp`.

2.3 Version parallèle

Conservez cette version séquentielle dans un coin, pour pouvoir comparer.

La version parallèle utilisant des threads est finalement relativement simple : plusieurs threads se partagent la liste des tâches et piochent dedans jusqu'à épuisement du stock. À vous d'implémenter ceci (attention à bien verrouiller la gestion des jobs et la mise à jour de la variable `minimum` !). Le nombre de threads à créer sera un argument de votre programme.

Faut-il vraiment toujours un verrouillage pour accéder à la variable `minimum` ? Faut-il même vraiment un verrouillage pour la variable `minimum`, ou peut-on s'en passer, au moins en partie ? Comment éviter d'avoir besoin d'un verrouillage pour le comptage des `cuts` ? En principe, une fois ces problèmes résolus, vous devriez obtenir une accélération proche de 6 pour le cas 14 villes et une graine 1234 avec 8 threads (et proche de 8 avec 16 threads).

Tracez des courbes : speedup en fonction du nombre de threads (avec un nombre raisonable de villes pour que la mesure soit correcte, i.e. au minimum de l'ordre de la seconde), speedup en fonction du nombre de villes (avec autant de threads que de processeurs), speedup en fonction de la profondeur `NUM` des jobs.

Adaptez votre programme pour effectuer la création des jobs après avoir lancé les threads de calcul (ce qui permet donc de le faire en parallèle).

3 Tracé de courbes avec gnuplot

Pour tracer des courbes, utilisez `gnuplot` : remplissez un fichier `data` contenant sur chaque ligne la valeur x et la valeur y, par exemple

```
1 15445
2 7829
3 6431
```

Lancez `gnuplot`, et tapez `plot "data" with linespoints` . Pour que gnuplot calcule le speedup lui-même, on peut utiliser `plot "data" using ($1):(15300/($2)) with linespoints` (où 15300 est le temps séquentiel, à remplacer par le votre !).

On peut tracer plusieurs courbes à la fois en les séparant par des virgules : `plot "data" with lines, "data2" with linespoints`

On peut faire écrire la sortie dans un fichier en utilisant la commande `set terminal png` puis `set output "monfichier.png"`

Pour faire lancer automatiquement les différentes expériences, utilisez votre shell :

```
(for i in $(seq 1 10); do ./tsp 14 1234 $i; done) | tee data
```

Notez également qu'à l'aide de la commande shell `join`, on peut fusionner deux fichiers de données, ce qui permet ensuite de faire faire des calculs par `gnuplot`.