TP Systèmes 3: Parallélisme

Ensimag 2A

Édition 2014-2015

1 Introduction

Le but de ce TP est de vous familiariser avec la parallélisation d'une application en mémoire partagée. Pour cela nous allons utiliser la bibliothèque POSIX Thread.

Cette parallélisation doit calculer le plus vite possible la réponse au problème donné. Vous n'êtes pas autorisé à changer l'algorithme. Il faudra utiliser une machine parallèle (multi-cœurs, multi-processeurs) et répartir le travail entre plusieurs threads. La bonne synchronisation des calculs est nécéssaire pour obtenir une solution correcte.

2 Le voyageur de commerce

> Le problème du voyageur de commerce consiste, étant donné un ensemble de villes séparées par des distances données, à trouver le plus court chemin qui relie toutes les villes (Wikipedia `http://fr.wikipedia.org/wiki/Probl%C3%A8me_du_voyageur_de_commerce`).

Nous vous fournissons un « petit » programme de résolution du problème du voyageur de commerce par *Branch-and-Bound*.

Pour information, nous visons des instances environ 25-30 villes en quelques minutes quand les meilleurs algorithmes résolvent des instances avec plusieurs dizaines de milliers de villes en quelques jours.

2.0.1 Compilation et utilisation du programme

Vous devez modifier le script `CMakeLists.txt` pour y insérer vos logins.

```
cd ensimag-threads
cd build
cmake ..
make
make test
make check
```

Le programme de tests va démarrer une série d'exécution avec une instance connue et un nombre variable de threads. Il utilise valgrind (drd et helgrind).

Vous devriez vérifier que votre programme va effectivement plus vite avec une machine ayant plusieurs coeurs.

Le programme se lance avec la commande suivante :

```
./ensitsp nombre_de_ville graine nombre_de_threads
```

Avec l'option `-s`, il affiche aussi la solution sous la forme d'une image SVG (lisible avec firefox, chromium, inkscape, eog, display, ...). Cela permet de vérifier visuellement que votre programme construit une solution correcte.

Avec l'option `-p`, il affiche aussi la taille de la file de jobs au moment du retrait d'un job. Cela permet de suivre la progression de votre algorithme.

2.0.2 Génération des instances

Les instances sont générées pseudo-aléatoirement. La graine du générateur est le deuxième argu-

ment du programme. Le générateur (*rand48*) n'est pas un excellent générateur mais il est normalisé. Cela permet de rejouer facilement les mêmes instances quelque soit la machine.

2.1 Les heuristiques de coupe

Le *branch-and-bound* utilise un parcours en profondeur d'abord. L'algorithme part de la ville 0, puis ajoute la ville 1, puis 2, etc. Cela donne une première boucle. Il va ensuite essayer toutes les combinaisons possibles en partant des variations modifiants les dernières villes.

Il y a 20! permutations pour 20 villes. Il faut environ 77 ans de calcul à la vitesse d'une permutation par nano-seconde (et 600 000 fois l'age de l'univers pour 30 villes). Il est donc crucial de ne pas tester des permutations inutiles.

Quatre heuristiques sont utilisées pour diminuer l'espace des solutions exploré :

1. Les villes sont triées en fonction de leur argument complexe par rapport au barycentre des villes. Combiné au parcours en profondeur d'abord, cela permet de trouver rapidement une bonne solution ressemblant à un cercle.
2. Si il reste k villes à parcourir, il faudra au moins la somme des $k+1$ plus petites distances distinctes entre villes, sans symétrie. En ajoutant cette valeur à la longueur de la solution partielle actuelle, cela indique une borne inférieure de la taille du tour. Si la borne est plus grande ou égale à la meilleure solution, il est inutile d'explorer cette solution partielle.
3. L'arbre couvrant des villes restantes est aussi une borne inférieure du meilleur tour possible.
4. Si il reste encore un grand nombre de ville à placer, un programme linéaire simple calcule l'ensemble de sous-tours minimum. La solution à plusieurs sous-tours déconnectés. Elle est une borne inféreure de la solution avec un seul tour.

3 Travail demandé

Le travail consiste à paralléliser le code fourni. Pour vous aider, le code met dans une file d'attente un ensemble de tâches à calculer. Vous aurez donc principalement à modifier `tsp-main-para.c` pour calculer les tâches dans différents threads (déplacer du code, ajouter des fonctions, lancer les threads ...). Il faudra néanmoins aussi éditer les autres fichiers pour éviter les problèmes avec certaines structures de données partagées.

3.1 Synchronisation

Trois points notable dans la synchronisation :

1. La manipulation de la file de *jobs* à faire en exclusion mutuelle.
2. La manipulation des variables globales comptants les coupes et la valeur de la solution minimum à faire en exclusion mututelle. Pour éviter d'avoir à utiliser constamment un mutex sur le minimum, ce qui reduira les performances, vous pouvez le copier dans une variable locale au thread pour l'utilisation courante et ne faire la mise à jour (protégé par un mutex) qu'après un certain nombre de lecture (à calibrer)
3. Pour faciliter la portabilité, le solveur linéaire est utilisé en écrivant un fichier au format LP, lisible par tous les solveurs. Il faudra utiliser des noms différents pour chaque thread.

4 Rendu des sources et des mesures

L'archive des sources que vous devez rendre dans `Teide` est généré par le makefile créé par cmake :

```
cd ensimag-threads
cd build
make package_source
```