Programmation Parallèle

Fiche 2 : Parallélisme imbriqué

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/pap/

1 Le voyageur de commerce

On cherche à optimiser une tournée d'un commercial, tournée passant par un ensemble de villes et revenant à son point départ. Ici on considère un espace euclidien dans lequel les villes sont toutes connectées deux à deux.

1.1 Quelques mots sur le code

Le nombre de villes est contenu dans `nbVilles` et la variable `minimun` contient la longueur de la plus petite tournée connue.

Lors d'un appel `void tsp (etape, lg, chemin, mask)`, le paramètre `chemin` contiendra un chemin de `etape` entiers (villes) tous distincts ; la longueur de ce chemin est `lg` ; le i-ième bit de la variable `mask` est à 1 si et seulement si la ville i apparait dans le tableau chemin.

La variable `grain` contient le niveau de parallélisme imbriqué demandé (0 - pas de parallélisme ; 1 - une seule équipe de thread est créée au premier niveau de l'arbre de recherche ; 2 - des équipes de threads sont en plus créées au niveau 2 de l'arbre, etc).

1.2 Version séquentielle

Étudiez rapidement l'implémentation fournie. Essayez-la pour vérifier qu'elle fonctionne (avec 12 villes et une `seed` 1234, on trouve un chemin minimal 278) :

```
./tsp-main 12 1234 seq
```

À des fins de calcul d'accélération, mesurez le temps nécessaire pour le cas 13 villes et une `seed` 1234. Notez que l'analyse de l'arbre de recherche est exhaustive et donc l'analyse engendrée par deux débuts de chemins de `k` villes nécessitent le même nombre d'opérations (leur complexité ne dépend finalement que `k` et du nombre de villes).

2 Parallélisation en créant de nombreux threads

Dans un premier temps, on va chercher à paralléliser correctement le programme, sans vraiment chercher à optimiser ses performances. Recopier dans la fonction `tsp_ompfor` le code de la fonction `tsp_seq`. Puis ajouter le pragma suivant juste avant la boucle qui lance l'exploration récursive des sous arbres.

```
#pragma omp parallel for if (etape <= grain)
```

Tester la version parallèle :

```
./tsp-main 12 1234 0 ompfor  # grain == 0 pas de parallélisme
./tsp-main 12 1234 1 ompfor  # grain == 1 1 seul niveau
```

Poursuivre la parallélisation du code en faisant maintenant attention aux variables partagées ou privées. En particulier, il s'agit d'éviter aux threads de tous travailler sur le même tableau partagé. Il s'agit aussi de protéger les accès concurrents à la variable `minimun`.

Une fois votre programme au point, comparez les performances des versions parallèle et séquentielle. Les performances obtenues ne devraient pas être spectaculaires car ce programme recopie beaucoup de chemins et, de plus, l'utilisation du `#pragma parallel` introduit un surcoût même lorsque qu'une clause **`if`** désactive le parallélisme.

En particulier, observez les performances obtenues en faisant varier le paramètre graine (3e paramètre). Notons que pour créer des threads récursivement il faut positionner la variable d'environnement `OMP_NESTED` [1] à `true` (ou bien exécuter `omp_set_nested(1)` au début de la fonction `main`) car, par défaut, le support d'exécution d'OpenMP empêche la création récursive de threads.

2.1 Optimisation du surcoût de la parallélisation

Tout d'abord, il s'agit d'éliminer les surcoûts inutiles en passant sur la fonction séquentielle dès qu'on ne génère plus de parallélisme :

```
void tsp_ompfor(...)
{
  if (etape > grain) { // version séquentielle
    tsp_seq(...);
  } else { // version parallèle
    #pragma omp parallel for ...
     ...
  }
}
```

2.2 Optimisation du surcoût de l'accès à la variable minimum

Observez qu'il n'est pas utile de protéger par une section critique tous les accès à la variable minimum : seules doivent se faire en section critique les comparaisons susceptible d'entrainer une modification du minimum.

2.3 Parallélisation à l'aide de la directive collapse

Une autre façon de paralléliser le code est de distribuer aux threads tous les chemins de `n` étapes depuis la ville 0. Il s'agit de distribuer des n-uplets de numéros de ville en utilisant la directive `collapse(n)`.

Insérez les fonctions présentes dans le fichier `collapse.c` et faites en sorte qu'elles puissent être appelées depuis le `main()`.

Comparez les temps d'exécution des différentes méthodes.

1. Note : la variable `OMP_MAX_ACTIVE_LEVEL` remplace `OMP_NESTED` à partir d'OpenMP 5.0.

3 Optimisation de nature algorithmique

Il est vain de poursuivre l'évaluation d'un début de chemin lorsque sa longueur est supérieure au minimum courant (correspondant à la longueur du chemin complet le plus petit qu'on a déjà trouvé). Pour mettre en oeuvre cette optimisation, insérez le test suivant au début des fonctions `tsp_xxx()` récursives.

```
if (lg +  distance[0][chemin[etape-1]]>= minimum)
    return;
```

Comparez les performance des différentes fonctions le cas 15 villes et la graine 1234.

Commentaires : Un des effets de cette optimisation est de déséquilibrer le calcul car l'analyse des chemins n'est plus exhaustive : certaines branches sont coupées très haut, d'autres très bas. Le code optimisé a un comportement peu prévisible car la complexité du calcul dépend des résultats intermédiaires. On dit que le code optimisé a un comportement *irrégulier* : en séquentiel et à nombre de villes égales, deux configurations différentes auront des temps de calcul différents. Renforcé par le caractère non déterministe du parallélisme, le comportement de l'application parallèle est maintenant peu prédictible : pour un même cas on observe des variations significatives (eg. 25%) du temps de calcul entre différentes exécutions parallèles.