

# Parallélisme imbriqué en OpenMP

## 1 Le problème du 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, mesurer 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. Par exemple il s'agit d'éviter aux threads de tous travailler sur même tableau partagé. Il s'agit aussi de protéger les accès concurrents à la variable `minimum`.

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

En particulier observez les performances obtenues en faisant varier le paramètre graine (3ième paramètres). Notons que pour créer des threads récursivement il faut positionner la variable d'environnement `OMP_NESTED` à `true` (ou bien faire l'appel `omp_set_nested(1)`) 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

Tout d'abord observer 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'entraîner une modification du `minimum`.

## 2.3 Parallélisation à l'aide de la directive collapse

Insérer la fonction suivante et faire en sorte qu'elle puisse être appelée depuis le `main()` :

```
void tsp_ompcol()
{
    int i,j,k;
    #pragma omp parallel for collapse(3)
    for (i=1; i < nbVilles; i++)
        for(j=1; j < nbVilles; j++)
            for(k=1; k < nbVilles; k++)
                if(i != j && i != k && j != k)
                {
                    chemin_t chemin;
                    chemin[0] = 0;
                    chemin[1] = i;
                    chemin[2] = j;
                    chemin[3] = k;
                    int dist = distance[0][i] + distance[i][j] + distance[j][k];
                    tsp (4, dist, chemin, 1 | (1<<i) | (1<<j) | (1<<k)) ;
                }
}
```

```

    }
}

```

## 2.4 Mesure de performance

On considère le cas 13 villes et seed 1234. Recopiez et adaptez les scripts donnés pour le mini-projet pour tracer les courbes d'accélération pour 2 4 8 12 et 24 threads pour la version `collapse` et la version parallèle avec un grain de 1. Ajouter les points correspondant aux accélérations pour la version imbriquée pour les grains 2, 3 et 4 (en prenant 12 threads).

## 3 Optimisation de nature algorithmique

Il est stupide 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_...()` récursives.

```

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

```

Comparer les performance des différentes fonctions le cas 15 villes et la graine 1234. Faire une courbe de speed-up pour la version collapse.

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.