OpenMP

Copiez le répertoire `/net/cremi/rnamyst/etudiants/pmg/TP2` sur votre compte.

Par défaut OpenMP utilise autant de threads que le système d'exploitation lui présente de cœurs. Cependant, le nombre de threads peut être fixé depuis le shell via une variable d'environnement :

```
export OMP_NUM_THREADS=4
./a.out
```

ou bien

```
OMP_NUM_THREADS=4 ./a.out
```

De même il est possible de définir une politique de distribution des indices en utilisant de façon combinée la variable `OMP_SCHEDULE` et la politique de distribution `schedule(runtime)`.

Par exemple : `OMP_SCHEDULE="STATIC,4" ./a.out`

1 Calcul de la somme d'un tableau - réduction

Le programme `sum.c` effectue la somme d'un tableau d'entiers strictement positifs. Parallélisez la boucle avec un `schedule(static)` et sans utiliser de verrouillage. Observez que le résultat devient incorrect. Il faut protéger l'accès à la variable `sum`.

Recopiez le code parallélisé pour essayer différentes techniques de protection :

1. une variable partagée protégée par un verrou OpenMP (lock),
2. une variable partagée accédée en section critique,
3. une variable partagée accédée de manière atomique,
4. une variable utilisant la réduction.

On implémentera les différentes stratégies dans le programme `sum.c`. Comparez les temps d'exécution obtenu par les politiques de distribution `static`, `static,1` et `dynamic`.

2 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.

2.1 Quelques mots sur le code

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

Lors d'un appel `void tsp (hops, len, path)`, le paramètre `path` contiendra un chemin de `hops` entiers (villes) tous distincts ; la longueur de ce chemin est `len`.

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).

2.2 Version séquentielle

Etudiez l'implémentation fournie. Essayez-la pour vérifier qu'elle fonctionne (avec 12 villes et une `seed` 1234, on trouve un chemin minimal 278). Vous pouvez décommenter l'appel à `printPath` pour observer la solution, mais pour les mesures de performances on ne gardera pas l'affichage. A 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).

2.3 Parallélisation en créant de nombreux threads

Dupliquer le répertoire source. Puis insérer le pragma suivant :
`#pragma omp parallel for if (hops <= grain)`
juste avant la boucle qui lance l'exploration des sous arbres :
`for (i=1; i < NrTowns; i++)`
de la fonction `tsp`.

Poursuivez la parallélisation du code en faisant attention aux variables partagées ou privées. Par exemple il s'agit d'éviter aux threads de tous travailler sur un unique et même tableau. Notons qu'un tableau ne peut être rendu privé, il est donc nécessaire de recopier le tableau `path` dans un nouveau tableau (alloué dans la pile). Protégez également les accès concurents à la variable `minimun`.

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.

Les perfomances obtenues ne devraient pas être terrible du tout car ce programme recopie beaucoup trop de chemins et, de plus, l'utilisation du pragma parallel a un surcoût même lorsque qu'une clause `if` désactive le parallélisme.

2.4 Optimisations de la parallélisation

Tout d'abord il s'agit d'éliminer les surcoûts inutiles en dupliquant ainsi le code :

```
if (hops <= grain) { // version parallèle
#pragma omp parallel for ...
    for (i=1; i < NrTowns; i++) {
        ...
    }
} else { // version séquentielle
    for (i=1; i < NrTowns; i++) {
        ...
    }
}
```

Ensuite il faut faire en sorte de ne créer que le nombre nécessaire de threads et, par conséquent, d'utiliser une politique de répartition dynamique.

Enfin on observe 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.

Noter les performances obtenus pour le cas 13 villes et pour les grains allant de 0 à 5. Calculer les accélérations obtenues par rapport à la version séquentielle.

2.5 Parallélisation à l'aide de la directive collapse

Dupliquer le répertoire source initial. Puis insérer la fonction suivante et l'appeler directement dans le main() :

```
void par_tsp ()
{
  int i,j,k;
#pragma omp parallel for collapse(3) schedule(runtime)
 for (i=1; i < NrTowns; i++)
   for(j=1; j < NrTowns; j++)
     for(k=1; k < NrTowns; k++)
       if(i != j && i != k && j != k)
         {
          int chemin[NrTowns];
          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) ;
         }
}
```

Calculer les accélérations obtenues pour le cas (13 villes et seed 1234) pour les codes suivants : collapse + distribution dynamique ; collapse + distribution statique ; équipes imbriquées.

Noter les performances obtenus pour le cas 13 villes et collapse(3). Calculer l'accélération obtenue par rapport à la version séquentielle.

2.6 Optimisation de nature algorithmique

Clairement, il est inutile 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 trois versions du tsp.

```
if (len +  distance[0][path[hops-1]]>= minimum)
    return;
```

En considérant le cas « 15 villes, seed 1234 », calculer à nouveau les accélérations obtenues dans cas suivante :

1. équipes imbriquées.
2. collapse(3) + distribution dynamique ;
3. collapse(3) + distribution statique ;
4. collapse(4) + distribution dynamique ;
5. collapse(4) + distribution statique ;

Les résultats sont-ils surprenant ?

Commentaires : Un des effets de cette optimisation est de désiquilibrer le calcul car l'analyse des chemins n'est plus exhaustive : certaines branches sont coupées très haut, d'autres très bas. Le code optimisé à 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.

3 Tâches OpenMP

Modifiez le programme `deux-taches.c` pour qu'il exécute deux tâches `A` et `B` en parallèle.

4 For en tâches

Modifiez le programme `for-en-taches.c` afin que les indices soient distribués au moyen de tâches (en lieu et place de la direcrive `omp for` tout en conservant un comportement globalement similaire.

5 Taskwait vs Barrier

Lancez plusieurs fois le programme `task-wait.c` avec 4 threads. Utilisez les trace produites pour analyser finement le comportement du programme. Remplacer la directive `taskwait` par une directive `barrier`. Observez les traces produites. Conclure.

6 Tâches et durée de vie des variables locales

Lancez plusieurs fois le programme `task.c` et analysez le comportement du programme. Observez ensuite le comportement du programme lorsque la directive `taskwait` est commentée [1].

7 Dépendances entre tâches

Dans le programme `depend.c` original, les tâches peuvent être exécutées dans un ordre aléatoire. Il s'agit de faire en sorte qu'une tâche traitant le couple d'indices `(i,j)` doivent attendre que les tâches traitant les couples `(i-1,j)` lorsque `i > 0` et `(i,j-1)` lorsque `j > 0` soient terminées pour pouvoir être exécutée.

1. Pour l'anecdote, observez si gcc sérialise ou pas l'exécution des tâches lorsque la clause `shared(chaine)` est remplacée par `firstprivate(chaine)`.