INF5171 : Cours–Laboratoire #8
Programmation parallèle avec `OpenMP/C`

Mardi, 14 novembre 2017

Introduction

Le but de ce laboratoire est de vous familiariser avec l'utilisation des diverses constructions du langage OpenMP/C.

- Pour obtenir le code source — à exécuter sur `japet` :

```
$ git clone http://www.labunix.uqam.ca/~tremblay/git/LaboOpenMP.git
```

- Pour ces exercices, les programmes fournis contiennent déjà certaines instructions `OpenMP`, mais ne contiennent toutefois aucune directive (`pragma`) pour rendre l'exécution parallèle. Votre travail consiste donc à ajouter de telles directives, et ce dans le but de comprendre l'effet des diverses directives.

- Un fichier `makefile` vous est fourni et permet de compiler les divers programmes sources, que vous pouvez ensuite exécuter avec les arguments appropriés. Exemple :

```
$ make pi
$ ./pi 100000 8


$ make max
$ ./max 1000000 8


$ make somme-1-a-n
$ ./somme-1-a-n 1000 4
```

1 Calcul de π par intégration numérique

Modifiez le programme `pi.c` pour le rendre parallèle, et ce de façon à obtenir une bonne accélération.

Voici les choses à essayer au niveau des pragmas, et à comparer en exécutant le programme avec divers nombres de processeurs **pour les items** c.**,** d. **et** e. :

a. Introduisez une directive `omp parallel` simple et voyez l'effet qui en résulte.

b. Introduisez une directive `omp for` et voyez l'effet.

c. Introduisez une directive `omp critical` et voyez l'effet.

d. Remplacez la directive `omp critical` par `omp atomic` et voyez l'effet **sur le temps d'exécution**.

e. Modifiez la directive `omp for` en ajoutant une clause `reduction` et voyez l'effet **sur le temps d'exécution**.

f. Mettez en commentaire l'instruction «`omp_set_dynamic(1);`» et voyez l'effet lorsque vous spécifiez (sur la ligne de commande ou via le `makefile`) un **très grand nombre de *threads*** — plus grand que le nombre de processeurs!

Cible `makefile` **:** `run_pi`

2 Recherche du maximum parmi un tableau d'entiers

a. Modifiez le programme `max.c` pour le rendre parallèle, et ce en utilisant du parallélisme de boucles **avec réduction**.

Indice : Dans ce cas, l'opérateur à utiliser est `max`, par exemple :

```
#pragma ... reduction(max: leMax)

if ( a[i] > leMax ) {
  leMax = a[i];
}
```

b. Dans l'instruction `if` qui détermine si la variable pour le maximum doit être mise à jour, que se passe-t-il si vous mettez un autre opérateur que «>»? Pouvez-vous expliquer ce qui se passe?

c. Dans la directive `omp for`, comparez l'effet d'ajouter l'une ou l'autre clause suivante de répartition du travail entre les *threads* :

```
... schedule(static) ...
... schedule(static, 3) ...
... schedule(dynamic) ...
... schedule(dynamic, 3) ...
... schedule(guided) ...
```

Pour comprendre l'effet, définissez la constante `DEBUG` à 1, ce qui indiquera, pour chacune des itérations, quel *thread* exécute l'itération en question. Utilisez ensuite le script `analyser-traces-max.rb` — par l'intermédiaire de la cible `run_max_analyse` du `makefile` — pour identifier quelles itérations ont été exécutées par chacun des *threads*. Qu'en concluez-vous?

Cibles `makefile` **:** `run_max`, `run_max_analyse`

3 Somme de 1 à n avec parallélisme récursif

Le fichier `somme-1-a-n.c` contient une fonction récursive qui permet de calculer la somme de `1` à `n`.

a. Parallélisez la fonction `somme` à l'aide de **directives** `task` et `taskwait`.

b. Activez la trace d'impression (instruction `printf` juste avant l'instruction `return`) à la fin de la fonction `somme` puis exécutez la cible `run_somme_analyse` en faisant varier le nombre de *threads*. Que constatez-vous?

   Exécutez

Cibles `makefile` : `run_somme`, `run_somme_analyse`

4 Fonctions pour effectuer le produit de polynomes

Le fichier `polynomes.c` contient une mise en oeuvre séquentielle d'opérations sur des polynomes. L'interface du type abstrait est dans le fichier `polynome.h`.

a. Modifiez la mise en oeuvre des fonctions `coefficient` et `fois` — en ajoutant des *pragmas* `OpenMP` — pour effectuer de façon parallèle **le produit de polynomes**.

b. Exécutez les mesures de temps d'exécution, avec la cible `bm`, et ce pour différents nombre de *threads* (cf. `makefile`). Constatez-vous une accélération?

c. Dans vos fonctions `coefficient` et `fois`, utilisez un mode de répartition du travail de type «`schedule(runtime)`», i.e., avec des boucles parallèles ayant l'allure suivante :

```
# pragma omp parallel for ... schedule(runtime)
```

Ensuite, dans le fichier `makefile`, modifiez la valeur de la variable `OMP_SCHEDULE`. Puis, exécutez à nouveau les mesures de temps d'exécution, et ce pour différentes valeurs.

Quelle semble être la(les) valeur(s) de «`schedule`» qui permet(tent) d'obtenir les meilleures performances? Pouvez-vous expliquer pourquoi il en est ainsi?

- Pour compiler :

  `make compiler_polynomes`

- Pour tester que les résultats sont corrects :

  `make tester_polynomes`

- Pour mesurer les performances :

  `make bm`