OpenMP

Copiez le répertoire `/net/cremi/rnamyst/etudiants/pmg/TP-OpenMP` 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 soit en entrant

```
export OMP_NUM_THREADS=4

./a.out
```

ou même

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

Pour gagner du temps et éviter de recompiler vos programmes pour tester différentes stratégies, il est possible de définir une politique de distribution des indices « `schedule(runtime)` » dans le code source, et de définir la stratégie voulue au moyen de la variable d'environnement `OMP_SCHEDULE`. Vous pourrez typiquement lancer vos programmes de la façon suivante :

```
OMP_SCHEDULE="STATIC,4" ./a.out
```

1 Politique de distribution

Paralléliser la boucle for des programmes `boucle-for-1.c` et `boucle-for-2.c` en utilisant les politiques de scheduling static, dynamic et guided.

Observez la différence de comportement entre ces trois politiques : la distribution des indices pour le premier programme, et le temps d'exécution pour le second. Utilisez la commande unix `sort -n -k 3` pour visualiser plus facilement le travail réalisé par chaque thread.

2 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 accédée en section critique,
2. une variable partagée accédée de manière atomique,
3. 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` et `static,1` (c'est-à-dire cyclique [1]).

1. Contrairement à ce que j'ai dit en cours lundi dernier, le mot-clé « cyclic » est utilisé dans plusieurs langages mais pas en OpenMP... ;)

3 Parallélisation du programme film.c

Paralléliser brutalement le programme film. Mesurer le speddup obtenu sur votre machine et sur un serveur AMD. On va maintenant essayer de réduire le temps d'exécution du programme en optimisant l'accès aux données :

1. améliorer la localité du programme séquentiel en échangeant les boucles i et n ;
2. dérouler la boucle en i de telle façon à traiter 4 lignes à la fois (on compare ainsi par bloc de 20 ko de données au lieu de 4,25) ;
3. paralléliser le traitement des pixels (attention aux variables partagées) ;
4. limiter la contention sur les variables partagées ;
5. aligner le début du tableau sur une frontière de page (4096 octets) en utilisant : `__attribute__((aligned(409`
6. paralléliser l'initialisation de façon à ce que les pixels soient stockés à proximité du thread qui les traitera (il s'agit de paralléliser l'initialisation en suivant le même schéma d'accès aux données que celui du calcul - il s'agira aussi d'utiliser la variable `OMP_PROC_BIND=true`).

Comparer les temps d'exécution de la fonction différence obtenu suite à chaque optimisation.

4 Le problème du voyageur de commerce en OpenMP

4.1 Préambule

Il s'agit de mettre en œuvre une version parallèle d'un programme de résolution du problème du Voyageur de Commerce. On rappelle le problème à résoudre : un voyageur de commerce doit parcourir un ensemble de villes en passant par chacune des villes. L'ensemble des villes forme un graphe complet (i.e. on peut aller de n'importe quelle ville à n'importe quelle ville). Quel sera le chemin optimal, c'est-à-dire celui qui minimise la distance à parcourir ?

Dans un premier temps, vous pouvez examiner le programme séquentiel résolvant le problème (dans le répertoire `tsp/Version0/`). Le programme prend deux arguments, qui sont le nombre réel de villes à parcourir et un élément de départ (*seed*) qui servira pour initialiser les générateurs de nombres aléatoires utilisés dans les premières fonctions.

Étudiez son implémentation (basée sur une fonction récursive). Lancez le programme pour vérifier qu'il fonctionne (avec 14 villes et une `seed` 1234, on trouve un chemin minimal 296). Vous pouvez décommenter l'appel à `printPath` pour observer la solution, mais pour les mesures de performances on ne gardera pas l'affichage. Notez que nous considérons que la ville de départ est celle de numéro 0 (i.e. `path[0] = 0`).

Remarquer l'optimisation au début de la fonction `tsp` : si le chemin partiel qu'on a déjà parcouru est déjà plus long que le chemin complet le plus petit qu'on a déjà trouvé jusqu'ici, on arrête le parcours de ce chemin partiel, puisqu'il donnera forcément des chemins trop longs ; cela économise beaucoup de calculs. Le tableau `cuts` sert à observer combien de fois on effectue cette optimisation, et pour quelles longueurs de chemins partiels.

4.2 Expériences à réaliser

Placez une directive OpenMP de parallélisation devant la boucle `for` dans la fonction `tsp`.

On pourra faire varier le nombre de threads utilisés avec la variable d'environnement `OMP_NUM_THREADS` (pensez à essayer avec un seul thread pour comparer à la version purement séquentielle) ainsi que la valeur NUM. Penser aussi à enlever l'option `-fopenmp` du Makefile, comparez le temps obtenu à celui de la version séquentielle. Pensez éventuellement à ajuster votre code pour optimiser ce cas (en utilisant des `#ifdef _OPENMP` au besoin).

4.3 En créant de nombreux threads

Nous allons créer des threads récursivement et pour cela il faut positionner la variable d'environnement `OMP_NESTED` à `true` , car par défaut le support d'exécution d'OpenMP empêche la création récursive de threads.

Partez de la version 1 fournie avec ce TP (dupliquez son répertoire car on nous réutliserons bientôt ce code) qui est la séquentielle originelle. Après avoir correctement protégé la variable conservant le minimum, il "suffit" ici d'ajouter un

`#pragma omp parallel for if (hops < NUM)`

juste avant la boucle

`for (i=0; i < NrTowns; i++)`

de la fonction `tsp`.

Faites attention cependant à bien préciser quelles variables doivent être privées : la règle est de favoriser les variables privées. Notez qu'on a changé le type de `Path_t` pour bien expliciter le fait que la variable `path` n'est ici qu'un pointeur, et non plus un tableau. Aussi, il faudra recopier `path` dans un nouveau tableau (pas obligatoirement alloué dynamiquement), sinon les threads vont se partager le même tableau pointé ! (NB. dans la version pthread, on avait implémenté la copie dans `tsp-job.c`).

Observez les performances. Oui, ce n'est pas terrible du tout. C'est parce que l'implémentation d'OpenMP que l'on utilise (celle de gcc) n'optimise pas encore bien le cas où le `if` désactive le parallélisme, or ce surcoût est payé à chaque parcours de ville ! Pour obtenir des performances, il faut donc utiliser plutôt une structure de la forme :

```
if (hops < NUM) {
#pragma omp parallel for
    for (i=0; i < NrTowns; i++) {
        ...
    }
} else {
    for (i=0; i < NrTowns; i++) {
        ...
    }
}
```

On pourra limiter le nobre de threads créer au nombre de villes restant à visiter, on utilisera alors une distribution dynamique.

4.4 Tracé de courbes avec gnuplot

Pour tracer des courbes, vous allez utiliser `gnuplot` : remplissez un fichier `data` contenant sur chaque ligne la valeur x et la valeur y, par exemple

```
1 15445
2 7829
3 6431
```

Pour automatiser les différentes expériences, simplifiez la sortie du programme pour pouvoir l'exploiter directement à partir de la commande shell :

`(for i in $(seq 1 10); do ./tsp 14 1234 $i; done) | tee data`

Notez également qu'à l'aide de la commande unix `join`, on peut fusionner deux fichiers de données, ce qui permet ensuite de faire faire des calculs par `gnuplot`.

Lancez `gnuplot`, et tapez `plot "data" with linespoints` . Pour que gnuplot calcule le speedup lui-même, on peut utiliser `plot "data" using ($1):(15300/($2)) with linespoints` (où 15300 est le temps séquentiel, à remplacer par le vôtre !).

On peut tracer plusieurs courbes à la fois en les séparant par des virgules : `plot "data" with lines, "data2" with linespoints`

On peut faire écrire la sortie dans un fichier en utilisant la commande `set terminal png` puis `set output "monfichier.png"`

4.5 En utilisant les tâches d'OpenMP

À partir de la version séquentielle fournie, parallélisez-la à l'aide de tâches.

Penser aussi à protéger l'accès à `minimum`, à éviter de créer des tâches à tous les niveaux, Pourquoi faut-il allouer dynamiquement la mémoire du chemin emprunté ? Comment limiter allocations et copies à tous niveaux ? Quand désallouer ?.

Est-ce intéressant d'utiliser les tâches OpenMP plutôt que les faire soi-même ?

On pourra mesurer l'influence des clauses `mergeable` (limitant le coût de génération d'une tâche) et `final(condition)` pour créer une tâche parallèle qui sera elle exécutée séquentiellement.