TP 2 : Parallélisme de tâches avec *threads*

quicksort multi-thread

Systèmes et supports d'exécution — 2008–2009

Alexandre DENIS

`denis@labri.fr`

`http ://dept-info.labri.fr/˜denis/`

L'objectif de ce TP est d'étudier la parallélisation par *threads* d'un algorithme « simple » avec différentes méthodes et différentes bibliothèques de *threads*.

1. L'algorithme *quicksort*

L'algorithme *quicksort* est un algorithme de tri rapide. Nous proposons de le paralléliser à l'aide de *threads*.

– écrire un algorithme *quicksort* standard (récursif) qui travaille sur un tableau d'entiers généré aléatoirement.
– vérifier que votre algorithme fonctionne correctement avant de passer à la suite !
– en faire un benchmark :
  – faire une boucle pour effectuer le *quicksort* sur des tableaux de taille croissante (en progression géométrique).
  – mesurer les temps avec le `timing.h`.
  – tracer la courbe de temps en fonction de la taille à l'aide de `gnuplot`.

2. Parallélisation de *quicksort*

– remplacer chaque appel récursif par une création de *thread*. Que faut-il protéger par un *mutex* ? N'oubliez pas les `pthread_join` adéquats ! Vérifiez que le résultat du tri est toujours correct.
– tester sur les machines multi-processeur du cremi : votre machine locale (en salle 201 : 4 cores), `aragog` (8 cores), `hagrid` (8 cores) ou `graup` (24 cores). Mesurer les performances et tracer les courbes. Conclusion ?

3. Optimisation

– que pouvez-vous dire de l'efficacité de la parallélisation précédente ?
– jusqu'à quelle taille de données la parallélisation fonctionne-t-elle ? Pourquoi ? Que se passe-t-il alors ?
– nous allons borner le nombre de *threads* par deux méthodes et les comparer :
  1. modifier votre programme pour décider de créer un *thread* ou non selon le nombre de *threads* actuellement en cours d'exécution, *i.e.* ne pas créer de *thread* au-delà d'une certaine borne m sur le nombre de *threads* actifs. Quelles valeurs de m vous semblent pertinentes ?
  2. modifiez votre programme pour ne créer un *thread* pour l'appel récursif que pour les premiers niveaux de récursion (jusqu'à un niveau n réglable). Combien de *threads* sont actifs simultanément en fonction de n ? Quelles valeurs de n vous semblent pertinentes ?
– évaluer et comparer le comportement de ces deux versions optimisées. Laquelle équilibre le mieux la charge ? Laquelle est la plus efficace ? Pourquoi ?
– pour obtenir un bon parallélisme, vous semble-t-il nécessaire de créer un *thread* pour chacun des deux appels récursifs du *quicksort* ? Justifiez. Adaptez le code en conséquence.

4. Utilisation de *Marcel*

Marcel est la bibliothèque de *threads* de l'environnement de programmation parallèle PM2. Reportez-vous à la page web du cours pour les instructions sur la compilation et l'utilisation de *Marcel*.

En pratique, dans votre code remplacer les appels `pthread_*` par leur équivalent `marcel_*`. Par exemple, utiliser `marcel_create` à la place de `pthread_create`. Ajouter également un appel `marcel_init(&argc, argv);` en début de programme. Compiler avec le `Makefile` fourni sur la page web du cours SSECPD.

– essayer chacune des trois versions — sans borne sur le nombre de *threads*, et avec chacune des deux optimisations — avec *Marcel* à la place de `pthread`. Que constatez-vous ?