TP 3 : Parallélisme de tâches à grain fin

Ordonnancement de tâches

Systèmes et supports d'exécution — 2008–2009

Alexandre DENIS, Olivier AUMAGE

denis@labri.fr

olivier.aumage@inria.fr

http://dept-info.labri.fr/~denis/

http://runtime.bordeaux.inria.fr/oaumage/

L'objectif de ce TP est d'étudier l'ordonnancement de tâches *à la main* dans le cadre d'un parallélisme de tâches à grain très fin quand une solution basée sur des *threads* a un surcoût trop important.

1. L'algorithme *quicksort multi-thread*

Vous avez réalisé lors du TP 2 une version parallèle de *quicksort* basée sur des *threads*. À partir des mesures réalisées sur la version séquentielle récursive et sur une version *multi-thread* optimisée :

– quel est le surcoût de la parallélisation par *threads* ?
– quel est le gain obtenu sur machine SMP ?
– conclusions ?

Note : sur les versions de Linux installées au CREMI, les threads `pthread` *sont des threads noyau. L'ordonnanceur Marcel est un ordonnanceur hybride, avec un LWP par CPU physique.*

Note 2 : vous pouvez voir si une machine est SMP en regardant le contenu de `/proc/cpuinfo`*. Attention ! L'*HyperThreading *apparaît aussi comme un deuxième processeur ; il peut être détecté en regardant si l'attribut* `ht` *apparaît dans la liste* `flags`*. Les machines* `hagrid`*,* `infini1-4`*,* `vivaldi` *et* `mcgonnagal` *sont des SMP.*

2. Gestion des tâches « *à la main* »

Nous allons dorénavant gérer le parallélisme à grain fin **manuellement**, en n'utilisant les *threads* que pour enchainer des tâches de manière itérative sur chacun des processeurs.

Mise au point

L'idée générale est de mettre en place une *pile de travail*, consitutée d'un empilement de tâches. Il s'agit de la méthode habituelle de transformation d'un code récursif en code itératif.

Dans cette pile de travail, chaque tâche est une description d'un travail à faire, travail qui serait effectué par un appel récursif à la fonction `quicksort` dans la version récursive. Plutôt que de réaliser un appel récursif directement (version séquentielle récursive) ou de créer un *thread* (version récursive *multi-thread*), un appel récursif devient un empilement d'une tâche sur la pile de travail.

Un *thread* de travail (le *thread* principal, dans un premier temps) est constitué d'une boucle qui dépile une tâche, la traite, et continue jusqu'à ce que la file soit vide.

Pour l'instant, un seul *thread* exécute le travail, mais nous ne perdons pas de vue que l'objectif est qu'à terme il y en ait plusieurs.

– définir une structure de données `struct job_s` décrivant *une* tâche (les paramètres d'un appel récursif) à effectuer ainsi que les fonctions nécessaires pour manipuler une pile d'élements `struct job_s`.[1]
– adapter votre *quicksort* pour remplacer les appels récursifs par un empilement de tâche.
– comment gère-t-on la *dépendance* entre les tâches ?
– comment initialise-t-on la pile ? Comment détecte-t-on la fin du *quicksort* ?

Optimisation de la récursion terminale

Avant de passer à une version multiprocesseur, nous allons appliquer quelques optimisations de base :

– chaque tâche génère la plupart du temps deux nouvelles tâches ; or, la première d'entre elles sera dépilée immédiatement. Est-il utile de l'empiler alors que l'on sait qu'elle sera immédiatement dépilée ? Adapter le code en conséquence.
– mesurer le temps d'exécution et évaluer le gain. Comparer avec la version sans optimisation, les versions récursives *multi-thread*, et la version séquentielle. Conclusions ?

3. Passage au *multi-thread*

Il s'agit maintenant d'ajouter d'autres *threads* noyau dans le programme de façon à observer (on l'espère !) une accélération sur une machine multi-processeur.

Pile de travail commune

– ajouter un *thread* noyau (`pthread`) qui consomme des tâches sur la pile de travail.

[1] Normalement, vous avez déjà écrit une telle structure pour le passage de paramètres aux *threads* lors du TP 2 ; il s'agit seulement de la compléter.

- comment gère-t-on les dépendances ? La tâche prête à exécuter est-elle forcém-ment celle en sommet de pile ?
- mesurer et comparer les performances obtenues sur une machine mono-processeur et en SMP. Essayez au moins avec 1, 2, et 4 *threads*. Conclusions ?

Une pile de travail par *thread*

La version précédente souffre d'un grave problème de contention pour l'accès à la pile de travail unique. Ce problème sera d'autant plus grave qu'il y aura plus de *threads* noyau. Une solution consiste en l'utilisation de piles *locales*.

- modifiez la version précédente pour que chaque *thread* soit doté de sa propre pile de travail locale.
- ajouter le *vol de travail* : quand la pile locale d'un *thread* est vide, il va vérifier chez les autres s'il y a encore du travail et en « *vole* » le cas échéant.
- évaluer le gain de performance apporté par cette méthode.