TP 3 : *Threads* et parallélisme à grain fin

Alexandre DENIS
denis@labri.fr

Systèmes et supports d'exécution — 2004–2005

L'objectif de ce TP est d'expérimenter et de comparer diverses implémentations d'un programme reposant sur un parallélisme à grain très fin sur une machine multiprocesseur afin de comprendre quels sont les différents facteurs qui influencent le temps d'exécution du programme.

Nous nous interésserons pour cette étude au calcul « jouet » du nième terme de la suite de Fibonacci.

1 La version séquentielle

Récupérez le contenu du répertoire `~aldenis/Enseignement/2004-SE/TP3/` dans votre répertoire de travail. Tapez `make fib_seq` et lancez le programme de cette façon :
`./fib_seq 16`

Mesurez et notez le temps d'exécution obtenu, à l'aide de la commande Unix `time` par exemple, pour quelques valeurs d'entrée raisonnables.

2 Version multithreadée

2.1 Au niveau utilisateur

Le fichier `fib_user_thr.c` contient un programme utilisant un *thread* utilisateur pour exécuter chaque appel de la fonction `fib`. Examiner le code pour comprendre le principe d'enchainement des tâches.

Pour compiler ce programme, vérifiez que vous avez bien positionné les variables d'environnement pour PM2. Ajoutez dans votre `.bashrc` les lignes :

```
export PM2_ROOT=/net/cremi/aldenis/soft/pm2
export PM2_CONF_DIR=/net/cremi/aldenis/.pm2
export PM2_FLAVOR=marcel1
export PATH=${PM2_ROOT}/bin:${PATH}
```

Tapez `make fib_user_thr` et comparez le temps d'exécution à la version séquentielle de référence. Hum... Alors ?

Vérifiez quand même sur une machine multiprocesseur (comme par exemple `magritte`) pour être sûr...

2.2 Au niveau noyau

Le fichier `fib_kernel_thr.c` contient un programme construit suivant le même principe, mais en utilisant cette fois des *threads* de niveau noyau.

Compilez ce programme (`make fib_kernel_thr`) et exécutez-le sur une machine monoprocesseur et sur une machine multiprocesseur. Comment le temps d'exécution se positionne-t-il par rapport aux versions précédentes ? Qu'en déduisez-vous ?

3 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.

3.1 Mise au point

Nous commençons par définir une *file de travail*. Définir une structure de données `job_t` décrivant *une* tâche (l'action et l'état) à effectuer ainsi que les fonctions nécessaires pour manipuler une pile d'élements `job_t`. Définir ensuite les fonctions de calcul réalisant le traitement d'une tâche.

Indication : il existe trois types de tâches dans Fibonacci

- le test (`n < 2`);
- la somme ;
- la terminaison (`return n`)

Le travail d'un *thread* consiste à dépiler une tâche, l'exécuter (appeler la fonction adéquate), et continuer jusqu'à ce que la file soit vide.

Faire toute la mise au point du programme avec un seul *thread* pour commencer ! Pensez à instrumenter votre programme pour savoir qui fait quoi et dans quel ordre.

3.2 Optimisations

Avant de passer à une version multiprocesseur, il est bon de commencer à optimiser le programme. Par exemple, le test génère la plupart du temps deux nouvelles tâches, et l'une d'entre elles sera dépilée immédiatement. Est-il utile de l'empiler alors que l'on sait qu'elle sera immédiatement dépilée ?

Mesurer le temps d'exécution, et comparer avec les trois premières versions de ce programme.

3.3 Ajout d'un second *thread* noyau

Il s'agit maintenant d'ajouter un second *thread* noyau dans le programme de façon à observer (on l'espère) une accélération sur une machine biprocesseur. Il faut essentiellement :

- ajouter un *thread* (`pthread_create`) doté de sa propre file de travail ;
- modifier le code de la fonction du *thread* de façon à aller *voler* du travail dans la file de l'autre *thread* lorsque l'on n'a plus rien dans sa propre file — attention à protéger correctement les accès concurrents aux files de travail !
- comparer le cas où chaque *thread* est doté de sa file de travail et le cas où les *threads* vont piocher dans une file de travail commune.

Vérifiez que votre programme fonctionne toujours, que le calcul est toujours exact, et surtout qu'il fonctionne plus vite qu'avant !