INF431

Processus légers et boucle parallèle
Sujet proposé par Gilles Schaeffer, sur une idée de François Pottier
Version: 1878:2611M

Le but de la PC est de réfléchir à l'implantation d'une primitive « parallel for » : on s'intéresse à des boucles de la forme :

```
for (int i = 0; i < n; i++)
  task(i);
```

Parfois, les tâches à effectuer pour les différentes valeurs de `i` sont « indépendantes », et on peut les effectuer en parallèle plutôt que séquentiellement. On voudrait écrire une fois pour toutes une librairie qui facilite cela. Comme déjà mentionné en cours, ces idées sont en vogue ces dernières années et diverses propositions de librairies ont été faites (Intel Thread Building Blocks, Microsoft Task Parallel Library, Java 7 ForkJoin, etc.) [1, 2, 3] : leur objectif est de permettre au programmeur lambda d'utiliser les processeurs multicœurs de sa machine « à peu de frais »... à condition toutefois de maîtriser les principes généraux de la programmation concurrente.

Question 1 Que signifie précisément « indépendantes » ? ◇

Question 2 Peut-on exécuter les boucles suivantes en parallèle au sens envisagé plus haut ?
- addition de deux vecteurs
- produit scalaire de deux vecteurs
- multiplication d'une matrice et d'un vecteur
- produit de deux polynômes (polynômes stockes par tableau de coefficients)
- évaluation d'un polynôme par la formule $a_0 + x(a_1 + x(a_2 + (\cdots(a_{n-1} + xa_n)\cdots)$. ◇

On se place dans le modèle des threads Java, mais on écrit dans un premier temps du pseudo-code pour éviter les lourdeurs de syntaxe. C'est le pseudo-code séquentiel « habituel », plus :
- une expression `spawn { I }` qui lance un nouveau thread chargé d'exécuter l'instruction `I` ; du point de vue du thread qui évalue cette expression, elle termine immédiatement et renvoie l'identifiant du thread nouvellement créé ;
- une instruction `join id` qui a pour effet d'attendre la fin du thread `id`.
- les opérations de création, acquisition, restitution d'un verrou :
  `l = new lock(); l.lock(); l.unlock();`
  L'acquisition d'un verrou est une opération bloquante : le thread qui tente d'acquérir un verrou est bloqué tant que le verrou est détenu par un autre thread. Si deux threads tentent simultanément d'acquérir un verrou, un seul y parvient.

Exemple : Un processus qui fait exécuter deux tâches à deux autres processus en parallèle :

```
TwoTasksInParallel {
  id1 = spawn { firsttask }
  id2 = spawn { secondtask }
  join id1
  join id2
}
```

Question 3 Écrire le pseudo-code d'une procédure `ParallelFor(n,task)` qui fait exécuter n tâches `task(i)` par n threads (un par tâche). ◇

Question 4 On veut utiliser notre procédure `ParallelFor` pour paralléliser un calcul de somme :

```
int sum = 0;
for (int i = 0; i < n; i++)
  sum += computation(i);
```

(indication : les n tasks à effectuer sont elles indépendantes ? quel problème faut il gérer ?) ◇

Question 5 Quels problèmes voyez vous au fait d'utiliser un thread par tâche ? ◇

Question 6 On suppose qu'on dispose de k processeurs et que les tâches nécessitent toutes à peu près le même temps de traitement. Comment accélérer le traitement ? Écrire le pseudo-code. ◇

Question 7 Que risque-t-il de se passer lorsque les tâches nécessitent des durées très différentes les unes des autres ? Proposer une solution pour améliorer le parallélisme lorsqu'on connaît à l'avance les temps de traitement des différentes tâches, en se ramenant à la résolution d'un problème d'optimisation (On ne demande pas ici de pseudo-code.) Que pensez vous de la complexité du problème ? ◇

Question 8 On suppose maintenant qu'on ne connaît pas à l'avance les temps de traitement des tâches. Proposer une solution utilisant une file d'attente et en écrire le pseudo-code. ◇

Parallélisme récursif pour le tri fusion. Nous avons vu en cours une approche récursive pour paralléliser l'addition de deux vecteurs jusqu'à une certaine granularité choisie à l'avance : tant que le morceau de tableau à traiter est trop grand, on crée un thread pour chaque moitié. Toujours en cours, il a été mentionné que `Mergesort` rentre plus ou moins dans le même cadre mais qu'il faut paralléliser la partie merge sous peine de n'avoir qu'un parallélisme de l'ordre de $O(\log n)$.

Question 9 Proposer un algorithme pour paralléliser la fusion de deux sous-tableaux du tableau `T` d'indice respectifs $[\ell_1, h1[$ et $[\ell_2, h_2[$. On pourra supposer que la procédure `ParallelMerge` reçoit en même temps que `T` un tableau annexe `B` et les bornes $[\ell_3, h_3[$ de la zone dans laquelle le tableau fusionné doit être placé. ◇

Question 10 Analyser la profondeur de calcul et le travail effectué par votre algorithme. ◇

Question 11 Analyser le travail, la profondeur et le parallelisme de l'algorithme de tri fusion parallèle utilisant votre fusion parallèle à la place de la fusion séquentielle. ◇

Question 12 Supposons qu'on dispose de k processeurs. Comment faire la répartition des tâches ? ◇

MergeSort est une instance élémentaire du problème général consistant à paralléliser sur k processeurs des tâches récursives de la forme

```
Task(T){
  if size(T)<s then TraitementIteratif(T)
  else
     divide(T,T1,T2)
     Task(T1)
     Task(T2)
     merge(T1,T2)
}
```

C'est, entre autre, une possibilité que se propose d'offrir facilement la bibliothèque Join/Fork de Java. Dans ce cadre général, il n'y a aucune raison que les sous-tâches soient de taille équilibrée : l'un des principaux apports des bibliothèques de ce type est qu'elles s'occupent de gérer automatiquement et dynamiquement la répartition des sous-tâches entre processeurs.

Références

[1] Intel threading building blocks. `http://threadingbuildingblocks.org/`.

[2] Microsoft task parallel library. `http://msdn.microsoft.com/en-us/library/dd460717.aspx`.

[3] Java fork/join. `http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html`.