INF431

Processus légers et boucle parallèle
CORRIGÉ

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 » ? ◇

Solution. Pour $i \neq j$, il ne faut pas que `task(i)` écrive à un emplacement en mémoire où `task(j)` lit ou écrit.

Remarquons que si `task(i)` et `task(j)` *lisent* toutes deux un même emplacement, alors tout va bien : l'accès en lecture seule à des données partagées ne constitue pas une *race condition*. □

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 $a_0 + x(a_1 + x(a_2 + (\cdots(a_{n-1} + xa_n)\cdots)$. ◇

Solution. L'addition de deux vecteurs est l'exemple du cours, les opérations sur chaque entrée sont indépendantes. Pour le produit scalaire il faut accumuler les résultats : on peut cependant paralléliser à gros grains, en traitant en parallèle k blocs de coordonnées et en terminant par une phase séquentielle pour sommer les k sommes partielles obtenues (cf Question 4). Dans la multiplication d'une matrice par un vecteur on parallélise facilement les calculs de chaque ligne : il s'agit de produits scalaires indépendants (plusieurs threads doivent *lire* en parallèle le vecteur). Dans le produit de deux polynômes c'est la même chose : chaque coefficient peut être calculé indépendamment. L'évaluation d'un polynôme via la formule de Horner est par contre intrinsèquement séquentielle. □

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). ◇

Solution. Puisqu'on a n tâches à effectuer indépendamment, l'implémentation la plus simple consiste à lancer n threads indépendants, chacun chargé d'exécuter `task(i)` pour une valeur différente de `i`. On utilise une première boucle pour lancer ces n threads puis une seconde boucle pour attendre qu'ils aient tous terminé (à l'aide de `join`).

```
ParallelFor (n, task){
  for (i = 0 ; i < n ; i++) id[i] = spawn { task(i) }
  for (i = 0 ; i < n ; i++) join id[i]
}
```

□

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 ?) ◇

Solution. Cette situation est courante et « facile » parce que l'opération + est associative/commutative et a un élément neutre. Lorsqu'on aura lancé k threads, chacun va terminer (dans un ordre arbitraire) en apportant une somme partielle, et on n'aura qu'à additionner ces sommes partielles (dans l'ordre où elles nous parviennent) dans une somme globale protégée par un verrou.

Avec la procédure `ParallelFor` de notre choix :

```
DynamicParallelSum (n, task){
  S = 0
  LS = new lock ()
  ParallelFor (n,
               proc (i) {
                 s = computation(i);
                 LS.lock()
                 S += s
                 LS.unlock()
               }
              )
  return S
}
```

Noter que `s` est locale à chaque thread, donc pas de conflit dessus. Par ailleurs, on peut noter par rapport à l'utilisation des vérrous faites en cours l'absence du groupe `try...finalize` inutile ici puisque l'opération `S += s` ne peut pas déclancher d'exception. □

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

Solution. Lancer un thread coûte cher (en temps et en mémoire), et lancer plus de threads qu'il n'y a de processeurs hardware coûte cher aussi (à cause du context switching) donc la solution naïve a toutes les chances d'être très lente, surtout si le calcul effectué par `task(i)` est relativement peu coûteux.

De même l'utilisation de lock/unlock est raisonnable si chaque tache prend du temps. Sinon le coût de lock/unlock risque de dominer. Par exemple dans le cas où le verrou est pris juste pour incrémenter un compteur, on peut utiliser des instructions plus specifiques comme `compareAndSet` (cf. le cours). □

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. ◇

Solution. Une idée naturelle est de lancer seulement k threads, pour une valeur k fixée (par exemple égale au nombre de processeurs), et de leur attribuer à chacun un intervalle de (environ) n/k indices à traiter.

```
ParallelFor (n, task, k){
  for (i = 0 ; i < k ; i ++) do
    id[i] = spawn {
               for (j = i ; j < n ; j += k) task(k)
            }
  for (i = 0 ; i < k ; i ++) do join id[i]
}
```

□

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 ? ◇

Solution. Dans les deux versions ci-dessus, on peut parler de « scheduling statique » : on décide dès le début quel thread s'occupera de quels indices. Cette technique est simple mais a un défaut : si le coût (en termes de temps de calcul) de la tâche `task(i)` est très variable suivant la valeur de `i`, on court le risque d'attribuer toutes les valeurs « coûteuses » de `i` au même thread et les valeurs « faciles » de `i` aux autres threads, ce qui fait que (après un certain temps) on se retrouvera avec un seul thread qui travaille. Ce n'est pas efficace.

Si on connaît les temps de traitement, l'idée est d'optimiser la répartion des tâches entre les différents processeurs : si on dispose de k processeurs pour n tâches indépendantes de durée d_i ($i = 1, \ldots, n$), il s'agit de trouver une partition $(I_j)_{1 \leq j \leq k}$ des tâches qui minimise

$$\max(\sum_{i \in I_j} d_i \mid j = 1, \ldots, k)$$

Ce problème est difficile (plus précisément il est NP-complet). L'étude de solutions approchées ou d'heuristiques pratiques à ce type de question de *scheduling* est une branche à part entière de la recherche opérationnelle. □

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. ◇

Solution. Lorsqu'on ne connaît pas les temps de traitement, il vaut mieux faire du « scheduling dynamique » : on construit une file d'attente dans laquelle on met des indices (ou des intervalles d'indices) à traiter, et nos k threads viennent piocher du travail dans cette file. Il faut alors que la file soit protégée par un verrou. Si un thread constate que la file est vide, il meurt.

```
DynamicParallelFor (n, task){
  F = new File;
  L = new lock();

  for (i = 0 ; i < n ; i++) F.put(i)

  for (i = 0 ; i < k ; i++)
     id[i] = spawn {
          while (true) {
     L.lock()
     if F.isempty() then { L.unlock(); exit() }
     j = F.take()
     L.unlock()
     task(j)
          }
  for (i = 0 ; i < k ; i++) join id[i]
}
```

L'instruction `exit()` dans ce code provoque la sortie de la boucle `while` et la fin du processus courant.

Il est important ici de pas oublier le unlock avant le break, sinon le verrou reste définitivement bloqué et les autres processus en attente ne termineront jamais.

Par ailleurs, on remarque que les instructions `F.isempty()` et `F.take()` doivent être ensemble (on ne peut pas par exemple sortir le `F.isempty()`, même si ce n'est qu'une lecture) : sinon ces deux opérations pourraient s'entrelacer avec celles effectuées par un autre processus et la pile pourrait se vider entre les appels `F.isempty()` et `F.take()`. □

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. ◇

Solution. Une réponse détaillée est dans la 3ème édition du Cormen (2009) [4], voir aussi les slides [5]. Une discussion assez complète de l'efficacité pratique se trouve par exemple aux adresses [6] (parallel merge) et [7] (parallel mergesort).

Pour paralléliser merge il faut pouvoir couper en deux sous-tâches. On le fait par divide-and-conquer sur le plus long des deux tableaux triés : on le coupe en deux, puis on utilise le premier élément de la deuxième moitié pour couper le deuxième tableau, en cherchant la position par « binary search ». On peut alors paralléliser le traitement des deux paires de demi-tableaux.

```
ParallelMerge(tableau T, int l1, int h1, int l2, int h2,
              tableau B, int l3, int h3){
  if (h1-l1 < h2-l2) ParallelMerge (T,l2,h2, l1,h1, B,l3,h3)
  else if (h1-l1 == 1){
                 if (h2-l2 == 1){ B[l3]=min(T[l1],T[l2])
                                  B[l3+1]=max(T[l1],T[l2])
                 }else B[l3]=T[l1]
  }else {
    int m1=l1+(h1-l1)/2;
    int m2=BinarySearch(T[m1],T,l2,h2);
    B[l3+(m1-l1)+(m2-l2)]=T[m1]
    int id1 = spawn{ParallelMerge(T,l1,m1,l2,m2,B,l3,l3+(m1-l1)+(m2-l2))}
```

```
      int id2 = spawn{ParallelMerge(T,m1+1,h1,m2,h2,B,l3+(m1-l1)+(m2-l2)+1,h3)}
      join id1; join id2;
    }
  }
```

Observons que dans le pire cas la taille du plus grand des deux appels récursifs est $3/4$ de la taille initiale : comme on a coupé le plus grand tableau en 2 exactement, chaque morceau du grand tableau est de taille au plus $\frac{1}{2}(h_1 - \ell_1)$; les deux morceaux du petit tableau sont chacun de taille au plus égale à la taille du petit tableau, $h_2 - \ell_2$ (dans le pire cas, on peut être amené à tout mettre dans l'un des deux tableaux). Au total on obtient que chacune des deux nouvelles instances à traiter est de taille au plus $\frac{1}{2}(h_1 - \ell_1) + h_2 - \ell_2 \leq \frac{3}{4}(h_1 - \ell_1) + \frac{3}{4}(h_2 - \ell_2)$ (c'est ici qu'on utilise l'hypothèse qu'on coupe en 2 le plus grand tableau : $h_2 - \ell_2 \leq h_1 - \ell_1$). □

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

Solution. La profondeur de calcul est déterminée par le temps d'exécution de la recherche binaire, $O(\log n)$, et par la récurrence $P(n) \leq P(3n/4) + O(\log n)$, donc bornée par $O(\log^2 n)$ (après $logn$ itération de la récurrence). On peut montrer de plus que le travail effectué reste $O(n)$ en analysant la récurrence : $C(n) \leq max_\alpha(C(\alpha n) + C((1-\alpha)n)) + \Theta(\log n)$ (montrer par induction que pour des constantes a et b bien choisies, $C(n) \leq an - b\log n$, cf [5]). □

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. ◇

Solution. Maintenant on parallélise mergesort comme dans la version naïve vue en cours, mais en faisant les opérations de merge par des appels à `ParallelMerge`.

```
ParallelMergeSort(tableau T, int l, int h, tableau B){
   if (h-l > 1){
     int m=l+(h-l)/2;
     int id1 = spawn {ParallelMergeSort(T,l,m,B)}
     int id2 = spawn {ParallelMergeSort(T,m,h,B)}
     join id1; join id2;
     ParallelMerge(T,l,m,T,m,h,B,l,h)
     Copy(B,l,h, T)
   }
 }
```

Le travail effectué reste $O(n \log n)$ comme dans un mergesort séquentiel. La profondeur est maintenant $O(\log^3 n)$ ($\log n$ étages récursifs de coût $O(\log^2 n)$). D'où un parallélisme de $n/log^2 n$ au lieu du $\log n$ de la version naïve. □

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

Solution. Les tâches à effectuer sont de coût prévisible. Il est donc envisageable de fixer une profondeur maximale au delà de laquelle on arrête de les mettre en parallèle. □

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.

Solution. **Complément : Work stealing** Une adaptation élémentaire de la réponse à la question 8 serait d'avoir une file partagée dans laquelle les processeurs vont chercher les tâches et de remplacer les appels récursifs par l'insertion des tâches correspondantes dans la file.

Le problème de cette solution est que le verrou de la file va être constamment l'objet de dispute entre les processeurs. On peut penser faire deux verrous différents, pour l'ajout de tâche en début de file et pour l'extraction d'une tâche en fin de file, mais cela ne résout pas vraiment le problème : lorsque la file est vide ou contient un seul élément, le «début» et la «fin» de la file coïncident, pour détecter ce cas on aurait besoin des 2 verrous...

La solution proposée par exemple dans la bibliothèque Fork/Join de Java [8] est d'avoir pour chaque thread une pile/file (qui combine des opérations de pile et de file), avec une politique d'usage qui permet de limiter le risque de contention :

- Chaque processeur utilise sa pile/file comme une pile et traite son sous-problème récursif à la manière séquentielle habituelle ;
- Un processeur qui a vidé sa pile prend une tâche dans une pile/file d'un autre processeur au hasard, en la retirant *à la manière des files*. (On parle de "work stealing".)

L'intérêt de cette approche est heuristique : on espère que la plupart du temps les threads vont travailler chacun sur sa pile, et que les opérations de files (le work stealing) qui risquent d'induire des conflits seront plus rares. La forme de la récursivité et la politique choisie impliquent en effet que les vols de tâche portent sur les tâches les plus anciennes, qui devraient aussi être les plus grosses.

Enfin le fait d'utiliser une pile/file permet de limiter l'usage des verrous : l'approche de Fork/Join est la suivante [8, Section 3.1] :

- Un verrou associé à chaque pile/file garantit qu'un seul thread essaie de voler dedans.
- Lorsque le propriétaire veut prendre dans une pile/file un élément (à la tête de la pile/file), il commence par décrementer la position de la tête, puis il teste si la pile risque d'être devenue vide. Dans ce cas il demande le verrou de la pile/file avant d'effectuer son opération.
- Lorsqu'un voleur veut prendre dans une pile/file un élément (à la queue de la pile/file), il demande le verrou associé, puis il incrémente la position de la queue, et il teste si la pile risque d'être devenue vide. Dans ce cas il rétablit la valeur de la position de la queue, rend le verrou et renonce provisoirement à prendre dans cette pile/file. Sinon il prend l'élément voulu.

L'important est ici que seuls deux acteurs peuvent interagir à la fois, qu'ils agissent sur des variables différentes, et qu'ils sont tous deux sûrs de pouvoir détecter une interaction éventuellement risquée. □

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`.

[4] Cormen, Leiserson et Rivest. *Introduction to algorithms*. MIT press, 3rd edition édition, 2009. Section 27.3, en ligne sur `http://mitpress.mit.edu/books/introduction-algorithms`.

[5] C. E. Leiserson. Multithreaded programming in cilk. `http ://supertech.csail.mit.edu/cilk/lecture-2.pdf`.

[6] Parallel merge, . `http://www.drdobbs.com/parallel/229204454`.

[7] Parallel mergesort, . `http://www.drdobbs.com/parallel/229400239`.

[8] Doug Lea. A java fork/join framework. *In Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM. `gee.cs.oswego.edu/dl/papers/fj.pdf`.