SI4 Programmation Parallèle
Epreuve écrite
5 mai 2022, durée 2h30
ELEMENTS de CORRECTION

Exercice 3 (bareme approximatif : 8 points)

– Algorithme pour le calcul du nombre d'inversions

Soit A un tableau (vecteur) de n nombres entiers. On veut concevoir une procédure qui permet de déterminer le nombre d'inversions d'éléments adjacents de A, c.a.d. le nombre d'éléments (immédiatement) adjacents qui ne sont pas correctement ordonnés. Voici une spécification en pseudo-langage

```
procedure nbInversions (tab d'entiers A, int n) returns int nbI
#Precondition n >=1
#Postcondition
# nbI = SUM( 1<= i <n tel que A[i]>A[i+1])
```

Quelques exemples :

- nbInversions ([1,1,1], 3) → 0
- nbInversions ([1,2,2], 3) → 0
- nbInversions ([10,9,8], 3) → 2
- nbInversions ([1,2,3,4,3,2,1,0,1], 9) → 4
- nbInversions ([8,7,6,4,5,3,1,2,3], 9) → 5
- nbInversions ([9,8,7,6,5,1,2,4,3], 9) → 6
- nbInversions ([9,8,7,6,5,1,4,3], 8) → 6

Le but de l'exercice est de donner une version séquentielle, puis des versions parallèles et de comparer les complexités.

1. Décrire en pseudo-langage (impératif) une version itérative séquentielle. Donner la complexité en temps notée Tseq(n).
2. Donner le principe d'une version parallèle récursive, c'est-à-dire par approche « divide and conquer » (sans l'écrire), qui suppose pour simplifier que la taille du vecteur (et donc des sous-vecteurs) est une puissance de 2

Principe est de couper la liste / vecteur en deux, puis en deux, etc, jusqu'à n'avoir plus que des sous vecteurs de longueur 2. Chaque sous vecteur remonte donc un 0 ou un 1. Puis, au niveau de la remontée récursive de ces 2 appels lancés en parallèle, celui qui a lancé ces 2 appels récupère 2 valeurs, va les sommer, mais, en plus, il va comparer le dernier élément du sous vecteur de gauche avec le premier élément du sous vecteur de droite, et si le test montre que il y a une inversion, ça ajoutera encore +1 au total , qui sera donc lui-même remonté au niveau récursif père.

On voit que ça crée une sorte d'arbre binaire des appels récursifs, et lors de la remontée, cet arbre se contracte. A la fin, à la racine, on a le total que l'on cherche.

Exemple sur nbInversions ([9,8,7,6,5,1,4,3], 8) → 6

1er niveau d'appels : appel avec [9,8,7,6] et appel avec [5,1,4,3]

2eme niveau d'appels : appel avec [9,8], et [7,6] et aussi en parallèle [5,1] et [4,3]

On est arrivé au niveau le + bas de la descente récursive, donc, ça donne 1, et 1 pour respectivement [9,8] et [7,6], et également 1 pour [5,1] et 1 pour [4,3].

On commence la remontée. A gauche le père reçoit donc +1 et +1 = 2, et il va donc maintenant comparer dans [9,8,7,6] les deux éléments du milieu, cad, le + à droite du sous vecteur gauche avec l'élément le + à gauche du sous vecteur droit. Donc, il va comparer 8,7, et ça donne encore +1. Donc, au total 3.

De l'autre côté, l'autre processeur va récupérer et faire le cumul égal 2, et va comparer 1 et 4, ce qui n'ajoutera rien.

Puis, on remonte encore d'un niveau : De la gauche, on a 3 ; et de la droite, on a 2. Il reste à ce processeur à comparer l'élement le + à droite du sous vecteur gauche, et l'élement le + à gauche du sous vecteur droit, donc, 6 avec 5. Il faut donc ajouter +1. Ça donne donc 3+2+1=6, qui est le total correct.

3. Donner le principe d'une autre version parallèle, cette fois plus appropriée pour une PRAM, toujours en supposant que la taille du vecteur est une puissance de 2 pour simplifier, en expliquant bien la démarche sous-jacente suivie et en précisant le modèle de PRAM utilisée, sachant que l'entier final *nbI* est en mémoire partagée, accessible en Read et en Write par tous les processeurs de la PRAM.
Bonus +2 points: écrire l'algorithme précis, avec tous les indices bien positionnés
Donner la complexité en temps Tpar(n) et le nombre de processeurs PRAM nécessaires.
Puis son coût « en travail ». Est-il efficace, et si non, dire s'il peut être rendu optimal en travail, et comment.
Le principe va être le même que celui qu'on met en œuvre dans un calcul de réduction avec une quelconque opération binaire (max, somme, etc) écrit en PRAM. Cependant, hormis sommer des valeurs accumulées, intermédiaires, il faut ajouter lors de chaque opération de réduction le test engendrant le +1 si une inversion est trouvée entre les deux valeurs en mémoire correspondant aux deux extrémités des 2 sous-tableaux concernés à chaque niveau de l'arbre, mais il n'y a rien de plus à faire.
Donc, on a les mêmes complexités qu'on avait pu noter pour réduction : O(log N) étapes, séquentielles, et dans chaque étape, N/2 maximum opérations de sommage qui sont parallèles. Plus on remonte dans l'arbre, moins on a de processeurs impliqués. Le nombre de processeurs maximum à impliquer est donc N/2. Au final, ça donne donc O(N) processeurs. Le travail de la version parallèle devient donc O(N)*O(log N). Ce qui n'est pas optimal comparé à O(N) comme travail séquentiel. L'efficacité est donc <1, puisqu'elle est de 1/O(N)=O(1/N).
Pour rendre l'algo optimal en travail. Il faudrait donc utiliser non pas O(N) processeurs, mais O(N/logN). Chacun va être en charge non pas de comparer une paire de valeurs mais O(logN) valeurs en séquentiel. Puis, N/logN/2 processeurs enclencheront l'algo de réduction sur l'arbre de taille réduite n'ayant que N/logN feuilles. Au final, la complexité sera majorée par les étapes faites en séquentiel, donc en temps O(log N), mais, le nombre de processeurs total étant de O(N/logN), le travail est en O(N) et donc, optimal en travail, et d'efficacité 1.
4. Une variante autour de ce problème (4 points). Au lieu de déterminer nbI (le nombre d'inversions) on veut déterminer pour chaque élément a de A quel est le nombre des éléments non correctement ordonnés qui précèdent l'élément a dans le tableau A
Quelques exemples :

- nbInversionsTab ([1,1,1], 3) → (0,0,0)
- nbInversions Tab([1,2,2], 3) → (0, 0, 0)
- nbInversions Tab([10,9,8], 3) → (0,1,2)
- nbInversions Tab([1,2,3,4,3,2,1,0,1], 9) → (0,0,0,0,1,2,3,4,4)
- nbInversions Tab([8,7,6,4,5,3,1,2,3], 9) → (0,1,2,3,3,4,5,5,5)
- nbInversions Tab([9,8,7,6,5,1,4,3], 8) → (0,1,2,3,4,5,5,6)
- nbInversions Tab([9,8,7,6,5,1,2,4,3], 9) → (0,1,2,3,4,5,5,5,6) : prenons le cas du dernier élément, 3 ; de son point de vue, il y a bien 6 éléments (ceux soulignés) qui le précèdent qui ne sont pas correctement ordonnés par ordre croissant
- nbInversions Tab([5,2,3,6,7,8,9,7,3,2,4,3,2,3,2,1]) → (0,1,1,1,1,1,1,2,3,4,4,5,6,6,7,8) pour un vecteur A de dimension 16.

a) Sans encore écrire ni une version séquentielle, ni une ou deux versions parallèles pour cette nouvelle procédure nbInversionsTab, expliquer en quoi elle diffère du problème initial.
b) Concernant une version PRAM, expliquer qu'on peut reposer sur le traditionnel Prefix mais avec une opération plus élaborée que juste une seule somme de valeurs entre voisins « dans l'arbre binaire ». Pour illustrer cette nécessité de sophistiquer les opérations réalisées, utiliser par exemple les deux

derniers exemples fournis ayant 8 valeurs dans A ou en ayant 16. Il y a besoin de modifier la phase de montée et celle de descente.
c) Décrire l'opération à réaliser dans chacune des 3 phases de l'algorithme préfix, en indiquant avec quelles valeurs opérer. Hint : vous commencez avec des 0 à chaque feuille de votre arbre binaire, qui représente le cumul des valeurs non correctement ordonnées. En plus du cumul qui va être accumulé, en remontant dans l'arbre, vous devez conserver de manière bien distincte à chaque nœud de l'arbre, un 0 ou un 1 (selon résultat du test) qu'il faudra additionner à ce qui monte ou à ce qui descendra. En gardant cette sorte de « retenue » dans une variable bien distincte, vous serez alors à même de réaliser correctement la phase de descente de la méthode Prefix.
Je traite b) et c) en même temps dans cette correction.

En fait, comme constaté en direct !, on pourrait utiliser le traditionnel Prefix en partant d'un tableau de même taille où l'on met un 0 ou un 1, selon que le précédent élément de chaque élément est supérieur. Donc, sur l'exemple à 8 valeurs, ([9,8,7,6,5,1,4,3]) , si on pourrait partir de (0,1,1,1,1,1,0,1) et ensuite on applique sur cette collection de valeurs l'algo de Prefix avec comme opération la somme de 2 entiers, ce qui fait qu'à la fin, on obtient bien (0,1,2,3,4,5,5,6)

Mais, comme expliqué en séance, on doit voir le problème d'une manière différente. On part avec un tableau de valeurs qui représente des cumuls initiaux, contenant initialement que des 0, c'est à-dire un cumul par processeur égal à 0.
Et, on va adapter l'algorithme Prefix, pour que dans sa phase de montée et dans sa phase de descente on puisse accumuler, stocker, et ensuite faire redescendre les bonnes valeurs.

On décrit la méthode en se basant sur l'exemple à 8 valeurs [9,8,7,6,5,1,4,3] partant de (0,0,0,0,0,0,0,0), libre à vous si vous voulez de vérifier qu'elle fonctionne bien sur l'exemple à 16 valeurs :
Phase de montée.
Etape parallèle 1 : cela concerne des comparaisons deux à deux, on part « du bas », on part donc de feuilles d'un arbre binaire, comme si c'était l'arbre obtenu par décomposition récursise/divide&conquer de la question 2. Pour bien comprendre, il faut bien que chaque nœud de l'arbre sache s'il est un fils droit ou un fils gauche. Ce sont les fils droits qui sont « actifs » dans la phase de montée
8 se compare avec 9 => 1 ; on conserve ce 1 dans une variable « retenue » stockée au niveau du nœud père des deux feuilles qui sont responsables des valeurs respectives 9 et 8.
6 se compare avec 7 => 1, on conserve ce 1 dans une variable « retenue »
1 se compare avec 5=> idem
3 se compare avec 4 => idem
Donc, on a bâti le premier niveau de l'arbre au dessus des feuilles, et dans les nœuds de ce premier niveau, on a stocké des retenues (dans l'exemple, elles sont par hasard toutes à 1)
Etape 2 : on recommence, en montant d'un niveau dans l'arbre ; les fils droits vont sommer leur valeur, avec celle de leur frère gauche dans l'arbre, et en plus, ils vont comparer (comme dans la question 2), la valeur la + à droite du sous vecteur détenu par leur frère gauche, avec la valeur la + à gauche du sous-tableau qu'ils détiennent. On est en PRAM, donc tous peuvent lire n'importe quel élément du vecteur
Cela donne donc :
Concernant le sous vecteur ([9,8], [7,6]), le noeud droit en charge de [7,6] va comparer 7 avec la valeur 8 détenue par le nœud gauche en charge de [9,8]. Ce qui ajoute +1 au cumul des 2 valeurs qui remontent vers le nœud père, donc 3. Ce +1 est mis dans la « retenue » de ce nœud père.
En parallèle, dans l'autre sous-arbre, 4 est comparé à 1, ce qui va ajouter 0 aux valeurs des deux fils qui sommées égal à 1+1=2. La « retenue » de ce nœud ci vaut donc 0.
Etape 3 : Même principe : le nœud (qui se trouve être la racine) cumule 3+2=5 qui remonte de ses 2 fils, et après comparaison de 5 avec 6, ajoute +1 qui est sa « retenue ». Donc, le total est de 5+1=6 à la racine.

Maintenant la **phase de descente**. Le même modèle que dans la descente d'un Prefix : les données d'un frère gauche sont passées (ou lues par) au frère droit. Difficulté par rapport à l'algo classique du préfix est de modifier quelles données doivent s'accumuler et descendre.

Un nœud qui est à droite doit sommer la valeur cumul qu'il reçoit de son frère gauche et la « retenue » qui est détenue sur son père, ainsi que la valeur que son père lui fait descendre. Ce nouveau total va être propagé à chacun de ses fils
Un nœud qui est à gauche doit simplement propager à ses deux fils ce qui descend de son père.

On démarre avec le nœud au niveau de la racine qui n'a pas de frère, donc il est un peu particulier en ce sens qu'il fait descendre un 0 à ses 2 fils. Sur l'exemple, procédons par sous arbre et non par étapes comme durant la phase de montée :
Le nœud racine, qui pour rappel a une retenue de 1 ; fait descendre 0.
Le fils droit de la racine : récupère de son frère gauche 3, (c'est normal, c'est le cumul des inversions du plus grand sous-arbre à gauche), y ajoute la retenue détenue sur son père (la racine), qui montre bien qu'il y a un problème d'inversion à la frontière de ces deux plus grands sous-arbres (entre 6 et 5) et y ajoute le 0 venu de la racine. Donc, 4. Cette valeur 4, il va la propager à ses deux fils. Du côté de son fils droit : celui-ci a reçu la valeur de son frère, donc 1, à quoi il ajoute le 4, et la retenue (qui vaut 0 sur le père)=5. Cette valeur de 5, il la fait descendre plus bas : son fils gauche étant une feuille, il n'a qu'à prendre ce 5, et c'est la valeur finale. Son fils droit, reçoit de son frère feuille, 0, qu'il ajoute au 5 qui est descendu, et à quoi il ajoute la retenue stockée sur le nœud père, qui vaut 1 (conséquence du test entre 4 et 3 fait lors de la phase de montée). Cela nous donne 6, qui est la valeur finale pour le dernier élément.
On décrit maintenant la descente de ce 4 vers le côté gauche : ce 4 arrive sur le fils gauche, il va la propager à ses deux fils. Son fils de gauche considère que c'est la valeur finale ; son fils de droite ajoute à ce 4 ce qu'il récupère de son frère, qui est 0, et la retenue sur leur père qui est 1, ce qui lui donne 5, sa valeur finale. On a donc traité la moitié droite de ([9,8,7,6,5,1,4,3]) pour l'instant, obtenant ces valeurs de nbInversionsTab(x,x,x,x,4,5,5,6).
Passons à la moitié gauche, le plus grand sous-arbre à gauche. Le 0 venu de la racine arrive sur le nœud gauche ; il va propager aux deux fils. Celui de droite va récupérer la valeur 1 (le cumul de son frère), y ajouter le 0 descendu du père, et la retenue au niveau du père qui est de 1 (qui, rappelons-le, est la conséquence du test entre 8 et 7 lors de la montée). Ça donne donc un total de 2, qui va être propagé aux deux fils. Celui de gauche en fait sa valeur finale, celui de droite y ajoute le 0 (cumul initial) venu de son frère, et la retenue de son père (résultat du test entre 7 et 6) qui vaut 1, soit un total de 3.
Il reste les 2 cas des feuilles les plus à gauche. Le 0 de la racine arrive à leur nœud père. Ce 0 est à nouveau propagé aux deux fils. Celui le plus à gauche, (détenant le 9 du vecteur) obtient ainsi le 0 final. Son frère droit reçoit donc 0 de son frère gauche, le 0 qui descend, et ajoute la retenue du père de 1, soit un total de 1 qui est la valeur finale.
nbInversionsTab(x,x,x,x,4,5,5,6).est donc devenue nbInversionsTab(0,1,2,3,4,5,5,6).