Parallélisation de tri

Exercice 1. Parallélisation de tri bitonique

Il existe de nombreux algorithmes de tri. Traditionnellement, ils ont été distingués par leur complexité, c'est-à-dire, étant donné un tableau de **n** éléments, combien d'opérations faut-il pour les trier, en fonction de **n**.

Théoriquement, on peut montrer qu'un algorithme de tri doit avoir au moins une complexité en $Olog\ n$. En effet, il existe plusieurs algorithmes qui garantissent d'atteindre cette complexité, mais un algorithme très populaire, appelé tri-rapide, a une complexité au meilleur des cas de $Olog\ n$, et de $O(n\^2)$ dans le pire des cas. Ce comportement résulte du fait que le tri-rapide doit choisir des '**éléments pivots**', et si ces choix sont systématiquement les pires possibles, la complexité optimale n'est pas atteinte.

- **Lorsque le tableau d'entrée a une longueur > 1,**
    - **Trouver un élément pivot de taille intermédiaire,**
    - **Diviser le tableau en deux en fonction du pivot,**
    - **Trier les deux sous-tableaux.**

Algorithme 1 : L'algorithme de tri rapide (quicksort).

D'autre part, l'algorithme de tri à bulles, très simple, a toujours la même complexité, car il a une structure statique :

- **Pour chaque passe de 1 à n - 1,**
    - **Pour chaque élément e de 1 à n - passe,**
        - **Si les éléments e et e + 1 sont mal ordonnés alors**
            - **les permuter**

Algorithme 2 : L'algorithme de tri à bulles.

Certains algorithmes de tri fonctionnent indépendamment des données d'entrées réelles, tandis que d'autres prennent des décisions basées sur ces données. La première catégorie est parfois appelée réseau de tri. On peut le considérer comme un dispositif personnalisé qui implémente un seul algorithme ; l'élément matériel de base est l'élément de comparaison et d'échange, qui a deux entrées et deux sorties. Pour deux entrées **x** et **y**, les sorties sont max(x,y) et min(x,y).

Dans la figure ci-dessous à gauche, nous montrons le tri à bulles, construit à partir d'éléments de comparaison et d'échange.

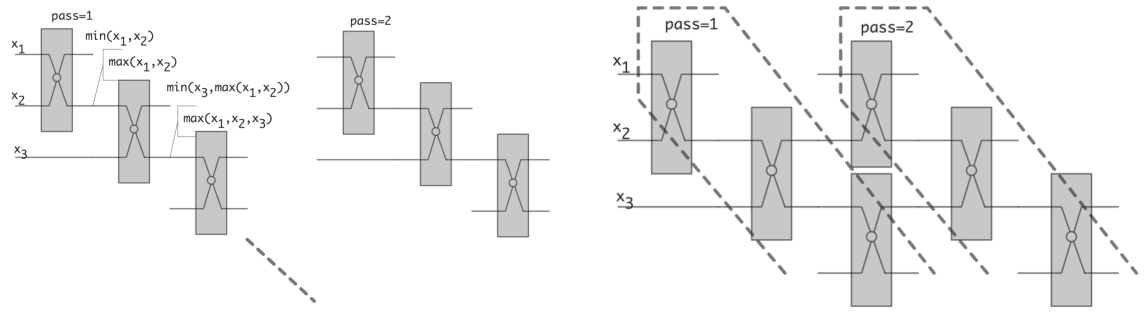


Fig. Algorithme de tri-à-bulles - illustration de deux passes (gauche: séquentiel, droite: parallèle)

Ci-dessus, nous avons remarqué que le tri séquentiel prend au moins un temps en O(N log N). Si nous pouvions obtenir une accélération parfaite, en utilisant **P** processeurs, nous aurions un temps parallèle en O(log N). En examinant à nouveau la figure de droite, vous remarquez que la deuxième passe peut en réalité être entamée bien avant que la première passe ne soit totalement terminée. Si nous examinons maintenant ce qui se passe à un moment donné, nous obtenons l'algorithme de **tri par transposition impair-pair**.

Le **tri par transposition impair-pair** est un **algorithme de tri parallèle simple**, dont la principale vertu est qu'il est relativement facile à implémenter sur une zone linéaire de processeurs. D'autre part, il n'est pas particulièrement efficace. Une seule étape de l'algorithme consiste en deux sous-étapes :

- Chaque processeur de numéro pair effectue une comparaison et un échange avec son voisin de droite ;
- Puis chaque processeur de numéro impair effectue une comparaison et un échange avec son voisin de droite.

Questions:

1. Discutez de l'accélération et de l'efficacité du tri par transposition impair-pair, où nous trions N nombres avec P processeurs ; pour simplifier, nous fixons N = P de sorte que chaque processeur contienne un seul nombre. Exprimez le temps d'exécution en opérations de comparaison et de permutation.
2. Combien d'opérations de comparaison et de permutation le code parallèle prend-il au total ?
3. Combien d'étapes séquentielles l'algorithme prend-il ?
4. Quels sont T_1, T_p, T_∞, S_p, E_p pour le tri de N nombres ?
5. Quelle est la quantité moyenne de parallélisme ?
6. Le tri par transposition impair-pair peut être considéré comme une implémentation parallèle du tri à bulles. Maintenant, que T_1 se réfère au temps d'exécution du tri à bulles (séquentiel), comment cela affecte-t-il S_p et E_p ?

Exercice 2. Tri parallèle avec Tri-rapide

Quicksort est un algorithme récursif qui, contrairement au tri à bulles, n'est pas déterministe. C'est une procédure en deux étapes, basée sur un réarrangement de la séquence :

- **Algorithme :** ***Réarrangement***
- **Entrée** : Un tableau d'éléments et une valeur de "pivot"
- **Sortie** : Le tableau avec les éléments ordonnés en rouge-blanc-bleu,
    - éléments rouges : plus grands que le pivot
    - éléments blancs : égaux au pivot
    - éléments bleus : plus petits que le pivot.

Nous affirmons sans preuve que cela peut être fait en O(n) opérations. Avec cela, quicksort devient :

- **Algorithme :** ***Quicksort***
- **Entrée** : Un tableau d'éléments
- **Sortie** : Le tableau d'entrée, trié
- **Tant que** le tableau a plus d'un élément, faire :
    - Choisissez une valeur arbitraire comme pivot.
    - Appliquez l'algo du ***Réarrangement*** à ce tableau.
    - ***Quicksort***(les éléments bleus)
    - ***Quicksort***(les éléments rouges)

L'indétermination de cet algorithme, et la variance dans sa complexité, proviennent du choix du pivot. Dans le pire des cas, le pivot est toujours le plus petit élément (unique) du tableau. Il n'y aura alors aucun élément bleu, le seul élément blanc est le pivot, et l'appel récursif se fera sur le tableau de n éléments rouges. Il est facile de voir que le temps d'exécution sera alors en **O(n^2)**. D'autre part, si le pivot est toujours (proche de) la médiane, c'est-à-dire l'élément de taille intermédiaire, alors les appels récursifs auront un temps d'exécution à peu près égal, et nous obtenons une formule récursive pour le temps d'exécution :

$$T_n = 2T_{n/2} + O(n)$$

qui est de **O(n log n)**.

Une parallélisation simple de l'algorithme quicksort peut être réalisée en exécutant les deux appels récursifs en parallèle. Cela est le plus facile à réaliser avec un modèle de mémoire partagée et des threads pour les appels récursifs.

```
function par_qs( data,nprocs ) {
  data_lo,data_hi = split(data);
  parallel do:
    par_qs( data_lo,nprocs/2 );
    par_qx( data_hi,nprocs/2 );
}
```

Cette parallélisation divise le tableau en deux parties et exécute récursivement les appels **par_qs()** sur chacune des parties en parallèle.

Cependant, bien que cela semble simple, il y a des défis dans cette approche.

- La division du tableau en deux parties peut ne pas être trivialement parallélisable, car elle nécessite de prendre en compte la répartition des éléments autour du pivot.
- En conséquence, la complexité temporelle de cette approche peut ne pas être aussi favorable que prévu, en particulier dans les cas où le tableau initial est déséquilibré ou lorsque le pivot est mal choisi.

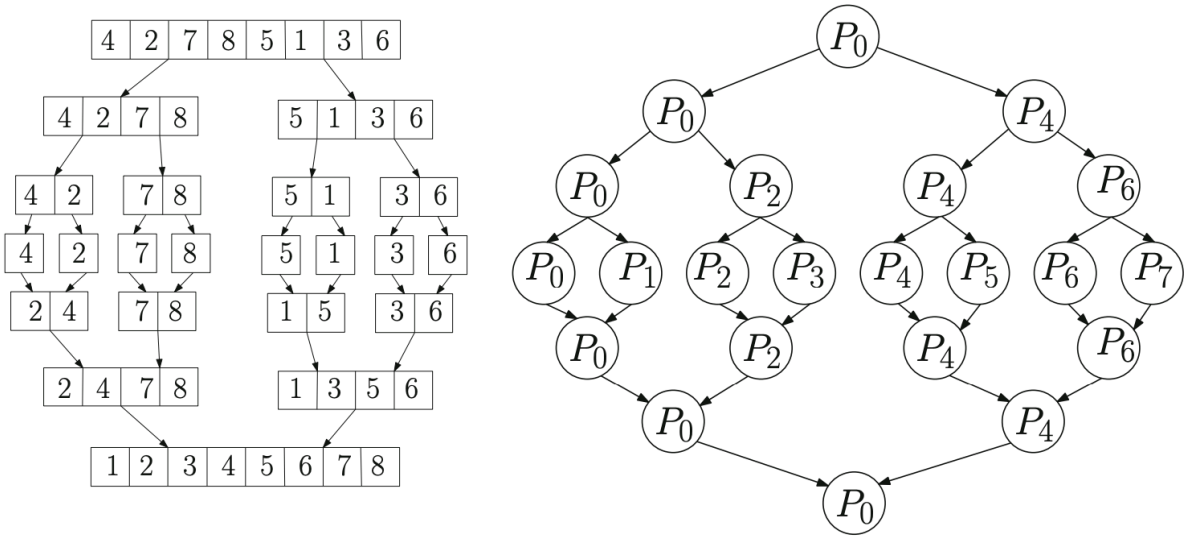
Dans certains cas, une approche plus sophistiquée, telle que la parallélisation du tri par fusion, pourrait être plus efficace pour tirer parti du parallélisme dans le tri de grands ensembles de données.

Questions

1. **Discuter de la proportion de code parallélisable et séquentielle pour ce cas de tri-rapide.**
2. **Quel est le temps d'exécution total, l'accélération et l'efficacité de la parallélisation de l'algorithme quicksort de cette manière ?**
3. **Existe-t-il un moyen de rendre la division du tableau plus efficace ?**
4. **Évaluer le besoin de communication dans cette stratégie.**

Exercice 3. Tri parallèle avec Tri par fusion

Fine grained parallelism with merge sorting



Cette figure illustre un algorithme de tri par fusion, avec des processus parallèles.

1. Rechercher et présenter l'algorithme de tri par fusion dans sa version séquentielle
2. Quelle type de granularité est adaptée à l'implémentation de cet algorithme?
    a. Justifier rigoureusement votre réponse.
3. Proposer un algorithme parallèle en se basant sur la version séquentielle.
4. Implémenter cet algorithme sur MPI.

Exercice 4. Mesure et concept d'efficacité parallèle

La plupart des algorithmes de tri sont basés sur une comparaison de la liste des valeurs complètes des éléments. En revanche, le tri par base effectue plusieurs étapes de tri partiel sur les chiffres du nombre. Pour chaque valeur de chiffre, un « bac » est alloué et les nombres sont déplacés dans ces bacs. La concaténation de ces bacs donne un tableau partiellement trié, et en passant à travers les positions des chiffres, le tableau devient de plus en plus trié.
Considérons un exemple avec un nombre d'au plus deux chiffres, donc deux étapes sont nécessaires :

array	25	52	71	12
last digit	5	2	1	2
(only bins 1,2,5 receive data)				
sorted	71	52	12	25
next digit	7	5	1	2
sorted	12	25	52	71

Il est important que l'ordre partiel d'une étape soit préservé lors de la suivante. Par induction, nous arrivons alors à un tableau totalement trié à la fin.
Un algorithme de tri à mémoire distribuée a déjà une "répartition" évidente des données, donc une implémentation parallèle naturelle du tri par base est basée sur l'utilisation de P, le nombre de processus, comme radix.
Nous illustrons ceci avec un exemple sur deux processeurs, ce qui signifie que nous examinons les représentations binaires des valeurs.

	proc0		proc1	
array	2	5	7	1
binary	010	101	111	001
stage 1: sort by least significant bit				
last digit	0	1	1	1
(this serves as bin number)				
sorted	010		101	111	001
stage 2: sort by middle bit				

stage 2: sort by middle bit				
next digit	1		0	1	0
(this serves as bin number)				
sorted	101	001	010	111	
stage 3: sort by most significant bit				
next digit	1	0	0	1	
(this serves as bin number)				
sorted	001	010	101	111	
decimal	1	2	5	7	

Questions :

1. Proposer une stratégie de parallélisation de cet algorithme en considérant une répartition du problème sur P processeurs.
2. Si plusieurs processeurs sont mobilisés, analyser la portion de code séquentielle et la portion de code parallèle et en déduire la limite de temps induite par la parallélisation.