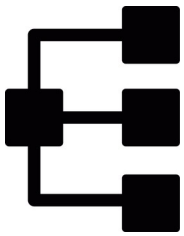
Parallélisation de tâches



Afin d'augmenter les performances des applications et de faire travailler un nombre maximum de cœurs d'une machine, il est possible de découper les traitements en une liste de sous-traitements (appelés tâches) et de les exécuter en parallèle plutôt qu'en séquentiel.

Une tâche est une procédure à exécuter qui peut attendre des paramètres en entrée et retourner un résultat. Les tâches parallèles sont entre autre utiles pour accélérer les temps de traitement de l'application. Plusieurs traitements sont alors exécutés en parallèle au lieu d'être exécutés séquentiellement : la vitesse de traitement est ainsi améliorée.

Objectif de la séance : reprendre l'algorithme du tri fusion et tirer partie de la méthode diviser pour régner pour effectuer ce travail avec des tâches parallèles.

1. Parallélisation des tâches de l'algorithme

Le programme incomplet de cet algorithme réparti sur plusieurs tâches parallèles, est fourni sur tri_fusion_parallele.py

Q1. **Rappeler** le principe de l'algorithme de tri fusion ainsi que les deux fonctions `tri_fusion()` et `fusion()`

Ce tri est basé sur la technique algorithmique diviser pour régner. L'opération principale de l'algorithme est la fusion, qui consiste à réunir deux listes triées en une seule. L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire.

La fonction tri_fusion() est une fonction qui découpe la liste en deux et trie chacune d'elle récursivement puis renvoie la fusion de ces deux listes triées.

La fonction fusion() fusionne deux listes triées en une seule elle aussi triée.

Q2. **Décrire** une méthode de parallélisation de cet algorithme.

Il est possible de découper la liste en N sous listes puis de triées chacune d'elles en parallèle avec N processus différents. Après, il faut fusionner ces N listes triées.

On souhaite maintenant effectuer cette parallélisation en prenant comme nombre de processus deux fois le nombre de cœurs disponibles sur la plateforme, et où l'on partitionne les données à trier équitablement en paquets.

La bibliothèque `Multiprocessing` propose un ensemble de fonctions utiles pour la réserver et gérer des tâches parallèles dont `cpu_count()` qui renvoie le nombre de cœurs disponibles

Q3. **Déterminer** et **tester** une instruction qui renvoie le nombre de processus exécutables selon la machine utilisée. **Compléter** l'affectation de `nb_proc` avec cette expression (ligne 30).

```
from Multiprocessing import *
nb_proc = multiprocessing.cpu_count()*2
```

Q4. On appelle `data` le tableau d'entiers à trier, **compléter** l'expression de la taille `np` des paquets qui seront triés par chaque processus.

```
from Multiprocessing import *
nb_proc = multiprocessing.cpu_count()*2
np = len(data)//nb_proc
```

`tdata` contient la liste des paquets à trier par les processus. La construction par compréhension du `tdata` défini ligne xx est valable seulement si la longueur de la liste à trier est proportionnelle au nombre de processus.

Q5. **Modifier** la construction par compréhension du tableau `tdata` en considérant cette fois que la longueur de la liste à trier n'est pas forcément proportionnelle au nombre de processus.

```
tdata = [data[i*np:(i+1)*np] if i < nb_proc-1 else data[i*np:] for i in range(nb_proc)]
```

La fonction fonction `tri_fusion_parallèle` trie chaque paquet avec un Pool du module `multiprocessing` qui permet de créer plusieurs tâches parallèles puis stocke les résultats dans une variable `pdata`.

Une liste ltest de 1000 données aléatoires désordonnées peut être obtenue par les commandes :

```
>>> import random
>>> ltest = [random.randint(0,10000) for _ in range(1000)]
```

Q6. **Exécuter** la fonction `tri_fusion_parallele` puis **décrire** et **justifier** le contenu de `pdata`.

pdata contient nb_proc liste triées, ces listes ont été triées par chaque processus.

Pour achever le tri, il faut fusionner les paquets de `pdata` 2 à 2 avec la fonction `fusion`. Cette fusion peut être effectuée par l'utilisation d'une file. Le principe consiste à copier `pdata` dans la file puis de `fusionner` les deux premiers éléments défilés dans une même liste et d'enfiler le résultat. Cette opération est répétée tant que la file contient plus d'un élément. La classe `file` du fichier `file.py` est disponible pour ce traitement.

Q7. **Compléter** la fonction `fusion_multiple()` à partir de la ligne 40, afin de fusionner avec la fonction `fusion()` les données contenues dans `pdata`. **Tester** la fonction `fusion_multiple()` puis **compléter** la fonction `tri_fusion_parallèle()` afin de fusionner les listes triées par chaque processus.

```
def fusion_multiple(pdata) :
    file = File()
    for e in pdata : file.enfiler(e)
    while file.nb_elements() > 1 :
        file.enfiler(fusion(file.defiler(), file.defiler()))
    return file.defiler()
```

Q8. La fonction `tri_fusion_parallele` doit maintenant être opérationnelle, **tester** la avec plusieurs jeux de données.

2.Etude des performances de la parallélisation des tâches

La fonction `performance()` permet de tester les performances de la parallélisation du tri de donnée.

Q9. **Etudier** la fonction `performance` et **décrire** le protocole de test de performance retenu.

Cette fonction mesure et affiche le temps d'exécution de tri_fusion() et tri_fusion_parallele() pour des jeux de données de longueur paramétrable

Q10. **Exécuter** la fonction `performance` et **commenter** les performances respectives des différentes fonctions pour des jeux de données de taille allant de 2^8 à 2^{18}. **Discuter** de la performance de cette parallélisation et **identifier** son point faible.

Résultats :

[2** 8] tri_fusion : 0.003159046173095703
[2** 8] tri_fusion_parallele : 0.029020309448242188
[2** 9] tri_fusion : 0.00519251823425293
[2** 9] tri_fusion_parallele : 0.030073881149291992
[2** 10] tri_fusion : 0.007531404495239258
[2** 10] tri_fusion_parallele : 0.029635906219482422
[2** 11] tri_fusion : 0.014852046966552734
[2** 11] tri_fusion_parallele : 0.06415867805480957
[2** 12] tri_fusion : 0.03564882278442383
[2** 12] tri_fusion_parallele : 0.0586848258972168
[2** 13] tri_fusion : 0.08428001403808594
[2** 13] tri_fusion_parallele : 0.08729243278503418
[2** 14] tri_fusion : 0.2255392074584961
[2** 14] tri_fusion_parallele : 0.22089004516601562
[2** 15] tri_fusion : 0.5995888710021973
[2** 15] tri_fusion_parallele : 0.4839169979095459
[2** 16] tri_fusion : 1.7863919734954834
[2** 16] tri_fusion_parallele : 1.4595530033111572
[2** 17] tri_fusion : 5.775151014328003
[2** 17] tri_fusion_parallele : 5.135091304779053
[2** 18] tri_fusion : 23.027045011520386
[2** 18] tri_fusion_parallele : 20.951560020446777

La parallélisation n'est pas efficace pour les jeux de données de faible longueur (< 2^{14}). Le temps nécessaire à la création de processus est trop important comparé au temps de traitement. Ensuite le gain devient de plus en plus intéressant.

La parallélisation est utilisée pour trier des sous listes, il faut ensuite fusionner ces résultats. Cette fusion n'est pas parallélisée alors que ce temps de traitement est linéaire (O(n)).