

TP5 : Parallélisme régulé

```
31 def __init__(self, settings):
32     self.file = None
33     self.fingerprints = set()
34     self.logdupes = True
35     self.debug = debug
36     self.logger = logging.getLogger(__name__)
37     if path:
38         self.file = open(os.path.join(path, 'requests.json'),
39                         'a')
40         self.file.seek(0)
41         self.fingerprints.update(self._get_fingerprints())
42
43 @classmethod
44 def from_settings(cls, settings):
45     debug = settings.getbool('DEBUG', False)
46     return cls(job_dir(settings), debug)
47
48 def request_seen(self, request):
49     fp = self.request_fingerprint(request)
50     if fp in self.fingerprints:
51         return True
52     self.fingerprints.add(fp)
53     if self.file:
54         self.file.write(fp + os.linesep)
55
56 def request_fingerprint(self, request):
57     return request_fingerprint(request)
```

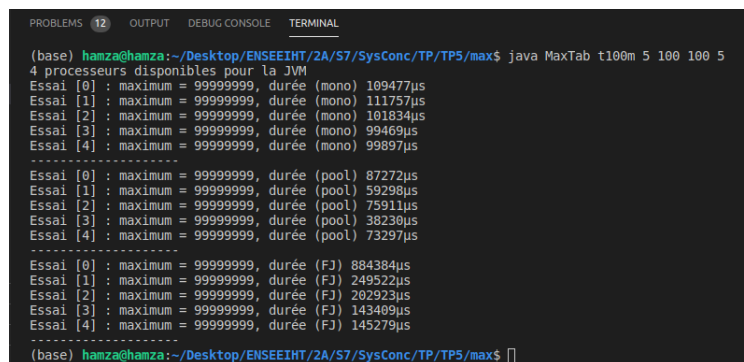
Hamza Mouddene

22 novembre 2020

1 Calcul du maximum d'un tableau

- En pratique, **Pool Fixe** est utilisé pour traiter des demandes indépendantes. Alors que **Fork Join** est plutôt utilisé pour accélérer une seule tâche cohérente, il nous permet d'exécuter facilement les problèmes de type diviser pour régner donc des programmes récusifs. À mon avis, le schéma le plus naturel et le plus puissant est celui du **Pool Fixe**.
- La version **Pool Fixe** est plus efficace en temps que la version **séquentielle**, alors que la verion **séquentielle** est plus efficace en temps que la version **Fork Join**. Donc la version **Pool Fixe** est meillure que la version **Fork Join**.

C'est ce qu'on constate quand on génère un tableau de taille 100 millions et on exécute le programme.



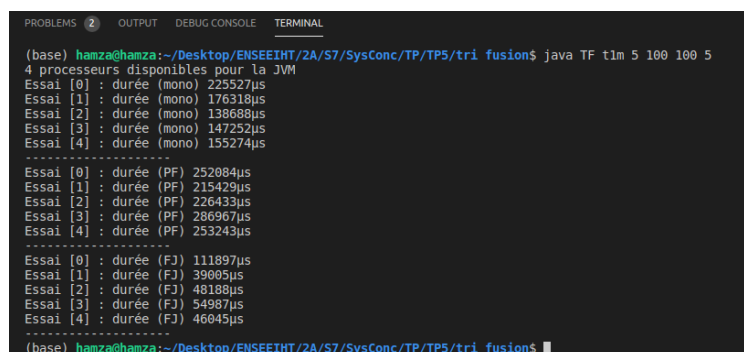
```
(base) hamza@hamza:~/Desktop/ENSEEIH/2A/S7/SysConc/TP/TP5/max$ java MaxTab t100m 5 100 100 5
4 processeurs disponibles pour la JVM
Essai [0] : maximum = 99999999, durée (mono) 109477µs
Essai [1] : maximum = 99999999, durée (mono) 111757µs
Essai [2] : maximum = 99999999, durée (mono) 101834µs
Essai [3] : maximum = 99999999, durée (mono) 99469µs
Essai [4] : maximum = 99999999, durée (mono) 99897µs
-----
Essai [0] : maximum = 99999999, durée (pool) 87272µs
Essai [1] : maximum = 99999999, durée (pool) 59298µs
Essai [2] : maximum = 99999999, durée (pool) 75911µs
Essai [3] : maximum = 99999999, durée (pool) 38230µs
Essai [4] : maximum = 99999999, durée (pool) 73297µs
-----
Essai [0] : maximum = 99999999, durée (FJ) 884384µs
Essai [1] : maximum = 99999999, durée (FJ) 249522µs
Essai [2] : maximum = 99999999, durée (FJ) 209292µs
Essai [3] : maximum = 99999999, durée (FJ) 143409µs
Essai [4] : maximum = 99999999, durée (FJ) 145279µs
-----
(base) hamza@hamza:~/Desktop/ENSEEIH/2A/S7/SysConc/TP/TP5/max$
```

FIGURE 1 – Exécution de MaxTab

2 Tri d'un tableau

- **Fork Join** est particulièrement bien pour les problèmes récursives, où une tâche implique l'exécution de sous-tâches, puis le traitement de leurs résultats. Alors que si on essaye de résoudre de tels problèmes de type **Pool Fixe** on se retrouve avec des threads bloqués en attendant que d'autres threads leur fournissent des résultats.

En terme de conception, la version **Pool Fixe** est plus manuelle à implémenter. D'autre part, **Fork Join** à construire.



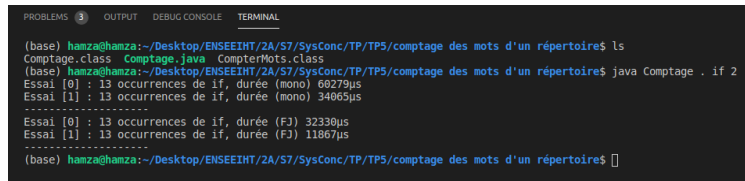
```
(base) hamza@hamza:~/Desktop/ENSEEIH/2A/S7/SysConc/TP/TP5/tri fusion$ java TF t1m 5 100 100 5
4 processeurs disponibles pour la JVM
Essai [0] : durée (mono) 225527µs
Essai [1] : durée (mono) 176318µs
Essai [2] : durée (mono) 138688µs
Essai [3] : durée (mono) 147252µs
Essai [4] : durée (mono) 155274µs
-----
Essai [0] : durée (PF) 252084µs
Essai [1] : durée (PF) 215429µs
Essai [2] : durée (PF) 226433µs
Essai [3] : durée (PF) 286967µs
Essai [4] : durée (PF) 253243µs
-----
Essai [0] : durée (FJ) 111897µs
Essai [1] : durée (FJ) 39005µs
Essai [2] : durée (FJ) 48188µs
Essai [3] : durée (FJ) 54987µs
Essai [4] : durée (FJ) 46045µs
-----
(base) hamza@hamza:~/Desktop/ENSEEIH/2A/S7/SysConc/TP/TP5/tri fusion$
```

FIGURE 2 – Exécution de TF

la figure précédente montre l'exécution de l'algorithme de tri fusion pour un jeu de données de tailles un million, ce qui montre que la version **Fork Join** est plus efficace que la version *séquentielle*. Par ailleurs, on trouve que la version *séquentielle* est plus rapide que la version **Pool Fixe**.

3 Comptage de mots

- En terme de facilité, **Fork Join** est plus simple que la version *séquentielle*, mais en terme de performance **Fork Join** est plus efficace que la version la version *séquentielle*.



```
(base) hamza@hamza:~/Desktop/ENSEEINT/2A/S7/SysConc/TP/TP5/comptage des mots d'un repertoire$ ls
Comptage.class  Comptage.java  CompterMots.class
(base) hamza@hamza:~/Desktop/ENSEEINT/2A/S7/SysConc/TP/TP5/comptage des mots d'un repertoire$ java Comptage . if 2
Essai [0] : 13 occurrences de if, durée (mono) 60279µs
Essai [1] : 13 occurrences de if, durée (mono) 34865µs
-----
Essai [0] : 13 occurrences de if, durée (FJ) 32338µs
Essai [1] : 13 occurrences de if, durée (FJ) 11867µs
-----
(base) hamza@hamza:~/Desktop/ENSEEINT/2A/S7/SysConc/TP/TP5/comptage des mots d'un repertoire$
```

FIGURE 3 – Exécution de TF

la figure précédente montre l'exécution de l'algorithme de comptage de mots dans un répertoire qui contient 3 fichiers avec un fichier java et on va chercher le mot clé *if*, l'exécution montre que la version **Fork Join** est plus efficace en temps que la version *séquentielle*.