Cours	INF5171
Groupe	20
Session	Automne 2016
Etudiant	David Maignan
Code MS	MAID10077306
Projet	TP3

1. Présentation

Synchronisation de fichiers entre un compte Google drive nuagique et un laptop. Les fichiers retournés ne connaissent que l'identifiant du parent direct. Il est donc nécessaire de les insérer dans un arbre pour faciliter la sauvegarde et de connaître le chemin complet du fichier pour l'écriture sur le disque.

1. Envoi des requêtes pour recevoir la totalité des fichiers
2. Insertion des fichiers d'un arbre représentant la hiérarchie entre les répertoires et les fichiers

L'approche retenue est l'utilisation d'une structure de synchronisation entre un producteur et des consommateurs.

Notes techniques:

1. les requêtes vers l'api nécessitent l'identifiant du dernier fichier reçu et donc ne peuvent pas être faites en parallèle.
2. Chaque réponse retourne 1000 fichiers maximum (limite technique de l'api) qui peuvent être traité par des threads parallèlement
3. L'ajout un noeud dans l'arbre peut échouer si le parent n'est pas encore présent. En premier lieu, je remettais le fichier dans la liste du moniteur mais cela entraînait une erreur en cascade si les noeuds suivants qui en dépendait.
4. Approche retenue est de mettre en attente le thread. Le parent est obligatoirement en cours de traitement par un autre thread et donc pas de situation de blocage possible.
5.

inf5171/monitor/producer/FileProducer

- un service producteur qui simule les requêtes/réponses vers l'API. Il est exécuté dans un thread unique avec un sleep de 100ms pour 'simuler' le temps entre les réponses vers l'api. Contrainte technique d'un maximum de 1000 fichiers par réponses.

inf5171/moniteur/consumer/TreadConsumer
inf5171/moniteur/consumer/ForkJoinConsumer

- des consommateurs qui sont activés par des des signaux lorsque le moniteur reçoit des fichiers par le producteur pour insertion dans un arbre.

inf5171/monitor/MStrutureMonitor
Structure qui joue le rôle de synchronisation entre le producteur et les consommateurs. Utilisation d'un ArrayDeque pour garder l'ordre des fichiers depuis le root jusqu'aux branches pour une insertion ordonnée.

model/tree/TreeNode
Structure ConcurrentLinkDeque pour la liste des enfants d'un noeud parent. Cette structure est partagée entre les différents consommateurs.

Plus outils pour mesures et tests de résultats.

Code:

```
MStructureMonitor {
Queue queue;
public Boolean push(T v){
lock.lock()
queue.add(v);
notEmpty.signalAll();
lock.unlock();
}

@Override
public Boolean push(List<T> list) {
  lock.lock();
  Boolean added = false;
  try {
      added = queue.addAll(list);
      if (added) {
          notEmpty.signalAll();
      }
  } finally {
      lock.unlock();
  }

  return added;
}

@Override
```

```
public T shift() {
  lock.lock();
  T value = null;
  try {
      while (queue.size() == 0 && ! completed) {
          try {
              notEmpty.await();
          } catch (InterruptedException e) {
              e.printStackTrace();
          }
      }

      value = queue.poll();

  } finally {
      lock.unlock();
  }

  return value;
}
```

```
FileProducer implements Runnable{
            @Override
            public void run() {
              T[] list = (T[]) fileList.toArray();
              int total = list.length;
              int index = 0;

              while(index < total){
                  try {
                      sleep(100L);    // estimation assez optimiste du temps de réponse de l'api
                  } catch (InterruptedException e) {
                      e.printStackTrace();
                  }

                  int last = (index < total - threshold)? index + threshold : total;
```

```
                monitor.push(Arrays.asList(Arrays.copyOfRange(list, index, last)));
                index += threshold;
            }

            monitor.setCompleted(true);
        }
}

ThreadConsumer implements Runnable{
        @Override
        public void run() {
          while(uncompleted() || completedNotEmpty()) {
            File file = fileMonitor.shift();

            TreeNode node = null;

            if (file != null){
                while(node == null){
                    node = treeBuilder.insertFile(file);
                    if(node == null) {
                        try {
                            Thread.sleep(50L);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
          }
        }
}
```

2. Analyse des résultats:

Lorsque le nombre de fichiers est peu élevé, les trois versions ont un temps d'exécution assez proche. La version séquentielle peut occasionnellement être plus performante si l'on crée un trop grand nombre de

threads par rapport au nombre de fichiers à traiter. Surcoût pour la création et gestion des threads et un enjambement entre les threads pour les fichiers parents / enfants (parent absent de l'arbre mais en cours de traitement).

A partir d'un nombre de fichiers de plusieurs milliers, les versions parallèles présentent un gain de performance significatif par rapport à la version séquentielle.

1. Une première raison est que le temps pour le producteur de produire tous les fichiers est plus important. La version séquentielle qui attend que tous les fichiers soient produits avant de commencer sa tâche est ainsi pénalisé par rapport au version parallèle qui débute dès la réception de fichiers.
2. On constate que pour les versions parallèles avec 1 thread consommateur leur temps d'exécution est plus ou moins égale au temps séquentielle moins le temps total pour produire les fichiers.

   Temps sec = Temps parallèle (1 thread) + Temps Producteur

3. L'augmentation du nombre de fichiers permet d'apprécier des gains de performance avec les versions parallèles. La version avec un forkJoinPool donne des resultats sensiblement meilleur qu'avec des threads instancies manuellement. Un seuil semble etre atteint et l'ajout de plus de consommateurs ne semblent pas influer sur la performance
4. La performance n'augmente plus significativement apres un seuil de threads qui est possiblement du à la limitation du producteur qui ne produit que 1000 fichiers tous les 100 ms.

Tableau de résultats:

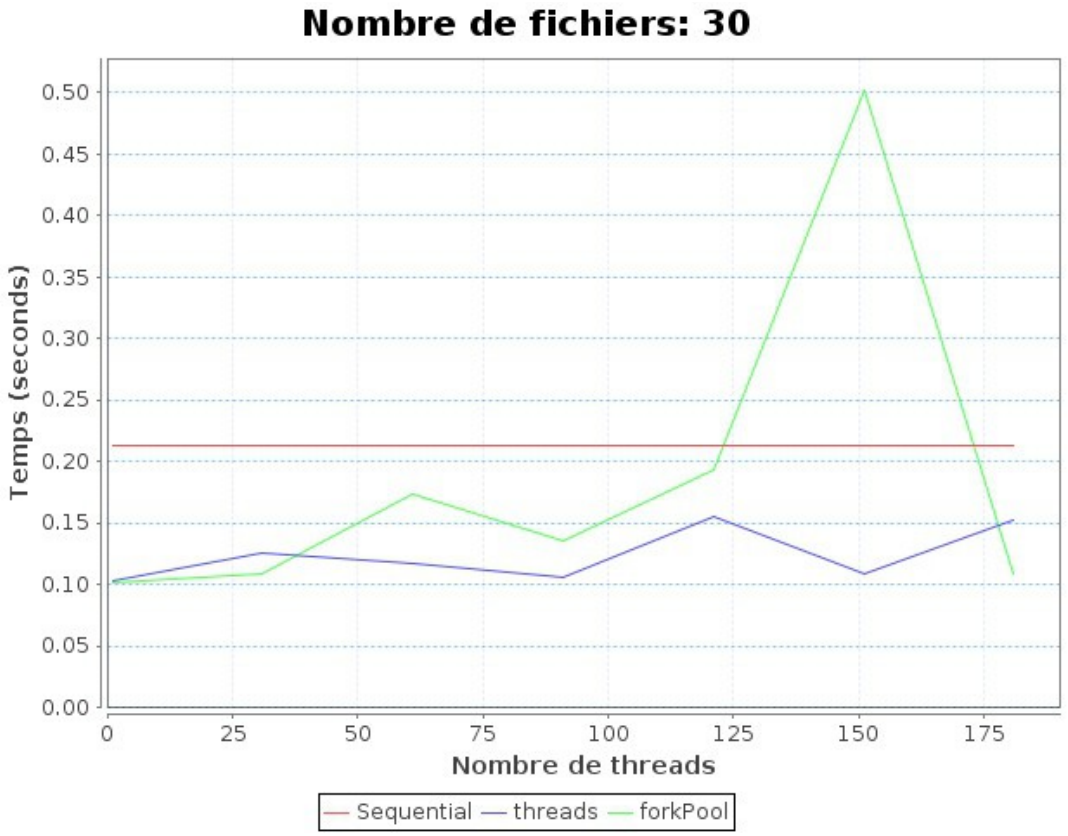
Nombre de fichiers: 30 à 55980
Nombre de threads: 1 à 181

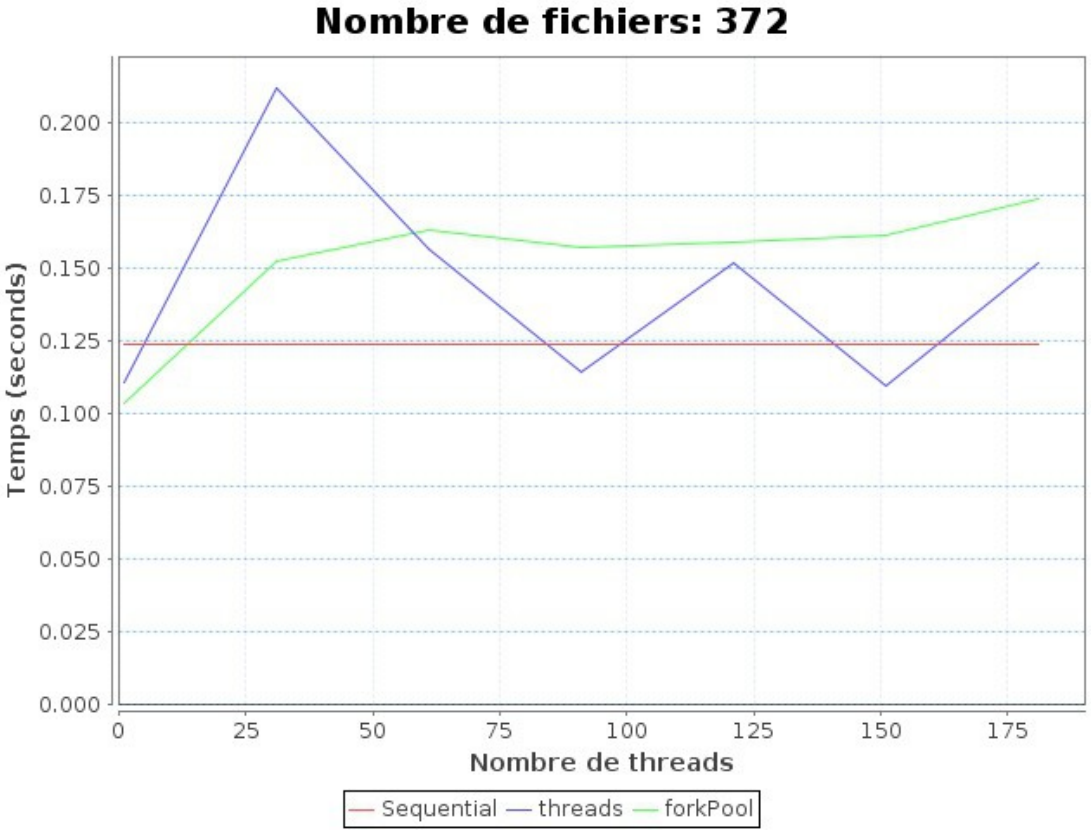
nbThreads	nbFiles	nbNodes	Tps (secs)	Acceleration
sequential				
1	30	31	0.2124444610	1.0000000000
1	372	373	0.1240736750	1.0000000000
1	2178	2179	0.3950975600	1.0000000000
1	8184	8185	2.6015148830	1.0000000000
1	23430	23431	16.6199906280	1.0000000000
1	55980	55981	114.8180211810	1.0000000000
threads				
1	30	31	0.1036344660	2.0499402293
31	30	31	0.1256719050	1.6904690114
61	30	31	0.1175601330	1.8071131393
91	30	31	0.1060185310	2.0038427150
121	30	31	0.1549589730	1.3709723089
151	30	31	0.1082609080	1.9623376981
181	30	31	0.1528413420	1.3899672577
1	372	373	0.1111129220	1.1166448759

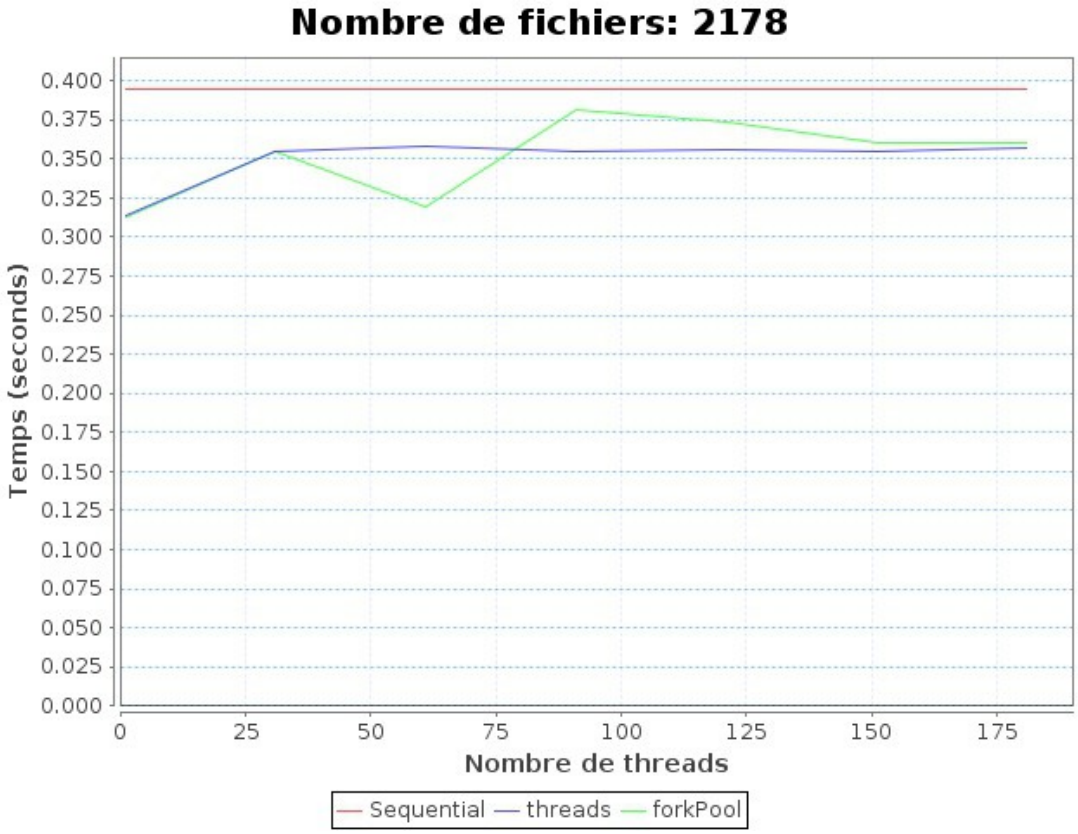
31|	372|	373|	0.2122415010|	0.5845872481
61|	372|	373|	0.1564208150|	0.7932043763
91|	372|	373|	0.1143715350|	1.0848300235
121|	372|	373|	0.1520433060|	0.8160416809
151|	372|	373|	0.1096745150|	1.1312899355
181|	372|	373|	0.1516237870|	0.8182995390
1|	2178|	2179|	0.3138097610|	1.2590352790
31|	2178|	2179|	0.3544930650|	1.1145424241
61|	2178|	2179|	0.3578603190|	1.1040552389
91|	2178|	2179|	0.3545108770|	1.1144864252
121|	2178|	2179|	0.3562693850|	1.1089854381
151|	2178|	2179|	0.3551656330|	1.1124318439
181|	2178|	2179|	0.3572853920|	1.1058318332
1|	8184|	8185|	1.6742828410|	1.5538084840
31|	8184|	8185|	0.9620044830|	2.7042648231
61|	8184|	8185|	0.9597754490|	2.7105453528
91|	8184|	8185|	0.9638654890|	2.6990435000
121|	8184|	8185|	0.9697927880|	2.6825471536
151|	8184|	8185|	0.9613525560|	2.7060986802
181|	8184|	8185|	0.9679782830|	2.6875756705
1|	23430|	23431|	14.2843150680|	1.1635133045
31|	23430|	23431|	2.9374568890|	5.6579521865
61|	23430|	23431|	2.6563707320|	6.2566532705
91|	23430|	23431|	2.7222460690|	6.1052491974
121|	23430|	23431|	2.6026478960|	6.3858006508
151|	23430|	23431|	2.5316853890|	6.5647930427
181|	23430|	23431|	2.6440440330|	6.2858221802
1|	55980|	55981|	142.3855174140|	0.8063883411
31|	55980|	55981|	37.2855344370|	3.0794253835
61|	55980|	55981|	29.6313895200|	3.8748780614
91|	55980|	55981|	30.6574578540|	3.7451905415
121|	55980|	55981|	21.1377199310|	5.4319019060
151|	55980|	55981|	20.7372957070|	5.5367885381
181|	55980|	55981|	19.0909957840|	6.0142499888
forkPool				
1|	30|	31|	0.1015962940|	2.0910650638
31|	30|	31|	0.1091391750|	1.9465463341
61|	30|	31|	0.1739255780|	1.2214676153
91|	30|	31|	0.1355911210|	1.5668021581
121|	30|	31|	0.1930202200|	1.1006331927
151|	30|	31|	0.5025441140|	0.4227379350
181|	30|	31|	0.1092150510|	1.9451939916
1|	372|	373|	0.1035696700|	1.1979730649
31|	372|	373|	0.1524613960|	0.8138038760
61|	372|	373|	0.1631092110|	0.7606785309
91|	372|	373|	0.1573262360|	0.7886394422
121|	372|	373|	0.1593358080|	0.7786929790
151|	372|	373|	0.1614887320|	0.7683116553
181|	372|	373|	0.1740102870|	0.7130249432
1|	2178|	2179|	0.3131188730|	1.2618133050
31|	2178|	2179|	0.3544183540|	1.1147773684
61|	2178|	2179|	0.3190460590|	1.2383715418
91|	2178|	2179|	0.3812130230|	1.0364219902
121|	2178|	2179|	0.3736659190|	1.0573550862
151|	2178|	2179|	0.3600746510|	1.0972656889
181|	2178|	2179|	0.3601688310|	1.0969787666
1|	8184|	8185|	1.5483153700|	1.6802228625
31|	8184|	8185|	0.9673900120|	2.6892099885
61|	8184|	8185|	0.9606772260|	2.7080009941

91|	8184|	8185|	1.0168325030|	2.5584497696
121|	8184|	8185|	0.9838423490|	2.6442395833
151|	8184|	8185|	1.1498966760|	2.2623901236
181|	8184|	8185|	1.0139594160|	2.5656992203
1|	23430|	23431|	15.6055266430|	1.0650067126
31|	23430|	23431|	3.3190282620|	5.0074869257
61|	23430|	23431|	3.4228156650|	4.8556487567
91|	23430|	23431|	2.7941660740|	5.9481040811
121|	23430|	23431|	2.8941840680|	5.7425478952
151|	23430|	23431|	3.0617813270|	5.4282095463
181|	23430|	23431|	2.6651749460|	6.2359848658
1|	55980|	55981|	132.8006730530|	0.8645891511
31|	55980|	55981|	44.7393125550|	2.5663787534
61|	55980|	55981|	35.0598410620|	3.2749156215
91|	55980|	55981|	26.6855040590|	4.3026364024
121|	55980|	55981|	28.5805375800|	4.0173499487
151|	55980|	55981|	23.0312454680|	4.9853153335
181|	55980|	55981|	16.6820596610|	6.8827245265

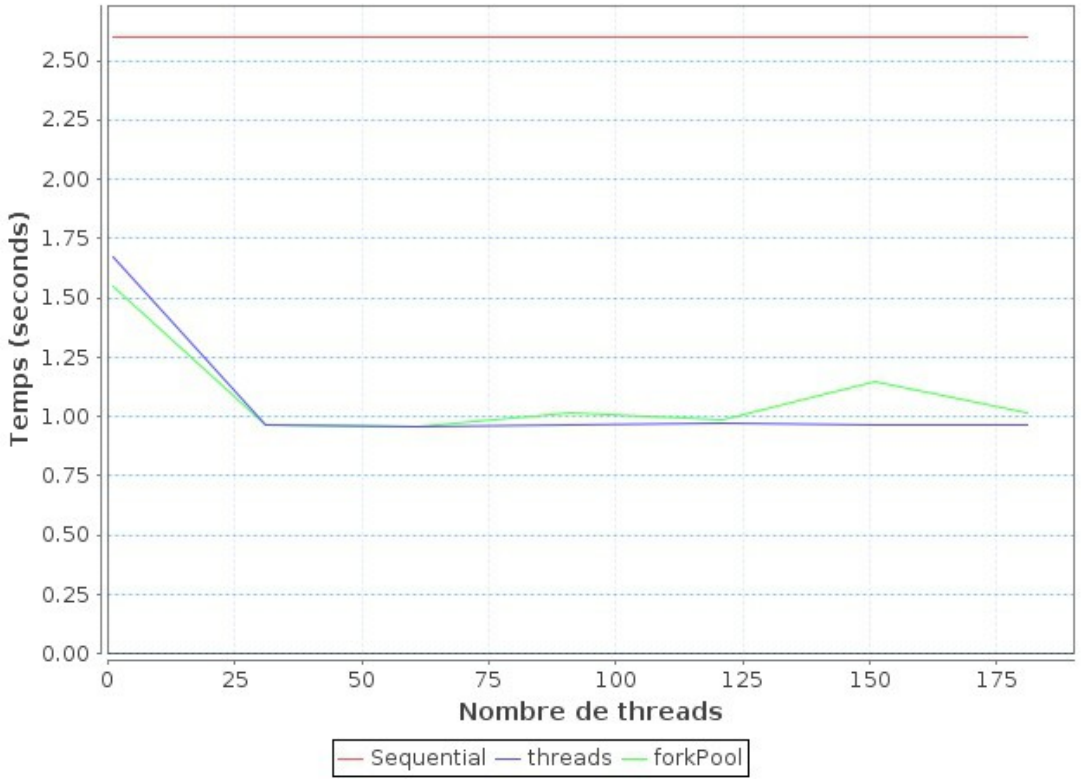
Graphiques des résultats



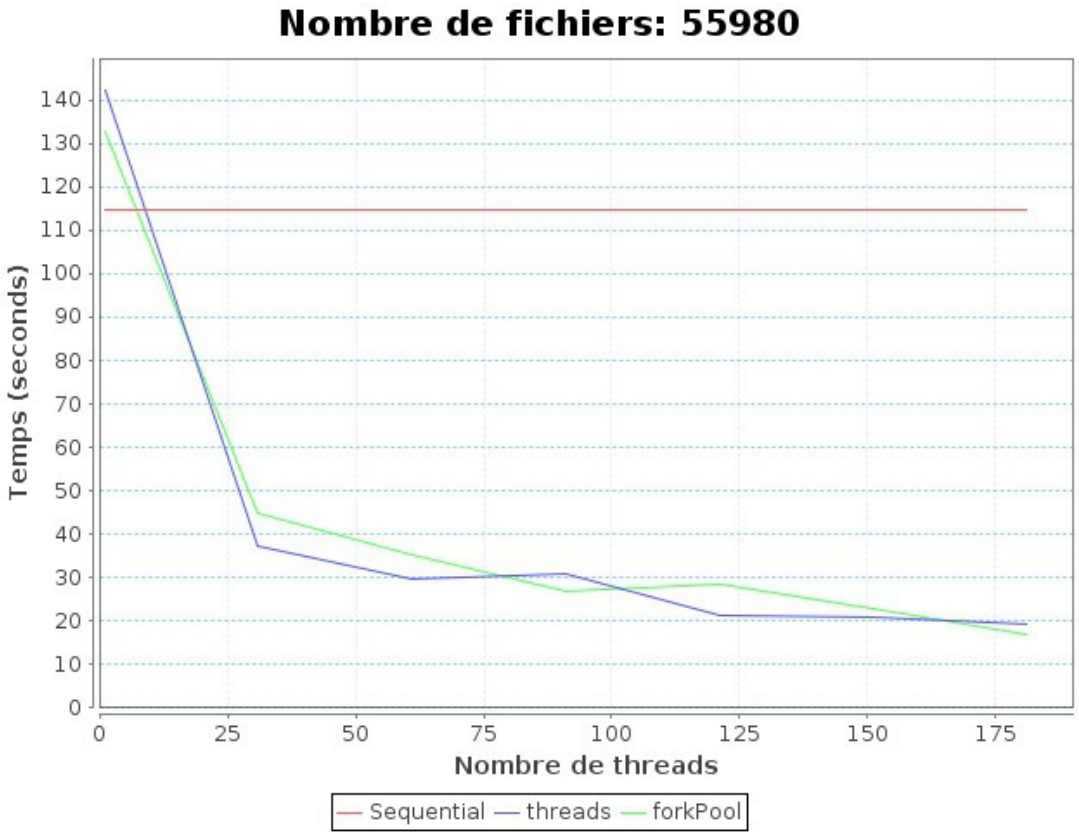

Nombre de fichiers: 372
Temps (seconds)
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
25
50
75
100
125
150
175
Nombre de threads
Sequential
threads
forkPool


Nombre de fichiers: 2178
Temps (seconds)
0.400
0.375
0.350
0.325
0.300
0.275
0.250
0.225
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
25
50
75
100
125
150
175
Nombre de threads
Sequential
threads
forkPool

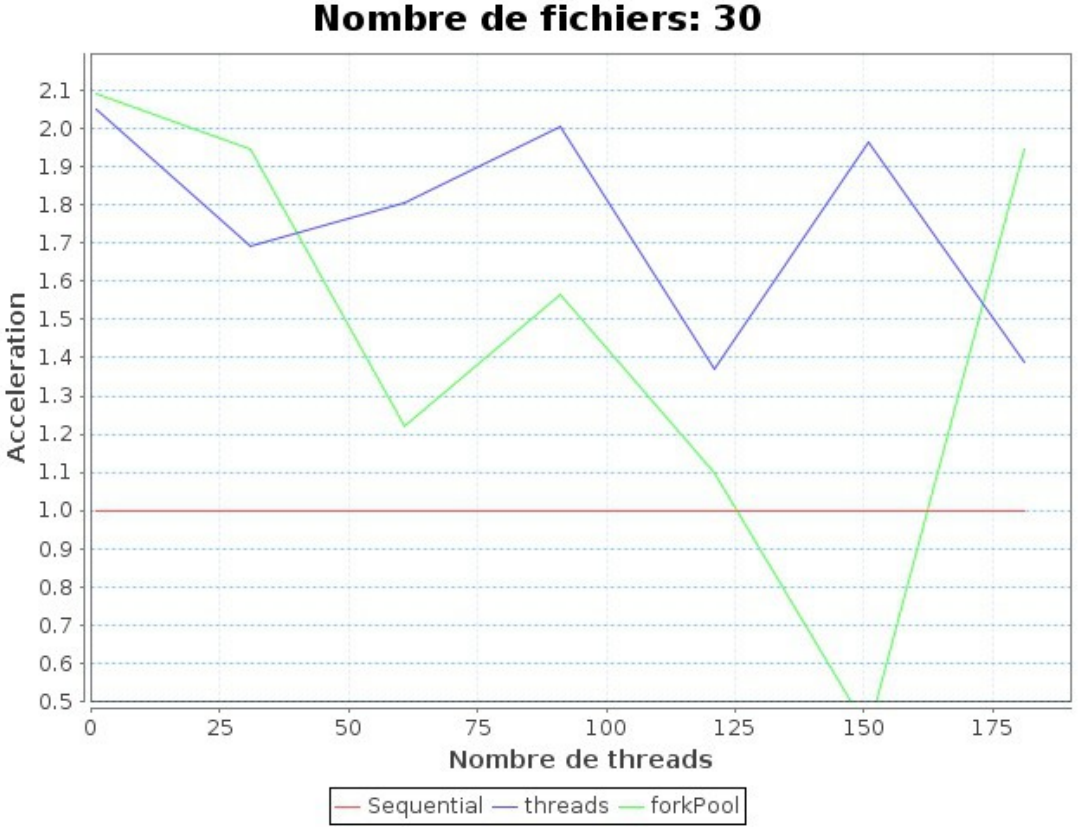
Nombre de fichiers: 8184

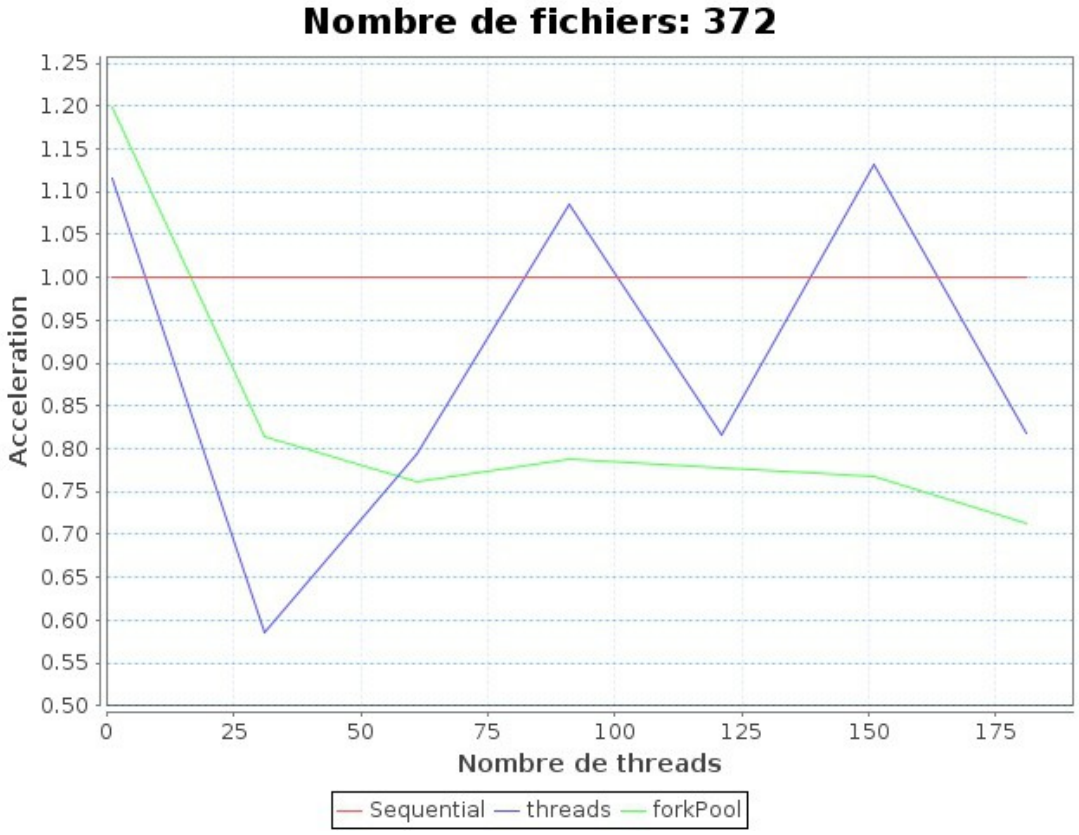


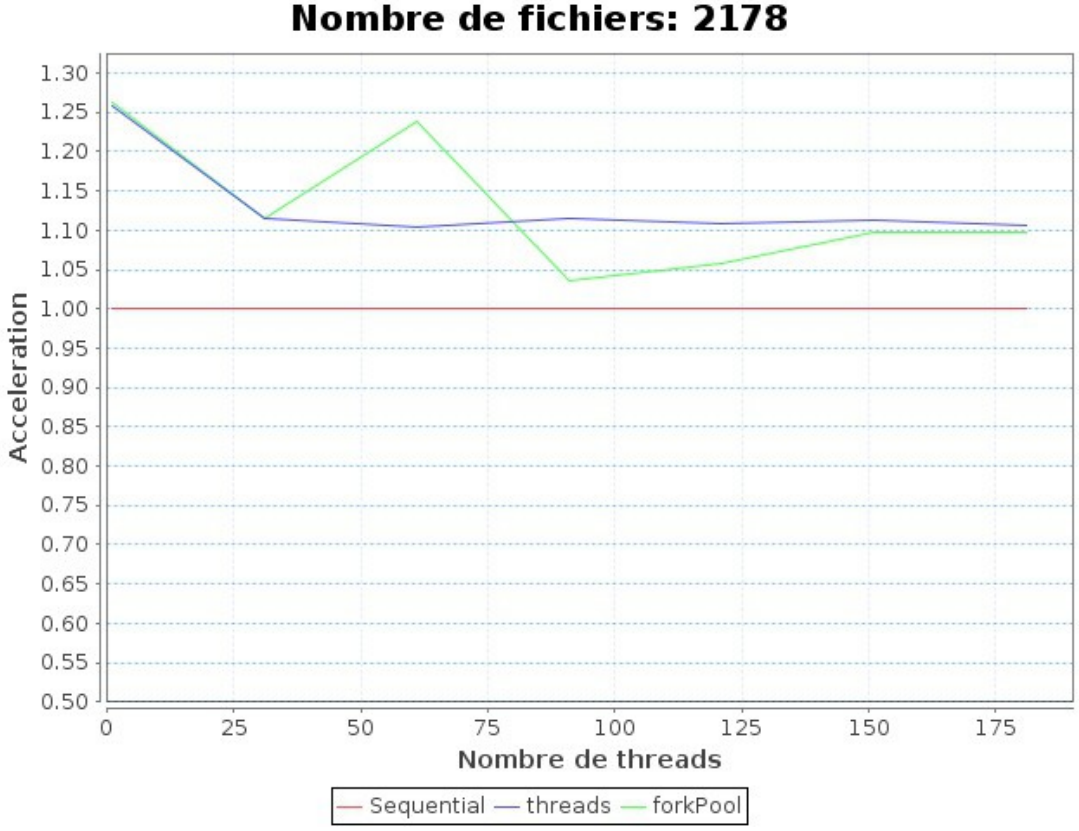
Nombre de fichiers: 23430

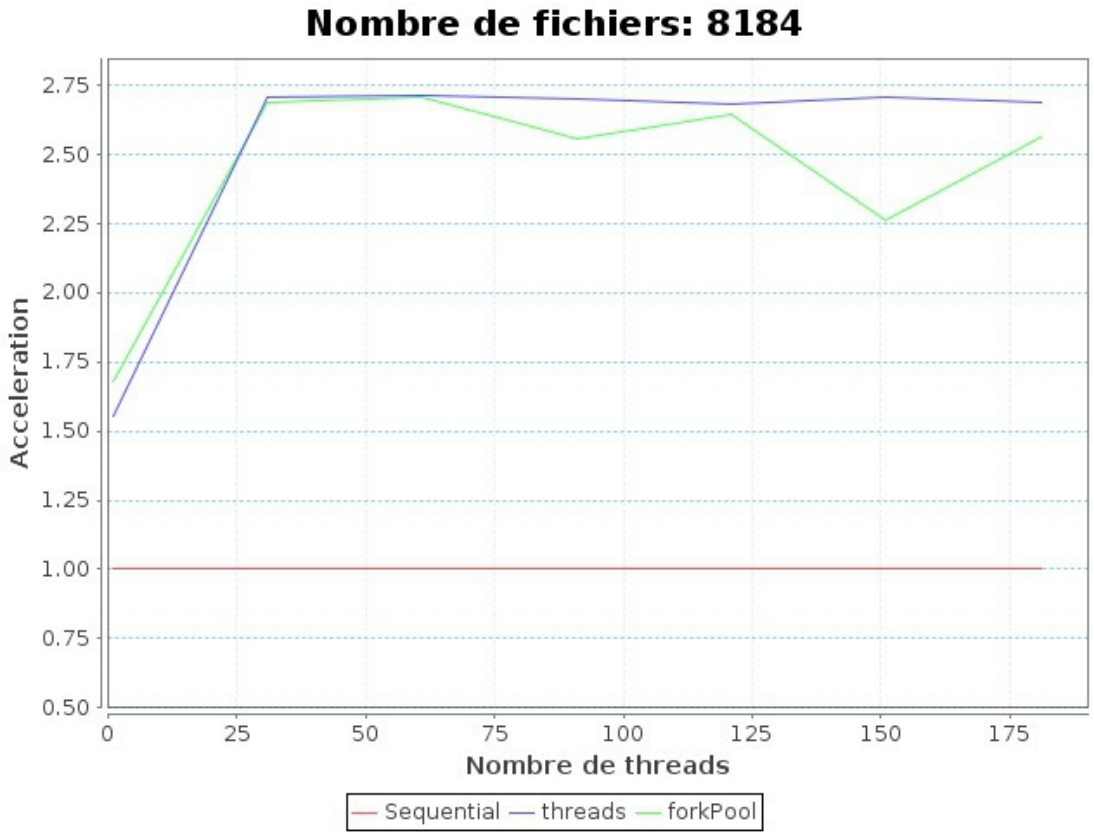


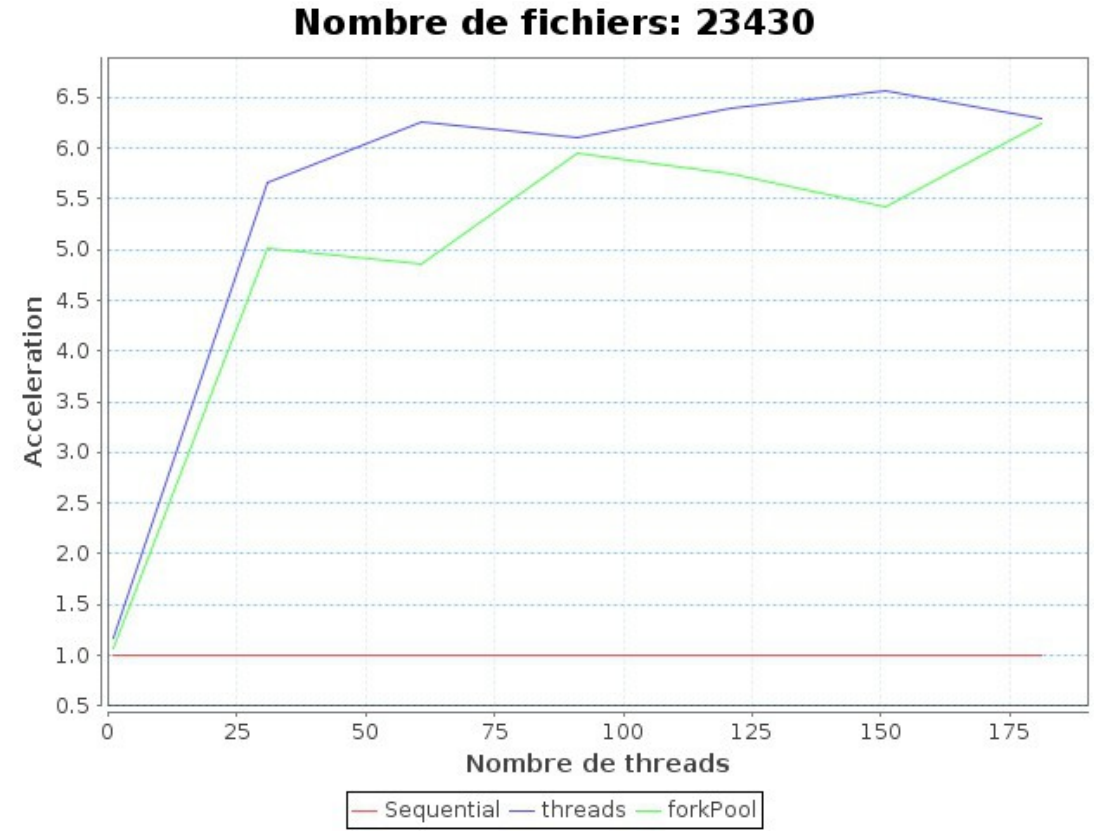
Acceleration.

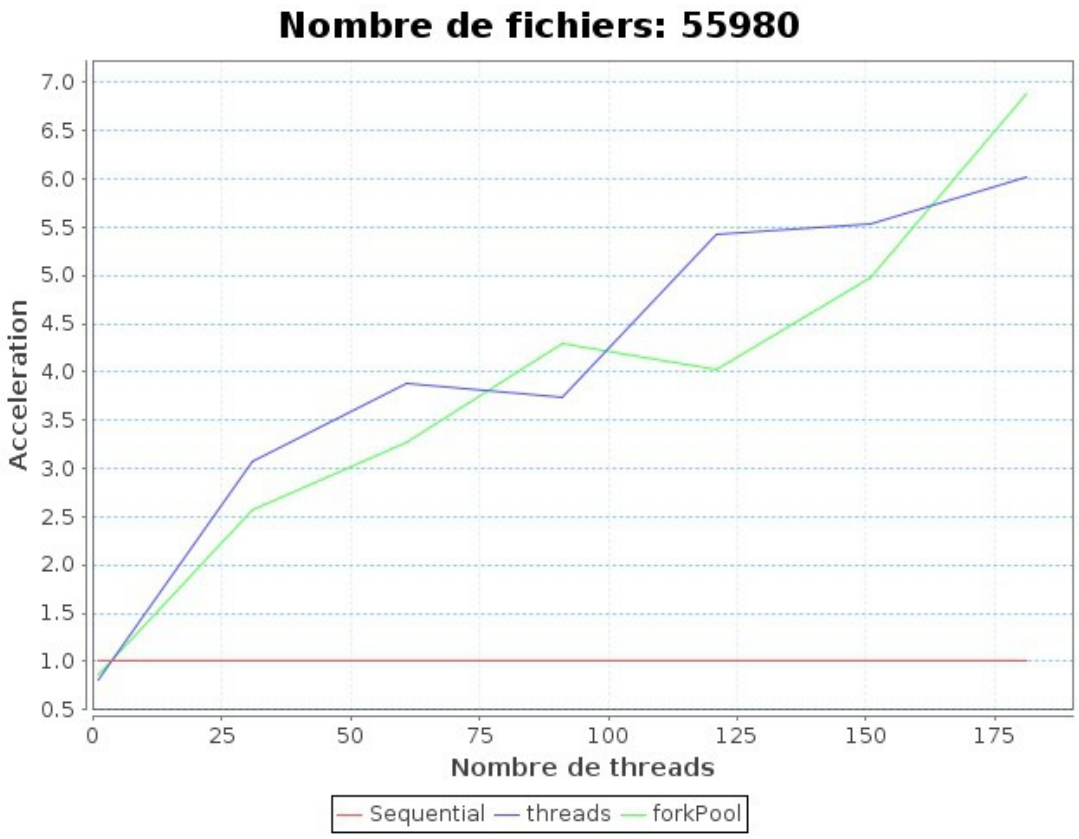



Nombre de fichiers: 372
Acceleration
1.25
1.20
1.15
1.10
1.05
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0
25
50
75
100
125
150
175
Nombre de threads
Sequential
threads
forkPool


Nombre de fichiers: 2178
Acceleration
1.30
1.25
1.20
1.15
1.10
1.05
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0
25
50
75
100
125
150
175
Nombre de threads
Sequential
threads
forkPool


Nombre de fichiers: 8184
Acceleration
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0
25
50
75
100
125
150
175
Nombre de threads
Sequential
threads
forkPool


Nombre de fichiers: 23430
Acceleration
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
25
50
75
100
125
150
175
Nombre de threads
Sequential
threads
forkPool



3. Stratégie de tests

Un minimum de deux boucles est fait pour les versions séquentielles pour avoir une moyenne et pondérer un peu les différences entre deux exécutions.

Tests unitaires couvrent:
L'ensemble des méthodes d'insertion des noeuds dans l'arbre pour garantir le bon emplacement des fichiers dans la structure.
Des tests unitaires couvrent les différents outils (Node Count, FileCount, Report) qui comptent le nombre des noeuds qui composent l'arbre final, le nombre de fichiers écrits sur le disque et les calculs pour le rapport final.

Test de performance et de résultat:
Chaque exécution produit un nombre connu de fichiers uniques. Chaque fichier doit être retrouvé sous forme de noeud dans l'arbre final. Ces calculs sont affichés dans les colonnes nbFiles et n Nodes. Note: nbNode = nbFiles + 1 (noeud root de l'arbre). Un timer de la librairie apache commons est utilise pour mesurer les temps au cours de l'exécution.

Conclusion:
Cette première expérience de programmation avec le langage Ruby m'a enthousiasmé. J'ai encore des difficultés significatives pour lire et comprendre la syntaxe. Ce qui ne me permet pas gagner en de productivité. Je passe malheureusement beaucoup plus de temps à déboguer du code plutôt qu'à en écrire.