Introduction à Spark

Enseignant : François Taiani

Etudiants : Ellyn Bleas & Cédric Dubois

Code : https://github.com/EllynB/ABD

Introduction

Au cours de ce TP, nous avons pu découvrir et avoir une première approche de Spark. Mais tout d'abord, qu'est ce que Spark ? Spark est un framework open-source et développé par la fondation Apache (gérant également les projets Hadoop, Cassandra, Tomcat mais le plus connu étant OpenOffice). Ce framework permet et facilite la gestion de l'exécution parallèle de programme sur une ou plusieurs machines (cluster).

Au cours de ce TP, les différents programmes ont été rédigé en Scala.

Partie I : Expérimentation sur une seule machine

Intégrale

$$\int_1^{10} \frac{1}{x} dx$$

Manuellement, la valeur cette intégrale est d'environ 2,3 (plus exactement ln(10)).

Après rédaction d'un programme Spark utilisant la méthode des rectangles afin de calculer une intégrale, nous avons réalisé une série de mesures en faisant varier le nombre de rectangle, le nombre de thread et enfin le nombre de slice. Ceci a été fait dans le but de voir l'effet de chacun de ces paramètres sur les performances du programme.

Les paramètres par défaut sont 500 rectangles, 4 threads et 2 slices.

Seul le temps de calcul est mesuré (omission de l'initialisation).

Variation du nombre de rectangle :

Nous avons ici fait varier le nombre de rectangle utilisé par la méthode de calcul d'intégrale (Méthode des Rectangles) de 3 à 50 000 000. Nous avons pu remarquer une variation du résultat obtenu (de plus en plus juste et précis jusqu'à 500 000 puis une perte de précision réapparaît). Pour ce qui est du temps d'exécution :

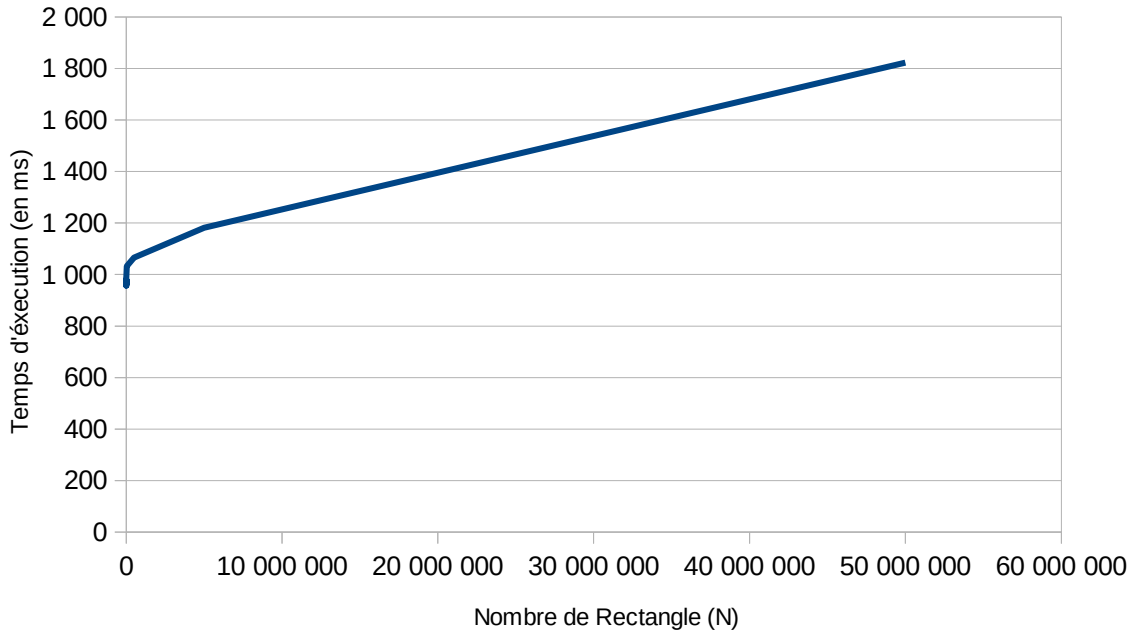

Illustration 1: Temps d'exécution du programme en fonction du nombre de N de rectangle

N	Temps moyen (ms)	Temps 1	Temps 2	Temps 3	Temps 4	Temps 5	Temps 6	Temps 7	Resultat
3	958	955	945	943	964	965	957	974	0,75
8	961	946	951	982	964	958	964	959	1,65
10	962	963	965	953	964	972	953	966	1,77
20	980	963	958	1 026	956	972	1 024	964	2,02
50	957	949	959	957	963	969	938	963	2,18
100	979	960	965	999	954	1 027	985	963	2,24
500	966	974	950	963	961	963	965	984	2,29
5 000	986	985	974	986	992	990	978	997	2,30
50 000	1 033	1 033	1 037	1 042	1 021	1 012	1 047	1 036	2,30
500 000	1 065	1 062	1 072	1 056	1 039	1 078	1 076	1 074	2,30
5 000 000	1 182	1 167	1 184	1 162	1 186	1 156	1 165	1 252	2,29
50 000 000	1 822	1 810	1 815	1 852	1 824	1 816	1 816	1 824	1,88

Le temps d'exécution a visiblement tendance à augmenter proportionnellement au nombre de rectangle utilisé. Ce qui est le résultat attendu car plus il y a de rectangles, plus il y a de calculs à faire.

Variation du nombre de thread :

Nous avons ici fait varier le nombre de thread de 1 à 500 000, mais, pour des questions de visibilité, le schéma présent ci-dessous ne représente qu'une partie de nos mesures.

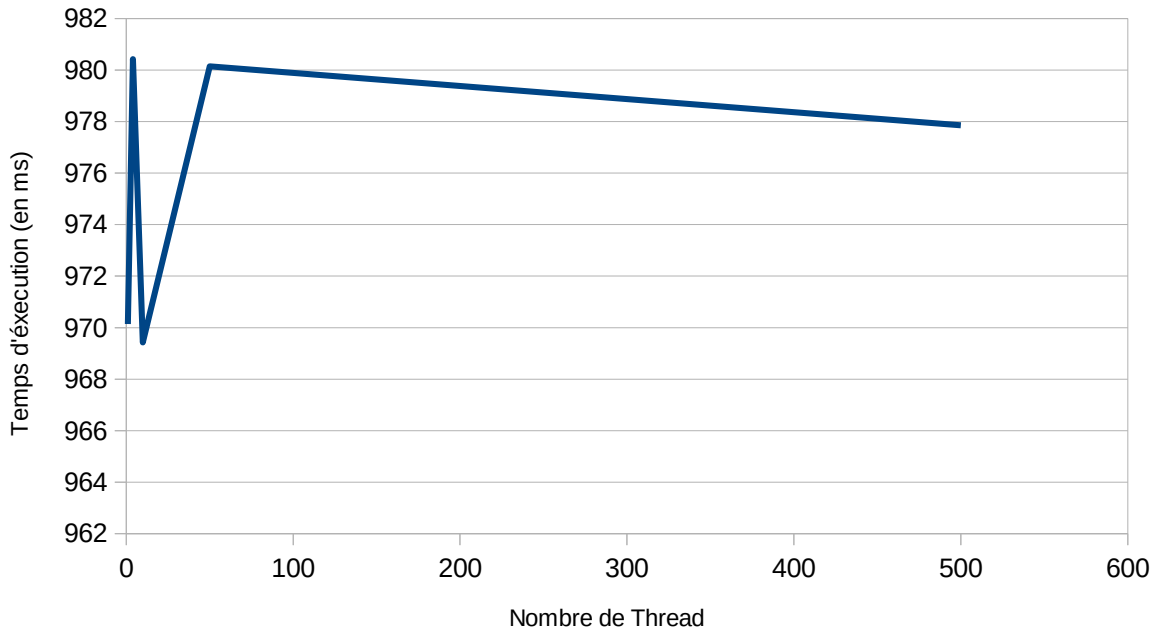

Illustration 2: Temps d'exécution du programme en fonction du nombre de thread

Thread	Temps moyen (ms)	Temps 1	Temps 2	Temps 3	Temps 4	Temps 5	Temps 6	Temps 7
1	970	974	970	967	964	974	984	958
4	980	979	966	1 032	967	967	962	990
10	969	981	974	952	971	972	963	973
50	980	970	990	967	982	983	984	985
500	978	960	1 055	962	958	982	965	963
5000	965	961	972	971	957	965	959	972
50000	967	953	966	952	975	971	990	960
500000	985	1 022	966	1 038	964	980	966	960

L'on peut observer sur cette courbe que le temps d'exécution du programme varie (apparemment) grandement en fonction du nombre de thread utilisé. Cependant, si nous regardons les variations, nous pouvons remarquer que celles-si sont très faibles (seulement de 20 millisecondes).

Variation du nombre de slice :

Pour ces dernières mesures, nous avons fait varier le nombre de slice dans lesquelles les données sont été réparties (de 1 à 5 000).

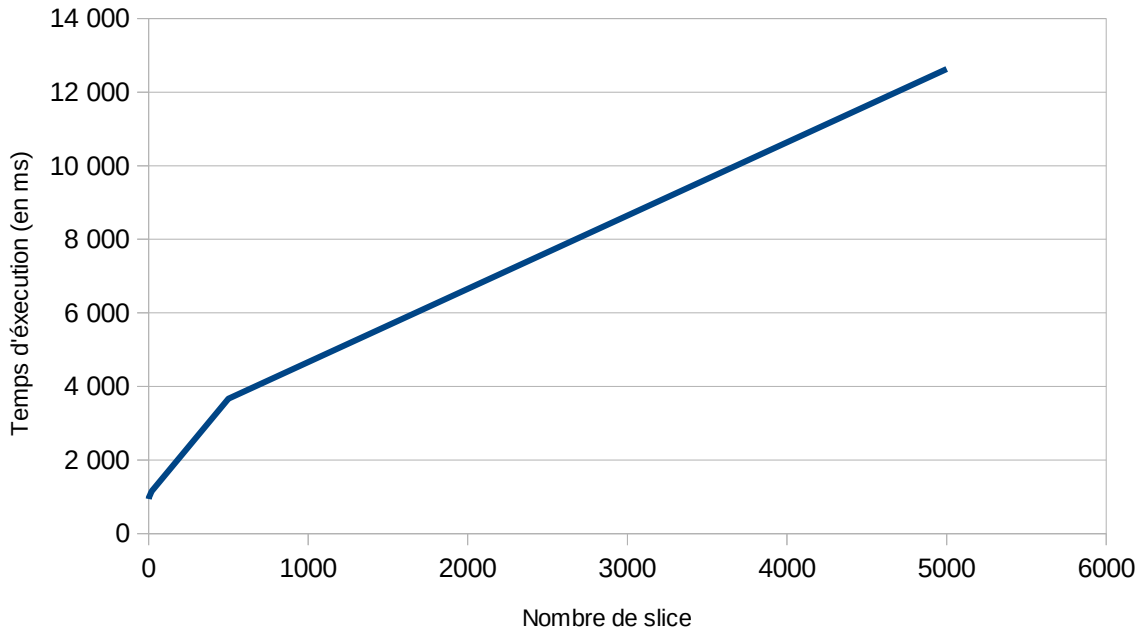

Illustration 3: Temps d'exécution du programme en fonction du nombre de slice

Slices	Temps moyen (ms)	Temps 1	Temps 2	Temps 3	Temps 4	Temps 5	Temps 6	Temps 7
1	941	932	951	940	941	935	947	942
2	981	959	998	991	977	1 016	971	958
3	993	994	984	975	971	996	992	1 040
4	998	991	1 002	1 001	995	997	992	1 010
10	1 062	1 059	1 062	1 053	1 113	1 051	1 044	1 053
15	1 102	1 086	1 115	1 103	1 106	1 111	1 095	1 097
20	1 159	1 176	1 162	1 152	1 154	1 155	1 158	1 155
500	3 666	3 602	3 558	3 669	3 619	3 737	3 693	3 782
5000	12 630	12 710	12 835	12 238	12 231	13 224	13 142	12 030

Sur le schéma présent ci-dessus, nous pouvons observer une augmentation significative du temps d'exécution lors de l'ajout de slices. En effet, plus les données sont divisées, plus le programme devra passer du temps à les partager et les synchroniser.

Digramme

Après rédaction d'un programme Spark afin de rechercher les digrammes dans un texte (ici le tome 1 des *Misérables*) et de calculer leurs nombres d'apparitions, nous avons réalisé une série de mesures en faisant varier le nombre de thread puis le nombre de slice. Ceci a été fait dans le but de voir l'effet de chacun de ces paramètres sur les performances du programme.

Les paramètres par défaut sont 4 threads et 2 slices.

Seul le temps de construction des digrammes et leur comptage est prit en compte (omission de l'initialisation et de l'écriture du résultat dans un fichier).

Variation du nombre de thread :

Nous avons ici fait varier le nombre de thread de 1 à 500 000.

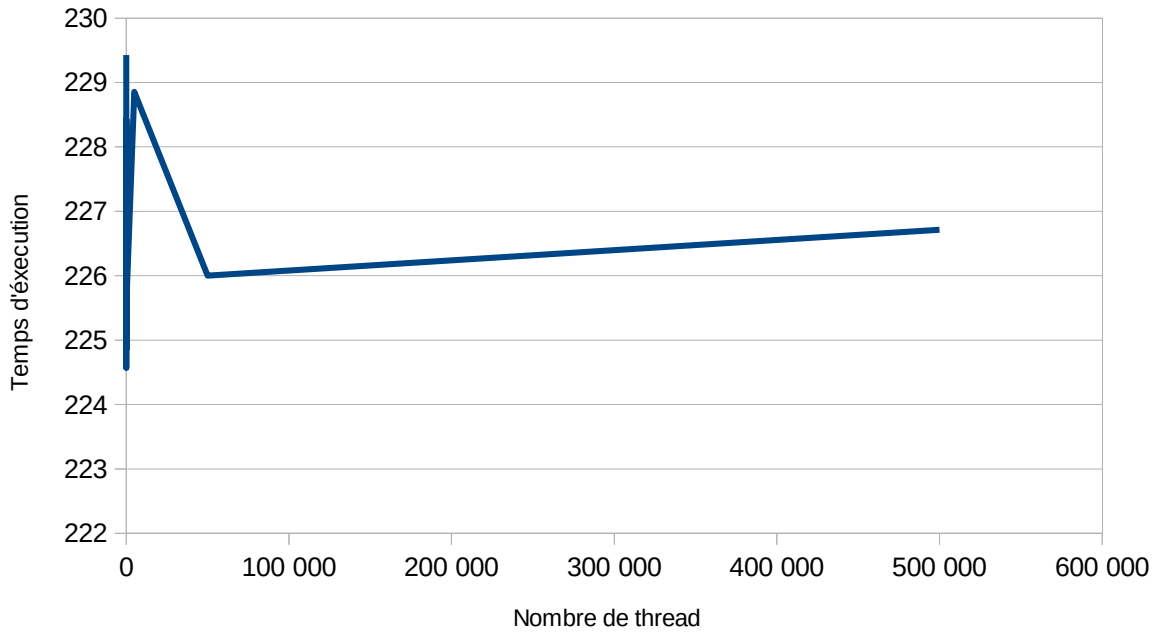

Illustration 4: Temps d'exécution du programme en fonction du nombre de thread

Cette courbe nous donne l'impression que le temps d'exécution varie grandement en fonction du nombre de thread (et ce de façon non attendu). Mais si l'on regarde les chiffres, nous pouvons remarquer que ces variations sont en faite minimes (de 225 à 229 ms).

Nombre de thread	Temps moyen (ms)	Temps 1	Temps 2	Temps 3	Temps 4	Temps 5	Temps 6	Temps 7
1	229	232	226	239	226	223	237	223
4	225	225	235	223	222	224	221	222
10	228	233	239	227	222	229	226	223
50	225	226	229	222	223	222	223	231
100	225	223	224	225	222	229	227	224
500	226	221	225	222	231	221	240	221
5 000	229	226	228	232	230	241	222	223
50 000	226	224	229	224	220	231	229	225
500 000	227	225	241	224	226	223	224	224

Variation du nombre de slice :

Pour ces dernières mesures, nous avons fait varier le nombre de slice dans lesquelles les données sont réparties (de 1 à 50 000 000).

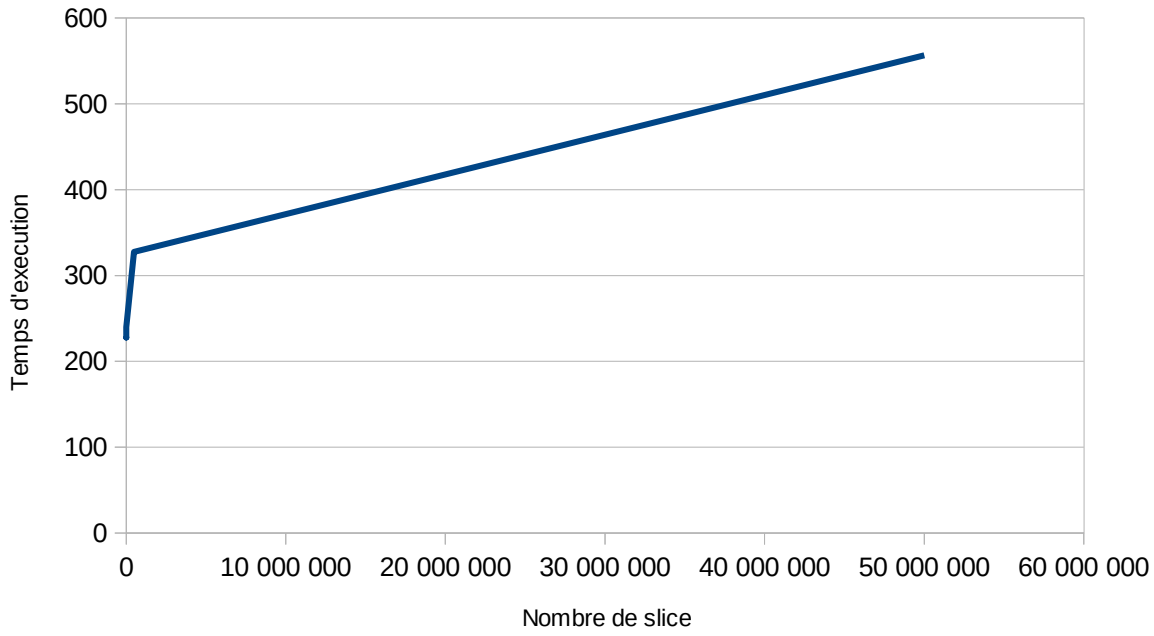

Illustration 5: Temps d'exécution du programme en fonction du nombre de slice

Nombre de slice	Temps moyen (ms)	Temps 1	Temps 2	Temps 3	Temps 4	Temps 5	Temps 6	Temps 7
1	229	228	223	240	224	225	225	235
2	228	225	223	237	222	237	225	226
500	231	230	239	227	238	230	223	228
5 000	240	236	254	237	236	237	238	240
500 000	328	325	325	327	339	327	328	322
50 000 000	557	554	552	558	553	567	555	557

Sur le schéma présent ci-dessus, nous pouvons observer une augmentation significative du temps d'exécution lors de l'ajout d'un nombre important de slices. Ce résultat concorde avec les résultats obtenus avec l'intégrale. En effet, cette augmentation et probablement du à la répartition et la synchronisation des données.

Partie 2 : Expérimentation sur un cluster de machines

Au cours de cette partie, nous avons du créé une seconde machine virtuelle afin de mettre en place un cluster de machine.

Dans un premier temps, nous avons donc du lancer une machine en tant que master (sur l'adresse spark://ESIR-ABD-Bleas-Dubois:7077) et la seconde en tant que slave. Il y a deux façon de voir si la connexion a été établie :

- Du côté du slave :

```
16/01/26 10:10:59 INFO Worker: Connecting to master ESIR-ABD-Bleas-Dubois:7077...
16/01/26 10:10:59 INFO Worker: Successfully registered with master spark://ESIR-ABD-Bleas-Dubois:7077
```

- Du coté du maître, en regardant sa page internet de gestion on peut trouver le tableau suivant :

Worker ID	Address	State	Cores	Memory
worker-20160126101059-148.60.11.118-33607	148.60.11.118:33607	ALIVE	1 (0 Used)	2.9 GB (0.0 B Used)

La machine slave effectuera donc les calculs lancés sur le maître lorsque nous le lui demanderons en modifiant la commande d'exécution comme tel :

```
../spark-1.6.0-bin-hadoop2.6/bin/spark-submit
--class "SimpleApp"
--master local[4]
target/scala-2.10/simple-project_2.10-1.0.jar
```

```
../spark-1.6.0-bin-hadoop2.6/bin/spark-submit
--class "SimpleApp"
--master spark://ESIR-ABD-Bleas-Dubois:7077
target/scala-2.10/simple-project_2.10-1.0.jar
```

Mesure du temps d'exécution :

Nous avons utilisé ce cluster et lancé notre programme de recherche et de comptage de digrammes sur le texte des Misérables (comme précédemment). Voici un tableau des temps d'exécution obtenus :

	Temps moyen (ms)	Temps 1	Temps 2	Temps 3	Temps 4	Temps 5	Temps 6	Temps 7
1 machine	228	225	223	237	222	237	225	226
1 master + 1 slave	322	313	322	322	318	332	318	327

Ces résultats n'étaient pas ceux attendu. En effet, nous pourrions penser que la programme s'exécuterait plus vite dans la seconde condition (un maître et un esclave). Mais, une explication à ces résultats vient du fait que le maître ne fait qu'envoyer les ordres à l'esclave et recevoir les résultats de ce dernier. En définitive, le temps de calcul est le même (même configuration pour les deux machines). Mais il faut y ajouter le temps d'envoi et de réception de données.

Pour obtenir un meilleur temps exécution, il faudrait mettre en place une troisième machine qui ferait office de second slave. Dans ce cas de figure, on peut penser que le temps d'exécution diminuerait bien qu'il y aurait (là aussi) un temps d'envoi et de réception des données.