Rapport de Travail d'Etude et de Recherche

Utilisation de Spark pour Map-Reduce sur des objets combinatoires

Thomas Foltête Adrien Pavão

M1 Informatique
Université de Paris-Sud

Avril 2017

Sommaire

Introduction

Framework Spark :
Haut niveau.
Jusqu'à 100 (10) fois plus rapide qu'Hadoop en mémoire (sur disque).
Utilisation classique sur des données stockées.

Combinatoire :
Étudie les collections d'objets, leurs combinaisons et leur dénombrement.
L'informatique est adapté à ce genre de calcul.

Distribution du calcul pour qu'il soit exécuté simultanément (en parallèle) par plusieurs agents.

- Multi threads (mono machine)
- Multi processeurs (mono machine)
- Multi machines

Ensemble récursivement énumérable :

- Une racine (élément initial)
- Une fonction successeur

Numerical semi-group :

- Sous-ensemble de $\mathbb{N}$ contenant 0
- Stable par addition
- Complémentaire fini

Propriétés d'un semi-group :

- Multiplicité
- Genus
- Frobenius
- Conducteur

$$S = \{0, \mathbf{3}, 6, \mathbf{7}, 9, 10\} \cup [12, +\infty[= <3, 7>$$

Le sous-ensemble <3, 7> est l'ensemble générateur.

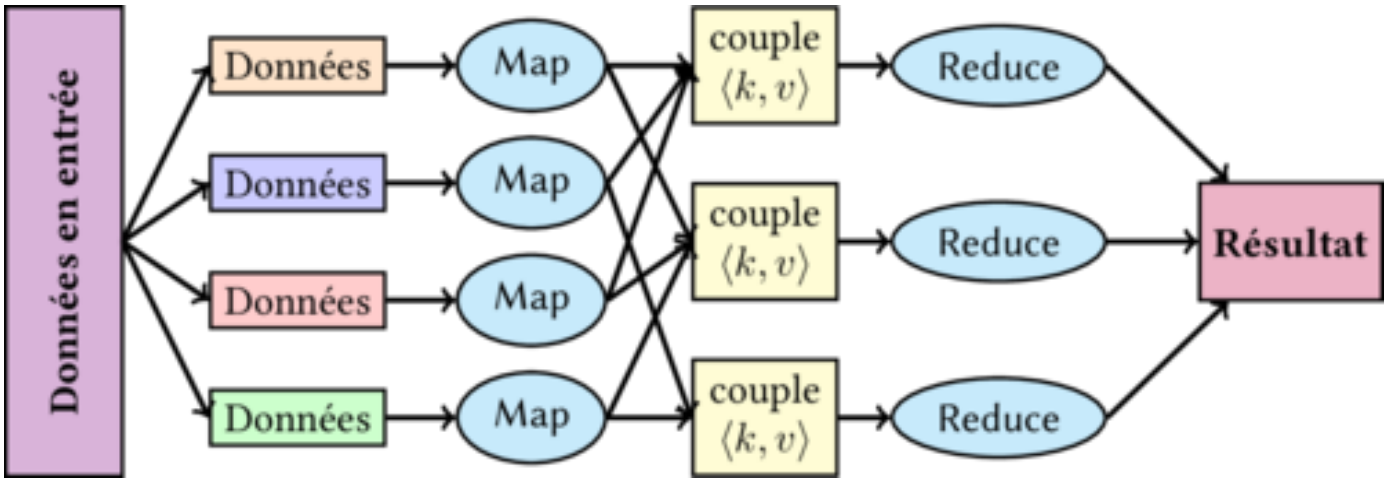
Son ensemble complémentaire est {1, 2, 4, 5, 8, 11}

Très utilisé en calcul distribué

Se base sur deux fonctions :

- **Map :** Application d'une fonction à tous les éléments d'une collection.
- **Reduce :** Réduction de tous les éléments à l'aide d'un opérateur associatif.

Figure – Schéma explicatif du MapReduce



Avantage de Spark :

- **Évaluation paresseuse :** L'évaluation réelle n'est faite que si elle est nécessaire. Résultats non préservés par défaut.
- **Aplatissement en mémoire :** Fixe les données en mémoire en les évaluant.

Spark se base sur les RDDs. Ils sont tolérants à la panne et ont une structure parallèle. Ils permettent à l'utilisateur de :

- Garder les données intermédiaires explicitement en mémoire
- Contrôler leur partitionnement afin d'optimiser l'emplacement des données
- Manipuler les données en utilisant un ensemble important d'opérateurs

Création d'un RDD à partir d'une liste Python :

```
rdd = sc.parallelize([1,2,3,4,5])
```

Il y a deux différents types de méthodes qu'on peux appliquer aux RDDs :

- Les **transformations** : Union, Map, etc.
- Les **actions** : First, Reduce, etc.

Ensembles des mots d'un alphabet : Produit cartésien d'un alphabet A avec lui-même.
Implémenté en Python plutôt qu'en Sage.

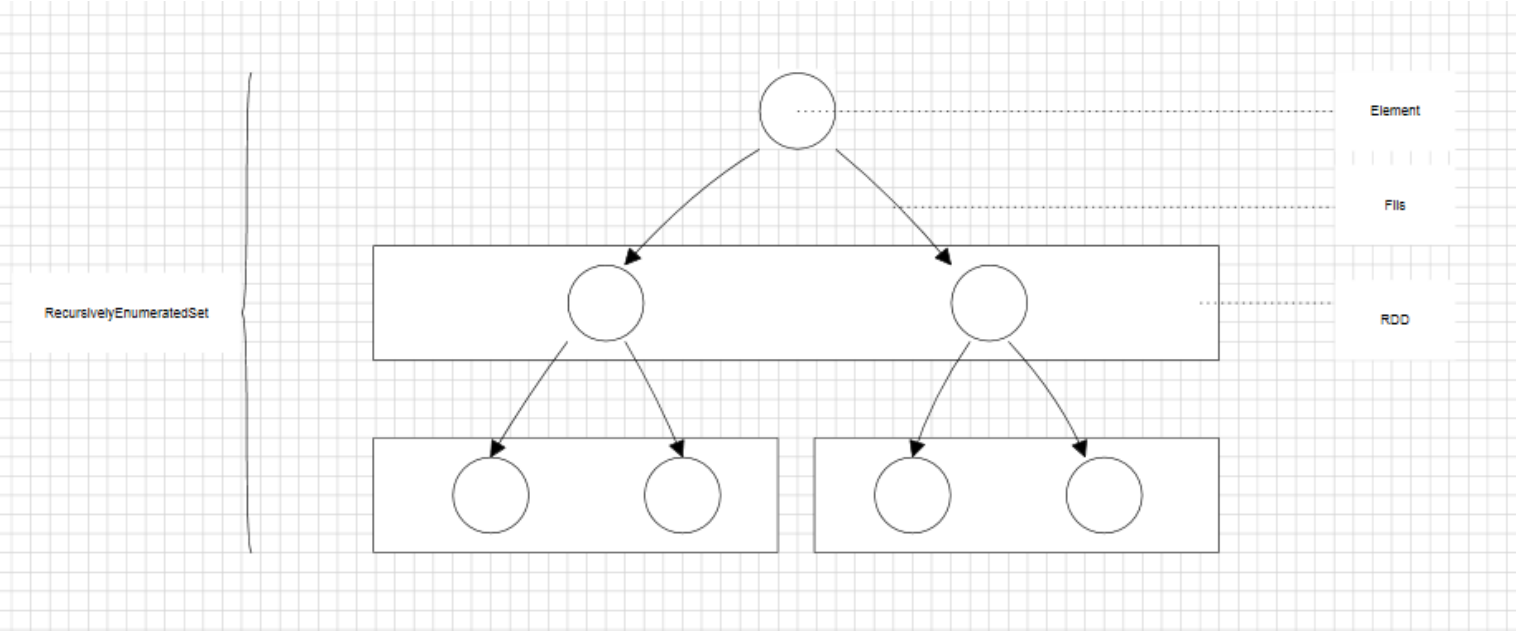
Ensemble des arbres binaires à n nœuds :
Cet ensemble se modélise ainsi :

$$\begin{cases} T_0 = Leaf \\ T_n = \cup_i T_i \times Node \times T_{n-i-1} \end{cases} \quad (1)$$

On peut ensuite par exemple compter les arbres ou compter les arbres par profondeur. Ce modèle peut également permettre de manipuler des polynômes.

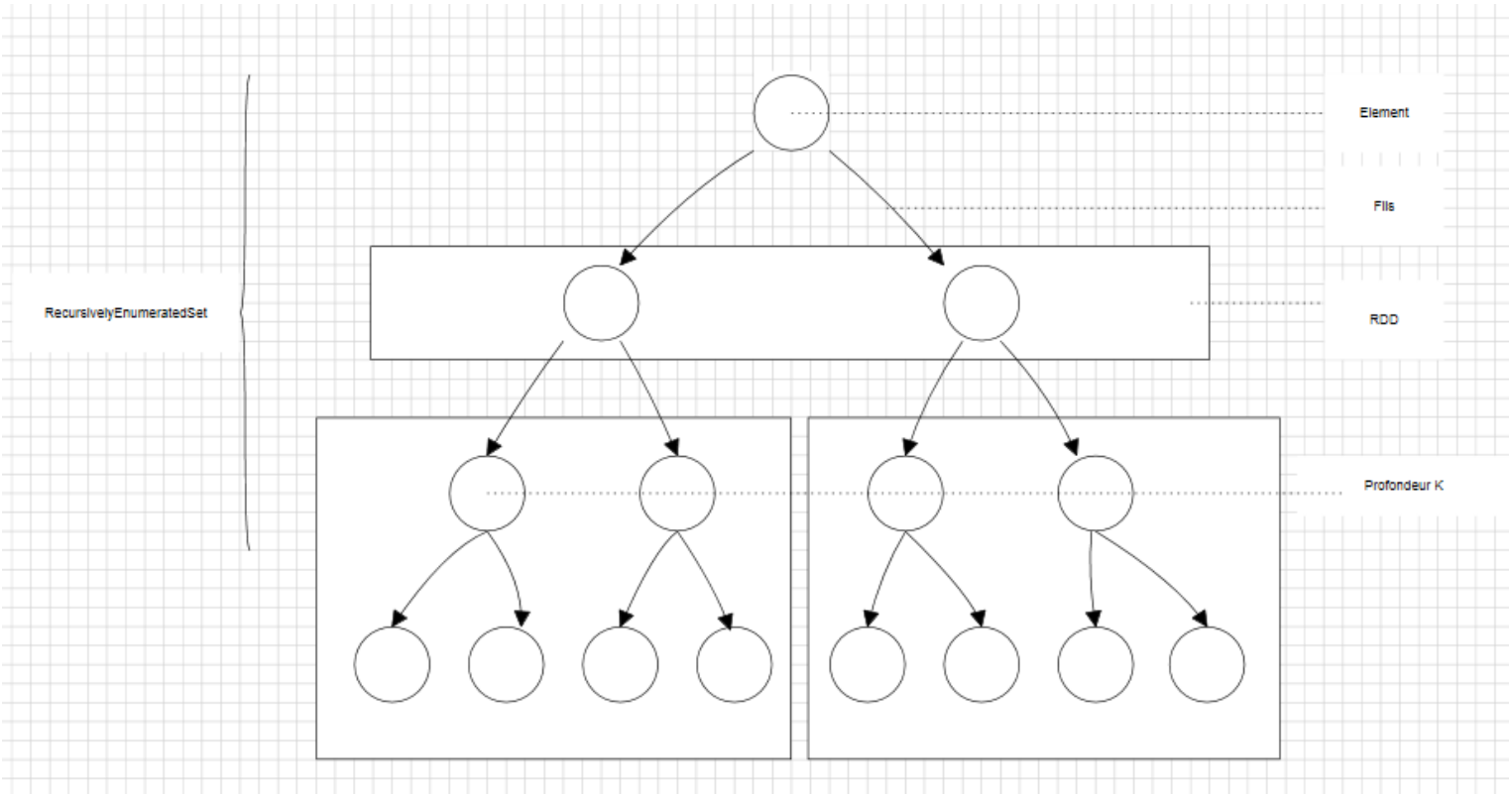
Approche A

Figure – Schéma explicatif de l'approche A



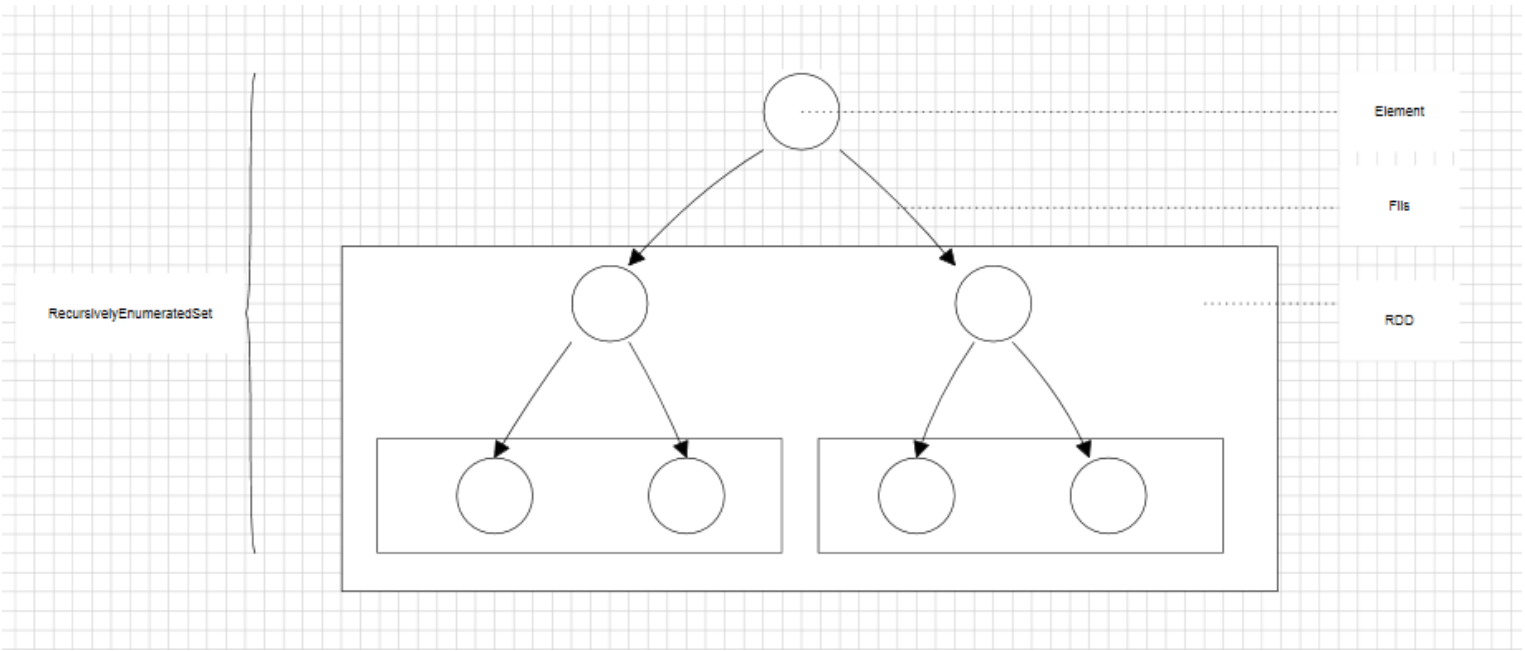
Approche B

Figure – Schéma explicatif de l'approche B



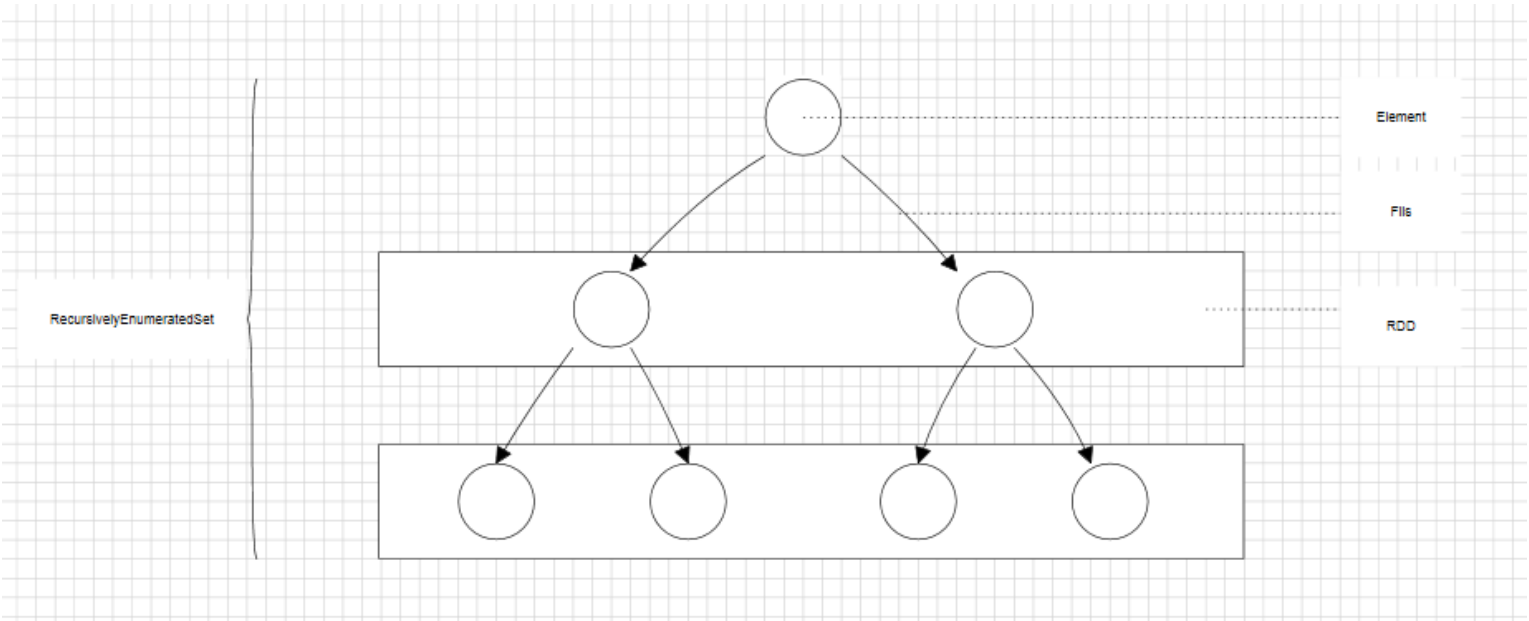
Approche C

Figure – Schéma explicatif de l'approche C



Approche D

Figure – Schéma explicatif de l'approche D



On observe :

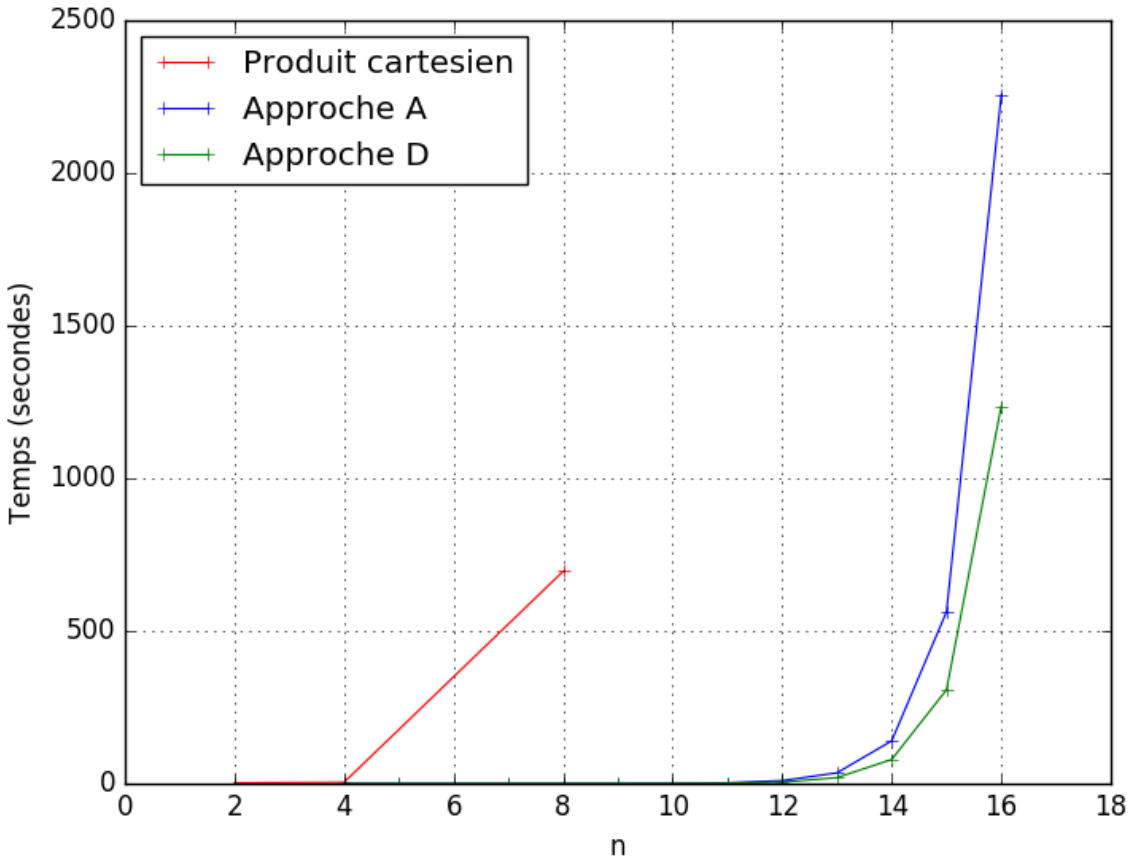
- Le temps de calcul : On souhaite le minimiser
- L'utilisation mémoire : Savoir si tout est développé en mémoire
- La distribution du calcul et le partitionnement des RDDs

Résultats et optimisations

Protocole de test

- Dénombrement des mots de taille n sur l'alphabet A = {a, b, c, d}
- Machine : 2 processeurs à 2.40GHz et 8 Go de RAM
- Count avec un Map-Reduce
- Mesure du temps dans le code Python avec *time.time()*
- Observation des processus et de la mémoire avec *top*
- Nombre de partitions du RDD en temps réel avec la méthode *getNumPartitions()*

Figure – Temps d'exécution en fonction de n



Conjecture pour ces résultats décevants :

- Mauvaise parallélisation due à notre implémentation Python, ou aux Strings.
- Certains RDDs sont réévalués plusieurs fois.
- Création récursive de petits RDDs qui prend beaucoup de temps.

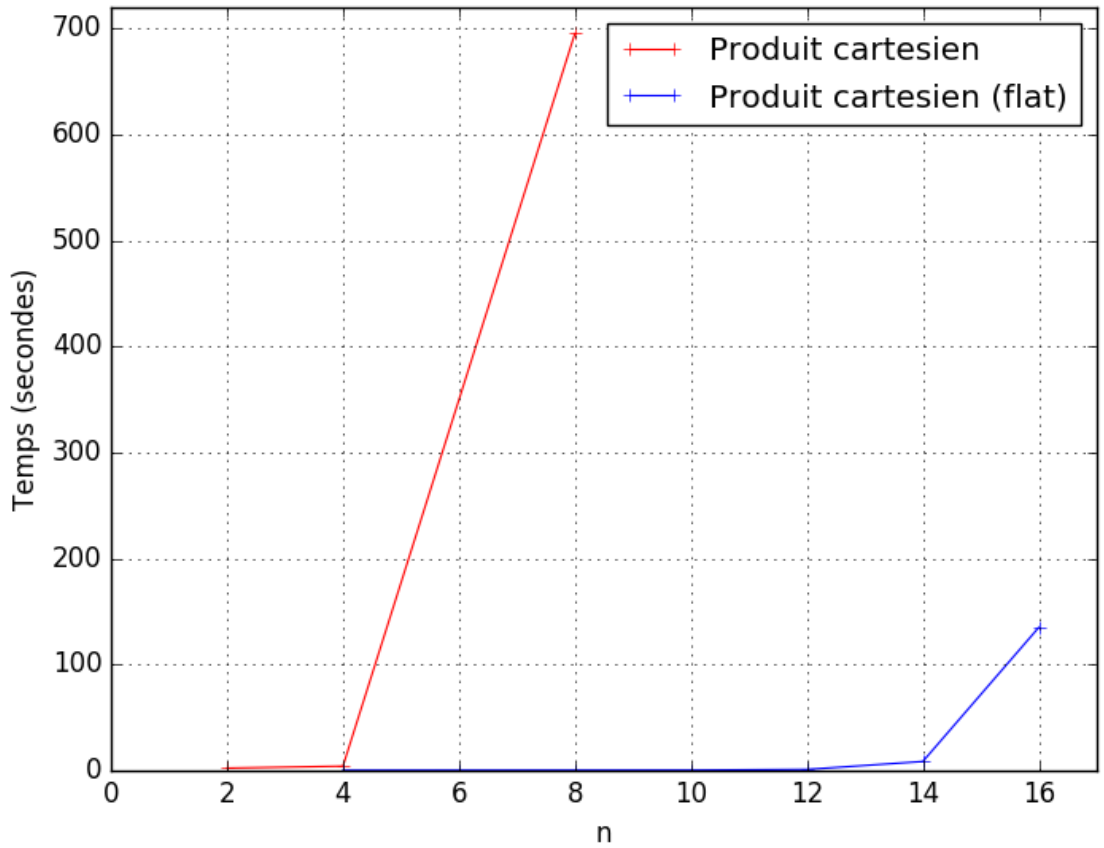
Idées de solution :

- Faire du profiling.
- Aplatir le RDD en mémoire à un certain niveau. L'aplatissement semble permettre un grand grain de performance.
- Faire un algorithme hybride.

On essai de collecter le RDD dans sa totalité.
Saturation de la mémoire pour $n > 12$.

```
n = 12
count = 16777216
Temps d'execution: 4.88712310791 secondes.
```

Figure – Schéma comparatif du produit cartésien avec son homologue avec aplatissement mémoire



Commande top :

```
PID  USER      [..]  CPU   MEM   TIME+ COMMAND
29700 ubuntu   [..]  99.7  0.3   1:07.20 python
29701 ubuntu   [..]  99.7  0.3   1:06.79 python
29705 ubuntu   [..]  99.7  0.3   1:06.62 python
29709 ubuntu   [..]  99.7  0.3   1:06.78 python
```

Il s'agit pour le moment du meilleur temps que nous ayons obtenu :

```
n = 16
count = 4294967296
Temps d'execution: 129.296506166 secondes.
```

Figure – Schéma comparatif de l'approche D et de sa version améliorée avec aplatissement en briques de taille 1024

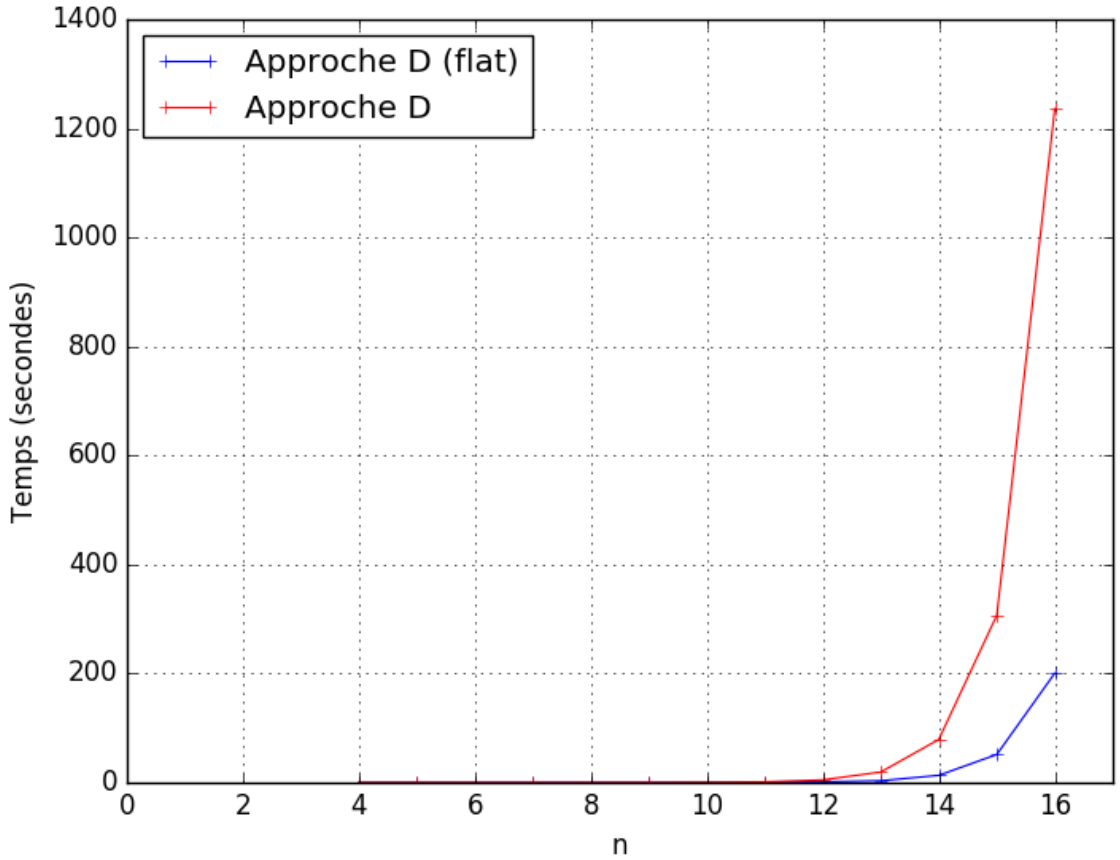
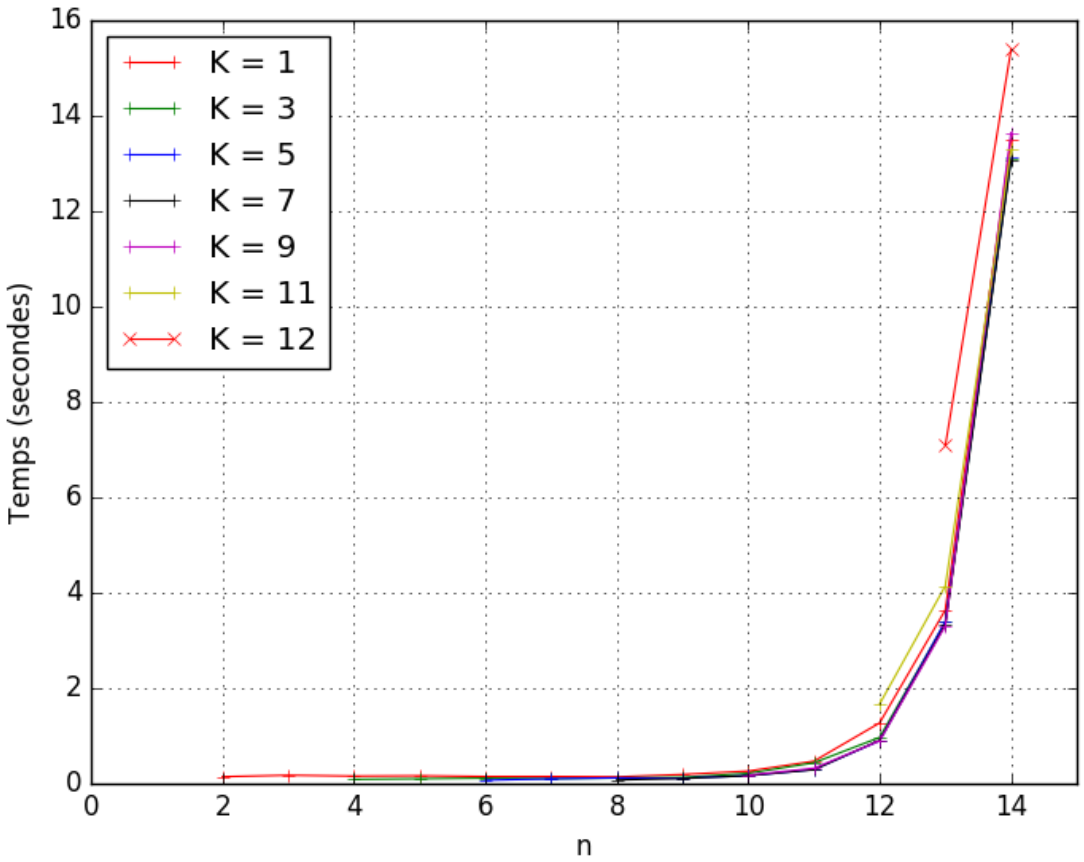


Figure – Temps d'exécution en fonction de n pour différents K



Un arbre déséquilibré en largeur :

```
rdd = sc.parallelize(range(1,5))

for i in range(n):
    rdd = rdd.flatMap( \\
    lambda x : range(1,4*x+1))
```

Les processus s'arrêtent chacun leur tour.

Exemple d'utilisation mémoire pendant un calcul avec Spark :

PID	USER	[...]	CPU	MEM	TIME+	COMMAND
8214	ubuntu	[...]	71.1	0.3	0:02.14	python
18122	ubuntu	[...]	43.2	3.9	0:12.74	java
18207	ubuntu	[...]	12.6	0.3	0:00.38	python
18210	ubuntu	[...]	4.3	0.3	0:00.13	python
18211	ubuntu	[...]	4.3	0.3	0:00.13	python
18217	ubuntu	[...]	3.3	0.3	0:00.10	python

Conclusion

- Possibilité de calcul combinatoire
- Un peu lent mais plus rapide en Scala
- Peut-être pas de vol de tâches prévu par Spark
- Les structures de données influent grandement les performances et la distribution du calcul

Notre étude de Spark ne répond pas forcément à toutes les questions qui l'entourent, cependant ce projet nous aura été très pédagogique.