Rapport de Travail d'Etude et de Recherche

Utilisation de Spark pour Map-Reduce sur des objets combinatoires

Master 1 Informatique

2016/2017

Fait par :

Adrien Pavão

Thomas Foltête

Enseignants:

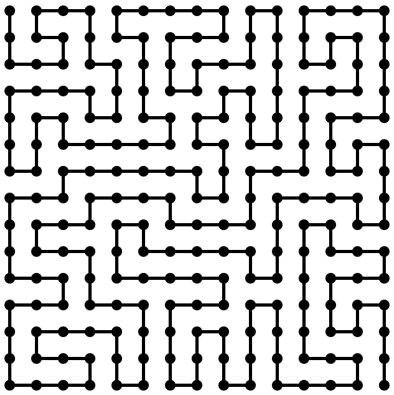
Nicolas Thiery

Contents

1 Introduction

L'objectif principal de ce travail consiste à prendre en main la technologie *Spark*[1] afin de connaître ses possibilités d'utilisation en calcul combinatoire. Spark est un framework permettant la distribution de calculs, développé par Apache. L'utilisateur ne se soucie pas de la parallélisation puisqu'il code en haut niveau. Il est décrit comme très efficace[2].
L'utilisation classique de Spark se fait sur grande quantité de données stockée. Nous voulons cependant essayer une utilisation en *combinatoire*: sur des données générées à la volée. C'est le défi principal de notre projet.
La combinatoire est la branche des mathématiques qui étudie les collections d'objets, leurs combinaisons et leur dénombrement.

Figure 1: Exemple de problème combinatoire: Combien y a-t-il de chemins sans contact distincts dans une grille ?



L'arrivée des ordinateurs qui met en game, possibilité de compter salement.
Ce TER individualisé est un projet d'introduction à la recherche. Les contours du sujet sont assez flous et nous offre de la liberté. Nous espérons que notre contribution à la découverte de Spark pourra être utile aux chercheurs de l'équipe GALaC. Quoiqu'il en soit, ce travail aura été très pédagogique pour nous avec la découverte de notions mathématiques, la mise en pratique, etc. Nous étions encadré par Nicolas Thiery, avec qui nous avions régulièrement rendez-vous au Laboratoire de Recherche en Informatique (LRI) pour faire le point et avoir de nouvelles pistes de réflexion.

2 Révisions préliminaires

2.1 Outils utilisés

Tout d'abord, avant d'attaquer le travail, il était important de revoir l'utilisation de certains outils que nous allions utiliser.

- **Sage**: Sage[3] est un logiciel de mathématiques open-source basé sur différentes bibliothèques python et autres logiciels mathématiques. Racontage de life.

  Concernant l'installation, nous avons téléchargé les fichiers sources de Sage, puis nous l'avons compilé. La compilation a duré 6 heures, 51 minutes et 19 secondes.

[1] https://spark.apache.org/
[2] https://databricks.com/spark/about
[3] http://sagemath.org/

Nous avons commencé par une prise en main en testant différents Recursively Enumerated Set, et en appliquant un map-reduce dessus. Prise en main. Voir fichier set.sage.

- **Python**: La quasi-totalité du code que nous avons écrit durant ce TER était en Python. Spark prend en charge le Scala (son langage natif), le Python et le Java. SageMath utilise également python, ainsi que sur les fonctionnalités d'autres logiciels mathématiques. Prise en main de Notebook, très utile.
- **Git**: Mise en commun du travail avec GitHub sur le dépôt Didayolo/spark.
- **Cloud du LAL**: Le Laboratoire de l'Accélérateur Linéaire a mis à notre disposition une machine de leur cluster d'ordinateurs. Possibilités de lancer les calculs en SSH. Script shell de configuration setup_virtual_machine fait au fur et à mesure, il ne reste maintenant qu'à exécuter ce script pour les prochains utilisateurs de sage et spark. Si les méthodes s'avèrent efficaces, on veux pouvoir les reproduire sur d'autres machines, d'autres clusters afin de lancer un très gros calcul (nombre de semi-groupes en fonction du genus).

2.2 Notions mathématiques

Dans cette sous-section, nous détaillerons certaines notions de mathématiques et d'informatique vues au cours du projet.

2.2.1 Permutations

Un problème de base en combinatoire consiste à énumérer les *permutations* d'un groupement d'objets. Il s'agit de les arranger selon tous les ordres possibles.

Voici un exemple de fonction récursive en python qui pour tout entier n renvoi toutes les permutations possibles de ses chiffres:

```
def permutation(n):
        """ Engendre toutes les permutations de n """
        strn = str(n)

        if len(strn)==1:
                return [strn]

        result = []
        for i, v in enumerate(strn):
                for p in permutation(strn[:i] + strn[i+1:]):
                        result.append(v + p)

        return result
```

Dont voici une exécution:

```
>>> permutation(2345)
['2345', '2354', '2435', '2453', '2534', '2543',\\
'3245', '3254', '3425', '3452', '3524', '3542',\\
'4235', '4253', '4325', '4352', '4523', '4532',\\
'5234', '5243', '5324', '5342', '5423', '5432']
```

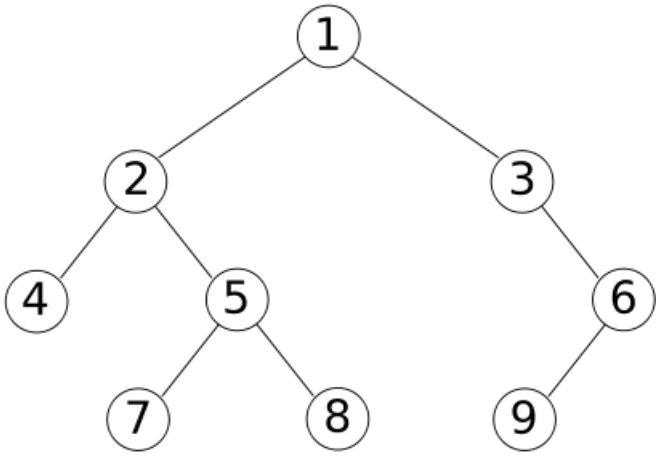
Un autre exemple: tous les mots de longueur n d'un alphabet. Voir fichier code_python/permutation.py

2.2.2 Arbres et arbres binaires

Il est courant de structurer les données sous formes d'*arbres*. Qu'est-ce qu'un arbre en mathématiques ?

Arbre de tous les arbres binaires...

Figure 2: Un exemple simple d'arbre binaire[4]



2.2.3 Recursively Enumerated Set

Un *ensemble énuméré récursivement* (recursively enumerated set en anglais) est un ensemble dont un ordinateur peut enumérer les éléments (éventuellement au bout d'un temps infini).

Le principe est simple. On a une fonction successeur qui pour chaque élément donne son ou ses élément(s) suivants (fils). On définit un élément initial et, par récurrence, on peux donc obtenir l'ensemble entier. Si l'ensemble ainsi généré est infini, il s'agit donc d'un ensemble semi-décidable: Un élément existant sera toujours trouvé en un temps fini tandis qu'il ne sera pas possible de prouver l'absence d'un élément à l'aide d'un parcours (seulement en le prouvant à l'aide de la définition de l'ensemble). (a rediger)

La base, génération de grosse quantité de données par énumération.

Seeds + fonction successeur

Différentes topologies possibles (arbres, listes ?)

Sage et implémentation python (citer exemple ?) ref Rapport (voir readme)[2]

Voir fichiers code_sage/set.sage et code_sage/semigroup.sage.

2.2.4 Monoïdes numériques

Les explications et les algorithmes liés au monoïdes numériques nous proviennent de l'article "Exploring the tree of numerical semi-groups", écrit par Jean Fromentin et Florent Hivert.[1]

Un *monoïde numérique* (numerical semi-group en anglais) est un sous-ensemble de $\mathbb{N}$ contenant 0, stable par addition et dont le complémentaire est fini. Il s'agit donc d'un ensemble infini d'entier. La propriété 'stable par addition' signifie que la somme de deux éléments quelconques de l'ensemble donne un élément de l'ensemble.

Soit S un semi-group, on définit les valeurs suivantes:

- **Multiplicité**: Le plus petit élément de S mis à part 0.

[4] https://fr.wikipedia.org/wiki/Arbre_binaire

- **Genus**: Le nombre d'entiers ne faisant pas partis de S, c'est-à-dire appartenant à l'ensemble complémentaire de S.
- **Frobenius**: Le plus grand nombre ne faisant pas parti de S.
- **Conducteur**: Le frobenius + 1, dont on sait qu'il fait parti de S.

L'arbre de tous les semi-groups est très irrégulier (déséquilibré?). Cette propriété le rend intéressant à étudier pour concevoir des algorithmes reparties sur des arbres. En effet, pour obtenir des bonnes performances il est important que l'algorithme répartisse bien les tâches et ne se contente pas de donner les différentes branches de l'arbre aux différentes unités de calcul.

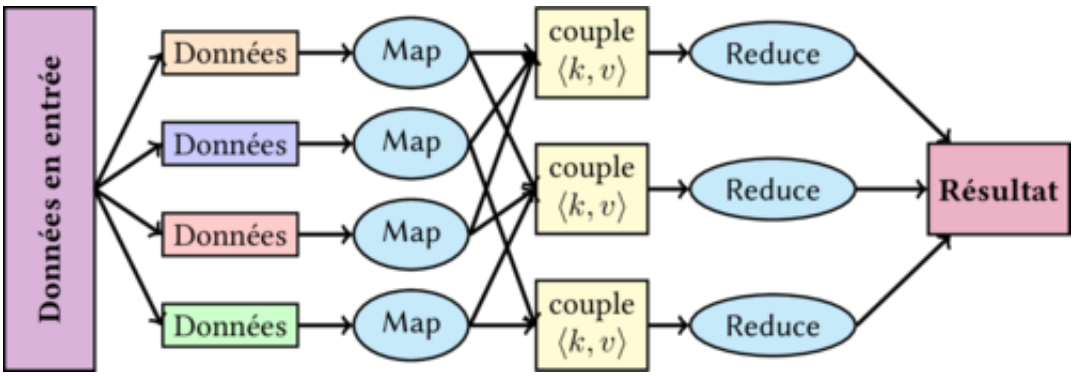
Implémentation python...

2.2.5 Map-Reduce

Nous allons maintenant aborder un point très important de notre projet. Le *Map-Reduce* est très utilisé en calcul distribué. C'est est un modèle de programmation qui se base sur deux fonctions: Map et Reduce:

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

Figure 3: Schéma explicatif du MapReduce[5]



Prenons comme exemple de Map-Reduce le dénombrement des éléments d'un RecursivelyEnumeratedSet en Sage:

```
def count(rec_set):
    """ Renvoi la taille d'un RecursivelyEnumeratedSet """
    map_function = lambda x: 1                                          #(1)
    reduce_function = lambda x,y: x+y                                   #(2)
    reduce_init = 0                                                     #(3)
    return rec_set.map_reduce(map_function, reduce_function, reduce_init) #(4)
```

1. Map prend un élément en entrée et renvoi 1: Chaque élément compte pour 1.
2. On additionne les nombre d'éléments entre eux (opération associative).
3. On commence a compter a partir de 0.
4. Le tant attendu Map-Reduce.

[5] https://fr.wikipedia.org/wiki/MapReduce

2.2.6 Parallélisation

La *parallélisation* du calcul est une notion clé de notre projet. L'idée est la suivante: Que faire lorsque qu'un travailleur ne travaille pas assez vite sur la tâche que l'on attend de lui ? On multiplie le nombre de travailleurs afin qu'ils effectuent des tâches simultanément. Le gain de productivité peut être très grand. En informatique, on distingue trois types de parallélisation:

- **Multi threads** (mono machine): Lancements de processus fils afin de paralléliser les tâches.
- **Multi processus** (mono machine): Plusieurs coeurs d'un processeur (ou encore plusieurs processeurs d'une même machine) se répartissent le travail.

  Afin de vérifier qu'un programme est bien parallélisé entre les différents coeurs de notre machine, nous effectuons la commande "top" qui permet de voir les processus actifs, avec ceux qui utilisent le plus de ressources en haut de la liste.

PID	USER	PR	NI	[...]	SHR	S	CPU	MEM	TIME+	COMMAND
8214	ubuntu	20	0	[...]	3764	R	71.1	0.3	0:02.14	python
18122	ubuntu	20	0	[...]	30152	S	43.2	3.9	0:12.74	java
18207	ubuntu	20	0	[...]	7324	S	12.6	0.3	0:00.38	python
18210	ubuntu	20	0	[...]	3812	S	4.3	0.3	0:00.13	python
18211	ubuntu	20	0	[...]	3812	S	4.3	0.3	0:00.13	python
18217	ubuntu	20	0	[...]	3812	S	3.3	0.3	0:00.10	python

  Ici par exemple, nous pouvons voir qu'il y a 5 processus pythons actifs, dont 4 consacrés à l'exécution de notre programme. On voit également qu'un processus java est actif, qui utilise 43% du CPU : cela s'explique par le fait que Spark fonctionne sur la JVM. En effet, étant écrit en Scala, il fonctionne sur la JVM.

- **Multi machines**: Plusieurs ordinateurs se répartissent le travail.

  Pas de mémoire partagée, il faut diviser l'information. Malgré cela, sur un arbre qui est très déséquilibré, il semble impossible de savoir comment répartir le travail équitablement entre toutes les machines. Il faut donc également un algorithme de vol de tâche : lorsqu'une machine n'a plus de travail, elle en récupère sur une machine qui a du travail en attente, sans aucune "permission" [6].

3 Spark

Les notions importantes étant tirées au clair, nous pouvons maintenant aborder de plus près ce fameux framework. On rappelle que l'objectif de Spark est d'écrire des calculs parallèles en utilisant des opérateurs de haut niveau, sans avoir à se soucier de la distribution des unités de calcul ni de la tolérance aux pannes.

3.1 Installation et mise en place de Spark sur le cluster de machines

Nous avons eu quelques difficultés pour notre première installation de Spark : malgré une installation qui semblait fonctionner, aucun des exemples de la documentation de Spark ne fonctionnait. Nous avons résolu ce problème en ajoutant la ligne 127.0.0.1 combispark dans le fichier "/etc/hosts" (ou combispark est le nom de notre machine virtuelle). En effet, Spark étant censé être utilisé sur un ensemble de machine, cet ajout est nécessaire lorsque l'on l'utilise sur une machine seule.

La compilation et l'exécution de spark, pour le python, se font par la commande spark-submit "nom_fichier".py. C'est d'ailleurs pour cela que pour combiner sage et spark, nous devons ajouter la bibliothèque sage dans le fichier python, et pas le contraire (puisque un fichier sage se compile par la commande "sage").

[6] https://en.wikipedia.org/wiki/Work_stealing

3.2 Premiers pas avec Spark

Nous avons rencontré quelques difficultés lorsque nous essayions d'apprendre à manier Spark. Ce qui nous a grandement aidé a été de participer à L'école Spark, organisé par Cecile Germain au LAL. Durant cette formation de deux jours nous avons pu découvrir cette technologie et la tester lors de travaux pratiques. Nous en avons également profité pour poser quelques questions au formateur puisque nous aspirions à une utilisation particulière de Spark: le calcul sur des données générées à la volée. Penchons-nous d'un peu plus près sur le sujet.

3.2.1 Resilient Distributed Datasets

Spark se base sur une nouvelle structure de données: les *Resilient Distributed Datasets* (RDDs). Les RDDs permettent de réutiliser efficacement les données dans une large famille d'applications a été proposée. Les RDDs sont tolérants à la panne et proposent des structures de données parallèles qui laissent les utilisateurs :

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

3.2.2 Opérations sur les RDD

Spark dispose d'un éventail d'opérations applicables aux RDDs. On y retrouve la plupart des opérations sur des bases de données.

- **Produit cartésien**: L'ensemble de tous les couples dont la première composante appartient au premier RDD et la seconde au second RDD.
- **Union**: Renvoi un nouveau RDD contenant tous les éléments des deux RDDs. L'union ne supprime pas les doublons ainsi créés.
- **Join**: Jointure (base de données).
- **Map et Reduce**: cf. plus haut dans le rapport.
- **FlatMap**: Renvoi un RDD en appliquant une fonction à tous les éléments du RDD d'origine, plus aplati le résultat. Contrairement au Map classique, l'entrée et la sortie ne sont pas nécessairement de la même taille.
- **Filter**: Filtrage des éléments avec une fonction de type 'élément -> booléen'.
- Et bien d'autres qu'il n'est pas nécessaire de citer ici.

3.3 Différentes implémentations utilisant des RDDs

Nous sommes ici dans le vif du sujet: l'implémentation pratique d'objets mathématiques avec des RDDs. Nous allons pouvoir savoir ce qui fonctionne ou non, analyser les performances des différentes méthodes (en temps de calcul et en mémoire).

Les implémentations sont faites en python car nous ne sommes pas parvenus à exécuter du code Sage avec Spark (ni du code spark avec sage). En effet sage utilise une version de python légèrement differente de celle classique, et qui n'est donc pas compatible avec l'utilisation de Spark.

3.3.1 Modélisation de l'ensemble des mots sur un alphabet par un RDD

Soit A un alphabet. On modélise A par un RDD, puis A^n par un RDD construit comme le produit cartésien de A avec lui même.

Exemple: Alphabet composé des strings 'a', 'b', 'c' et 'd'. Produit cartésien 16 fois. Tanani.

Lancement de Map-Reduce count, resultat 4 milliards, temps exe et usage mémoire... Spark a ou pas développé en mémoire A^n ?

Voir fichier code_spark/cartesian_product.py

3.3.2 Modélisation de l'ensemble des arbres binaires à n noeuds par un RDD

Cet ensemble se modélise ainsi:

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

Lancer un Map-Reduce. Par exemple, compter les arbres, ou compter les arbres par profondeur (utiliser $p(t) = X^{profondeurdet}$ et sympa pour manipuler des polynomes).

Voir fichier code_spark/tree.py Référence au chapitre combinatoire du livre calcul maths avec Sage (scanner) [3]

3.3.3 Modélisation d'un RecursivelyEnumeratedSet par un RDD

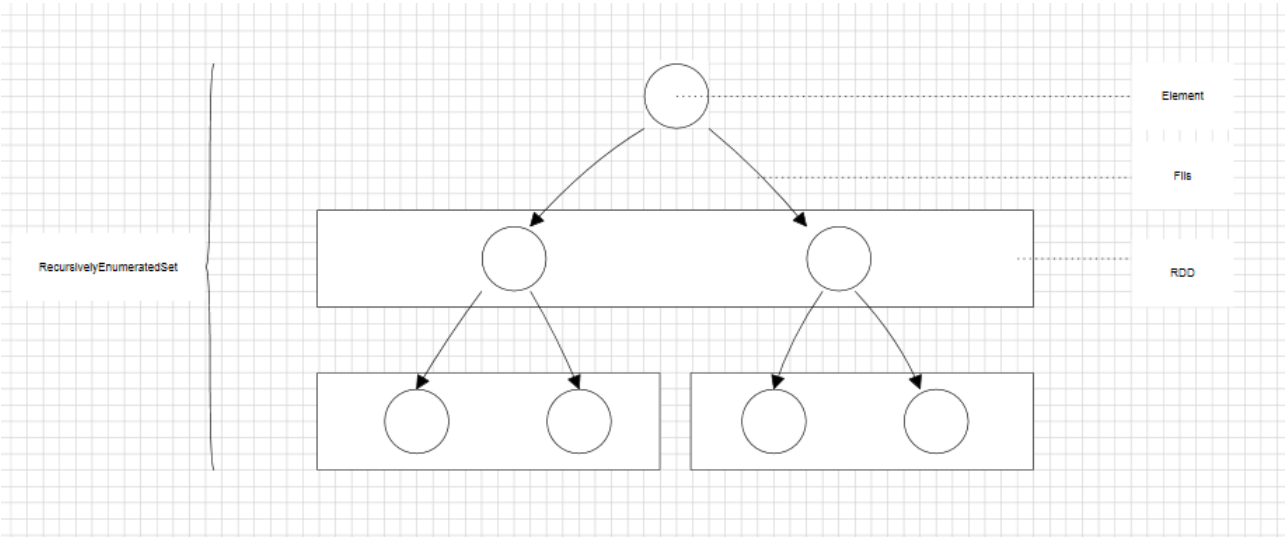
Nous recherchons à présent la meilleure méthode pour modéliser un RecursivelyEnumeratedSet avec Spark.

Soit T un ensemble défini récursivement. Les *descendants* d'un noeud sont tous les noeuds sous lui (fils, petits fils, ...).

Différentes approches testées: (Schéma...)

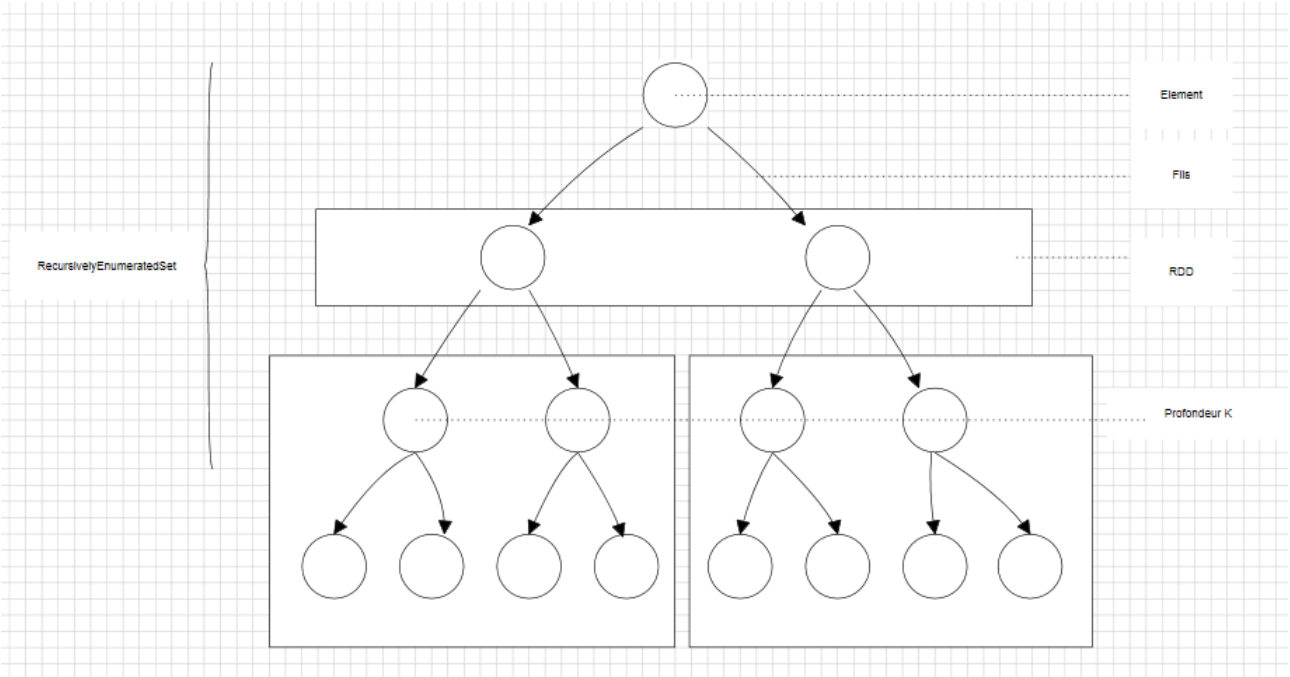
- **Approche A**: Pour chaque noeud de l'arbre, modéliser par un RDD l'ensemble de ses fils. Voir fichier code_spark/rdd_a_b.py

Figure 4: Schéma explicatif de l'approche A



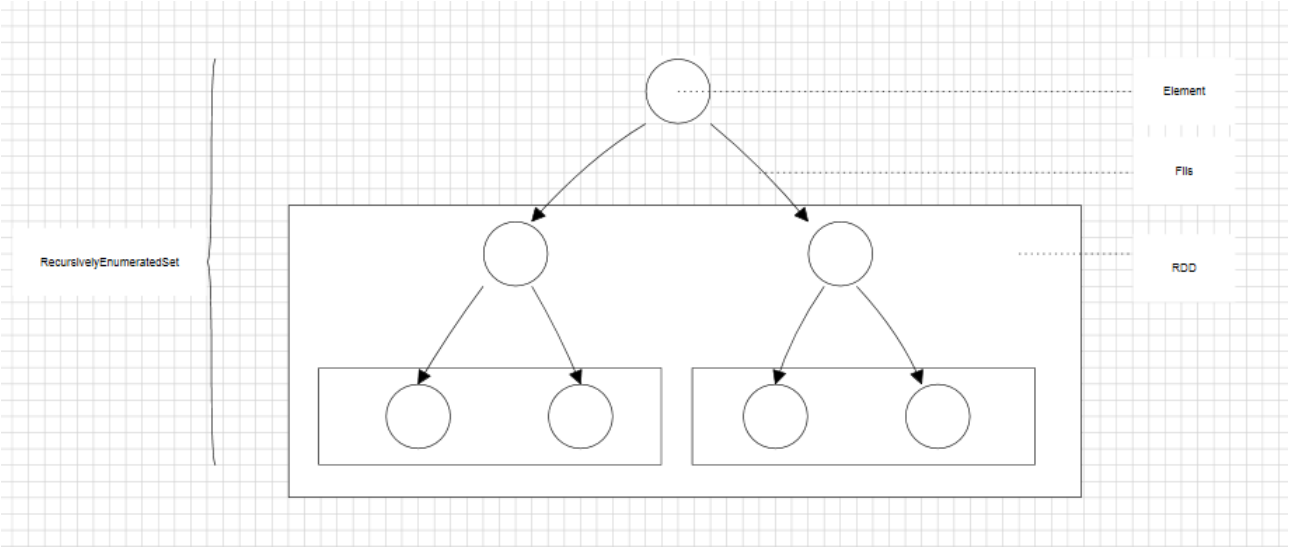
- **Approche B**: Idem, mais on s'arrête à une profondeur k donnée, et ensuite on travaille en local: le RDD d'un noeud à profondeur k contient tous les descendants du noeuds: a priori ne gère pas bien les arbres déséquilibrés "en largeur". Voir fichier code_spark/rdd_a_b.py

Figure 5: Schéma explicatif de l'approche B



- **Approche C**: Pour chaque noeud de l'arbre, modéliser par un RDD l'ensemble de ses descendants.

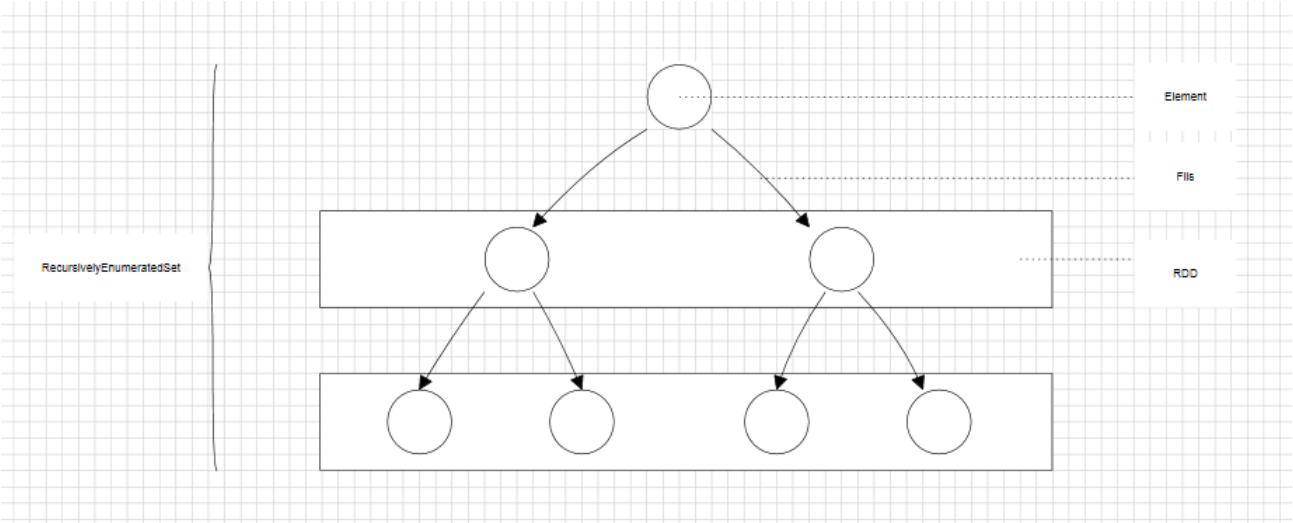
Figure 6: Schéma explicatif de l'approche C



- **Approche D**: Pour chaque profondeur k, modéliser par un RDD l'ensemble des noeuds à profondeur

k. Cette approche semble être la plus naturelle. On l'obtient à l'aide de flatmap. Voir fichier code_spark/rdd_d.py

Figure 7: Schéma explicatif de l'approche D



- **Approche A'**: Comme a, mais avec génération au vol / paresseuse des RDD.
- Gestion des arbres déséquilibrés "en profondeur" ?

Gestion des arbres déséquilibrés "en profondeur".

4 Résultats et optimisation

On souhaite à présent tester les performances des différentes méthodes implémentées dans le but de les comparer et des les améliorer. Nos observations se portent principalement sur:

- **Le temps de calcul**. Celui-ci est mesuré à l'intérieur du code python (voir code_python/time.py). On évite ainsi de compter le lancement de Spark, etc. comme c'est le cas avec la fonction time de Linux. Cet aspect est le plus important, il s'agit de la valeur que nous voulons minimiser en priorité.
- **L'utilisation de la mémoire**. On observe celle-ci à l'aide de la commande Linux *top*. La question principale est de savoir si les RDDs sont développés en mémoire où s'ils sont développés et jetés à la volée tel qu'on le souhaite.
- **Le partitionnement du calcul** (distribution). Nous avons placé à certains passages de l'exécution la méthode *getNumPartitions()* qui renvoie le nombre de partitions d'un RRD. Nous observons également le résultat de la commande Linux *top* en temps réel afin de savoir le nombre de processus distincts sur lesquels les calculs s'exécutent.

Nous avons décidé d'exécuter tous les tests sur le même exemple. En effet, cette homogénéité permet de comparer efficacement les performances des différentes approches. L'exemple que nous avons choisi est le comptage des mots de longueur n sur l'alphabet A = {a, b, c, d}, dont on sait que la valeur attendue est 4^n. Il n'est pas nécessaire d'entrer dans des problèmes de combinatoire compliqués pour le moment puisque l'on souhaite simplement implémenter de gros calculs en Spark.

Figure 8: Tableau afin d'avoir une idée du nombre d'éléments comptés par les différents programmes.

n	4^n
1	4
2	16
3	64
4	256
5	1024
6	4096
7	16384
8	65536
9	262144
10	1048576
11	4194304
12	16777216
13	67108864
14	268435456
15	1073741824
16	4294967296

Les fichiers correspondant aux exécutions se situent dans le dossier code_spark.

Nous avons choisi de dénombrer les mots avec des Map-Reduces. Nous pouvions également compter avec la méthode count() qui donne la taille d'un RDD, les résultats étant d'après nos tests similaires. Il est probable que count() utilise un Map-Reduce.

Configuration de la machine sur laquelle on a lancé les tests...

Quelques définitions:

- L'**évaluation paresseuse** (en anglais : lazy evaluation)[7], appelée aussi appel par nécessité ou évaluation retardée est une technique d'implantation des programmes récursifs pour laquelle l'évaluation d'un paramètre de fonction ne se fait pas avant que les résultats de cette évaluation ne soient réellement nécessaires. Ces résultats, une fois calculés, sont préservés pour des réutilisations ultérieures.

  Il s'agit d'une propriété très importante de Spark. On distingue les actions et les je-sais-plus-quoi...

- **Aplatir en mémoire** : Fixer les valeurs des données paresseuses en les évaluant. Les actions aplatissent. Lorsque l'on veux volontairement aplatir on utilise flatMap(x => x).

Notre protocole de test étant à présent décrit, penchons-nous sur les résultats.

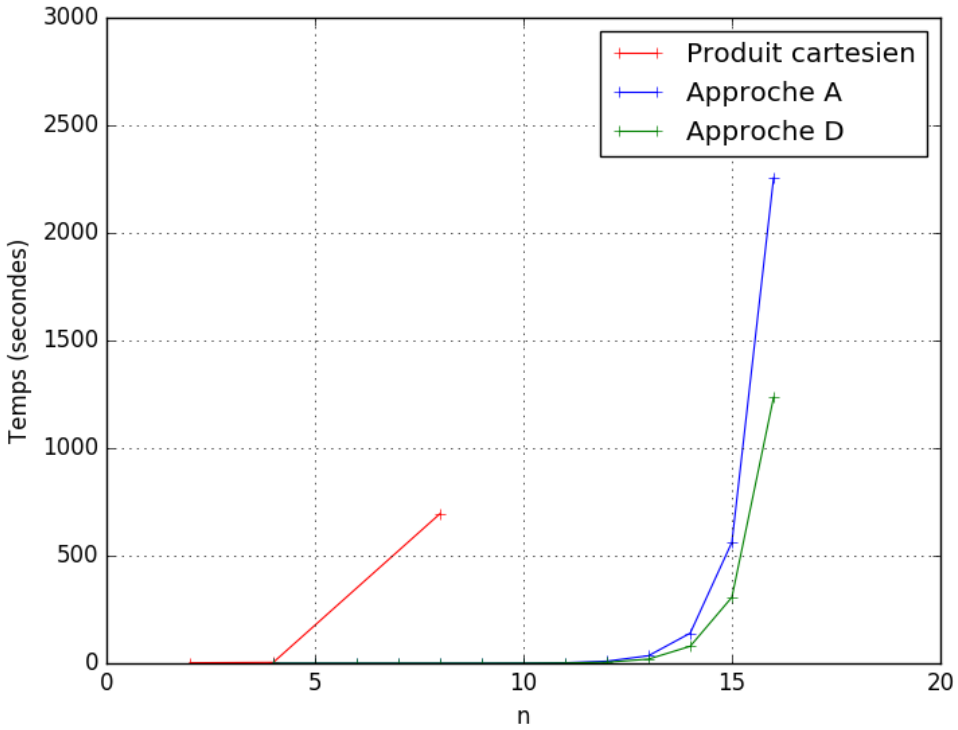
4.1 Résultats des approches testées

Concernant l'utilisation de la mémoire, le programme parvient effectivement à développer les éléments de RDDs à la volée. Les processus mis en cause dans le fonctionnement du programme ne dépasse pas 0.3% d'utilisation mémoire malgré les tailles conséquentes des RDDs (4294967296 par exemple).

Observons maintenant les temps d'exécution des différents programmes.

[7] https://fr.wikipedia.org/wiki/%C3%89valuation_paresseuse

Figure 9: Temps d'exécution en fonction de n



- Produit cartésien naïf: extrêmement lent
- Approche A: Pas de parallélisation malgré le partitionnement en 4
- Approche D: idem

Nous sommes arrivé à la conclusion que les calculs sont bien trop longs par rapport à ce qui aurait été obtenu avec une méthode classique. Voici quelques conjectures pouvant expliquer les raisons de ces résultats décevant:

Raisons ?

Parallélisation qui s'effectue mal: durant le map mais pas le reduce ? Certains morceaux de RDD sont réévalués de nombreuses fois inutilement Création récursive de petits RDDs qui prend beaucoup de temps

Les résultats de nos premières approches naïves étant analysés, nous allons à présent essayer de les améliorer en explorant différentes pistes.

4.2 Optimisation et nouvelles approches plus performantes

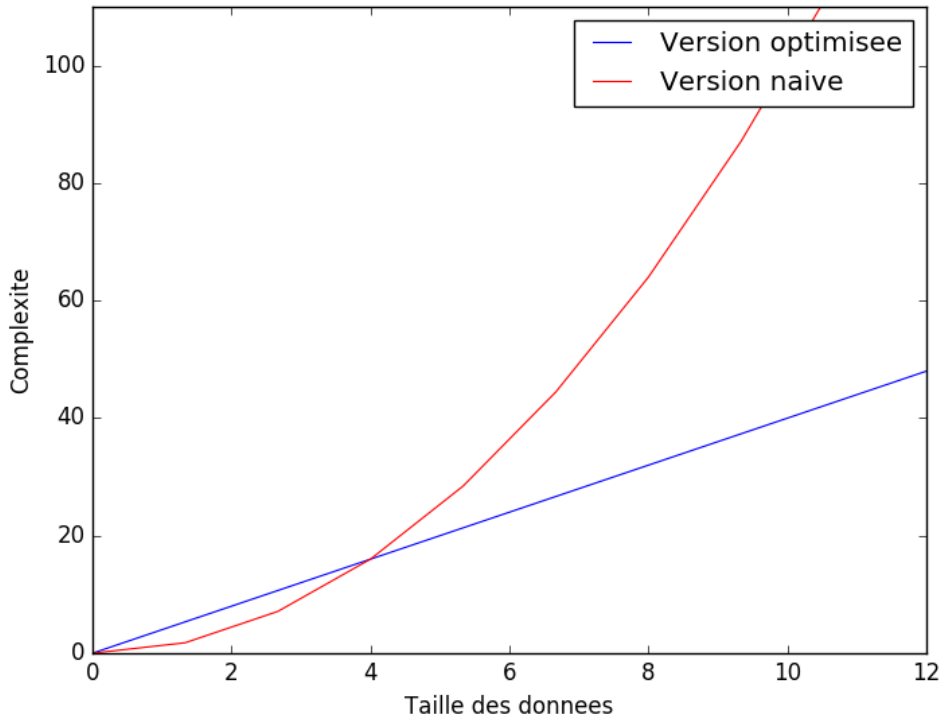
Le principal problème que nous avons rencontré lorsque nous souhaitions comprendre les performances pour les améliorer est que les exécutions de Spark sont très opaques. On a très peu d'informations sur ce qui est fait. Nous allons mettre en œuvre des solutions potentielles aux conjectures proposées plus tôt.

Pistes d'optimisation:

- Faire du profiling pour déterminer quelles parties de l'exécution prennent beaucoup de temps.
- Aplatir le RDD en mémoire à un certain niveau.

- Faire un algorithme hybride, qui alterne entre plusieurs méthodes, selon la taille du RDD.

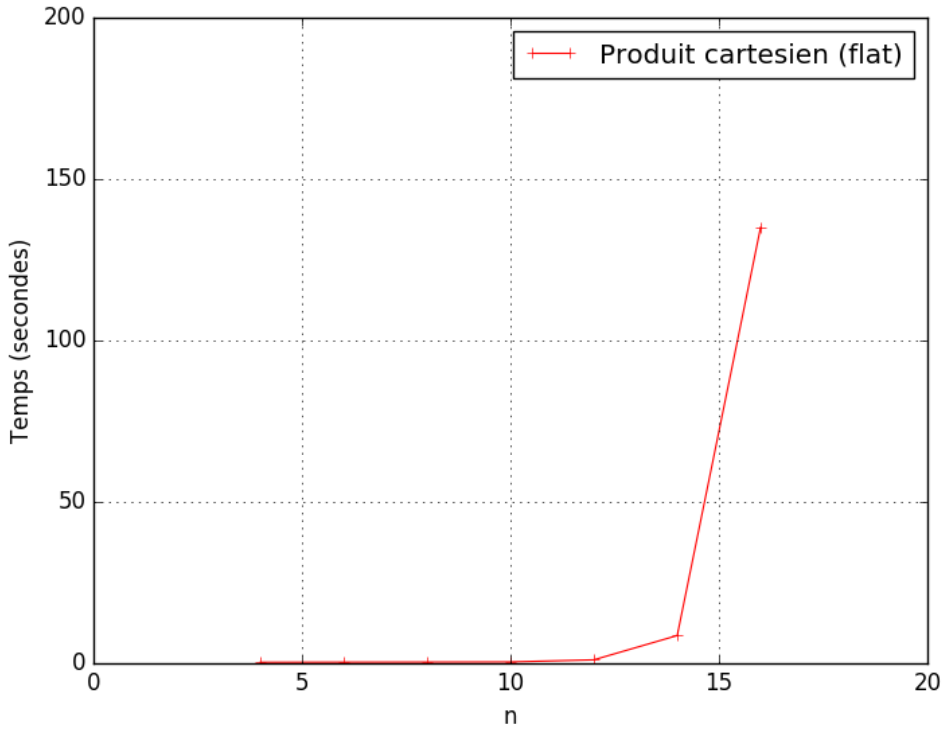
Figure 10: Graphique explicatif du principe de l'algorithme hybride



Dans cet exemple, la taille 4 est le seuil à partir duquel on change d'algorithme. L'exécution étant récursive, on gagnera du temps sur les calculs formant les briques de base qui permettent de construire les calculs à plus grande échelle.

Aplatissement -> grand grain de performance (voir fichier code_spark/flat_cartesian_product.py) Produit cartésien avec aplatissement à $\frac{1}{4}$ de la taille du RDD total, soit la profondeur $n - 1$.

Figure 11: Temps d'exécution en fonction de n



Le temps d'exécution avec $n = 16$ est de 135.075317144 secondes. Les performances sont améliorées d'un facteur 16.7. C'est un bon début mais ce n'est pas suffisant.

Si on développe entièrement le RDD en mémoire, évidemment on a une saturation de la mémoire (on ne dépasse pas n = 12) et les performances ne sont pas bonnes:

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

On souhaite donc définir à quelle profondeur relative il est le plus intéressant de développer les RDDs en mémoire (puisque cette profondeur sera atteinte un certain nombre de fois de par la nature même de la récursivité).

Schéma performance nouvelles approches
- Autres flat
- Flat a profondeur K
- Hybride

5 Conclusion

Fin du film. Section 3 et 4 à vraiment approfondir (c'est l'essentiel du TER).

References

[1] Jean Fromentin and Florent Hivert
Exploring the tree of numerical semigroups

[2] Florent Hivert
Turn the Python prototypes for tree exploration into production code, integrate to SAGE

[3] Nicolas Thiery, tanani Calcul mathématique avec Sage