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 **3**

2 Révisions préliminaires **3**
- 2.1 Outils utilisés . . . 3
- 2.2 Notions mathématiques . . . 4
  - 2.2.1 Permutations . . . 4
  - 2.2.2 Arbres et arbres binaires . . . 5
  - 2.2.3 Recursively Enumerated Set . . . 5
  - 2.2.4 Monoïdes numériques . . . 5
  - 2.2.5 Map-Reduce . . . 6
  - 2.2.6 Parallélisation . . . 7

3 Spark **7**
- 3.1 Installation et mise en place de Spark sur le cluster de machines . . . 7
- 3.2 Premiers pas avec Spark . . . 8
  - 3.2.1 Evaluation et récursivité . . . 8
  - 3.2.2 Resilient Distributed Datasets . . . 8
  - 3.2.3 Opérations sur les RDD . . . 8
- 3.3 Différentes implémentations utilisant des RDDs . . . 9
  - 3.3.1 Modélisation de l'ensemble des mots sur un alphabet par un RDD . . . 9
  - 3.3.2 Modélisation de l'ensemble des arbres binaires à n noeuds par un RDD . . . 9
  - 3.3.3 Modélisation d'un RecursivelyEnumeratedSet par un RDD . . . 9

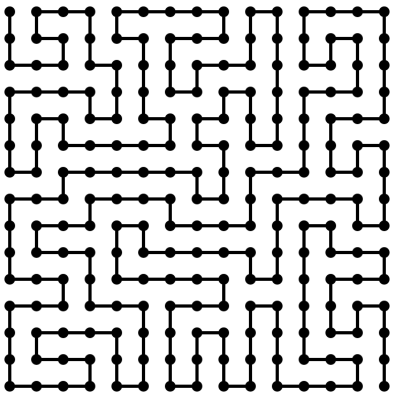
4 Résultats et optimisation **11**
- 4.1 Résultats des approches testées . . . 13
- 4.2 Optimisation et nouvelles approches plus performantes . . . 13
  - 4.2.1 Aplatissement mémoire naïf . . . 14
  - 4.2.2 Produit cartésien avec aplatissement mémoire . . . 15
  - 4.2.3 Approche D améliorée . . . 16
  - 4.2.4 Hybride et Flat dynamique . . . 17

5 Conclusion **17**

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. De plus, depuis l'arrivée des ordinateurs, nous avons la possibilité d'effectuer des calculs à une vitesse prodigieuse. C'est pour cette raison que la combinatoire est étudiée à l'aide de l'informatique.

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



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 et leur mise en pratique. 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. Il est porté par une communauté qui se charge d'implémenter de nouvelles fonctionnalités jour après jour. Concernant l'installation, nous avons téléchargé les fichiers sources de Sage, puis nous l'avons compilé. La compilation a duré 6 heures, 51

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

minutes et 19 secondes. Nous avons pris le temps de prendre en main Sage en codant différents RecursivelyEnumeratedSet, en appliquant des Map-Reduce dessus, ou encore en affichant des graphes représentatifs de fonctions mathématiques.

Voir le fichier code_sage/set.sage.

- **Python**: La quasi-totalité du code que nous avons écrit durant ce TER est en Python. Spark prend en charge le Scala (son langage natif), le Python et le Java. SageMath utilise également Python. Ce choix est donc justifié. Nous avons également prise en main ipython Notebook, qui s'est avéré très utile pour clarifier le code et communiquer nos idées entre nous.
- **Git**: Le travail a été mis en commun 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, allumé à tout heure du jour et de la nuit. Nous avions la possibilité de s'y connecter et de lancer des calculs en SSH. Nous avons écrit un script shell de configuration nommé *setup_virtual_machine* au fur et à mesure des installations nécessaires au bon fonctionnement de nos programmes. Ainsi, les prochains utilisateurs de Sage et Spark pourront s'éviter certaines étapes d'installation fastidieuses en exécutant simplement ce script. Si les méthodes que nous allons tester s'avèrent efficaces, on veux pouvoir les reproduire sur d'autres machines et d'autres clusters de machines avec une certaine adaptabilité.

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 renvoie 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:

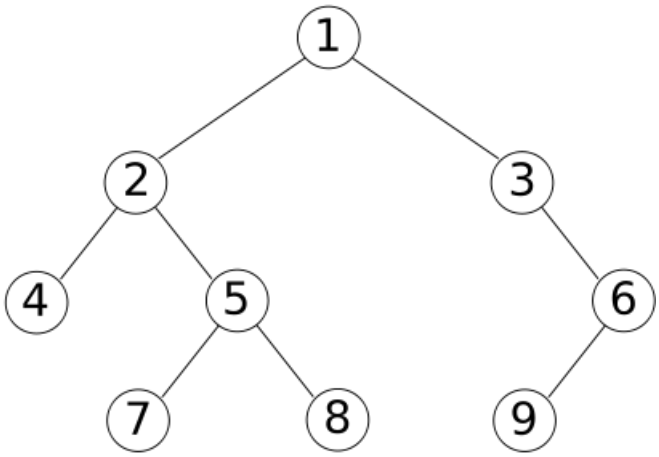
```
>>> permutation(2345)
['2345', '2354', '2435', '2453', '2534', '2543',\\
'3245', '3254', '3425', '3452', '3524', '3542',\\
'4235', '4253', '4325', '4352', '4523', '4532',\\
'5234', '5243', '5324', '5342', '5423', '5432']
```

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

2.2.2 Arbres et arbres binaires

Il est courant de structurer les données sous formes d'*arbres*. Un arbre est un ensemble de nœuds liés de façon orienté. Un sens est ascendant, l'autre descendant (fils).

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



Le calcul parallèle et la récursivité amènent bien souvent à des topologies en arbre; cette notion est donc très présente tout au long de ce projet.

2.2.3 Recursively Enumerated Set

Un *ensemble énuméré récursivement* (recursively enumerated set en anglais) est un ensemble dont une machine peut énumérer les éléments suivants à partir des éléments précédents. Le principe est simple. On a une fonction successeur qui pour chaque élément donne son ou ses élément(s) suivant(s) (fils). On définit un élément initial (la racine) 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). *Voir le fichier code_sage/set.sage pour une implémentation en 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.
- **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.

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

- **Conducteur**: Le frobenius + 1, dont on sait par déduction qu'il fait parti de S.

L'arbre de tous les semi-groups est très déséquilibré. Cette propriété le rend intéressant à étudier pour concevoir des algorithmes repartis 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. Ceci peut par exemple être mené à bien à l'aide d'un algorithme de vol de tâches.

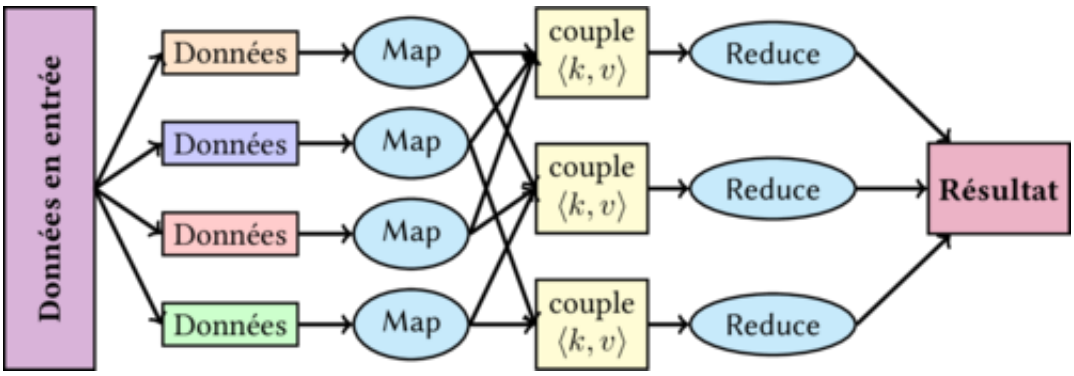
Voir le fichier code_sage/semigroup.sage pour une implémentation en Sage.

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 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**: Réduction de tous les éléments 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 qui lui incombe ? 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 cœurs de notre machine, nous effectuons la commande "top" qui permet de voir les processus actifs classés par taux d'utilisation des ressources.

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, c'est otut à fait normal puisqu'il est écrit en Scala.

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

  Dans ce cas, il n'y a pas de mémoire partagée, il faut diviser donc l'information entre les différents agents. Malgré cela, sur un arbre très déséquilibré, il semble difficile 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 : Aucun des exemples de la documentation ne fonctionnait. Nous avons résolu ce problème en ajoutant la ligne *127.0.0.1 combispark* dans le fichier */etc/hosts* (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 par Spark du code Python se fait 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 (puisqu'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 Evaluation et récursivité

Quelques définitions utiles à la compréhension du fonctionnement et des avantages qu'offre Spark:

- 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, peuvent être préservés pour des réutilisations ultérieures.

  Il s'agit d'une propriété très importante de Spark. On distingue les **actions** et les **transformations**. Une action renvoie un résultat et nécessite donc d'évaluer les données récursives tandis qu'une transformation renvoie un RDD dont les valeurs réelles ne seront évaluées qu'en cas de besoin.

- **Aplatir en mémoire** : Fixer les valeurs des données paresseuses en les évaluant. Dans nos implémentations, lorsque l'on veux volontairement aplatir un RDD on utilise une action telle que la méthode first(), qui renvoie le premier élément du RDD. En Spark, l'appel d'une action provoque donc un aplatissement, contrairement à l'appel d'une transformation.

3.2.2 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. Les RDDs sont tolérants à la panne et proposent une structure parallèle qui laisse 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.

3.2.3 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. Ces opérations sont des transformations.

- **Produit cartésien**: L'ensemble de tous les couples dont la première composante appartient au premier RDD et la seconde au second RDD.
- **Union**: L'ensemble de tous les éléments des deux RDDs. L'union ne supprime pas les doublons ainsi créés.
- **Join**: La jointure au sens entendue en base de données.
- **Map et Reduce**: Les opérateurs permettant d'appliquer des Map-Reduce (cf. section 2.2.5 du rapport) aux RDDs.

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

- **FlatMap**: Renvoi un RDD en appliquant une fonction à tous les éléments du RDD d'origine. 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. Ceux-ci sont visibles dans la documentation de Spark.

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 et analyser les performances, en temps de calcul et en mémoire, des différentes méthodes. Les implémentations sont faites en Python car nous ne sommes pas parvenus à exécuter du code Sage avec Spark. En effet Sage utilise une version particulière de Python, et n'est donc pas compatible avec l'utilisation de Spark. Ce problème reste donc à résoudre pour qui souhaite exécuter du Sage avec Spark. Cela semble à priori possible, mais nous avons choisi de ne pas chercher plus loin dans cette voie puisque ce n'était pas indispensable au travail que nous voulions effectuer. L'objectif final de ces implémentations est de lancer des Map-Reduce dessus.

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. Par exemple prenons un alphabet A composé des strings {'a', 'b', 'c' et 'd'}. En faisant 16 fois un produit cartésien avec lui-même, on obtient donc un ensemble de 4^{16} mots possibles de 16 lettres contenant les lettres 'a', 'b', 'c' et 'd'. Nous verrons plus loin dans le rapport que c'est sur cet exemple que nous avons testé la plupart de nos algorithmes.

Voir les fichier code_spark/cartesian_product.py et code_python/permutation.py.

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

Cet ensemble se modélise ainsi:

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

On peut ensuite par exemple compter les arbres ou compter les arbres par profondeur (utiliser *p(t) = $X^{\text{profondeur de } T}$*. Ce modèle peut également permettre de manipuler des polynômes. Nous avons eu accès à plus de détails sur ce sujet dans le chapitre *combinatoire* du livre "Calcul mathématique avec Sage"[3].

Voir le fichier code_spark/tree.py.

3.3.3 Modélisation d'un RecursivelyEnumeratedSet par un RDD

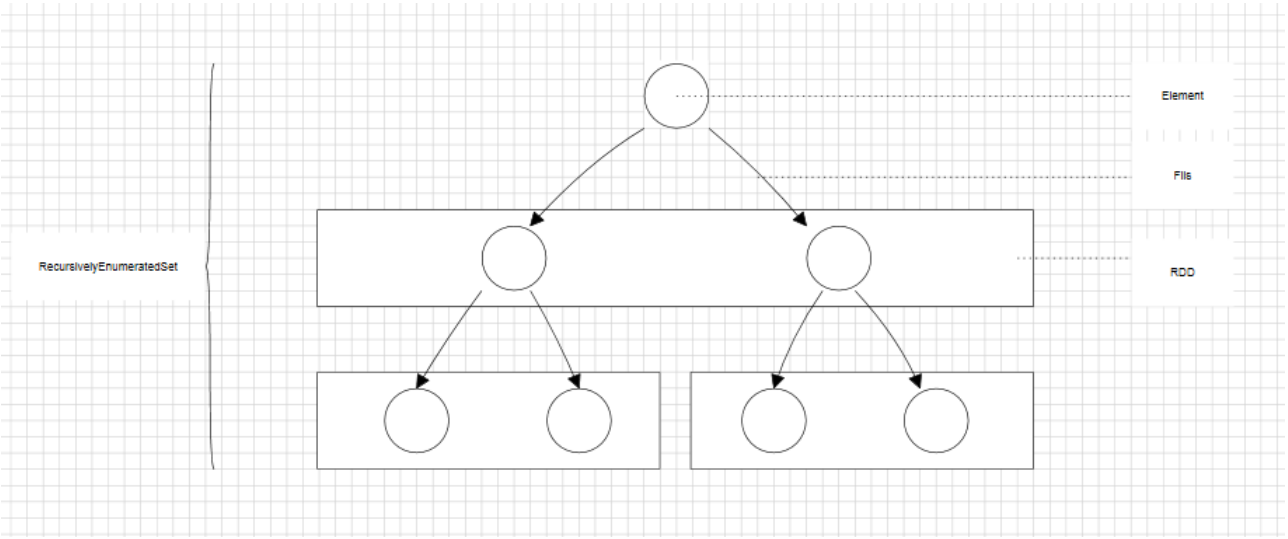
Nous recherchons à présent la meilleure méthode pour modéliser un RecursivelyEnumeratedSet avec Spark.

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

Nous avons pensé à différentes approches:

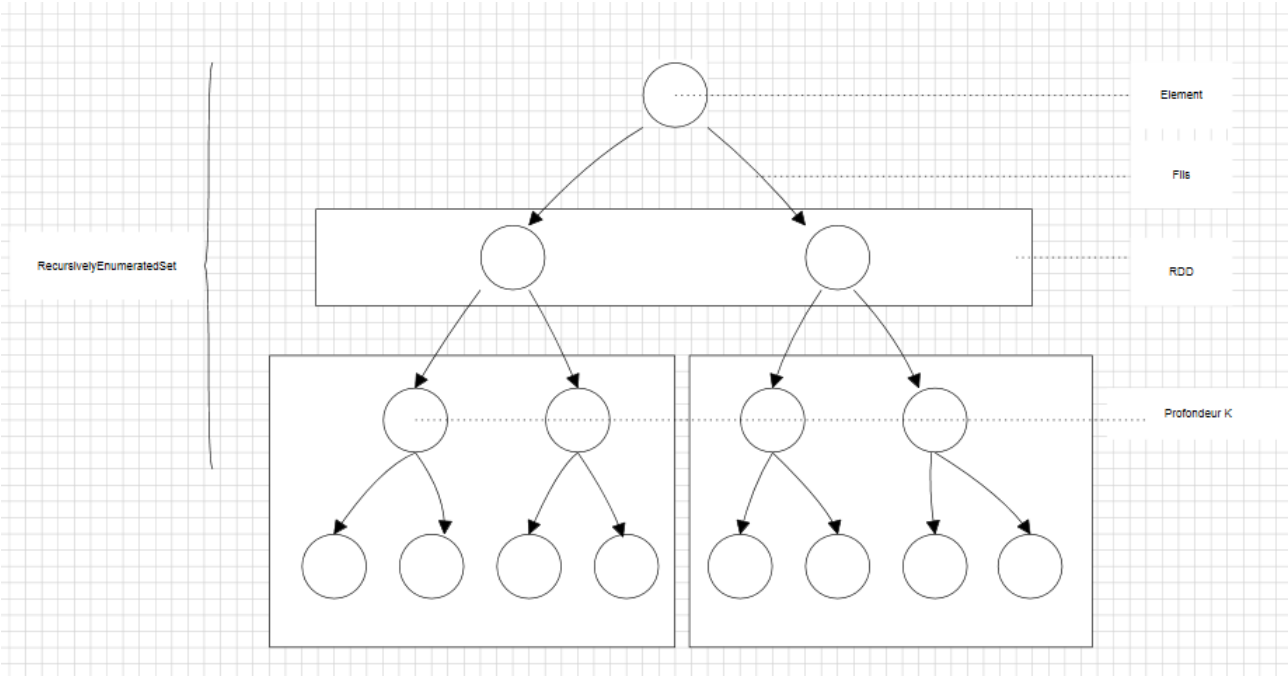
- **Approche A**: Pour chaque noeud de l'arbre, modéliser par un RDD l'ensemble de ses fils. *Voir le fichier code_spark/rdd_a_b.py.*

Figure 4: Schéma explicatif de l'approche A



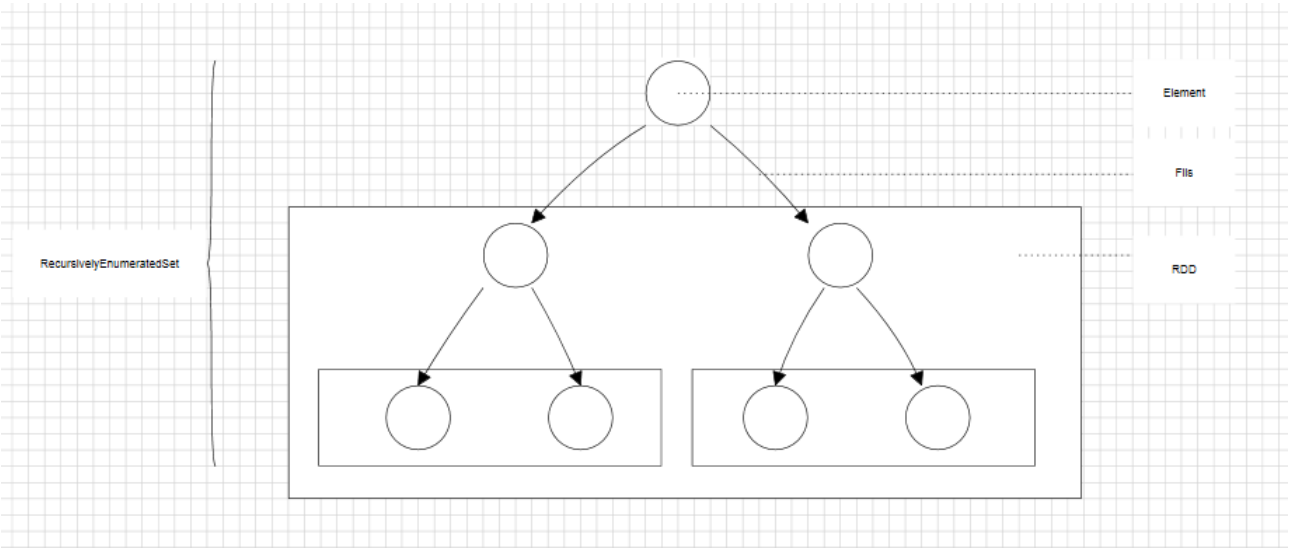
- **Approche B**: Comme l'approche A, mais on s'arrête à une profondeur K donnée, et ensuite on travaille en local: le RDD d'un noeud à profondeur k contient tous les descendants du noeuds. A priori, cette méthode ne gère pas bien les arbres déséquilibrés en largeur. *Voir le fichier code_spark/rdd_a_b.py.*

Figure 5: Schéma explicatif de l'approche B



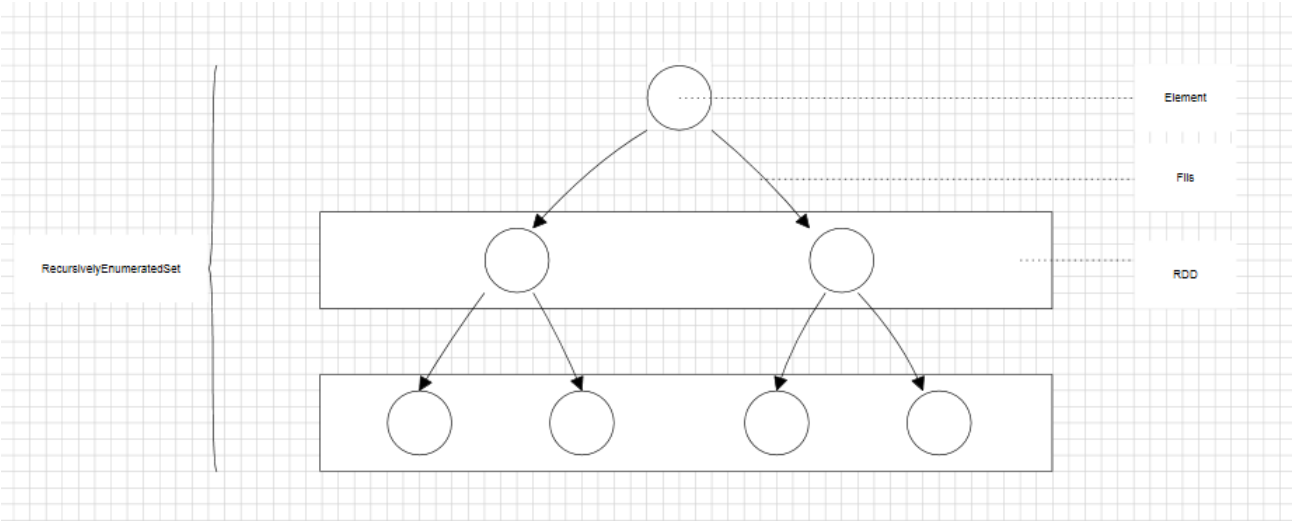
- **Approche C**: Pour chaque noeud de l'arbre, modéliser par un RDD l'ensemble de ses descendants.

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 la méthode flatmap. *Voir le fichier code_spark/rdd_d.py.*

Figure 7: Schéma explicatif de l'approche D



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 le fichier code_python/time.py pour un exemple*). On évite ainsi de comptabiliser le lancement de Spark - ou tout autres effets

pouvant altérer le temps de calcul - 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

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 d'ailleurs probable que *count()* utilise un Map-Reduce, ou une forme de calcul distribué proche.

Voici la configuration de la machine sur laquelle nous avons lancé les tests:

- 4 processeurs identiques: Intel(R) Core(TM)2 Duo CPU T7700 @ 2.40GHz.

- 8 Go de mémoire RAM.

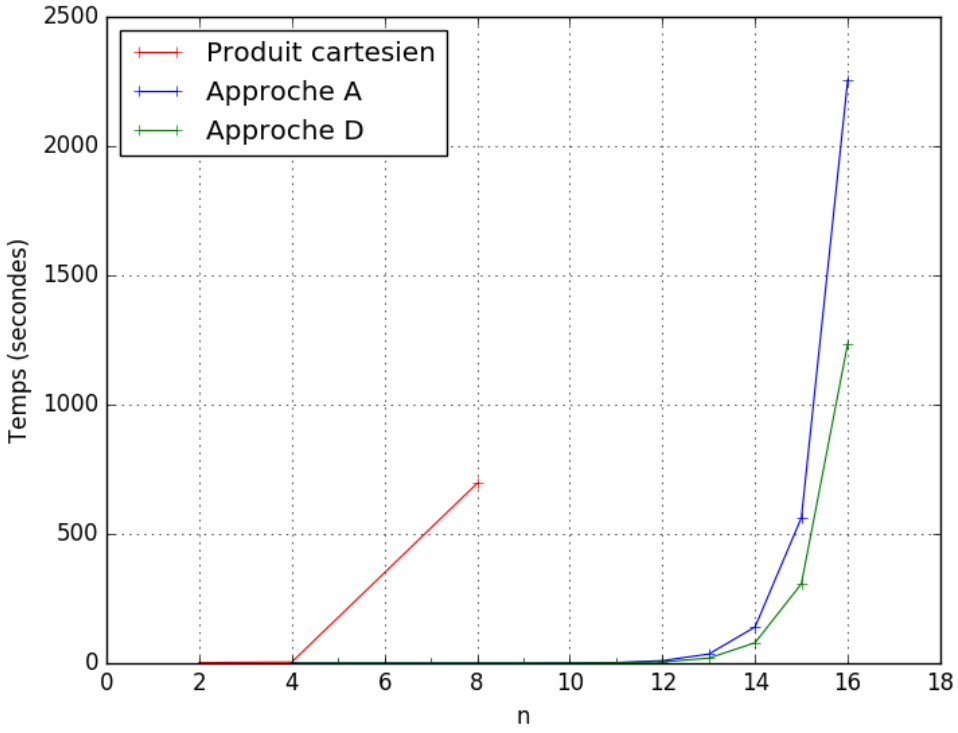
Les fichiers correspondant aux exécutions se situent dans le dossier *code_spark*. Notre protocole de test étant à présent décrit, penchons-nous sur les résultats.

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.

Figure 9: Temps d'exécution en fonction de n



On observe que le produit cartésien naïf est extrêmement lent. L'approche A et l'approche D sont lentes également. Nous avons remarqué qu'il n'y a pas de parallélisation du calcul malgré le partitionnement en 4 des RDDs. Ces implémentations basiques sont donc à revoir. Les calculs durent bien trop longtemps par rapport à ce qui aurait été obtenu avec des méthodes classiques. Voici quelques conjectures pouvant expliquer les raisons de ces résultats décevant :

- La parallélisation s'effectue mal à cause de l'implémentation en Python de nos ensembles récursifs. La structure de données utilisant des strings et les opérations utilisées sont mal choisies.
- Certains morceaux des RDDs sont réévalués de nombreuses fois inutilement au lieu d'être mémorisés.
- Il y a un création récursive de petits RDDs qui prend au total beaucoup de temps.

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

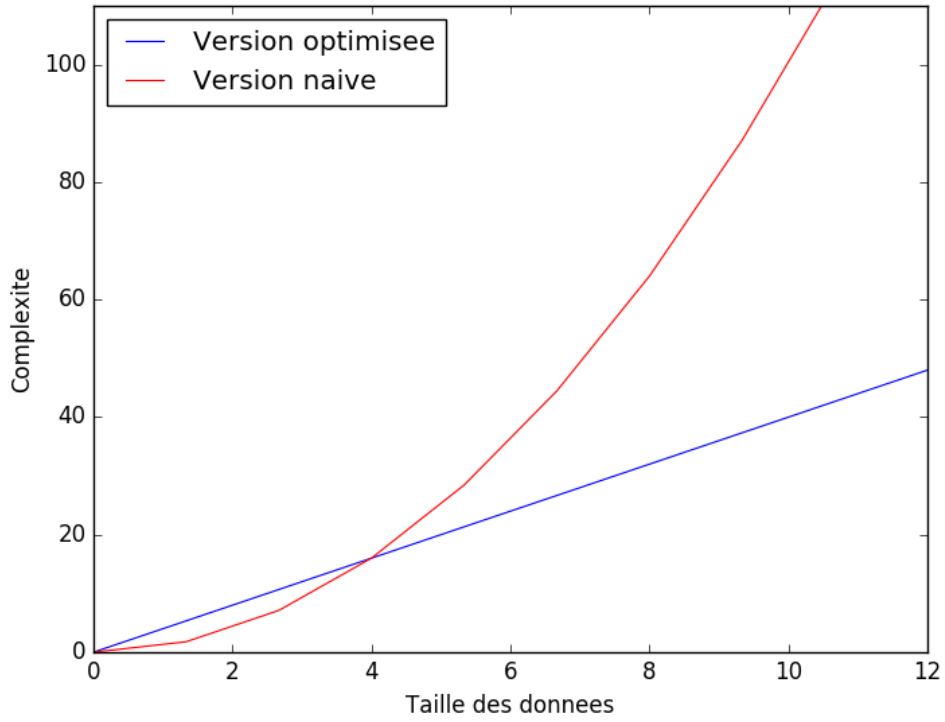
4.2 Optimisation et nouvelles approches plus performantes

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.
Voici nos pistes concernant l'optimisation :

- Faire du profiling pour déterminer quelles parties de l'exécution prennent beaucoup de temps.
- Aplatir le RDD en mémoire à un certain niveau afin d'éviter les créations multiples de RDD. L'aplatissement semble permettre un grand grain de performance (*voir le fichier code_spark/flat_cartesian_produc*
- 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.

Les nouvelles approches testées sont les suivantes:

4.2.1 Aplatissement mémoire naïf

On fait l'essai de collecter le RDD dans sa totalité. Si on développe entièrement le RDD, on provoque évidement une saturation de la mémoire pour des n trop élevés. Sur notre machine, on ne parvient pas à dépasser n = 12 et les performances sont plutôt mauvaises:

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

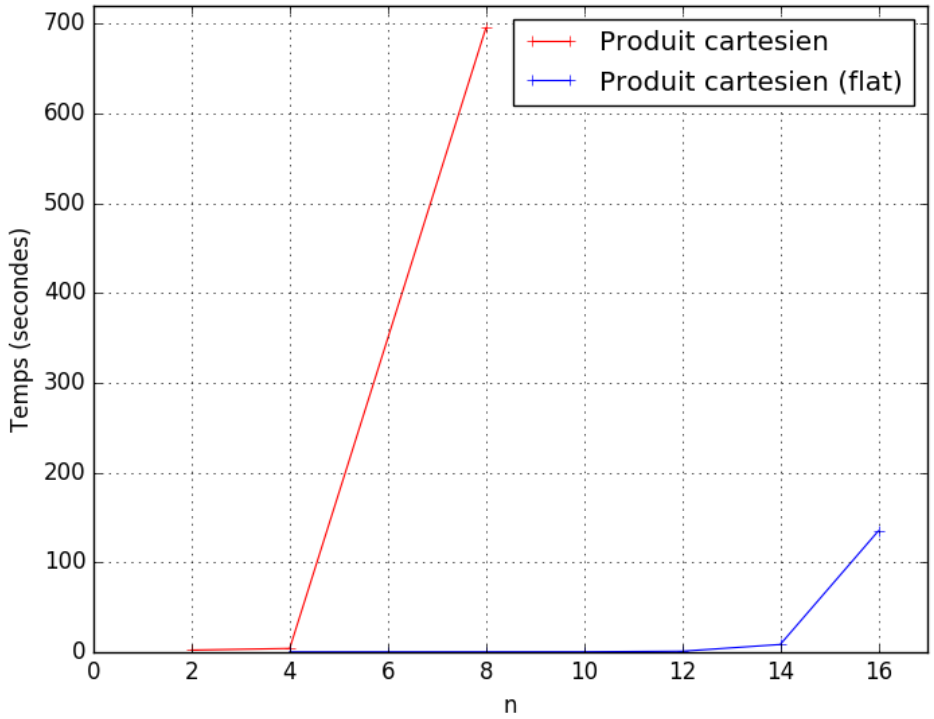
On souhaite donc définir à quelle profondeur relative il est le plus intéressant de développer les RDDs en mémoire (puisque cette profondeur sera atteinte un certain nombre de fois à différents endroits de l'évaluation de part la nature de la récursivité).

Voir le fichier code_spark/simple_calc.py.

4.2.2 Produit cartésien avec aplatissement mémoire

On test tout d'abord un produit cartésien avec aplatissement à $\frac{1}{4}$ de la taille du RDD total, soit la profondeur $n-1$. Cette méthode risque une explosion de la mémoire pour des n trop élevés.

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.

Voir le fichier code_spark/flat_cartesian_product.py.

En faisant varier la taille des RDDs développés en mémoire, nous sommes parvenus à un résultat meilleur. Les RDDs aplatis étaient de taille 4^5. Tous les processeurs tournaient à 99.7%. Les ressources disponibles étaient donc bien employées comme on peut le voir ci-dessous :

```
PID  USER        PR   NI   [...]   \%CPU  \%MEM     TIME+  COMMAND
29700 ubuntu     20    0   [...]   99.7   0.3    1:07.20  python
29701 ubuntu     20    0   [...]   99.7   0.3    1:06.79  python
29705 ubuntu     20    0   [...]   99.7   0.3    1:06.62  python
29709 ubuntu     20    0   [...]   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
```

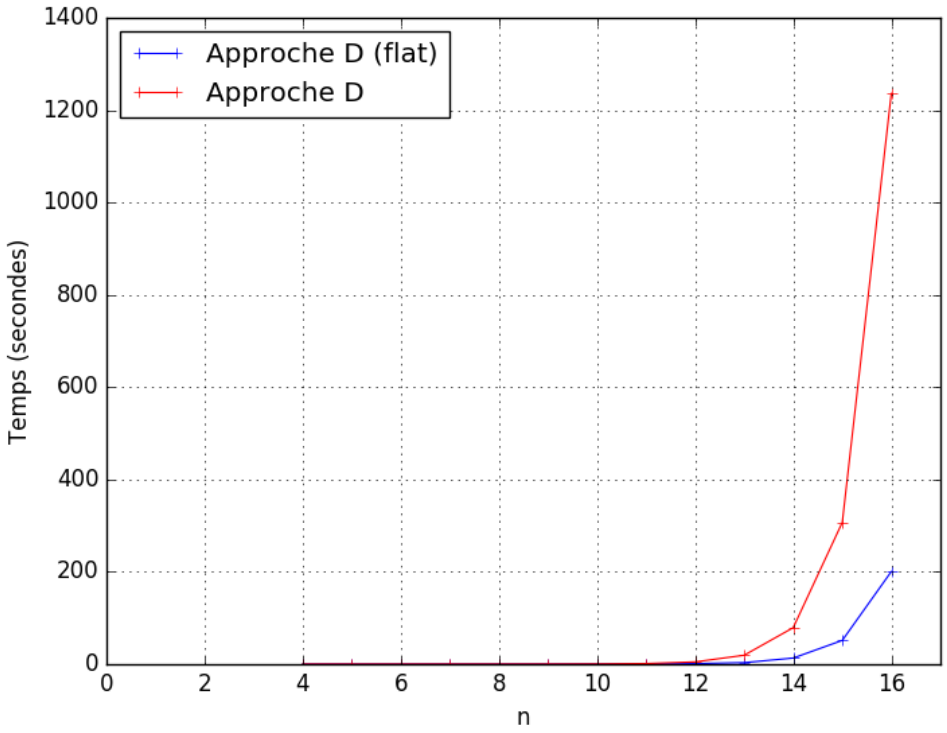
Temps d'execution: 129.296506166 secondes.

Voir le fichier code_spark/flat_simple_calc.py.

4.2.3 Approche D améliorée

Parmis les approches naïves, nous avons vu que la D était la plus concluante. C'est pourquoi nous avons cherché à l'optimiser en priorité. On y trouve maintenant un aplatissement des branches en des feuilles de taille 1024, et une substition des strings par des entiers. L'approche D parvient enfin à être parallélisé sur les 4 processus et on améliore les performances d'un facteur 6. Ce facteur d'amélioration est cohérent puisque le calcul est effectué sur 4 processeurs plutôt que 1 dans la version précédente de l'algorithme.

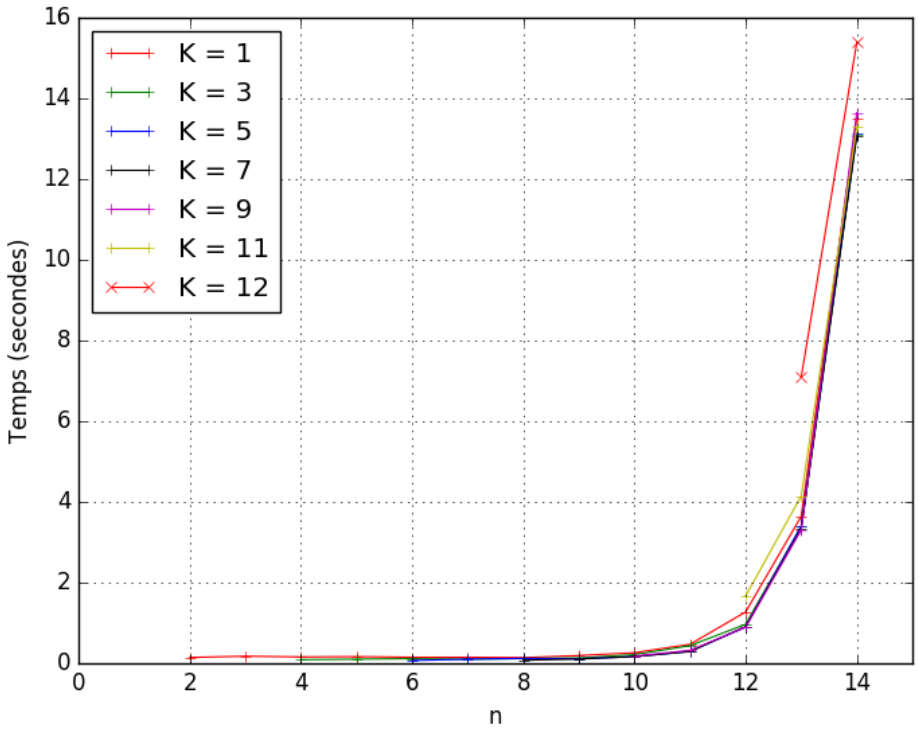
Figure 12: Schéma comparatif de l'approche D et de sa version améliorée avec aplatissement en briques de taille 1024



Voir le fichier code_spark/rdd_d_better.py.

On souhaite à présent généraliser cette idée, et on essaie de déterminer la taille optimale des RDDs développés en mémoire. On définit donc un paramètre K représentant la profondeur à partir de laquelle on aplati les RDDs. Les RDDs ainsi obtenus seront de taille 4^K. On fait varier incrémentalement le paramètre K afin de comparer les résultats.

Figure 13: Temps d'exécution en fonction de n pour différents K



Cette approche semble moins influencée par le développement en mémoire des RDDs que le produit cartésien. On remarque tout de même une légère évolution du temps de calcul. Il est également intéressant de noter que nous n'avons pas pu dépasser la valeur K = 12. En effet, comme lors de la méthode de l'aplatissement naïf, de trop gros RDDs entraînent une saturation de la mémoire.

4.2.4 Hybride et Flat dynamique

Ces idées d'améliorations ne sont pas encore concluantes. Nous souhaitons faire varier K en fonction de n, ou encore combiner différentes modélisations de RDDs. Les résultats de ces approches seront ajoutés à la version finale de ce rapport si elles permettent une amélioration radicale des performances.

5 Conclusion

En conclusion, nous avons au cours de ce projet appris de nouvelles notions mathématiques et perfectionner notre usage de certains outils informatiques. La prise en main de Spark était assez fastidieuse au début, cependant après la formation l'Ecole Spark et quelques essais, nous sommes parvenues à des résultats. Nous avons touché du doigt la méthode scientifique et la tentative d'amélioration de performances dans le cadre de calculs combinatoires. L'objectif était de savoir si Spark pourrait permettre d'obtenir de bonnes performances de calculs sur des ensembles énumérés de façon récursive plutôt que sur son utilisation habituelle, c'est-à-dire des grandes quantités de données enregistrées en mémoire. Nous pouvons conclure que Spark permet effectivement ce genre de calcul sur des données générées à la volée, cependant les résultats que nous avons obtenus ne permettent pas de dire si cette technologie a la possibilité d'obtenir de meilleurs performances que les méthodes de calcul classiques en terme de temps écoulé. Cette possibilité reste tout à fait envisageable d'après nous.

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] A. Casamayou, N. Cohen, G. Connan, T. Dumont, L. Fousse, F. Maltey, M. Meulien, M. Mezzarobba, C. Pernet, N. M. Thiéry, P. Zimmermann
Calcul mathématique avec Sage