Rapport INF727 - WordCount distribué

Philéas SAMIR
MS BGD 21-22
Télécom Paris
phileas.samir@telecom-paris.fr

***Résumé*—Nous avons implémenté un WordCount distribué. Nos expérimentations à partir de cette implémentation nous permettent de redémontrer la loi d'Amdahl. Nous montrons aussi l'intérêt des méthodes de calcul distribué en travaillant sur des volumes de données qui dépassent les capacités matérielles d'une machine seule. Nous retrouvons que les limites les plus contraignantes deviennent les écritures et lectures disque et les temps réseau. Nos résultats sont cohérents avec les limites connues du paradigme MapReduce et les innovations telles que Spark.**

I. Introduction

Le paradigme de programmation MapReduce [2] permet de produire des applications distribuées au sein d'un cadre conceptuel défini. C'est une avancée majeure qui est au coeur de l'écosystème Big Data tel que le connaissons. Des fonctions MapReduce sont toujours packagées avec des logiciels comme Hadoop [3] et malgré l'avènement de nouveaux paradigmes comme Spark [5], il existe encore des cas d'usage pour lesquels MapReduce demeure la meilleure option.
L'application est divisée conceptuellement en plusieurs parties :

- Un input est divisé en *splits* qui sont envoyés sur des nodes,
- Une fonction *map* est appliquée à ces splits, les outputs sont écrits sur disque,
- Les outputs sont envoyés sur des machines déterminées en fonction de leur clef lors du *shuffle*,
- Les clefs ayant été distribuées, chaque node peut travailler en parallèle et effectuer une fonction *reduce*,
- Enfin, la machine qui avait lancé l'application récupère tous les outputs des nodes et effectue éventuellement un *sort*.

En partant des travaux pratiques guidés de M. Sharrock, nous implémentons un WordCount distribué en Python. Le code est disponible sur GitHub. Deux versions sont disponibles : l'une à partir d'un fichier .txt donné par l'utilisateur, l'autre à partir d'une liste d'URLs vers des fichiers stockés sur des serveurs (en l'occurrence, des fichiers d'environ 400 Mo fournis par Common Crawl stockés sur un bucket S3 AWS).
Comme le montrent les tables II et III, notre WordCount distribué sur un seul fichier Common Crawl s'effectue en moins d'une minute (environ 55 secondes), sur 12 de ces fichiers il s'effectue en moins de 4 minutes (232 secondes), sur 24 de ces fichiers s'effectue en moins de 6 minutes (332 secondes) en prenant en compte les temps réseau. La mémoire vive des machines fournies par l'école, ou de nos machines personnelles, ne permet pas d'effectuer ce WordCount en local pour un tel volume de données (24 fichiers de 400 Mo soit 9.7 Go environ).

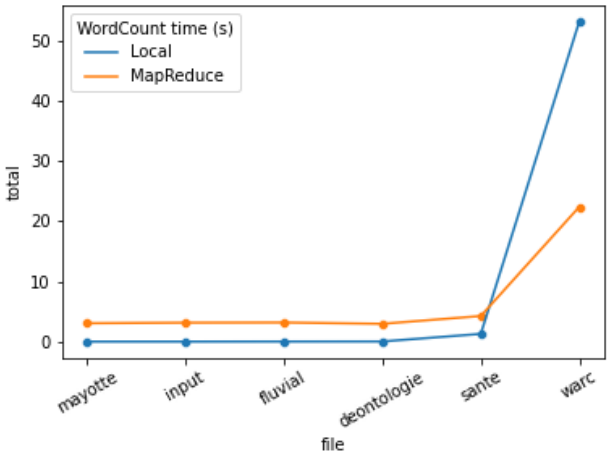

Figure 1. Temps d'exécution du WordCount en secondes, local vs MapReduce.

II. Implémentation

A. Version locale et premiers résultats

Notre première implémentation d'un WordCount local compte les mots à l'aide d'un dictionnaire, voir I. Cette version entièrement locale nous donne une base de comparaison pour comprendre à partir de quel volume de données la distribution devient pertinente, soit en temps d'exécution, soit du fait de contraintes matérielles.

file	wordcount (s)	sort (s)	total time (s)
warc	32.33	15.29	53.24
deontologie	0.00	0.00	0.02
fluvial	0.01	0.00	0.01
forestier	0.00	0.00	0.00
input	0.00	0.00	0.00
sante	1.16	0.07	1.28
12_warc	465.88	101.20	600.89

Table I
Temps d'exécution du WordCount en local.

Le WordCount distribué suivra les parties décrites plus haut :

— L'input est un fichier texte qui doit être divisé en splits plutôt égaux, avec une contrainte : ne pas diviser au milieu d'un mot,
— La fonction map lit le fichier split et assigne à chaque mot un hash afin de préparer les fichiers à envoyer lors du shuffle,
— La fonction shuffle envoie les fichiers aux autres machines à l'aide des hash,
— On compte le nombre d'occurrences de chaque mot dans les shuffles reçus,
— On récupère les outputs des reduces distribués et on les associe (chaque mot n'apparaît que sur une machine), puis on les ordonne par fréquence décroissante.

Après implémentation de la version proposée par le TP guidé, notre premier WordCount distribué sur le code de la santé publique mettait 697 secondes. Cette première version distribuée est *I/O intensive*. En effet, pour chaque mot unique présent dans un split, un fichier unique est créé dans la phase de map, puis on y écrit une nouvelle ligne pour chaque occurrence du mot dans le split (il y a une écriture disque par mot). Chaque fichier ainsi créé est envoyé à une machine durant la phase de shuffle. Ces nombreuses écritures disque prennent un temps considérable, de même que le shuffle : pour chaque fichier, on calcule à quelle machine envoyer, puis on envoie. Ainsi, une connexion ssh doit être ouverte *pour chaque mot unique pour chaque machine*. De quoi saturer un réseau si on n'est pas précautionneux...
Cette première version ne permet même pas d'envisager un WordCount sur un seul fichier Common Crawl.

B. Accélérations du MapReduce avec input donné

Nous avons amélioré à plusieurs reprises la fonction qui transforme l'input en splits. Les versions antérieures sont toujours consultables dans le code du master. La dernière version n'est pas plus rapide que l'itération précédente, en revanche, elle permet de splitter des plus gros fichiers sans saturer la mémoire.
Nos améliorations du temps d'exécution ont ciblé trois axes d'accélération :

— Réduire le nombre d'écritures sur disque,
— Réduire le nombre de connexions ssh ou simplifier les envois par ssh,
— Paralléliser le code.

Pour résoudre le problème des nombreux envois ssh, une première solution moins network intensive a été d'écrire dans des sous-dossiers du dossier Shuffles, avec au maximum un sous-dossier par machine du cluster. On copie l'intégralité du dossier vers la machine souhaitée, et on réduit ainsi le nombre de connexions ssh. Cette solution reste tout aussi I/O intensive. Pour réduire à la fois le nombre d'écritures et le nombre d'envois, on utilise des dictionnaires : les clefs sont les noms des machines du cluster, les valeurs correspondent à la liste des mots associés à cette machine (`index_machine = hash_mot % nmachines`). On n'écrit sur disque qu'une fois, lorsque le dictionnaire a été complètement généré, et pour chaque machine il n'y a dans le pire cas que n fichiers dans lesquels écrire et à envoyer via ssh, avec n le nombre de machines sur le cluster.
Cette nouvelle version du map implique de modifier le reduce pour s'y conformer. Initialement, étant donné qu'on a un fichier par mot unique, on peut regarder le mot en question et compter le nombre de lignes du fichier pour avoir un couple (mot/occurrences). Par la suite, on aura un fichier qui contient tous les mots destinés à une machine (non triés). Notre fonction reduce devient donc équivalente au WordCount local.
Lors de l'implémentation du map, il faut résister à la tentation de faire un premier WordCount avant le shuffle. En faisant une telle opération, on améliore significativement le temps global, mais on sort du paradigme MapReduce.
Pour paralléliser les commandes envoyées aux slaves, nous utilisions initialement la librairie `multiprocessing` qui permet de paralléliser l'exécution de fonctions dans un script Python. En réalité, lorsqu'il s'agit de paralléliser des commandes bash, générer des listes de `subprocess.Popen()` est beaucoup plus rapide. Le package `multiprocessing` nous permet en revanche de paralléliser facilement certaines fonctions des slaves, et de gagner en temps en envoyant plusieurs splits par slave. Le package `multiprocessing` nous permet aussi d'accélérer la dernière partie du reduce sur la machine master, en parallélisant la lecture des fichiers reduce puis en fusionnant des dictionnaires.

file	nmachines	nsplits	time (s)	time w/o network (s)
deontologie	1	5	11.79	2.96
fluvial	1	14	11.97	3.18
forestier	1	3	11.53	3.05
input	1	5	11.78	3.15
sante	4	7	13.90	4.25
warc	20	4	63.04	22.42

TABLE II
MEILLEURE COMBINAISON (NMACHINES, NSPLITS) EN TEMPS D'EXÉCUTION POUR CHAQUE FICHIER. LES TEMPS RÉSEAU SONT IGNORÉS.

On remarque sur la table II qu'à mesure que la taille du fichier en entrée augmente, le nombre de machines optimal augmente également. Notre implémentation finale met environ 55 secondes sur un fichier Common Crawl (la table II présente les meilleurs résultats en ignorant les temps réseau, empiriquement en prenant en compte les temps réseau, notre temps minimal est de 54 secondes). Les deux étapes qui prennent le plus de temps sont en première position l'envoi des splits, et en deuxième position le reduce (qui comprend le reduce distribué, la copie via ssh des slaves vers le master, la lecture en local et le sort). L'envoi des splits se fait via WiFi depuis une machine située sur un autre réseau que celui des slaves. Si ce temps pouvait être réduit (si toutes les machines étaient sur le même réseau, par exemple), on parviendrait certainement à un temps d'exécution plus faible que le temps d'exécution en local. Notre implémentation distribuée finale est presque aussi

rapide que notre implémentation locale.

total files	nmachines	nsplits	time (s)	time w/o network (s)
12	3	4	525.11	396.87
12	4	3	400.94	315.95
12	6	2	309.47	238.66
12	12	1	**231.66**	**167.43**
24	6	4	719.62	479.01
24	8	3	492.57	382.02
24	12	2	396.81	298.04
24	24	1	**331.93**	**227.77**

TABLE III
VERSION BIG : TEMPS TOTAL AVEC OU SANS LES TEMPS RÉSEAU POUR DIFFÉRENTES COMBINAISONS DE NOMBRE DE FICHIERS PAR NOMBRE DE MACHINES.

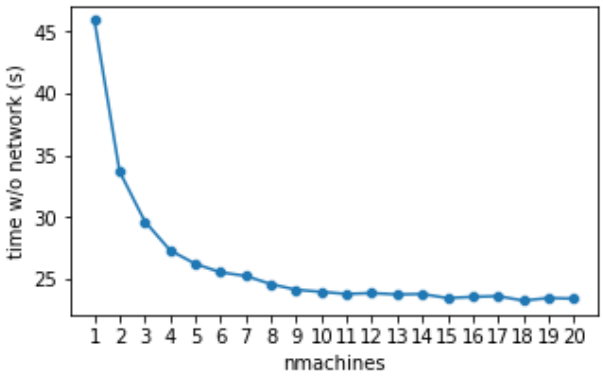

FIGURE 2. Temps d'exécution moyen par nombre de machines du WordCount sur un seul fichier Common Crawl.

C. *Version* BIG

Faire presque aussi bien de façon distribuée qu'en local ne permet pas de mettre en valeur l'intérêt du paradigme MapReduce. Nous avons donc implémenté une nouvelle version du WordCount qui télécharge sur chaque slave un nombre déterminé de fichiers Common Crawl. Afin de pouvoir tester différentes combinaisons de couples (n, m) avec n le nombre de machines et m le nombre de fichiers par machine, on a arbitrairement choisi de travailler sur $k = 12$ fichiers Common Crawl (taille totale environ 4.8 Go), puis $k = 24$ fichiers (taille totale environ 9.7 Go).
L'implémentation de cette version a demandé de nombreuses modifications. D'une part, on abandonne le package `multiprocessing` pour la parallélisation des fonctions des slaves, et on privilégie des boucles. En effet, transposer le script slave initial causait des problèmes de saturation de la mémoire vive des slaves. Ce sont les slaves qui téléchargent eux-mêmes depuis le S3 Common Crawl leurs splits. Le temps de téléchargement et d'extraction est inclus dans la mesure du temps de map. Dans la dernière version, on a réintroduit du parallélisme au sein du reduce. Cette modification pose problème lorsqu'on a plus de 4 fichiers par machine, mais accélère significativement le temps global de l'opération. Il a fallu choisir entre la robustesse et la rapidité.

D. *Remarques diverses et axes de progression*

Chaque exécution du script master utilise son process-id comme identifiant dans l'arborescence des machines slaves afin d'éviter toute collision (on peut lancer plusieurs jobs en parallèle sur les mêmes machines tant qu'on ne rencontre pas de contrainte matérielle, type saturation de mémoire). A la fin de l'exécution du script master, une commande est envoyée vers les machines slaves pour supprimer le dossier du WordCount qui vient de s'achever.
Pour régler définitivement les problèmes de saturation de mémoire, on pourrait calculer la taille des fichiers sur les slaves et choisir en fonction d'utiliser `multiprocessing` ou non.

La gestion des pannes est un aspect essentiel du paradigme MapReduce et de l'écosystème Hadoop. Dans l'état actuel, à part un léger sanity check dans le reduce, les pannes ne sont pas du tout gérées par notre implémentation. C'est une des premières fonctionnalités à implémenter à l'avenir.
Une autre amélioration possible est la conteneurisation de cette application, avec une éventuelle interface web par laquelle un utilisateur pourrait fournir un fichier .txt ou une collection d'URLs. Cette conteneurisation permettrait de surcroît un déploiement sur des machines plus proches qu'actuellement (négation des temps réseau). On se rapprocherait alors de l'idée de YARN, qui repose aussi sur des conteneurs.
Lors des expérimentations qui nous ont permis de générer les tables et figures de ce rapport, nous avons utilisé un système de logs pour vérifier que les outputs restaient cohérents. On pourrait implémenter un check automatique. Pour l'intégralité de nos résultats, chaque combinaison *(nombre de splits, nombre de machines, fichier)* a été testée plusieurs fois puis les temps moyennés pour s'assurer que la variabilité de certaines étapes ne pollue pas nos conclusions.

III. RÉSULTATS

Sauf mention contraire, nous ignorerons certains temps réseau dans nos résultats, en particulier le nettoyage des machines, l'envoi des splits et la réception des reduces. Ces temps étant d'une part variables, et d'autre part dûs à la distance entre la machine master et les slaves, nous avons choisi de les exclure.

A. *Comparaison des implémentations*

La figure 3 compare le temps d'exécution par nombre de machines et nombre de splits par machine de deux implémentations du WordCount distribué sur un seul fichier Common Crawl : l'une utilise la librairie `multiprocessing` dans le script master et l'autre non (dans les deux cas, on parallélise certaines fonctions dans le slave).
Le temps d'exécution semble baisser à mesure que l'on augmente le nombre de splits et le nombre de machines, jusqu'à un certain point. Dans la version précédente, la parallélisation à l'aide de `multiprocessing` dans le master (plutôt que `subprocess.Popen()`) menait à une dégradation des résultats à partir d'un certain nombre de machines/nombre de

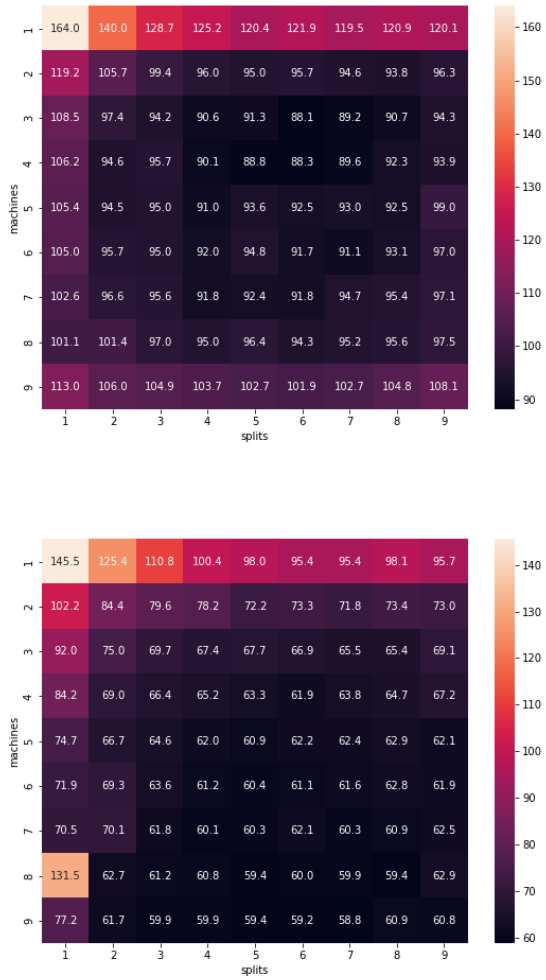

FIGURE 3. Temps d'exécution *avec les temps réseau* de deux implémentations, sur le fichier Common Crawl, par nombre de machines/nombre de splits. Les anomalies sont explicables par la variabilité des temps réseau.

splits. Dans la version finale, on tend vers un temps fixe à mesure que le nombre de machines et de splits augmente (en particulier si on ignore les temps réseau, voir 2). La parallélisation au sein même des slaves a donc son importance : dans le cas où chaque split est traité séquentiellement, la durée globale augmente à mesure qu'on augmente le nombre de splits par machine. C'est ce que montrent les résultats de la version *BIG* (voir III).

B. Loi d'Amdahl

Nous allons tenter de retrouver empiriquement les résultats prévus par la loi d'Amdahl [1]. La table IV nous montre l'évolution du temps de calcul à mesure que le nombre de machines augmente. C'est à partir de cette table que nous allons retrouver la loi d'Amdahl empiriquement. Rappelons la formule vue en cours :

$$S = \frac{1}{1 - P + P/N}$$

nmachines	time (s)	time w/o network (s)	speedup
1	101.15	45.94	1.00
2	75.32	33.67	1.36
3	68.18	29.64	1.55
4	67.91	27.30	1.68
5	63.17	26.20	1.75
6	62.36	25.52	1.80
7	62.14	25.23	1.82
8	64.05	24.56	1.87
9	60.96	24.11	1.91
10	60.64	23.94	1.92
11	60.35	23.77	1.93
12	60.78	23.84	1.93
13	60.26	23.73	1.94
14	63.30	23.76	1.93
15	61.46	23.44	1.96
16	60.12	23.53	1.95
17	59.93	23.60	1.95
18	63.39	23.23	1.98
19	63.52	23.44	1.96
20	63.44	23.39	1.96

TABLE IV
TEMPS MOYEN DU WORDCOUNT PAR NOMBRE DE MACHINES, SPEEDUP EMPIRIQUE.

avec S l'accélération (le speedup), P la portion parallèle du code et N le nombre de processeurs. N est directement proportionnel au nombre de machines, nous confondrons donc ces deux grandeurs. La valeur du speedup empirique est calculée de la façon suivante pour chaque valeur de n le nombre de machines :

$$\hat{S}_n = t_{N=1}/t_{N=n}$$

Nous prendrons pour valeur de P la valeur empirique suivante :

$$\hat{P} = (t_{map} + t_{shuffle} + t_{reduce})/t_{total}$$

soit le rapport entre le temps d'exécution du slave et le temps d'exécution total. La valeur de $\hat{P}$ est calculée à partir du temps de notre WordCount sur une seule machine (c'est la portion du code qui est parallélisée lorsqu'on passe à l'échelle). On trouve $\hat{P} = 0.62437$.

La figure 4 montre que notre courbe empirique se rapproche de la courbe théorique vue en cours. Ici, nous avons moyenné les résultats par nombre de machines, ainsi *l'exécution avec nombre de machines = 1 est déjà en partie parallélisée*, ce qui a des conséquences dans notre calcul de $\hat{S}$. Le speedup théorique explique 98.3% de la variance du speedup empirique.
Afin de ne pas prendre en compte notre "parallélisme dans le parallélisme" (dû à l'utilisation de `multiprocessing`), on fait les mêmes calculs en ne gardant que les exécutions avec nombre de splits = 1 (voir V). Dans ce cas, on trouve $\hat{P} = 0.8$ en reprenant la formule antérieure. La figure 5 montre des résultats similaires à ce qu'on trouvait en premier lieu. La différence entre le speedup théorique et le speedup empirique s'explique selon nous par le fait que la phase de shuffle, prise en compte dans le calcul de $\hat{P}$, n'est pas entièrement parallélisable (temps réseau irréductibles). En jouant avec

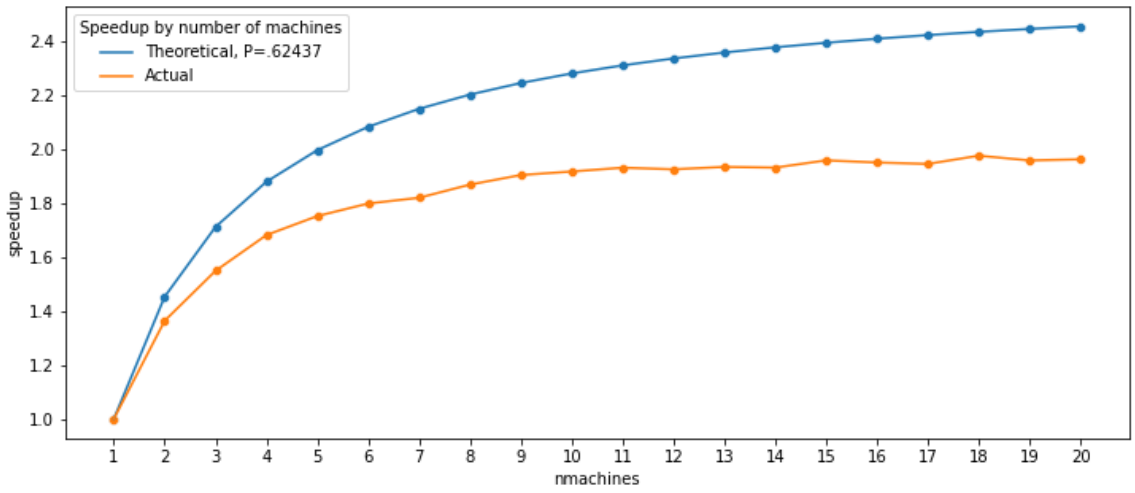

FIGURE 4. Speedup théorique et speedup empirique de notre implémentation par nombre de machines, indifféremment du nombre de splits. En entraînant un modèle de régression linéaire avec pour données l'accélération théorique et pour vecteur à prédire l'accélération empirique, on a un coefficient de détermination $R2 = 0.983$. On peut l'interpréter ainsi : le speedup théorique explique 98.3% de la variance du speedup empirique.

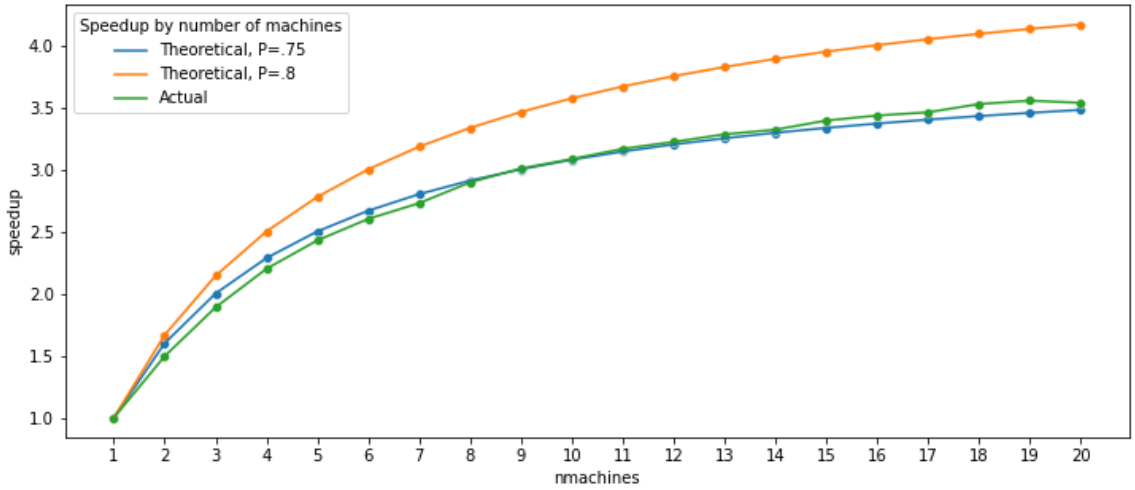

FIGURE 5. Speedup théorique pour $P = 0.8$ (valeur calculée), $P = 0.75$ (valeur trouvée par l'expérience) et speedup empirique de notre implémentation par nombre de machines, avec nombre de splits = 1.

quelques valeurs de P, on trouve que pour $P = 0.75$, la courbe d'accélération théorique et la courbe d'accélération empirique sont graphiquement très proches, mais que la valeur calculée de P explique mieux les variations du speedup empirique que la valeur trouvée graphiquement ($R2 = 0.996$ pour $P = 0.75$, $R2 = 0.999$ pour $P = 0.8$).

IV. CONCLUSIONS

Dans ce rapport, nous avons implémenté un WordCount distribué. Notre implémentation peut prendre en entrée un fichier .txt stocké sur disque, ou une liste d'URLs à partir desquels récupérer des fichiers. Nous montrons l'intérêt de distribuer de telles opérations, notamment lorsque la taille des fichiers d'entrée grandit. Nous montrons enfin que nos résultats sont cohérents avec ce que prévoit la loi d'Amdahl.
Comme le montre [4], l'exploration d'énormes volumes de données Common Crawl à l'aide de MapReduce peut avoir un intérêt scientifique concret. Le fait d'implémenter une application MapReduce *from scratch* nous a permis d'aborder des défis communs mais importants en informatique. Aujourd'hui, les capacités matérielles des machines ont progressé jusqu'à permettre de traiter de grands volumes de données

nmachines	time (s)	time w/o network (s)	speedup
1	145.53	91.33	1.00
2	102.20	61.07	1.50
3	91.97	48.35	1.89
4	84.16	41.53	2.20
5	74.74	37.62	2.43
6	71.88	35.12	2.60
7	70.51	33.49	2.73
8	***131.47***	31.55	2.89
9	77.17	30.37	3.01
10	69.86	29.63	3.08
11	64.29	28.86	3.17
12	64.88	28.36	3.22
13	63.76	27.83	3.28
14	67.84	27.51	3.32
15	63.23	26.92	3.39
16	64.16	26.60	3.43
17	62.40	26.40	3.46
18	64.90	25.91	3.53
19	64.76	25.70	3.55
20	66.03	25.83	3.54

TABLE V
TEMPS MOYEN DU WORDCOUNT PAR NOMBRE DE MACHINES AVEC NOMBRE DE SPLITS PAR MACHINE = 1, SPEEDUP EMPIRIQUE. LA VARIABILITÉ DES TEMPS RÉSEAU EST MISE EN VALEUR PAR LE TEMPS D'EXÉCUTION TOTAL POUR NMACHINES = 8.

en parallèle sur une seule machine. Les GPU, qui reposent sur la parallélisation, ont par exemple permis de démocratiser le Deep Learning et le minage de crypto-monnaies (pour le meilleur et pour le pire). Pour un développeur Python nanti de mémoire, se confronter aux limites matérielles d'une ou de plusieurs machines permet de réaliser à quel point la distribution et l'optimisation des ressources ont encore un intérêt — en particulier pour des applications qui traitent des grandes volumes de données.

REMERCIEMENTS

Merci à Monsieur Sharrock pour la qualité de son cours et de son TP guidé, et pour son accompagnement et ses conseils. Merci à Guillaume, Lucas et Julien pour leurs indications qui m'ont permis de faire des progrès considérables en temps de calcul, notamment comparé à mes premières implémentations.

RÉFÉRENCES

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies : Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, volume 30 of *AFIPS Conference Proceedings*, pages 483–485. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : Simplified data processing on large clusters. In *OSDI'04 : Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[4] Vassilis Kolias, Ioannis Anagnostopoulos, and Eleftherios Kayafas. Exploratory analysis of a terabyte scale web corpus. *CoRR*, abs/1409.5443, 2014.

[5] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark : A unified engine for big data processing. *Communications of the ACM*, 59(11) :56–65, nov 2016.