Ordonnancement dynamique des transferts dans MapReduce sous contrainte de bande passante

Sylvain Gault

ComPAS'2013 : RenPar'21 / SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Ordonnancement dynamique des transferts dans MapReduce sous contrainte de bande passante

Sylvain Gault *

Avalon, Inria, LIP, ENS Lyon,
46 Allée d'Italie
69364 Lyon Cedex 7 - France
Sylvain.Gault@ens-lyon.fr

Résumé
De nombreux domaines scientifiques font désormais face à un déluge de données. L'une des approches proposées pour permettre le traitement de tels volumes est le paradigme de programmation MapReduce introduit par Google. Ce schéma d'exécution très simple se compose de deux phases, *map* et *reduce* entre lesquelles a lieu une phase d'échange massif de données entre les machines exécutant l'application.
Dans cet article, nous proposons un système linéaire définissant un partitionnement des données à traiter et un algorithme d'ordonnancement dynamique des transferts afin d'optimiser cette phase intermédiaire. Nous comparons cette approche à celle reposant sur un programme linéaire et un ordonnancement statique par phases. Les expériences menées montrent que notre approche produit des ordonnancements plus compacts en un temps bien plus court.

Mots-clés : MapReduce, ordonnancement, communications

1. Introduction

La plupart des disciplines scientifiques s'appuient désormais sur l'analyse et la fouille de masses gigantesques de données pour produire de nouveaux résultats scientifiques. Ces données brutes sont produites à des débits sans précédents par divers types d'instruments tels que les séquenceurs d'ADN en biologie, le *Large Hadron Collider* (LHC) ou les grands télescopes tels que le *Large Synoptic Survey Telescope* (LSST) en physique, les scanners haute résolution en imagerie médicale, ou encore la numérisation d'archives en sciences humaines et sociales. Certains domaines de l'informatique produisent et traitent d'immenses volumes de données, par exemple par la multiplication des réseaux de capteurs, l'indexation du Web ou l'analyse des interactions dans les réseaux sociaux.
Ce déluge de données soulève de nombreux défis en termes de stockage et de traitement informatique. Selon le type d'utilisateurs et les traitements à effectuer sur ces données, différentes approches complémentaires peuvent être envisagées. Dans le cas du traitement des données issues du LHC, les utilisateurs sont réunis au sein de collaborations internationales et les traitements effectués sont des simulations de longue durée. Une infrastructure de calcul et de stockage distribuée à l'échelle mondiale, la grille, a donc été mise en place afin de mutualiser l'utilisation des ressources informatiques. A l'inverse, l'entreprise Google a proposé un modèle de programmation afin de traiter les problèmes d'indexation du web au sein de ses propres centres de données. Ce paradigme issu de la programmation fonctionnelle, appelé MapReduce [2], consiste à distribuer le traitement à réaliser sur de nombreux nœuds de calcul ayant accès à l'ensemble des données au travers d'un système de fichiers partagé. Une application MapReduce est généralement composée de deux fonctions de base fournies par l'utilisateur et exécutées en deux étapes, comme le montre la Figure 1.

*. Ces travaux ont été partiellement financés par le projet ANR MapReduce (ANR-10-SEGI-001).

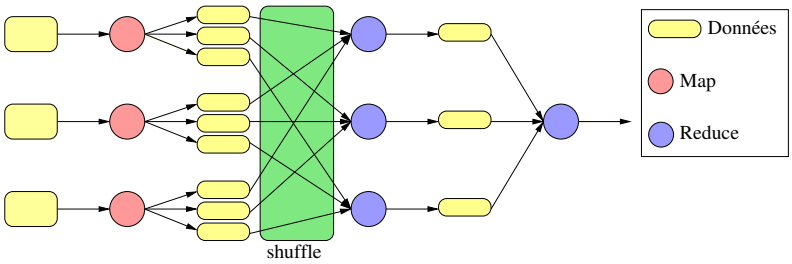


FIGURE 1 – Organisation classique d'une application MapReduce.

L'ensemble des données à traiter est tout d'abord découpé en morceaux qui sont passés en paramètre de la fonction *map*. Une instance de cette fonction est créée pour chaque sous-ensemble de données. L'ensemble des *maps* est ensuite distribué sur les ressources de calcul disponibles. Des données intermédiaires, typiquement des paires clé/valeur, sont produites et transmises aux instances de la fonction *reduce*. Une phase de *shuffle* intervient généralement entre les étapes *map* et *reduce*. Elle consiste à redistribuer les données intermédiaires. Chaque instance *map* va communiquer une partie des données qu'elle a produite à chacune des instances de *reduce*.
L'une des principales motivations à l'origine du paradigme MapReduce est l'ensemble des contraintes relatives à la bande passante disponible au sein d'un centre de traitement de données, tel que celui de Google [2]. Les calculs sont donc effectués au plus proche des données afin de privilégier les accès au stockage local aux transferts de données sur le réseau. Si cette stratégie s'avère efficace en ce qui concerne les données initiales qui sont traitées par la fonction *map*, de nombreux transferts de données sont inévitables lors de la phase de *shuffle*. Cette phase lors de laquelle tous les nœuds de calcul vont s'échanger des données est donc fortement consommatrice de bande passante et va par conséquent être limitée par la capacité du réseau d'interconnexion. De plus, la phase de *reduce* ne peut démarrer qu'une fois que l'ensemble des données transmises par les différents *maps* a été reçu, ce qui crée une barrière de synchronisation forte. Optimiser la phase de *shuffle* par le biais d'un ordonnancement efficace des transferts de données peut donc avoir un impact important sur les performances globales de l'application. Un tel ordonnancement devient encore plus crucial lorsque des phénomènes de congestion peuvent apparaître au sein du réseau d'interconnexion, du fait d'un commutateur sous-dimensionné par exemple. Une définition formelle de ce problème ainsi qu'un algorithme d'ordonnancement des transferts de données ont été proposés par Berlińska et Drozdowski [1].
Dans cet article, nous identifions les limitations de l'algorithme proposé par Berlińska et Drozdowski [1] et proposons un algorithme original d'ordonnancement des transferts de données lors de la phase de *shuffle* d'une application MapReduce. Les objectifs principaux de cet algorithmes sont de garantir la non-congestion du réseau d'interconnexion, de limiter les temps d'inactivité des nœuds de calcul lors de cette phase de communication, et enfin de minimiser le temps de complétion de l'application.
La suite de cet article est organisée de la façon suivante. La section 2 présente divers travaux précédents relatifs à cette problématique. Dans la section 3 nous détaillons les modèles d'application et de plateforme que nous avons utilisé pour concevoir des algorithmes d'ordonnancement et de partitionnement optimisant les transferts de données lors de la phase de *shuffle*. Nous évaluerons ensuite les performances de ces algorithmes dans la section 4. Enfin, nous conclurons cet article et présenterons nos futurs travaux dans la section 5.

2. Travaux connexes

De nombreux travaux se sont intéressés au problème du coût des transferts de données au sein des applications MapReduce. La plupart d'entre eux traitent de la localité des données lors de la phase de *map*. L'un des algorithmes proposés [8] améliore cette localité en introduisant un délai avant de migrer une tâche sur un autre nœud, si le nœud *préféré* n'est pas disponible. L'algorithme BAR [4] vise quant à lui à approcher la distribution optimale des données compte tenu d'une configuration initiale qui sera adaptée dynamiquement.
LEEN [3] est un algorithme de partitionnement des clés intermédiaires qui vise à équilibrer la durée des

reduce tout en essayant de réduire la consommation de bande passante lors du *shuffle*. Cet algorithme repose sur des statistiques de fréquences d'apparition des clés intermédiaires pour tenter de créer des partitions équilibrées et optimiser les transferts de données.
L'algorithme HMPR [5] propose un *pre-shuffling* qui tend à réduire la quantité de données à transférer ainsi que le nombre de transferts. Pour cela, il prédit la partition dans laquelle les données seront générées en sortie du *map* et fait traiter le morceau de données par le nœud qui exécutera le *reduce* de cette partition si cela est possible.
L'environnement d'exécution Ussop [6], ciblant les grilles de calcul, adapte la quantité de données à traiter par chaque *map* en fonction de la puissance de calcul de la machine qui l'exécute. De plus, cet outil tend à réduire les transferts de données intermédiaires en exécutant localement le *reduce* sur la machine ayant généré le plus de clés intermédiaires.
Une application MapReduce peut être considérée comme un ensemble de tâches divisibles puisque les données à traiter peuvent être réparties indifféremment entre les instances de *map*. Il est donc possible d'appliquer des résultats issus de la théorie des tâches divisibles [7] à ce type d'application. C'est l'approche qui a été suivie par Berlińska et Drozdowski [1]. Dans cet article, les auteurs considèrent un environnement d'exécution dont le nombre de nœuds de calcul est supérieur au nombre de communications pouvant avoir lieu simultanément sans provoquer de contention. Pour éviter l'apparition de ce phénomène, ils proposent de modéliser l'exécution d'une application MapReduce par un programme linéaire qui génère une répartition des données et un ordonnancement statique par phases des communications. Si cette approche s'avère intéressante, le recours à un programme linéaire la rend inapplicable pour des instances impliquant plus de quelques centaines de *maps* car le temps de résolution peut parfois dépasser plusieurs minutes. Aussi, il arrive que le solveur de programme linéaire échoue pour certaines instances. L'ordonnancement par phases induit, de plus, un grand nombre de temps d'inactivités sur les machines et le réseau lors du *shuffle*. Dans la section suivante, nous proposons des solutions originales à ces problèmes dans un cadre d'utilisation similaire.

3. Contributions

3.1. Modèles

Les modèles de plate-forme et d'application utilisés dans cet article sont similaires à ceux utilisés par Berlińska et Drozdowski [1] qui ont, en particulier, exprimé les débits en secondes par octet de manière à éviter les divisions dans leurs programmes linéaires. Nous reprenons ici ces unités de manière à rester homogène avec leurs travaux. Nous considérons donc une grappe de machines interconnectées par un unique commutateur. Le réseau ainsi formé est donc en étoile. Les liens connectant les machines au commutateur sont homogènes, sans latence et de débit C exprimé en secondes par octet. Le principal facteur limitant les performances d'une application MapReduce est la capacité du commutateur. Nous considérons en effet que celle ci est bien inférieure à la somme des bandes passantes des liens connectant chaque machine au commutateur. Par souci de simplicité, nous définissons le débit du commutateur comme étant un multiple du débit des liens : $\sigma = C/l$. Autrement dit, le commutateur est capable de servir l communications simultanément sans induire de contention. Au delà, il devient un goulet d'étranglement et les performances de toutes les communications en cours sont dégradées. Enfin, la capacité de traitement des machines est de A_i secondes par octet.
Une application MapReduce est principalement représentée par la quantité de données α_i (en octets) à traiter par chaque *map*. La quantité totale des données à traiter est appelée V, et est égale à la somme des α_i. Nous définissons également γ comme étant le ratio entre la quantité de données passées en paramètre d'un map et le volume de données intermédiaires produites. Lors de la phase de *shuffle*, chaque map devra donc distribuer $\gamma \times \alpha_i$ octets entre les différentes tâches *reduce*. Enfin, notre modèle introduit un délai au démarrage des tâches *map*. Chaque nœud est démarré séquentiellement avec un délai S entre chaque démarrage. Ceci peut être du, par exemple, au temps de chargement du code de l'application sur les nœuds. Pour des raisons de simplification, nous considérons que chaque nœud n'exécute qu'un seul processus *mapper* ou *reducer*.

3.2. Approche de Berlińska et Drozdowski

Dans leur article [1], Berlińska et Drozdowski tentent une optimisation globale du partitionnement et de l'ordonnancement. De manière à éviter de partager la bande passante des liens et éviter la contention,

ils optent pour un ordonnancement par phase de l transferts simultanés et ordonnés. La relation d'ordre entre les transferts peut être définie comme suit.

$$start(i,j) > end(i,j-1) \qquad \text{pour } i \in 1..m, j \in 2..r \tag{1}$$
$$start(i,j) > end(i-1,j) \qquad \text{pour } i \in 2..m, j \in 1..r \tag{2}$$

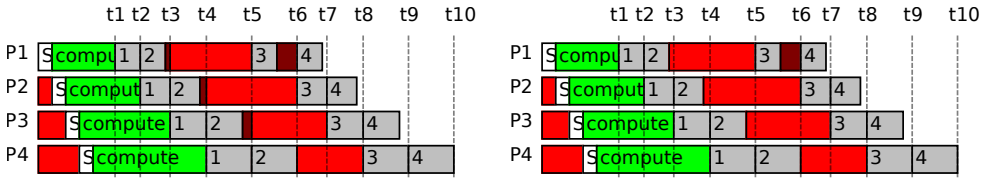
Avec $start(a,b)$ et $end(a,b)$ donnant respectivement la date de début et de fin du transfert du nœud a vers le nœud b, et m et r représentent respectivement le nombre de nœuds exécutant des tâches *map* et des tâches *reduce*.
Chaque processus *map* transférera donc ses données d'abord vers le processus *reduce* 1, puis vers le *reduce* 2, etc. Chaque processus *map* doit attendre que le processus *map* précédent ait terminé son transfert vers le *reduce* j pour commencer lui-même son transfert vers le *reduce* j.

$$itv(i,j) = \left(\left\lceil \frac{j}{l} \right\rceil - 1\right) m + i + (j-1) \mod l \qquad \text{pour } i \in 1..m, j \in 1..r \tag{3}$$
$$vti(i) = \{a | itv(a,b) = i, b \in \{1..r\}\} \tag{4}$$

L'équation (3) définit la fonction itv qui donne le numéro de l'intervalle de temps lors duquel le *mapper* i effectuera son transfert vers le *reducer* j. L'équation (4) définit quant à elle vti comme l'application réciproque de itv. Pour un intervalle i donné, elle fait correspondre l'ensemble des *mapper* qui effectueront un transfert durant cet intervalle. Enfin $itv(r,m)$ correspond au dernier transfert qui sera effectué.

$$\text{minimize } t_{itv(m,r)+1} \tag{5}$$
$$iS + A_i \alpha_i = t_i \qquad \text{pour } i = 1, ..., m \tag{6}$$
$$\frac{\gamma C}{r} \alpha_k \leq t_{i+1} - t_i \qquad \text{pour } i = 1, ..., itv(m,r), k \in vti(i) \tag{7}$$
$$\sum_{i=1}^{m} \alpha_i = V \tag{8}$$

Le programme linéaire ci-dessus (5) – (8) tente de minimiser la date de terminaison du dernier transfert. La contrainte (6) fait en sorte que le premier transfert commence quand le calcul termine. L'inégalité (7) indique que la taille d'un intervalle doit être assez grande pour y placer tous les transferts prévus. Et enfin la somme (8) assure que l'on traite toutes les données.



(a) Résultat produit par l'approche de Berlińska et Drozdowski

(b) Résultat produit par le système linéaire

FIGURE 2 – Comparaison des résultats produits par le programme linéaire de Berlińska et Drozdowski avec le système linéaire proposé et ordonnanceurs identiques.

La figure 2(a) présente un exemple de résultat produit par ce programme linéaire. Y figurent, le temps de démarrage séquentiel (S, en blanc), le temps de calcul des tâches *map* (compute, en vert), les transferts vers chaque processus *reduce* i (en gris), et enfin les temps d'inactivité (en rouge). L'exécution des tâches *reduce* n'est pas représentée ici car elles n'influent pas sur l'ordonnancement produit.

3.3. Partitionnement

On remarque sur la figure 2(a) que, le premier transfert d'un mapper i se termine au moment où le calcul du mapper $i+1$ se termine également, autrement dit :

$$iS + \alpha_i A_i + \frac{\alpha_i \gamma C}{r} = (i+1)S + \alpha_{i+1} A_{i+1} \tag{9}$$

Ceci est vérifié chaque fois que $\alpha_i < \alpha_{i+1}$. De plus, lorsque la plate-forme est homogène (tous les A_i sont égaux), alors $\alpha_i < \alpha_{i+1}$ est équivalent à :

$$S \times r \times m < \gamma \times V \times C \tag{10}$$

Or, le temps de démarrage S (de l'ordre de quelques secondes) est généralement largement inférieur au temps de transfert de l'ensemble des données à traiter (de l'ordre du téraoctet). Sous ces conditions, nous proposons donc de calculer le partitionnement par le système linéaire suivant.

$$iS + \alpha_i A_i + \frac{\alpha_i \gamma C}{r} = (i+1)S + \alpha_{i+1} A_{i+1} \qquad \text{pour } i = 1, ..., m-1 \tag{11}$$

$$\sum_{i=1}^{m} \alpha_i = V \tag{12}$$

Ce système linéaire peut être résolu en temps $O(m)$ et en espace $O(m)$. L'expérience montre que ce système linéaire calcule des α_i similaires au programme linéaire (5) – (8) aux erreurs d'arrondi près. La figure 2(b) illustre le partitionnement produit par ce système linéaire.

3.4. Ordonnanceur

Dans le modèle de plate-forme choisi, il est nécessaire d'éviter la contention sur le commutateur et les liens de communication. Pour cela, nous conservons les contraintes d'ordre (1) – (2). Néanmoins, la limite globale de l transferts simultanés impose de choisir les transferts à effectuer à un instant donné. L'heuristique que nous proposons est de conserver $i+j$ constant pour chaque paire de processus *mapper* i et de processus *reducer* j. En pratique, cela signifie que lorsque le *mapper* 1 effectue un transfert vers le *reducer* 5, le *mapper* 2, s'il n'est pas en train de transférer vers le *reducer* 4, sera *encouragé* à le faire.

L'algorithme 1 présente la stratégie que nous proposons. Dans cet algorithme, `nœud.état` contient la représentation de l'activité actuelle du nœud, et `nœud.cible` contient l'identifiant du *reduce* vers lequel `nœud` est en train de transférer ou veut effectuer son prochain transfert. Lorsque le transfert depuis un *mapper* i vers un *reducer* j se termine, pour chaque *mapper* inactif i' et son prochain *reducer* cible j', nous déterminons $p_{i'} = i' + j'$. Puis nous sélectionnons les nœuds qui minimisent $p_{i'}$. Ces nœuds sont considérés les plus *en retard* et leurs transferts doivent commencer au plus vite. Cela favorise la maximisation de l'utilisation de la bande passante disponible et permet de ne pas violer la contrainte (2) sans la rendre explicite dans l'algorithme.

La procédure `SUR_FIN_CALCUL` est appelée dès qu'une tâche *map* fini de traiter l'ensemble de ses données. Elle appelle la procédure `REQUÊTE_TRANSFERT` qui va démarrer le transfert demandé si cela ne viole pas les contraintes sur l'utilisation de la bande passante. La procédure `SUR_FIN_TRANSFERT` quant à elle, est appelée lorsqu'un transfert termine. Elle commence par lancer le transfert le plus prioritaire s'il existe. Puis elle lance le prochain transfert du nœud dont le transfert vient de terminer, si cela est possible.

Cet algorithme respectes les contraintes posées sur l'utilisation du réseau tout en l'occupant au maximum à chaque instant. En effet, si la bande passante du commutateur est déjà saturée, alors seul le transfert initié par l'appel à `REQUÊTE_TRANSFERT` de la ligne 28 sera effectif, la terminaison d'une communication ne permettant de démarrer qu'un seul nouveau transfert. Par ailleurs, si les contraintes d'ordre (1) et (2) ont empêché des transferts de commencer, le commutateur n'est pas saturé. La terminaison d'un transfert ne permet alors de démarrer qu'au plus deux nouveaux transferts, ce qui est effectué lignes 28 et 32.

Algorithm 1 Algorithme d'ordonnancement des transferts

```
1: procédure REQUÊTE_TRANSFERT(nœud)
2:     si le lien réseau du reducer cible est occupé ou la limite de débit du commutateur est atteinte alors
3:         nœud.état ← INACTIF
4:     sinon
5:         nœud.état ← TRANSFERT
6:         DÉPART_TRANSFERT(nœud, nœud.cible)
7:     fin si
8: fin procédure
9: procédure SUR_FIN_CALCUL(nœud)
10:    REQUÊTE_TRANSFERT(nœud)
11: fin procédure
12: fonction NŒUD_À_RÉVEILLER
13:    pour chaque nœud N en état INACTIF faire
14:        si le lien réseau du reducer cible est occupé alors
15:            passer au nœud suivant
16:        fin si
17:        p[N] ← numéro de N + numéro de N.cible
18:    fin pour
19:    si p est vide alors
20:        retourner la valeur indéfini
21:    sinon
22:        retourner le N dont le p[N] est le plus petit
23:    fin si
24: fin fonction
25: procédure SUR_FIN_TRANSFERT(nœud)
26:    n ← NŒUD_À_RÉVEILLER
27:    si n n'est pas indéfini alors
28:        REQUÊTE_TRANSFERT(n)
29:    fin si
30:    si nœud n'a pas effectué tous ses transferts alors
31:        nœud.cible ← nœud suivant
32:        REQUÊTE_TRANSFERT(nœud)
33:    sinon
34:        nœud.état ← TERMINÉ
35:    fin si
36: fin procédure
```

4. Évaluation

Dans cette section, nous évaluons notre algorithme en le comparant à celui proposé par Berlińska et Drozdowski. Pour cela nous avons développé un simulateur en Perl qui implémente ces deux algorithmes. La table 1 décrit les paramètres relatifs aux plates-formes et applications que nous avons utilisés dans les deux expériences suivantes. Dans ces expériences nous considérons des plates-formes homogènes, de capacité de traitement $A = A_i, \forall i$. Nous supposons également que la quantité de données produite par les *map* est égale à celle traitée, soit $\gamma = 1$.

Dans la première expérience, nous fixons l, le nombre de transferts simultanés ne provoquant pas de contention sur le commutateur et faisons varier le nombre de *mappers* (qui est égal au nombre de *reducers*, i.e., $m = r$). La figure 3 montre le temps d'exécution (en secondes) depuis le lancement de l'application jusqu'à la complétion du dernier transfert en fonction du nombre de processus *map*. D'après les paramètres choisis et l'inégalité (10), nous pouvons déduire qu'au delà de 282 nœuds, la condition $\alpha_i < \alpha_{i+1}$ n'est plus vérifiée. Les temps d'exécution pour moins de 50 nœuds ne sont pas représentés car dans ce cas, la bande passante du commutateur n'est pas limitante et les deux approches donnent le même temps d'exécution.

Nous remarquons que l'approche de Berlińska et Drozdowski donne des temps d'exécution globalement

Paramètres communs			
A	4 secondes par Go	C	8 secondes par Go
S	1 seconde	V	10 To
	Expérience 1		**Expérience 2**
l	50 transferts simultanés		de 50 à 300 transferts simultanés
m et r	de 50 à 300 nœuds		300 nœuds

TABLE 1 – Récapitulatif des paramètres des expériences.

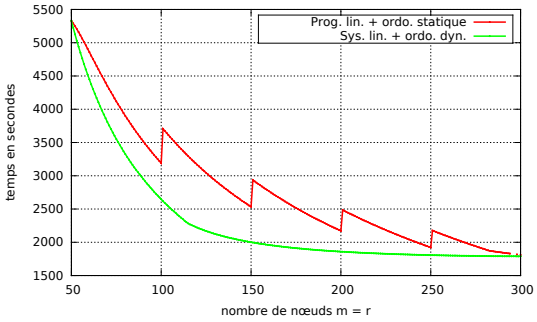


FIGURE 3 – Date de terminaison du dernier transfert en fonction de m et r.

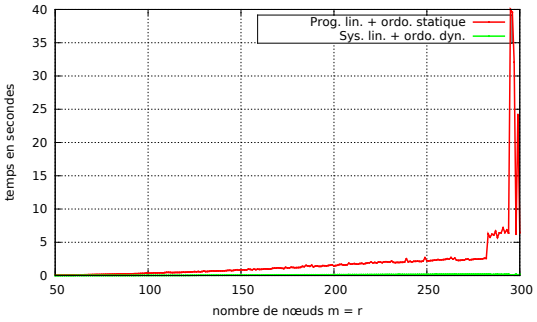


FIGURE 4 – Durée de résolution du partitionnement en fonction de m et r.

décroissants par intervalles. Ces ruptures sont dues à l'algorithme d'ordonnancement des transferts par phases. En effet, lorsque le nombre de transferts à effectuer pour chaque nœud est un multiple de l, alors la bande passante sera totalement utilisée lors de chaque phase. En revanche, s'il y a au moins un nœud supplémentaire, une phase supplémentaire est nécessaire, et celle-ci sous-utilisera les ressources. Ce phénomène est absent avec notre algorithme qui n'impose pas une exécution par phases tout en garantissant la non-congestion du commutateur.

De plus, le programme linéaire et l'ordonnanceur par phase de Berlińska et Drozdowski produisent systématiquement des ordonnancements plus longs que le système linéaire couplé à l'ordonnanceur dynamique que nous proposons. Le gain maximal est obtenu pour 151 nœuds et est de 47%.

La figure 4 présente les temps de résolution respectifs du programme linéaire et du système linéaire dans la configuration décrite par la table 1. Ces temps ont été mesurés sur une machine dotée d'un processeur *Intel core i5* à 2.40GHz et de 3Go de mémoire. La résolution du programme linéaire est effectuée par `lp_solve`. La résolution du système linéaire est implémentée directement dans notre simulateur.

On remarque sur cette figure que le temps de résolution du programme linéaire augmente progressivement jusqu'à 282 nœuds, c'est-à-dire, tant que $\alpha_i < \alpha_{i+1}$. Au delà, le solveur `lp_solve` requière beaucoup plus de temps pour ne pas toujours parvenir à identifier une solution. Sur cette figure, toutes les résolutions prenant plus de 10 secondes ont conduit à un échec. Le système linéaire que nous proposons ne nécessite quant à lui jamais plus de quelques dixièmes de secondes pour déterminer un partitionnement qui permettra un ordonnancement efficace des transferts de données.

Nous avons tenté de poursuivre les mesures jusqu'à $r = m = 1000$. Cependant, le taux d'échec de résolution du programme linéaire est supérieur à 90% au delà de 300 processus, ce qui rend les résultats non-significatifs. Cependant, le système linéaire et son ordonnanceur dynamique conservent tout de même leur comportement asymptotique.

Dans la seconde expérience, nous fixons le nombre de nœuds et faisons varier le un nombre de transferts concurrents sans congestion. Les figures 5 et 6 présentent respectivement le temps d'exécution et le temps de résolution du programme et du système linéaire.

On observe que le temps d'exécution induit par le programme linéaire décroit par segments. Ces discontinuités sont dues à l'algorithme par phases, qui, lorsque l augmente, peut supprimer des phases.

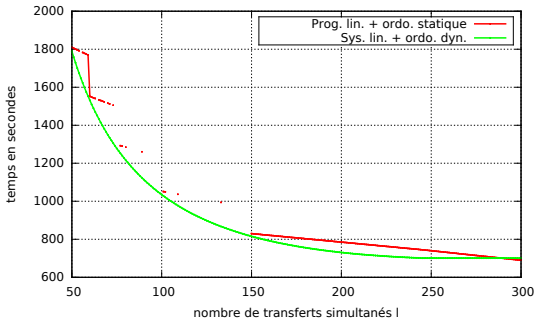


FIGURE 5 – Date de terminaison du dernier transfert en fonction de l.

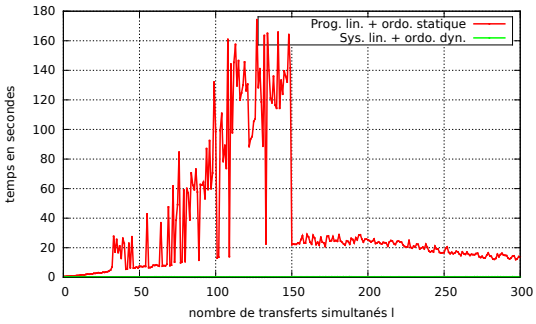


FIGURE 6 – Durée de résolution du partitionnement en fonction de l.

Notre système linéaire avec l'ordonnanceur dynamique conduit à des temps d'exécution décroissants de façon continue. La figure 6 montre le temps résolution du programme linéaire est beaucoup plus lente et très instable. Dans de nombreux cas, excédant plusieurs minutes, la résolution est même impossible. Le temps de résolution du système linéaire est quant à lui négligeable.

5. Conclusion

Dans cet article, nous nous somme attachés à optimiser la phase de *shuffle* d'une application MapReduce. Pour cela nous avons proposé un système linéaire couplé à un ordonnanceur dynamique des transferts. Par rapport à l'approche retenue par Berlińska et Drozdowski fondée sur un programme linéaire et un ordonnanceur statique par phase, nous avons montré de meilleurs temps d'exécution, une construction de l'ordonnancement plus rapide, plus stable et sans échec, ainsi qu'un meilleur passage à l'échelle.
Dans le futur, nous envisageons mettre en application ces algorithmes en production et de les comparer à des solutions existantes, telles que Hadoop. Nous considérons également le cas de plate-formes hétérogènes en termes de capacité de traitement.

Bibliographie

1. Berlinska (J.) et Drozdowski (M.). – Scheduling Divisible MapReduce Computations. *Journal of Parallel and Distributed Computing*, vol. 71, n3, mars 2010, pp. 450–459.
2. Dean (J.) et Ghemawat (S.). – MapReduce : Simplified Data Processing on Large Clusters. *In : Proc. of the 6th Symposium on Operating Systems Design & Implementation (OSDI)*. pp. 137–150. – San Francisco, CA, décembre 2004.
3. Ibrahim (S.), Jin (H.), Lu (L.), Wu (S.), He (B.) et Qi (L.). – LEEN : Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. *In : Proc. of the Second IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. pp. 17–24. – Indianapolis, IN, novembre 2010.
4. Jin (J.), Luo (J.), Song (A.), Dong (F.) et Xiong (R.). – BAR : An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing. *In : Proc. of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. pp. 295–304. – Newport Beach, CA, mai 2011.
5. Seo (S.), Jang (I.), Woo (K.), Kim (I.), Kim (J.-S.) et Maeng (S.). – HPMR : Prefetching and Pre-shuffling in Shared MapReduce Computation Environment. *In : Proc. of the 2009 IEEE International Conference on Cluster Computing (Cluster)*. – New Orleans, LA, septembre 2009.
6. Su (Y.-L.), Chen (P.-C.), Chang (J.-B.) et Shieh (C.-K.). – Variable-Sized Map and Locality-Aware Reduce on Public-Resource Grids. *FGCS*, vol. 27, n6, juin 2011, pp. 843–849.
7. Veeravalli (B.), Ghose (D.), Mani (V.) et Robertazzi (T.). – *Scheduling Divisible Loads in Parallel and Distributed Systems*. – IEEE Computer Society Press, 1996, 292p.
8. Zaharia (M.), Borthakur (D.), Sarma (J. S.), Elmeleegy (K.), Shenker (S.) et Stoica (I.). – *Job Scheduling for Multi-User MapReduce Clusters*. – Rapport technique n UCB/EECS-2009-55, EECS Department, University of California, Berkeley, avril 2009.