Stratégie de découpe de tâche pour le traitement de données massives

Quentin Baert, Anne-Cécile Caron, Maxime Morge, Jean-Christophe Routier

Stratégie de découpe de tâche pour le traitement de données massives

Quentin Baert
quentin.baert@univ-lille1.fr

Anne-Cécile Caron
anne-cecile.caron@univ-lille1.fr

Maxime Morge
maxime.morge@univ-lille1.fr

Jean-Christophe Routier
jean-christophe.routier@univ-lille1.fr

Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

Résumé

MapReduce est un patron de conception permettant de traiter un très grand volume de données distribuées sur un cluster de machines. Ses performances sont liées aux éventuelles distorsions des données. Pour contrer ces biais, nous proposons un système multi-agent adaptatif. Les agents interagissent durant l'exécution et l'allocation dynamique des tâches est le résultats de négociations afin de soulager l'agent le plus chargé et donc le temps global d'exécution. Dans cet article nous montrons comment, lorsqu'une tâche est trop coûteuse pour être négociée, un agent peut la découper afin d'en négocier les sous-tâches.

Mots-clés : *Résolution distribuée de problème, Négociation, Données massives, MapReduce*

Abstract

MapReduce is a design pattern for processing large data sets distributed on a cluster. Its performances are linked to data skews. In order to takle the latter, we propose an adaptive multi-agent. The agents interact during the job and the dynamic tasks allocation is the outcome of negotiations in order to relieve the most loaded agent and so the running time. In this paper, we show how, when a task is too expensive to be negotiated, an agent can split it in order to negotiate its sub-tasks.

Keywords: *Distributed problem solving, Negotiation, Big Data, MapReduce*

1 Introduction

La science des données vise à traiter de grands volumes de données pour y extraire de nouvelles connaissances (en anglais, *insight*). L'analyse de ces données massives nécessite de paralléliser les traitements comme cela peut l'être avec le patron de conception MapReduce [4]. Ce dernier tient son nom des fonctions sur lesquelles il s'appuie : la fonction *map* qui filtre les données et la fonction *reduce* qui les agrège. Le framework le plus populaire pour la distribution de MapReduce est Hadoop mais de nombreuses autres implémentations existent comme le framework Spark ou la base de données NoSQL distribuée Riak construite par Amazon Dynamo.

Le développeur d'une application distribuée qui s'appuie sur le patron de conception MapReduce doit connaître l'implémentation qu'il utilise (Hadoop par exemple) ainsi que les données à traiter et l'environnement d'exécution afin de paramétrer au mieux le job avant son exécution. Les choix d'implémentation et de paramétrage du développeur peuvent être remis en cause par une variation des données ou de l'environnement d'exécution. Dans [9] les auteurs identifient ainsi deux biais récurrents lors de la phase de *reduce*. Ces derniers, appelés biais de partitionnement et biais des clés coûteuses, sont ceux que nous cherchons à résoudre. Ils ont pour même conséquence un déséquilibre dans les charges de travail des reducers : le traitement global du job MapReduce est pénalisé par le reducer le plus chargé.

Nous proposons dans cet article un système multi-agents pour mettre en œuvre le patron de programmation distribuée MapReduce où des agents reducer négocient les tâches pour réduire la charge du reducer le plus chargé et donc le temps d'exécution. En particulier, l'adaptativité des SMA permet de contrer les différents biais de données liés à la phase reduce et ainsi optimiser l'exécution sans prétraitement des données. Nous apportons ici une solution au biais des clés coûteuses : certaines clés regroupent un grand nombre de valeurs. Pour y parvenir nos agents reducers appliquent de manière autonome une stratégie de découpe des tâches volumineuses, reposant sur des critères de décisions locaux, pour leur négociation afin de répartir la charge de travail de manière équitable entre les agents reducers. Ce processus adapte en continu la répartition des calculs à la dynamique du traite-

ment du job. Nous réduisons ainsi l'écart d'effort entre le reducer le plus chargé et celui le moins chargé. Cet équilibrage de la charge de travail permet d'accélérer le traitement du job MapReduce.

La section 2 présente le modèle MapReduce et différents travaux s'attaquant aux biais des reducers. Dans la section 3, après avoir introduit le protocole de négociation, nous présentons notre stratégie de découpe des tâches coûteuses. La section 4 présente des expérimentations qui mettent en évidence l'apport de notre système multi-agents adaptatif pour le traitement de job MapReduce. Finalement, la section 5 conclut et dresse quelques perspectives.

2 Travaux connexes

Les *jobs* MapReduce se composent deux ensembles de tâches, les tâches *map* et les tâches *reduce*, qui sont distribuées sur plusieurs machines. Le modèle de programmation MapReduce s'appuie sur ces deux fonctions fournies par l'utilisateur avec les signatures suivantes :

$$\begin{array}{rl} \textit{map}: & (K1, V1) \rightarrow list[(K2, V2)] \\ \textit{reduce}: & (K2, list[V2]) \rightarrow list[(K3, V3)] \end{array}$$

La figure 1 illustre le flux de données MapReduce tel qu'il est implémenté dans Hadoop :

1. le superviseur partage les données d'entrée en donnant une "tranche" des données à chaque mapper ;
2. les mappers appliquent la fonction *map* sur leur tranche et créent les couples intermédiaires clé-valeur $(key : K2, value : V2)$;
3. une fonction de partitionnement est appliquée sur la sortie des mappers afin de les diviser en sous-ensembles, un sous-ensemble par reducer de sorte que tous les couples avec la même clé soient envoyés au même reducer. La fonction de partitionnement peut être personnalisée afin de spécifier les clés qui doivent être traitées ensemble par le même reducer ;
4. les reducers agrègent les couples intermédiaires clé-valeur pour construire les couples $(K2, list[V2])$; ils exécutent la fonction *reduce* sur chaque groupe de valeurs associé à chaque clé ;
5. les couples finaux clé-valeur $(K3, V3)$ sont écrits dans un fichier du système de fichiers distribué.

Au niveau du système de fichiers, les tranches affectées en entrée des mappers peuvent être de grande taille, et découpées en blocs. De même, à la sortie des mappers, les données liées à une clé sont écrites dans une série de blocs, le reducer doit donc lire pour une clé fixée un groupe de valeurs formé physiquement de plusieurs blocs de données provenant de plusieurs mappers.

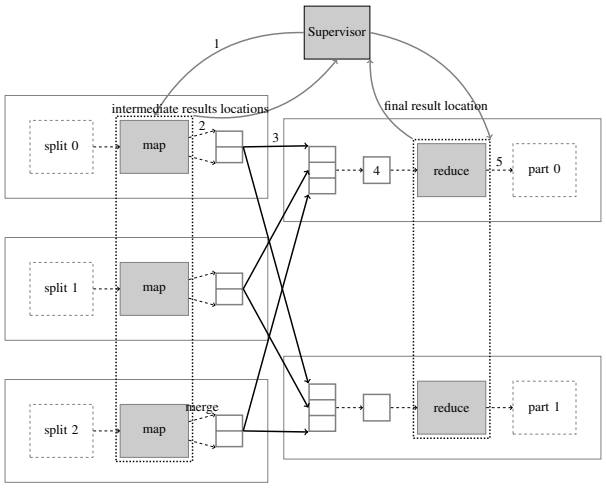


FIGURE 1 – Flux de données MapReduce.

L'application du modèle MapReduce présente le risque de l'occurrence de biais (*skew*) liés aux données, qui viennent pénaliser l'efficacité de l'exécution des calculs. Certains travaux comme [10] et [17] prédisent la performance avec le profil des jobs en recueillant des données lors des exécutions précédentes. Nous ne voulons pas pré-traiter les données et supposons n'avoir aucune connaissance a priori des données d'entrée car le sur-coût computationel nécessaire peut être élevé, en particulier pour des grandes masses de données.

Le biais de partitionnement est lié au problème du déséquilibre qui peut être dû à l'affectation par les mappers des clés aux reducers. Sans connaissance a priori des données la fonction de partitionnement ne peut garantir une répartition équitable des clés et donc des tâches de calcul entre les reducers. Plusieurs travaux ([9, 2, 13]) visent à contrer ce biais. Dans [1] nous adressons également ce problème en adoptant une approche basée sur une allocation dynamique des tâches qui repose sur des négociations entre les agents reducers. A la différence des travaux précédents, notre solution est décentralisée et ne nécessite pas de paramétrage.

Un autre biais, appelé biais des clés coûteuses, est dû aux clés qui regroupent un nombre très important de valeurs, relativement aux autres clés. Cette situation peut par exemple apparaître quand la répartition des données suit une distri-

bution de type Zipf [11] où quelques éléments regroupent un grand nombre de valeurs pendant que de nombreux autres éléments ne sont concernés que par très peu de valeurs. C'est le cas pour le nombre d'offres par mot-clé émises sur la plate-forme marketing de Yahoo ! (cf section 4). Le phénomène d'engorgement des reducers lié à une telle situation a notamment été étudié dans [12]. Ce problème ne peut être résolu par une simple répartition différente des clés entre les reducers, les reducers responsables des clés coûteuses se trouvent nécessairement surchargés.

Si dans [9] ce biais de données est évoqué, la solution proposée par les auteurs ne le résout pas. Dans [7] les auteurs s'attaquent également aux problèmes des biais de données dans MapReduce. Ils proposent une méthode pour mieux répartir la charge de travail entre les reducers et identifient le problème des clés coûteuses. Mais argumentant que le modèle MapReduce exige qu'une même clé soit traitée par un seul reducer, ils estiment que les possibilités d'adaptation du système à ce biais sont très reduites. Leur proposition se limite alors à suggérer que le système alerte l'utilisateur que ses données rencontrent ce problème afin qu'il tente de trouver une solution ad hoc. Dans [13] les auteurs s'attaquent plus concrètement au problème. Ils proposent une solution dans laquelle les sorties des mappers sont découpées en blocs dont la taille doit être paramétrée a priori par le créateur du job. Les valeurs liées à une même clé peuvent ainsi se retrouver réparties dans plusieurs blocs. Les auteurs introduisent alors la notion de tâche de *reduce intermédiaire* qui travaille donc sur un sous-ensemble des valeurs d'une clé pour produire un *résultat intermédiaire*. Ces tâches étant de taille « calibrée » il est plus facile d'en répartir équitablement la charge. Les différents résultats intermédiaires d'une même clé doivent être agrégés lors d'une *phase de reduce finale*. L'approche adoptée est totalement centralisée, un nœud maître regroupe toutes les informations sur les tâches intermédiaires et organise la répartition des tâches sur les différents reducers. De plus la solution repose sur des paramètres que doit définir a priori le créateur du job, ce qui implique qu'il ait une connaissance préalable de la forme des données. Enfin la découpe des tâches en blocs est systématique.

Dans cet article, nous adoptons une approche similaire en découpant les tâches liées aux clés coûteuses en sous-tâches. Ces dernières peuvent ensuite être négociées et donc dynamiquement redistribuées entre les reducers afin d'obtenir une distribution plus équitable de la charge de travail. Cependant à la différence de [13] la découpe des tâches coûteuses n'est réalisée qu'à la demande, lorsqu'elle permet effectivement d'atteindre une répartition plus équitable des tâches. De plus notre mécanisme de découpe ne nécessite pas de définir a priori un paramètre déterminant la taille de la découpe.

Comme expliqué précédemment, le découpage des tâches nécessite de modifier légèrement le patron de conception, en décomposant la phase de reduce en une phase intermédiaire et une phase finale. Plus formellement, on définit maintenant trois fonctions :

$$\begin{aligned} \textit{map}: &\quad (K1, V1) \rightarrow list[(K2, V2)] \\ \textit{IR}: &\quad (K2, list[V2]) \rightarrow (K2, list[V2]) \\ \textit{FR}: &\quad (K2, list[V2]) \rightarrow list[(K3, V3)] \end{aligned}$$

Etant donnée une fonction reduce R, le programmeur doit donc la décomposer en deux fonctions IR (*intermediate reduce*) et FR (*final reduce*) telles que pour toute clé k et les valeurs associées S,

$$R(k, S) = FR(k, < IR(k, S_1); \cdots, IR(k, S_n) >)$$

quelle que soit la partition de l'ensemble de valeurs $S = S_1 \cup \cdots \cup S_n$.

La phase reduce sert à agréger les données. Dans [6], les auteurs identifient trois familles de fonctions d'agrégation : les fonctions distributives où le reduce, le reduce intermédiaire et final sont une seule et même fonction (par exemple `sum`, `min`, `max`, `count` sont distributives) ; les fonctions algébriques qui peuvent être décomposées en utilisant un nombre borné de fonctions distributives (par exemple `avg` va être décomposée en utilisant un reduce intermédiaire calculant une somme (`sum`) et un nombre de valeurs (`count`), la division de la somme totale par le nombre total de valeurs se faisant dans le reduce final) ; et les fonctions holistiques qui ne sont ni distributives ni algébriques (comme la médiane). Il est difficile de trouver une "bonne" décomposition d'une fonction holistique, en particulier de trouver une fonction intermédiaire efficace qui ne nécessite pas de mémoriser trop d'informations. Dans la suite de l'article, nous ne considérons que des fonctions distributives ou algébriques.

D'une manière générale, nous adoptons ici une approche comportementale pour la résolution distribuée de problème, approche qui englobe la stigmergie [18], la résolution distribuée de contraintes [3] ou la négociation [15]. On peut

remarquer que, dans une large partie de la littérature sur les mécanismes de négociation, les agents favorisent leur intérêt individuel. A l'inverse, dans notre contexte de résolution distribuée de problème, les agents ont un but commun qui prime sur leur intérêt individuel. Contrairement à [16], notre résolution ne consiste pas affecter une fois pour toute des paquets de ressources en fonction des préférences des agents mais d'itérer les négociations de tâches individuelles en fonction d'une estimation locale des tâches restant à réaliser. Contrairement à [3], le problème abordé ici est dénué de contraintes d'ordonnancement et les agents n'ont ni des capacités ni des compétences limitées. A notre connaissance, l'unique SMA qui implémente le patron MapReduce s'appuie sur des agents mobiles pour assurer la réplication du code et des données afin de garantir la tolérance aux pannes [5]. Toutefois, ce travail ne met en œuvre aucune technique d'auto-organisation [14] pour que le système multi-agents s'adapte aux données ou à l'environnement informatique. Pour ce faire, nous avons opté ici pour les techniques de négociation multi-agents.

3 Proposition

Notre modèle corrige à ce jour deux biais de données à savoir : le biais de partionnement des clés et le biais des clés coûteuses.

Pour résoudre le biais de partitionnement, nos agents reducers possèdent une architecture complexe et utilisent un protocole de négociation basé sur le Contract Net Protocol afin de se répartir les tâches dynamiquement en cours d'exécution et d'équilibrer leurs charges de travail (ou contributions). Ce protocole a été présenté en détails dans [1]. Nous en rappelons les principes généraux au travers d'un exemple dans la section 3.1. Nous adressons ici le biais des clés coûteuses. Nous présentons un mécanisme de découpe de tâches basé sur les blocs de données qui les composent. Cette division permet aux reducers de partiellement traiter une tâche (grâce notamment à l'introduction d'une fonction de reduce intermédiaire) et d'affiner encore l'équilibre de la charge de travail. La section 3.2 décrit le principe de la découpe de tâche ainsi que les conditions nécessaires pour qu'un reducer amorce cette opération. Les sections 3.3 et 3.4 présentent dans un premier temps une heuristique de découpe de tâche puis la stratégie effective qui utilise l'heuristique précédente en considérant les blocs de données. Lorsqu'une tâche est divisée, elle est traitée par plusieurs reducers. Se pose alors la question de la cohérence des résultats et de la construction du résultat définitif de la tâche. La section 3.5 introduit le rôle de *reducer final* dont l'objectif est d'assurer qu'une tâche reduce divisée et exécutée sur plusieurs nœuds aboutisse finalement à un unique résultat par agrégation des multiples résultats intermédiaires.

3.1 Protocole de négociation

Afin d'illustrer la délégation des tâches par la négociation, nous considérons ici une enchère particulière au sein d'un job MapReduce simple.

Nous supposons que la phase des mappers a été réalisée et que les tâches reduce sont initialement allouées à un ensemble de quatre reducers, $\Omega = \{1, 2, 3, 4\}$. Nous appelons *contribution* d'un reducer la somme des coûts des tâches de reduce qui lui incombent. Nous nous focalisons sur la distribution de tâches à l'instant t telle que les contributions individuelles sont : $c_1(t) = 10$, $c_2(t) = 8$, $c_3(t) = 3$ et $c_4(t) = 5$, où $c_i(t)$ est la contribution de l'agent i au temps t (voir Figure 2a).

Les agents vont engager des négociations dont le but est de diminuer la contribution de l'agent le plus chargé. Afin de diminuer sa contribution, le reducer 1 initie une enchère pour la tâche τ de coût 3 ($c_\tau = 3$) via un cfp (*call for proposal*) aux autres reducers (voir figure 2b). Un cfp inclut la contribution de l'initiateur (ici $c_1(t)$) et le coût de la tâche à négocier (ici c_τ).
Pour déterminer s'il peut gérer la tâche τ au temps $t + 1$, le reducer i ($i \in \Omega \backslash \{1\}$) doit satisfaire le critère d'acceptabilité suivant :

$$c_i(t) + c_\tau < c_1(t)$$

Les reducers qui satisfont ce critère feront baisser la contribution la plus haute s'ils accueillent la tâche τ. Ils font donc une proposition pour τ alors que les autres refusent de participer à la délégation de tâche. Ainsi, le reducer 2 ne peut pas réaliser τ, sinon sa contribution résultante $c_2(t) + c_\tau$ serait plus élevée que $c_1(t)$. Pendant ce temps, les reducers 3 et 4 font des propositions pour τ en envoyant leurs contributions au reducer 1 (voir figure 2c).
Le reducer 1 reçoit donc des propositions des agents $\Omega' = \{3, 4\}$. Il choisit d'attribuer τ à l'enchérisseur le moins chargé, en appliquant le critère de sélection suivant :

$$\underset{j \in \Omega}{\operatorname{argmin}}(c_j(t))$$

De cette manière, le reducer 1 accepte la proposition du reducer 3 et rejette celle du reducer 4 (voir figure 2d).

Après la négociation (à l'instant $t+1$), on observe que :

— la tâche τ appartient maintenant au reducer 3 ;
— les nouvelles contributions sont $c_1(t+1) = 7$, $c_2(t+1) = 8$, $c_3(t+1) = 6$ et $c_4(t+1) = 5$.

En conséquence, le reducer 2 a maintenant la contribution maximale. Toutefois, on peut observer que la contribution du reducer le plus chargé a diminué car nous avons $c_2(t+1) < c_1(t)$. La négociation mène à une allocation plus efficace où la charge est plus équitablement répartie (voir figure 2e).

3.2 Découpe de tâche : amorce et principe

Le biais des clés coûteuses ne peut être résolu ni par une fonction de partition classique ni uniquement par notre processus de négociation (cf. Figure 3). Afin de permettre aux reducers avec des tâches trop coûteuses de faire décroître leur contribution à l'aide d'enchères, nous proposons de diviser ces tâches en sous-tâches moins coûteuses et négociables. Afin de réduire le coût lié aux négociations, le découpage n'est réalisé que si nécessaire. Dans ce but, nos reducers sont construits avec deux fonctions de reduce : la fonction *IR* et la fonction *FR* (voir section 2) et sont ainsi en mesure de traiter une sous-tâche de reduce (grâce à la fonction *IR*) comme une tâche complète (grâce à la fonction *FR*).

Considérons une population de n reducers. Pour que le reducer i puisse débuter une découpe de tâche, il doit remplir des conditions de divisibilité :

1. ne pas être le reducer le moins chargé du système, c.à.d. qu'il existe m reducers ($1 \leq m \leq n-1$) moins chargés que i ;
2. être en état de pause, c.à.d. que selon les croyances de i concernant les contributions des autres reducers de son réseau d'accointances, il ne peut pas déléguer de tâche ;
3. que la plus coûteuse de ses tâches soit au moins composée de deux blocs de données ; une division de celle-ci pouvant ainsi donner au moins deux sous-tâches.

Lors d'une découpe de tâche, l'objectif de i est de diminuer sa contribution. L'idée est de découper la tâche la plus coûteuse de i en $k+1$ sous-tâches de telle sorte que $1 \leq k \leq m$. La répartition des k sous-tâches est négociée avec les m reducers moins chargés que i, de telle sorte qu'à l'issue de ces négociations, aucun ne soit plus chargé que i avant ces négociations.

Les k sous-tâches sont créées de sorte à contenir le même nombre de blocs de données et donc à avoir approximativement le même coût, facilitant ainsi leur future délégation. Ces sous-tâches pourront être également découpées.

3.3 Heuristique de découpe de tâches

L'heuristique de découpe de tâches repose sur les croyances qu'a le reducer des contributions de ses accointances et sur la différence entre sa contribution et chacune de celles-ci.

Définition 1 (Delta de contribution)**.** *Soit $\Omega = \{1, \ldots, n\}$ une population de n reducers. Au temps t, chaque reducer i de contribution $c_i(t)$ possède un vecteur $\vec{r_i} = < r_{i_1}, \ldots, r_{i_{n-1}} > \in \Omega^{n-1}$ de ses accointances par ordre croissant de leur contribution. Soit $c_{i_k}(t)$ la contribution estimée du reducer r_{i_k} (c.à.d. la croyance qu'a i de la contribution du k^{ime} reducer de $\vec{r_i}$). Pour tout reducer $r_{i_k} \in \vec{r_i}$, nous définissons le delta de contribution comme :*

$$\Delta_i^k = c_i(t) - c_{i_k}(t)$$

D'après les conditions de divisibilité, si l'agent i est en situation de découper une tâche, alors il est en état de pause et il existe m reducers moins chargés que lui. Sa tâche la plus coûteuse τ est non négociable et on a : $c_\tau \geq \Delta_i^k, \forall k \in [1; m]$. Ainsi la découpe de la tâche τ, si on ne prend pas en compte les blocs de données des tâches, a pour objectif de déléguer k sous-tâches de même coût. Cette délégation doit permettre de diminuer le plus possible la contribution du reducer le plus chargé.

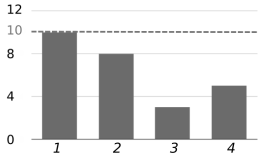
Le reducer i calcule k de la manière suivante :

$$k = \underset{k \in [1;m]}{\operatorname{argmin}}(c_i(t) - \tfrac{k\Delta_i^k}{k+1})$$

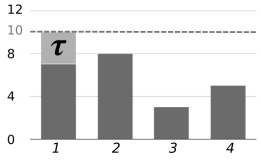
Ce qui amène à la création de $k+1$ sous-tâches $\tau_1, \ldots, \tau_{k+1}$ telles que :

— $c_{\tau_1} = \ldots = c_{\tau_k} = \frac{\Delta_i^k}{k+1}$;
— $c_{\tau_{k+1}} = c_\tau - \frac{k\Delta_i^k}{k+1}$.

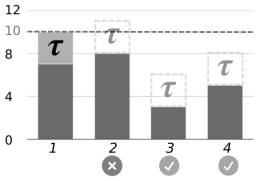
L'exemple suivant illustre comment l'indice k est choisi et l'impact qu'a ce choix sur les contributions du système après négociations.



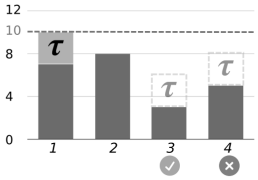
(a) Allocation initiale : le reducer 1 est le plus chargé

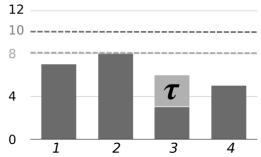


(b) Le reducer 1 propose la tâche τ



(c) Chaque enchérisseur applique le critère d'acceptabilité





(d) Le reducer 1 applique le critère de sélec- (e) Nouvelle allocation après la négociation tion

FIGURE 2 – Processus de négociation pas-à-pas : comment le reducer 1 délègue la tâche τ.

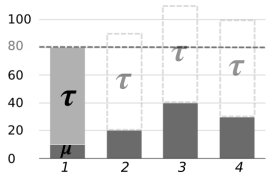


FIGURE 3 – Situation initiale, le reducer 1 ne peut pas négocier la tâche τ.

Exemple. Soit $\Omega = \{1, 2, 3, 4\}$, un ensemble de quatre reducers interconnectés avec les contributions respectives $c_1(t) = 80, c_2(t) = 20, c_3(t) = 40, c_4(t) = 30$. Le reducer 1 possède deux tâches : τ et μ. Puisqu'il est en train d'exécuter la tâche μ, le reducer 1 ne peut initier une enchère qu'avec la tâche τ qui n'est pas négociable (voir Figure 3). Le reducer 1 est donc en état de pause et il existe $m = 3$ reducers moins chargés que lui dans le système. Il peut découper la tâche τ afin de faire baisser sa contribution.

On observe :

- $\vec{r_1} =< 2, 4, 3 >$ (c.à.d. les accointances du reducer 1 dans l'ordre croissant de leur contribution)
- $\Delta_1^1 = c_1(t) - c_2(t) = 60$, $\Delta_1^2 = c_1(t) - c_4(t) = 50$, $\Delta_1^3 = c_1(t) - c_3(t) = 40$

Le nombre de sous-tâches et le nombre d'enchérisseurs pris en compte influencent les contributions résultantes à la division de tâche, et ce n'est pas toujours $k = m$ qui donne la contribution la plus faible pour le reducer le plus chargé. L'exemple que nous décrivons ici en est une illustration.

Si le reducer 1 partage τ avec un seul reducer ($k = 1$), il crée les sous-tâches afin d'équilibrer sa contribution avec celle du reducer le moins chargé. Ainsi, il va découper τ afin d'équilibrer sa contribution avec c_2 au terme d'une négociation. La meilleure découpe pour équilibrer c_1 et c_2 consiste à ne considérer que la partie de coût Δ_1^1 de τ et de la diviser en deux sous-tâches de même coût. Par conséquent, les sous-tâches τ_1 et τ_2 sont créées à partir de τ telles que :

- $c_{\tau_1} = \frac{\Delta_1^1}{2}$
- $c_{\tau_2} = c_\tau - c_{\tau_1} = c_\tau - \frac{\Delta_1^1}{2}$

De cette manière, le reducer 2 est en mesure d'accepter la tâche τ_1 amenant à une situation dans laquelle $c_1(t+1) = c_2(t+1) = 50$ (voir Figure 4).

En procédant au même raisonnement, les situations pour $k = 2$ (voir Figure 4) et pour $k = 3$ (voir Figure 4) donnent des nouvelles contributions différentes :

- pour $k = 2$, $c_1(t+1) = c_4(t+1) = 46$
- pour $k = 3$, $c_1(t+1) = c_3(t+1) = 50$

Plus généralement, après une négociation avec k enchérisseurs, le reducer 1 délègue k sous-tâches de coût $\frac{\Delta_1^k}{k+1}$ et sa nouvelle contribution

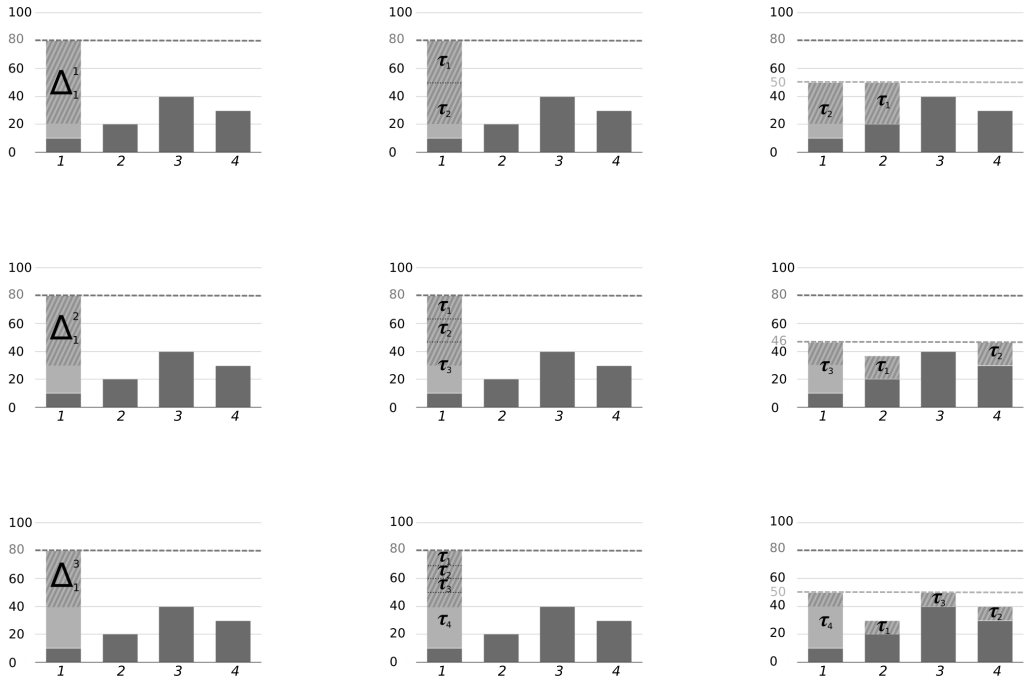


FIGURE 4 – Découpe de la tâche τ entre les reducers 1 et 2 (haut), entre les reducers 1, 2 et 4 (centre) et entre les reducers 1, 2, 3 et 4 (bas).

vaut $c_1(t+1) = c_1(t) - \frac{k\Delta_1^k}{k+1}$. Comme on peut le constater, il existe une valeur de k (ici $k = 2$) qui minimise c_1' :

$$k = \underset{k\in[1;3]}{\operatorname{argmin}}(c_1(t) - \tfrac{k\Delta_1^k}{k+1}).$$

3.4 Découpe effective de tâche

La section précédente présente une création *idéale* des sous-tâches si l'on considère des tâches complètement divisibles. Cependant, les tâches de reduce sont composées de blocs de données et ne peuvent pas être divisées avec une telle précision. Cette section présente la découpe effective de tâche qui repose sur l'heuristique ci-dessus.

Les tâches de reduce sont composées de plusieurs blocs de données dont il faut tenir compte lors de la découpe de celles-ci. Un bloc de données est dit complet s'il n'est pas possible de lui ajouter des données, il est dit partiel sinon. Soient $\mathcal{B}$ la taille d'un bloc et $c_\mathcal{B}$ la contribution des données d'un bloc complet. Les blocs d'une tâche sont initialement créés par les mappers qui produisent pour chaque clé qu'ils traitent, selon leur nombre de valeurs associées, des blocs complets et au plus un bloc partiel (de contribution inférieure à $c_\mathcal{B}$). Une tâche reduce, dont x mappers ont traité la clé, contient donc un certain nombre de blocs complets (la somme des blocs complets construits par les x mappers) et au plus x blocs partiels.

Afin de créer des sous-tâches composées d'au moins un bloc de données, le calcul de k pour un reducer i doit être revu de la façon suivante :

$$k = \underset{k\in[1;N]}{\operatorname{argmin}}(c_i(t) - \tfrac{k\Delta_i^k}{k+1})$$

avec $N = max(\{n \in [1;m] \mid \Delta_i^n \geq c_\mathcal{B}\})$.

Pour le reducer i découper une tâche τ consiste

(i) à déterminer k en utilisant la formule ci-dessus, k étant le nombre de reducers à qui l'on souhaite déléguer une sous-tâche ;

(ii) à diviser les blocs de données de τ pour construire $k+1$ sous-tâches en répétant la routine suivante : parmi les blocs de données non attribués de τ, le bloc le plus coûteux est attribué à la sous-tâche de plus faible coût.

Les $k+1$ sous-tâches ainsi construites sont proches de leur coût idéal (c.à.d. $\frac{\Delta_i^k}{k+1}$ pour k

d'entre elles et $c_\tau - \frac{k\Delta_i^k}{k+1}$ pour la dernière). En fait, pour une sous-tâche τ' de coût idéal $c^*_{\tau'}$ et de coût réel $c_{\tau'}$, on observe que $|c^*_{\tau'} - c_{\tau'}| \leq c_{\mathcal{B}}$.

3.5 Agrégation des résultats intermédiaires

Comme expliqué dans la section 2, le découpage des tâches se base sur la définition de deux fonctions pour la phase de reduce : une fonction de reduce intermédiaire appelée IR et une fonction de reduce final appelée FR. Lorsqu'une tâche de reduce n'est pas découpée, on lui applique directement la fonction FR. Lorsqu'une tâche de reduce est découpée, ses sous-tâches sont marquées comme tâches de reduce intermédiaire, et on doit leur appliquer la fonction IR. Ces sous-tâches sont considérées comme n'importe quelle autre tâche et sont donc candidates à la négociation ainsi qu'au découpage. Les résultats des différentes sous-tâches doivent cependant être regroupés pour permettre d'établir le résultat de la tâche initiale. C'est le reducer initiateur de la première découpe qui a cette responsabilité. Cette information est portée par les différentes sous-tâches.

Considérons par exemple une tâche T affectée au reducer R. Ce reducer décide de découper T en $\{T_1, T_2, T_3\}$, il traite lui-même T_1 avec la fonction IR, et délègue T_2 et T_3 aux reducers R_2 et R_3. Le reducer R_2 décide de découper à son tour T_2 en T_{21} et T_{22}, afin de déléguer T_{22} à un quatrième reducer. Les résultats de l'application de IR sur toutes ces tâches $\{T_1, T_{21}, T_{22}, T_3\}$ sont envoyés à R, le reducer qui a initié la division de la tâche de reduce T qui leur applique la fonction de reduce final FR.

Les résultats intermédiaires sont bien amenés à un même reducer final, quelque soit le nombre de découpes intermédiaires, ce qui préserve la cohérence du résultat final.

4 Expérimentations

Nos expérimentations ont pour objectifs de comparer notre proposition avec la distribution classique du MapReduce et d'évaluer la valeur ajoutée du découpage des tâches, c'est-à-dire de comparer notre SMA à celui proposé dans [1].

Nous avons implémenté notre prototype avec le langage de programmation Scala [1] et la boîte à outils Akka [2]. Cette dernière, en s'appuyant sur le modèle d'acteur [8], nous permet de réduire la distance entre les spécifications du SMA et son implémentation. De plus, grâce à Akka, le déploiement du SMA sur un cluster de PC est simple et direct.

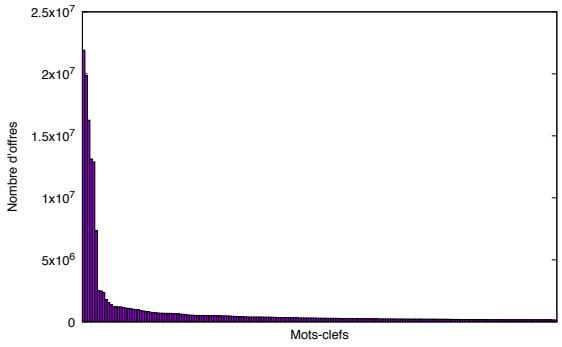


FIGURE 5 – Nombres d'offres par mot-clé.

Yahoo ! utilise un mécanisme d'enchères pour vendre des espaces publicitaires au sein de son moteur de recherche. Par exemple, une agence de voyage veut apparaître aux côtés des résultats de recherche « voyage à Caen ». L'offre de cet annonceur correspond au prix qu'il est prêt à payer chaque fois qu'un internaute clique effectivement sur son annonce. Nous avons analysé le jeu de données correspondant à la période allant du 15 juin 2002 au 14 juin 2003 qui contient $7.7\ 10^7$ offres (journée, identifiant de l'annonceur, liste de mots-clés, etc.), soit environ 8 Go [3]. Attendu que chaque offre porte sur une liste de mots-clés, nous considérons le job qui consiste à comptabiliser le nombre d'offres pour chacun des mots-clés. Comme illustré dans la figure 5, 6 mots-clés parmi 75 359 font l'objet d'un grand nombre d'offres. Ainsi, ce job doit exécuter 6 tâches particulièrement coûteuses.

Nous faisons l'hypothèse que l'équilibre des charges parmi les reducers et a fortiori le découpage des tâches permet de réduire le temps d'exécution de la phase de reduce. À cette fin, nous comparons le temps d'exécution de ces phases dans la distribution classique du MapReduce, celle du SMA proposé dans [1] et celle de notre SMA qui découpe les tâches. Nous avons exécuté le job présenté précédemment avec 10 mappers et 10 reducers. Nous présentons les durées d'exécution en fonction du nombre de machines utilisées, des PCs 3.30GHz Intel(R) Core(TM) i5 avec 4 cœurs et 8 Go de RAM. Pour chaque ensemble de paramètres, nous effectuons 3 exécutions. Comme l'écart-type dû au non-déterminisme de l'ordonnanceur est faible, nous n'exhibons que les moyennes sur les différentes exécutions.

1. http ://www.scala-lang.org/
2. http ://akka.io
3. `http://webscope.sandbox.yahoo.com/`

La figure 6 présente les temps d'exécutions des différentes phases. La figure 7 exhibe l'équité, c.à.d. le rapport entre le temps d'exécution du reducer le moins sollicité et de celui le plus sollicité. Comme on peut le constater le temps d'exécution de la phase de map décroît avec le nombre de machines car elle bénéficie du parallélisme. Quelque soit la méthode utilisée, le temps d'exécution de la phase de reduce décroît globalement. Cette décroissance n'est pas parfaitement proportionnelle au nombre de machines car la phase de reduce est pénalisée par la non-localité des données : un reducer peut avoir besoin de traiter des données issues d'un mapper qui se situe sur une autre machine. De plus, l'approche classique est pénalisée par les biais dus à la forme des données : ne s'adaptant pas à la présence des clés coûteuses, cette approche ne permet pas de réduire l'écart entre l'effort réalisé par le reducer le plus sollicité et ceux qui le sont moins. Selon cette approche, l'équité reste faible comme le montre la figure 7 : le reducer le moins sollicité travaille environ 50% moins que celui qui l'est le plus.

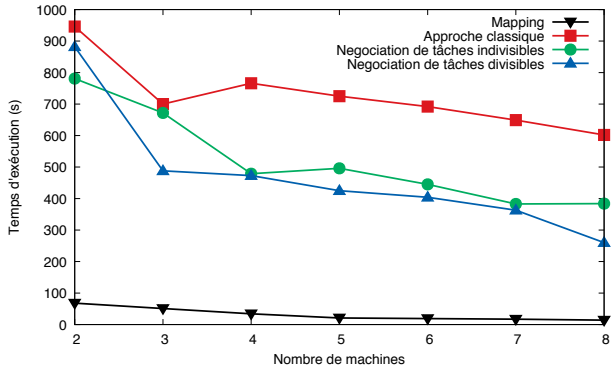


FIGURE 6 – Temps d'exécution des différentes phases.

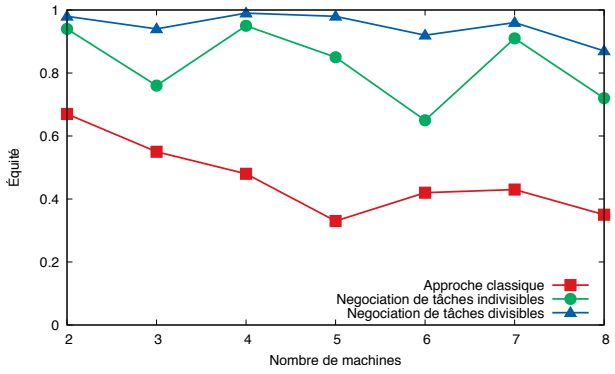


FIGURE 7 – Équité entre reducers.

À l'inverse, les approches basées sur la négociation terminent plus rapidement. Ce gain de temps s'explique par une meilleure exploitation de l'ensemble des ressources disponibles, i.e. une équité proche de 1 qui traduit le fait que le travail lié au job est réparti uniformément sur tous les reducers. En effet, la négociation permet de répartir dynamiquement et en continu les tâches vers les agents les moins chargés. Le jeu de données présentant 6 clés particulièrement coûteuses, dès que l'on atteint ce nombre de machines, l'intérêt de la découpe des tâches est réduit, d'autant que l'équité est déjà proche de 1. La négociation attribue rapidement au plus une clé coûteuse par reducer équilibrant ainsi naturellement les charges de travail même sans avoir recours à la division des tâches. Cependant notre SMA exploitant la division reste plus performant que celui basé uniquement sur la négociation, même si ce gain est parfois faible.

Ces expériences montrent que notre SMA bénéficie du parallélisme plus que les autres implémentations. En résumé, la division combinée à l'équilibrage par la négociation des tâches permet de réduire le temps consacré à la phase de reduce en améliorant l'équité de la charge.

5 Conclusion

Dans cet article nous montrons comment la mise en œuvre du patron de conception MapReduce à l'aide d'un SMA permet de contrer les biais de données pénalisant la phase de reduce, en particulier le biais des clés coûteuses. Notre système ne demande ni de pré-traitement ni de paramétrage dépendant des données. Notre implémentation de MapReduce à l'aide de SMA s'appuie sur des agents reducers qui découpent et négocient leurs tâches au cours de la phase de reduce. Leurs décisions se basent sur la quantité de données restant à traiter par chacun (i.e. la contribution). Cette prise de décision est locale, et ne nécessite pas de centralisation de l'information. Notons que si l'environnement d'exécution est hétérogène, le système s'adapte automatiquement. Nos expérimentations montrent que la division et l'équilibrage des tâches permettent de réduire le temps consacré à la phase de reduce grâce à une réduction des écarts d'effort entre les reducers.

Ces expériences nous amènent à envisager plusieurs perspectives. D'une part, nous souhaitons améliorer le comportement des enchérisseurs pour qu'ils participent simultanément à plusieurs enchères afin de réduire le taux d'échec (actuellement 50 %) et donc réduire le coût communicationnel. D'autre part, nous devons

intégrer un critère de localité des données dans la prise de décision lors des négociations et ainsi réduire le coût de transfert des données. A cette intention, nous envisageons de nous abstraite de notre problématique applicative pour considérer le problème générale de ré-allocation dynamique de tâches entre machines hétérogènes.

Remerciements

Ce travail est soutenu par le défi CNRS MASTODONS. Nous remercions le comité de programme des JFSMA qui, par ses remarques, nous a permis d'améliorer cet article.

Références

[1] Baert, Q., Caron, A.-C., Morge, M., and Routier, J.-C. (2016). Allocation équitable de tâches pour l'analyse de données massives. In *JFSMA*, pages 55–64. Cépaudès éditions.

[2] Chen, Q., Zhang, D., Guo, M., Deng, Q., and Guo, S. (2010). SAMR : A self-adaptive MapReduce scheduling algorithm in heterogeneous environment. In *ICCIT*, pages 2736–2743. IEEE.

[3] Clair, G., Gleizes, M.-P., Kaddoum, E., and Picard, G. (2008). Approches multi-agents auto-organisatrices pour un contrôle manufacturier intelligent et adaptatif. In *JFSMA*, pages 191–200. Cépaduès.

[4] Dean, J. and Ghemawat, S. (2004). MapReduce : Simplified data processing on large clusters. In *SOSDI*, pages 137–150.

[5] Essa, Y. M., Attiya, G., and El-Sayed, A. (2014). Mobile agent based new framework for improving big data analysis. *IJACSA*, 5(3) :25–32.

[6] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H. (1997). Data cube : A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1) :29–53.

[7] Gufler, B., Augsten, N., Reiser, A., and Kemper, A. (2011). Handling data skew in mapreduce. In *ICCCSS*, pages 574–583.

[8] Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245.

[9] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. (2012). Skewtune : Mitigating skew in mapreduce applications. In *ACM SIGMOD ICMD*, pages 25–36.

[10] Lama, P. and Zhou, X. (2012). Aroma : Automated resource allocation and configuration of mapreduce environment in the cloud. In *ICAC*, pages 63–72.

[11] Li, W. (1992). Random texts exhibit zipf's-law-like word frequency distribution. *IEEE Transactions on Information Theory*, 38(6) :1842–1845.

[12] Lin, J. (2009). The curse of zipf and limits to parallelization : A look at the stragglers problem in mapreduce. In *Workshop on Large-Scale Distributed Systems for Information Retrieval*, page 2009.

[13] Liroz-Gistau, M., Akbarinia, R., and Valduriez, P. (2015). FP-Hadoop : efficient execution of parallel jobs over skewed data. *VLDB Endowment*, 8(12) :1856–1859.

[14] Mathieu, P., Routier, J.-C., and Secq, Y. (2002). Principles for dynamic multi-agent organizations. In *PRIMA*, pages 109–122.

[15] Nongaillard, A. and Mathieu, A. (2011). Egalitarian negotiations in agent societies. *AAI*, 25(9) :799–821.

[16] Nongaillard, A., Mathieu, P., and Jaumard, B. (2008). La négociation du bien-être social utilitaire. In *JFSMA*, pages 55–64.

[17] Verma, A., Cherkasova, L., and Campbell, R. (2011). Aria : Automatic resource inference and allocation for mapreduce environments. In *ICAC*, pages 235–244.

[18] Wagner, I., M., L., and Bruckstein, A. M. (1999). Distributed covering by ant-robots using evaporating traces. *IEEE Trans. Robotics and Automation*, 15(5) :918–933.