Allocation équitable de tâches pour l'analyse de données massives

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

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

prenom.nom@univ-lille1.fr

RÉSUMÉ. *De nombreuses entreprises utilisent des applications MapReduce pour le traitement de données massives. Afin d'optimiser l'allocation des tâches, plusieurs systèmes collectent des données à partir des exécutions précédentes et prédisent les performances en faisant une analyse du job. Cependant, ces systèmes ne sont pas efficaces durant la phase d'apprentissage ou quand un nouveau type de tâches ou de données apparaît. Dans cet article, nous présentons un système multi-agent adaptatif pour l'analyse de données massives avec MapReduce. Nous ne prétraitons pas les données mais nous adoptons une approche dynamique où les agents reducers interagissent au cours de l'exécution. Pour réduire la charge de travail du reducer le plus chargé, et donc le temps d'exécution, nous proposons une réallocation des tâches qui s'appuie sur la négociation. Nous prouvons que le processus de négociation se termine et mène à une meilleure répartition des tâches. Nos expérimentations sur des données réelles confirment la valeur ajoutée de la négociation.*

ABSTRACT. *Many companies are using MapReduce applications to process very large amounts of data. In order to optimize the task allocation, several systems collect data from previous runs and predict the performance doing job profiling. However they are not effective during the learning phase, or when a new kind of job or data set appears. In this paper, we present an adaptive multiagent system for large data sets analysis with MapReduce. We do not preprocess data but we adopt a dynamic approach, where the reducer agents interact during the job. In order to decrease the workload of the most loaded reducer - and so the running time - we propose a task re-allocation based on negotiation. We prove that the negotiation process terminates and leads to a better task allocation. Our experimentations over real-world data confirm the added-value of negotiation.*

MOTS-CLÉS : *système multi-agent, résolution distribuée de problème, négociation, données massives, MapReduce.*

KEYWORDS: *multiagent system, distributed problem solving, negotiation, big data, MapReduce.*

DOI:10.3166/RIA.31.401-426

1. Introduction

La science des données vise à traiter de grands volumes de données pour y extraire de nouvelles connaissances (en anglais, *insight*). Comme le potentiel technologique et la demande sociale ont augmenté, de nouvelles méthodes, de nouveaux modèles, systèmes et algorithmes sont développés. L'analyse de ces données, en raison de leur volume et de leur vitesse d'acquisition, demande de nouvelles formes de traitements. À cette intention, le patron de conception MapReduce (Dean, Ghemawat, 2004) est parallélisable et utilisable, par exemple pour mettre en œuvre l'algorithme PageRank, le calcul d'un index inversé, identifier les articles les plus populaires sur Wikipedia ou réaliser le partitionnement en k-moyennes. Le framework le plus populaire pour MapReduce est Hadoop (White, 2015) mais de nombreuses autres implémentations existent comme le framework Spark (Zaharia *et al*., 2012) ou la base de données NoSQL distribuée Riak construite par Amazon Dynamo (DeCandia *et al*., 2007). Dans ces approches, l'extraction des données ainsi que leur traitement sont distribués et exécutés sans échantillonnage.

Les données et les flux d'entrées peuvent faire l'objet de biais, de pics d'activités périodiques (quotidiens, hebdomadaires ou mensuels) ou de pics d'activités déclenchés par un événement particulier. Ces distorsions peuvent être particulièrement difficiles à gérer. Dans les frameworks existants, une affectation efficace des tâches (c.à.d. la répartition des clés) demande une connaissance *a priori* de la distribution des données. À l'inverse, nous défendons la thèse selon laquelle les systèmes multi-agents (SMA) sont particulièrement appropriés pour s'adapter à des données inconnues, à des flux qui évoluent constamment ou à un environnement informatique dynamique. Les systèmes multi-agents constituent un paradigme de premier ordre pour l'analyse, la conception et l'implémentation de systèmes composés d'entités interagissantes ; le traitement de données distribuées avec le patron d'architecture logicielle MapReduce en est un exemple. Un SMA se caractérise par le fait que : (i) chaque agent dispose d'informations incomplètes et de capacités limitées ; (ii) il n'y a pas d'autorité centrale ; (iii) les données sont décentralisées ; (iv) les traitements sont réalisés de manière asynchrone. Dans les SMA, l'autonomie des agents permet au système de s'adapter dynamiquement aux variations et perturbations de l'environnement.

Dans cet article, nous proposons un système multi-agent adaptatif pour mettre en œuvre le patron de conception MapReduce afin d'analyser des données. Le traitement des données est distribué parmi deux types d'agents : (i) les agents mappers qui filtrent les données ; (ii) les agents reducers qui agrègent les données. Afin d'équilibrer la charge de travail des reducers, les tâches sont dynamiquement re-allouées parmi les reducers durant le processus et sans échantillonnage. Pour y parvenir les reducers sont impliqués dans de multiples enchères concurrentes. Nos agents négocient les tâches sur la base de leur charge individuelle afin de faire diminuer celle de l'agent le plus chargé, c.-à-d. celui qui retarde le traitement des données. Nous prouvons que ce processus de négociation se termine et améliore l'équité qui mesure si le traitement est effectué au détriment du plus chargé des agents. Nous avons confronté notre système

multi-agent à des données réelles et nos observations confirment la plus-value des négociations.

Cet article est une version étendue de (Baert *et al.*, 2016). Nous présentons ici une description détaillée du cadre théorique, notamment des exemples illustratifs. Au cours du traitement des données, chaque reducer détermine les tâches exécutées localement et celles déléguées par négociation. Nous proposons et évaluons ici deux stratégies différentes qui le permettent. Enfin, nous présentons plusieurs expériences supplémentaires mesurant l'équité de l'affectation des tâches et l'accélération du traitement due à notre SMA.

Cet article est structuré comme suit. La section 2 introduit le patron de programmation MapReduce qui est le contexte d'application de nos travaux. La section 3 présente les travaux connexes. La section 4 décrit le cœur de notre proposition. Puis nous présentons nos résultats expérimentaux dans la section 5. Finalement, la section 6 conclut et dresse quelques perspectives.

2. MapReduce

Le patron de programmation MapReduce permet de traiter de grands volumes de données (Dean, Ghemawat, 2004). Dans ce modèle, la fonction map filtre les données et la fonction reduce les agrège.

L'implémentation distribuée Hadoop permet aux développeurs sans expérience avec les systèmes parallèles et distribués d'utiliser facilement les ressources au sein d'un tel système. Les jobs MapReduce sont divisés en deux ensembles de tâches, les tâches map et les tâches reduce, qui sont distribuées sur une grappe de PCs. 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 :

1. le superviseur partage les données d'entrée entre les mappers ;

2. les mappers appliquent la fonction *map* sur les données d'entrée 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. une fois qu'un mapper a traité son entrée, il en informe le superviseur ;

5. lorsque tous les mappers ont traité leur entrée, le superviseur informe les reducers pour lancer la seconde phase ;

6. les reducers agrègent les couples intermédiaires clé-valeur pour construire les couples $(K2, list[V2])$;

7. les reducers prennent les couples $(K2, list[V2])$ et exécutent la fonction *reduce* sur chaque groupe de valeurs associé à chaque clé. Par la suite, un tel groupe est appelé **groupe de clés** ;

8. les couples finaux clé-valeur $(K3, V3)$ sont écrits dans un fichier du système de fichiers distribué ;

9. enfin, les reducers informent le superviseur de la localisation du résultat final. Lorsque tous les reducers ont terminé leurs tâches, le travail est terminé.

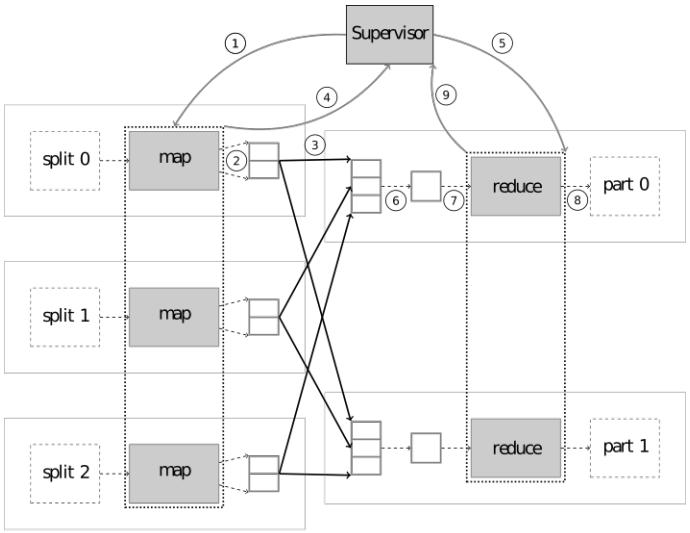


Figure 1. Flux de données MapReduce

Qu'il soit réalisé par une fonction par défaut ou par une fonction spécifique, le partitionnement est fixé *a priori*. Par exemple, pour déterminer quel reducer traite une clé, la partition Hadoop effectue un modulo sur le nombre de reducers :

```
key.hashCode() % numberOfReducers.
```

Cette fonction est fixée sans connaissance *a priori* sur les données. Ainsi la charge de travail n'est pas nécessairement répartie uniformément entre les reducers.

Nous avons réimplémenté et expérimenté le patron MapReduce avec un jeu de données météorologiques [1]. Ce jeu contient plus de 3 millions d'enregistrements (identifiant de station, horodatage, température, pluviométrie, ...) issus de 62 stations au

1. https://donneespubliques.meteofrance.fr/?fond=produit&id_produit=90&id_rubrique=32g

cours des 20 dernières années. Nous avons écrit un job simple qui compte le nombre d'enregistrements par demi-degré de température :

1. la fonction *map* lit un enregistrement ($K1$) et renvoie un couple ($K2, V2$)=`(Temp,1)` où `Temp` est la température arrondie à son demi-degré le plus proche pour la station et l'horodatage correspondant ;

2. la fonction *reduce* somme pour une clé donnée `Temp` tous les compteurs (toujours 1) fournis par la fonction *map* dans `s` pour produire ($K3, V3$)=`(Temp,s)`.

La figure 2 montre que, même si le job est réalisé et le résultat obtenu est celui attendu, la charge de travail des reducers observée après une exécution avec la fonction de partitionnement par défaut d'Hadoop est inéquitable car seulement un quart des reducers a travaillé pendant que les autres reducers étaient inactifs tout au long du processus. Le job aurait pu être terminé plus tôt si tous les reducers avaient été sollicités. Une autre partition peut conduire à un meilleur partitionnement des clés, mais sans connaissance des données, cette observation est nécessairement *ex post facto*.

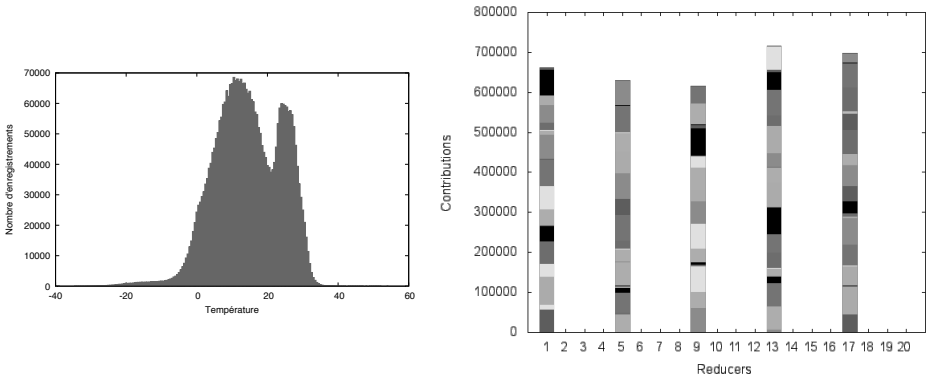


Figure 2. Le nombre d'enregistrements par demi-degré de température (à gauche) et le nombre de valeurs traitées par reducer avec la fonction de partition par défaut d'Hadoop (à droite)

Outre ce problème de partition de données, les nœuds qui exécutent un job MapReduce (et donc les reducers) n'ont pas nécessairement des performances homogènes. Leurs performances peuvent varier durant l'exécution. En conséquence, le temps de calcul est déterminé par le nœud le plus chargé voire le plus lent.

Notre but ici est de résoudre ce problème lié aux biais des données (ou à l'hétérogénéité des performances) en réaffectant dynamiquement les tâches des reducers pendant le traitement. Notre proposition ne dépend ni de la fonction de partitionnement initiale, ni de connaissances sur les données. Notre framework est basé sur un SMA où les reducers sont des agents négociant des tâches pendant le traitement. Ces négociations s'appuient sur le *Contract Net Protocol* (Smith, 1980).

3. Travaux connexes

Dans cet article, nous nous concentrons sur la négociation multi-agent afin de réduire le temps de calcul de la phase reduce (c'est-à-dire la charge du reducer le plus chargé). Premièrement, nous examinons les approches existantes pour le partitionnement des tâches dans la phase reduce. Deuxièmement, nous mettons en exergue les avantages d'un SMA adaptatif, en particulier la négociation multi-agent.

Nous présentons ici différents travaux qui ont étudiés l'optimisation de la phase de reduce. Plusieurs critères sont à considérer pour comparer ces travaux à notre proposition, à savoir :

1. la non nécessité d'avoir une connaissance préalable des données. En considérant la taille des données, il n'est pas envisageable de les lire et les traiter deux fois ;

2. la gestion des biais de données, c.à.d. un mauvais équilibrage de la charge de travail lié à une mauvaise répartition des tâches de reduce à la sortie des mappers (cf. section 2) ;

3. la dynamicité de la solution. Un processus dynamique est à même de réagir à l'évolution des performances des nœuds ;

4. la décentralisation du processus. Un processus décentralisé est plus réactif, moins couteux et moins fragile qu'un processus centralisé ;

5. le faible paramétrage nécessaire. Avoir un processus fortement paramétré implique de connaitre les données et l'environnement informatique. Sans lecture préalable des données (qui est pour rappel très coûteuse), une phase d'affinement des paramètres semble indispensable avant d'obtenir un système correctement calibré. De plus, une fois les paramètres trouvés pour un job, rien n'assure qu'ils soient adaptés à un nouveau traitement ou à de nouvelles données.

Le tableau 1 compare notre solution aux travaux existants à l'aide de ces critères.

Comme indiqué dans la section 2, le partitionnement des clés par Hadoop (Dean, Ghemawat, 2004) est fixé et statique. Par contre, (Lama, Zhou, 2012) et (Verma *et al.*, 2011) prédisent la performance avec le profil des jobs en recueillant des données lors des exécutions précédentes. En fait, nous ne voulons pas prétraiter les données, par exemple avec une phase d'apprentissage automatique utilisant un échantillon de données. En effet, nous supposons n'avoir aucune connaissance *a priori* des données d'entrée car le surcoût computationel nécessaire peut être élevé, en particulier pour des grandes masses de données.

SAMR (Chen *et al.*, 2010) est un algorithme d'ordonnancement qui utilise l'historique des exécutions précédentes pour identifier les nœuds lents. De cette façon, cet ordonnanceur centralisé attribue les tâches en fonction de la performance des nœuds. Cette approche pénalise les premières exécutions d'un même job, et l'historique est difficilement exploitable pour un nouveau job. Nous préférons une approche dynamique où les reducers sont adaptatifs et interagissent pendant le job. Notre SMA est

intrinsèquement adaptatif vis-à-vis du calcul car les négociations sont réalisées au cours du traitement des données.

(Kwon *et al*., 2012) identifient quatre biais dans la mise en œuvre du patron MapReduce : deux au cours de la phase de mapping et deux autres pendant la phase reduce (le biais de partitionnement et celui du groupe de clés coûteux). Comme les auteurs étudient des situations de déséquilibre entre les mappers ou les reducers, SkewTune (Kwon *et al*., 2012) a pour objectif d'atténuer les biais de partitionnement dus à une distribution inégale des données. Quand un nœud est disponible après avoir terminé sa tâche, SkewTune identifie le reducer le plus chargé et répartit ses données non traitées parmi tous les reducers. Notre approche est semblable à SkewTune et nous n'abordons ici que le biais de partitionnement. Cependant, le partitionnement est, dans notre approche, le résultat d'un choix collectif de reducers et nous n'attondons pas qu'un reducer soit inactif pour remettre en cause la répartition des tâches. En outre, nous pensons que la négociation multi-agent convient pour traiter le partitionnement d'un ensemble de valeurs associées à la même clé (c'est-à-dire la gestion du biais du groupe de clés coûteux par division de tâches). Nous reportons à des travaux futurs la prise en compte des autres biais.

FP-Hadoop (Liroz-Gistau *et al*., 2015) traite ces deux biais de la phase de reduce avec un processus centralisé de division des tâches. Les auteurs introduisent une nouvelle phase intermédiaire qui parallélise la phase de réduction d'une clé. Cette phase intermédiaire est gérée par le superviseur avec une structure de donnée centralisée et sa configuration nécessite une connaissance *a priori* des données. Notre objectif est de proposer un processus entièrement décentralisé où les décisions sont locales. Un tel processus décentralisé est plus adaptatif car il permet d'obtenir un processus d'allocation plus réactif qu'un processus centralisé. De plus, il gère un environnement hétérogène et nécessite moins de paramétrage de la part de l'utilisateur.

L'optimisation de la phase de mapping telle qu'elle est étudiée dans (Vernica *et al*., 2012) est hors de la portée de cet article, complémentaire à notre approche et elle pourrait être mise en œuvre par un SMA.

Comme le tableau 1 le résume, notre contribution est dynamique, décentralisée, sans connaissance *a priori* des données ni historique et elle ne nécessite pas de paramétrage dépendant des données.

Tableau 1. Différentes approches pour le partitionnement des tâches

	Hadoop	SAMR	SkewTune	FP-Hadoop	Ici
Pas de conn. *a priori*	✓	✗	✓	✓	✓
Dynamique	✗	✗	✓	✓	✓
Gestion des biais	✗	✓	✓	✓	✓
Décentralisation	✗	✗	✗	✗	✓
Paramétrage faible	✗	✓	✓	✗	✓

Dans notre approche, l'allocation dynamique des tâches repose sur une négociation entre reducers. La théorie du choix social offre des méthodes pour concevoir et analy-

ser des décisions collectives en combinant les préférences ou biens-êtres individuels. Le calcul du choix social est souvent considéré comme un problème d'optimisation qui se résout par une approche centralisée (une enchère) où les agents divulgent leurs préférences au commissaire-priseur central et omniscient qui détermine l'allocation en conséquence (Brandt *et al.*, 2016). Cependant, une telle approche comporte des limitations qui nous semblent trop fortes : (i) il peut être trop coûteux de collecter et traiter toutes les informations sur une seule et même machine ; (ii) si les données évoluent au cours du processus de résolution, il doit être redémarré pour tenir compte des nouvelles données ; (iii) elle suppose que les agents sont complètement connectés sans restriction et qu'ils peuvent tous communiquer les uns avec les autres. Nous traitons ici les points (i) et (ii) en utilisant plusieurs enchères distribuées et simultanées auxquelles participent des agents adaptatifs qui prennent des décisions basées sur des informations locales. Comme nous considérons des agents complètement connectés, l'inconvénient (iii) demeure l'une de nos préoccupations. Cependant, nous pouvons facilement adapter le réseau d'accointances des agents pour construire des groupes qui négocient indépendamment les uns des autres. Typiquement dans un système distribué, le coût de communication dépend de la topologie du réseau, c'est-à-dire des contraintes physiques. Une solution consiste à adapter les groupes au réseau physique et donc répartir physiquement les négociations.

On peut remarquer que, dans une large partie de la littérature sur les mécanismes de négociation, les agents favorisent leur intérêt individuel. Par exemple, (Damamme *et al.*, 2015) suppose que les agents sont individuellement rationnels. À l'inverse, dans notre contexte de résolution distribuée de problème, les agents ont un but commun qui prime sur leur intérêt individuel. Concrètement, dans notre travail, les agents ne sont pas individuellement rationnels car ils peuvent accepter de nouvelles tâches qui ont un coût.

4. Proposition

Nous souhaitons diminuer la charge de travail du reducer le plus chargé pour clore au plus tôt la phase reduce [2]. Dans ce but, nous réallouons dynamiquement les tâches par des négociations multi-agents et sans orchestration centralisée.

Dans cette section, nous présentons le cœur de notre proposition. Nous commençons par une vue d'ensemble de notre proposition et illustrons la négociation par un exemple. Ensuite, nous présentons l'architecture de notre agent reducer, et introduisons les différents protocoles d'interaction et les comportements des agents impliqués. Finalement, nous présentons quelques résultats formels à propos de notre processus de réallocation.

2. Le problème de tolérance aux pannes est hors de la portée de cet article. Concrètement, nous supposons que les tâches map/reduce peuvent être ralenties par des éléments exogènes mais elles ne sont ni mises en échec ni abandonnées.

4.1. Vue d'ensemble

Notre contribution vise à fournir un partitionnement équilibré des tâches. Dans ce but, nous proposons une réallocation des tâches basée sur des décisions locales où chaque reducer est incarné par un agent. Cet agent est caractérisé par l'ensemble des tâches qu'il doit exécuter. Nous supposons que chaque tâche possède un coût (intrinsèque) et que tous les agents possèdent les mêmes fonctionnalités. Par conséquent, tous les agents estiment leur propre contribution comme la somme des coûts des tâches qui leur reste à réaliser.

DÉFINITION 1 (Allocation de tâches / Contribution). — *Pour un ensemble donné $\mathcal{T}$ de m tâches $\tau_1, \ldots, \tau_m$ avec leurs coûts associés $c_{\tau_1}, \ldots, c_{\tau_m}$ et une population $\Omega = \{1, \ldots, n\}$ de n agents reducers, une* allocation *$\mathcal{A}$ de tâches est représentée par une liste ordonnée d'ensembles de tâches disjoints deux à deux $\mathcal{T}_i \subseteq \mathcal{T}$ (avec $\biguplus \mathcal{T}_i = \mathcal{T}$) qui décrivent l'ensemble des tâches détenues par chaque agent i :*

$$\mathcal{A} = [\mathcal{T}_1, \ldots, \mathcal{T}_n] \text{ avec } 1, \ldots, n \in \Omega \tag{1}$$

La contribution *de l'agent i à l'instant t au sein de l'allocation $\mathcal{A}$ est définie telle que :*

$$c_i^{\mathcal{A}}(t) = \sum_{\tau \in \mathcal{T}_i} c_\tau + w_i(t) \tag{2}$$

où $w_i(t)$ est le coût estimé de la tâche courante réalisée par l'agent i. Avant la phase reduce, $w_i(0) = 0$.

La phase des mappers ne diffère pas du modèle classique MapReduce. Les mappers fournissent aux reducers des couples intermédiaires clé-valeurs. Toutefois, les mappers ajoutent l'information sur le coût de la tâche pour ces valeurs (partielles). Comme nous ne supposons aucune connaissance *a priori* sur les données, la méthode de répartition par défaut est ensuite utilisée pour effectuer la première allocation aux reducers.

Les reducers reçoivent leurs paires $(K2, liste[V2])$ et débutent leur travail. Simultanément, la phase de négociation commence afin de diminuer la contribution du plus chargé des reducers et donc de terminer au plus tôt la phase de reduce. Les agents reducers communiquent entre eux pour négocier la tâche à déléguer. En fait, ils sollicitent leurs pairs avec un appel à proposition - en anglais, *call-for-proposal* (cfp)- afin d'alléger leurs propres contributions. Un message cfp contient le coût de la tâche à négocier ainsi que la contribution actuelle de l'initiateur.

Un reducer enchérit afin de prendre la responsabilité de la tâche et donc diminuer la pire contribution. Un enchérisseur fait une proposition si et seulement si, après le transfert de tâches, la pire contribution est inférieure à la pire contribution initiale. Formellement, sa décision est fondée sur le critère local suivant :

DÉFINITION 2 (Critère d'acceptabilité). — *Soit $\mathcal{A}$ une allocation de tâches sur n agents à un instant t. L'enchérisseur j acceptera le transfert d'une tâche $\tau \in \mathcal{T}_i$ sollicité par le commissaire-priseur i ssi :*

$$c_j^{\mathcal{A}}(t) + c_\tau < c_i^{\mathcal{A}}(t) \tag{3}$$

Autrement dit, un participant accepte d'être impliqué comme enchérisseur dans une négociation si, en cas de négociation réussie, sa contribution est strictement inférieure à la contribution initiale du commissaire-priseur. Ainsi, pour les deux agents impliqués, la plus grande contribution après le transfert est plus petite que la plus grande contribution avant cela.

Réciproquement, le commissaire-priseur peut recevoir plusieurs offres pour son cfp. Une offre comprend la contribution du sous-traitant potentiel. Le commissaire-priseur sélectionne alors parmi les enchérisseurs l'agent avec la plus petite contribution. Formellement,

DÉFINITION 3 (Critère de sélection). — *Soit $\mathcal{A}$ une allocation de m tâches $\mathcal{T}$ entre n agents dans Ω à un instant t. Si l'enchérisseur i souhaite déléguer la tâche τ et il a reçu des propositions d'agents de $\Omega' \subset \Omega$, il sélectionne :*

$$\operatorname{argmin}(\{c_j^{\mathcal{A}}(t) \mid j \in \Omega'\}) \tag{4}$$

De cette façon, le transfert de tâche permet de charger le reducer le moins chargé afin d'équilibrer autant que possible la charge de travail. Il est important de noter que l'évaluation du critère de décision pour le transfert de tâches requiert uniquement des informations locales.

Les reducers envoient des cfp tant que leur cfp précédent n'a pas été rejeté par l'ensemble de ses interlocuteurs. Le protocole assure que, lorsque les négociations sont terminées, il n'y a plus aucun transfert de tâches qui pourrait mener à une baisse de la contribution la plus haute : le fardeau du plus chargé des agents ne peut pas être allégé. Comme nous le verrons dans la section 4.5, un reducer reprend l'envoi de cfp quand il acquiert la connaissance que certaines de ses accointances peuvent enchérir.

4.2. Exemple

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 de mapping a été réalisée et que les tâches reduce sont initialement allouées à un ensemble de quatre reducers, $\Omega = \{1, 2, 3, 4\}$. Nous nous focalisons sur l'allocation à l'instant t ($\mathcal{A} = [\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4]$) telle que les contributions individuelles sont : $c_1^{\mathcal{A}}(t) = 10$, $c_2^{\mathcal{A}}(t) = 8$, $c_3^{\mathcal{A}}(t) = 3$ et $c_4^{\mathcal{A}}(t) = 5$ (voir figure 3(a)).

Afin de diminuer sa contribution, le reducer 1 initie une enchère pour la tâche $\tau \in \mathcal{T}_1$ avec $c_\tau = 3$ (voir figure 3b). Ce reducer envoie un cfp qui inclut sa contribution

$c_1^{\mathcal{A}}(t)$ et le coût de la tâche c_τ. Comme le reducer 1 a la contribution maximale, cette enchère peut réussir.

Pour répondre, les autres reducers doivent décider s'ils souhaitent réaliser la tâche τ. En utilisant les critères d'acceptabilité (voir définition 2) chacun de ces reducers choisit de faire une proposition pour τ ou de refuser cette délégation de tâche. Le reducer 2 ne veut pas réaliser τ, sinon sa contribution résultante $c_2^{\mathcal{A}}(t) + c_\tau$ serait plus élevée que $c_1^{\mathcal{A}}(t)$. Pendant ce temps, les reducers 3 et 4 font des propositions pour τ en envoyant leurs contributions au reducer 1 (voir figure 3c).

Le reducer 1 doit maintenant sélectionner l'enchérisseur avec la contribution la plus faible. En utilisant le critère de sélection (voir définition 3), le reducer 1 accepte la proposition du reducer 3 et rejette celle du reducer 4 (voir figure 3d).

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

– l'allocation est $\mathcal{A}' = [\mathcal{T}_1 \setminus \{\tau\}, \mathcal{T}_2, \mathcal{T}_3 \cup \{\tau\}, \mathcal{T}_4]$;

– les contributions sont $c_1^{\mathcal{A}'}(t+1) = 7$, $c_2^{\mathcal{A}'}(t+1) = 8$, $c_3^{\mathcal{A}'}(t+1) = 6$ et $c_4^{\mathcal{A}'}(t+1) = 5$.

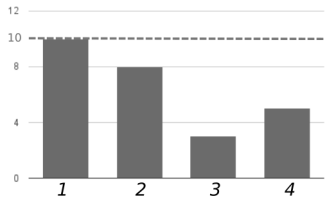
En conséquence, le reducer 2 a maintenant la contribution maximale. Toutefois, on peut observer que la contribution du reducer le plus chargé a diminué car nous avons $c_2^{\mathcal{A}'}(t+1) < c_1^{\mathcal{A}}(t)$. La négociation mène à une allocation plus efficace où la charge est plus équitablement répartie (voir figure 3e).

4.3. Architecture de l'agent reducer

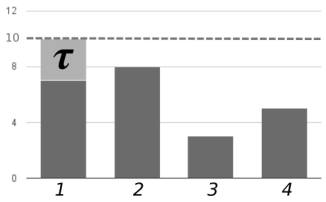
Nous considérons ici le modèle d'acteur par passage de message asynchrone proposé par (Clinger, 1981) pour la programmation concurrente. Inspiré par (Hewitt *et al.*, 1973), nous considérons qu'un agent : (i) a une adresse unique ; (ii) est déclenché par les messages livrés dans sa boîte aux lettres ; et (iii) peut créer d'autres agents.

Afin de réduire la complexité liée à la conception de l'agent reducer, nous optons pour une architecture modulaire d'agent (Morge *et al.*, 2008). Cette approche permet : (i) la séparation des préoccupations ; (ii) la concurrence de la phase de négociation et de celle de reduce ; (iii) une meilleure compréhension du comportement des agents composants. Ainsi, un agent reducer crée trois sous-agents (voir figure 4) :

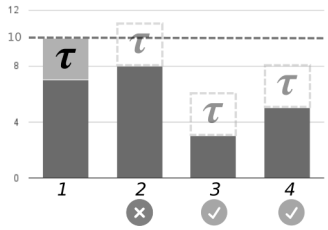
1. un agent worker qui exécute localement les tâches ;
2. un agent broker qui négocie les tâches en tant que commissaire-priseur ou enchérisseur ;
3. un agent manager qui orchestre les négociations menées par le broker avec les tâches réalisées par le worker. Le manager est responsable du lot de tâches classées selon leur coût. La stratégie pour générer le lot sera abordée dans la section 4.6.



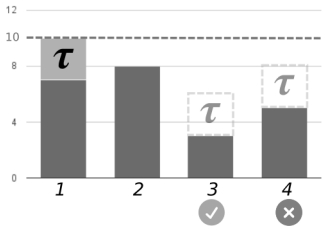
(a) Allocation initiale. Le reducer 1 est le plus chargé



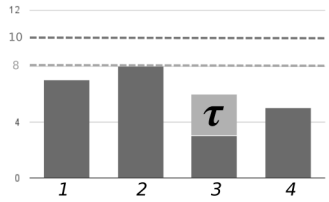
(b) Le reducer propose la tâche τ



(c) Le critère d'acceptabilité de chaque enchérisseur



(d) Le reducer 1 applique le critère de sélection



(e) L'allocation après la négociation

Figure 3. Processus de négociation pas-à-pas : comment le reducer 1 *délègue la tâche* τ

Contrairement à l'agent worker, les deux autres peuvent communiquer avec d'autres agents via leur reducer. Alors que l'agent manager reçoit la sortie du mapper, le broker négocie avec d'autres brokers. En fait, l'agent reducer joue le rôle de proxy pour transmettre des messages depuis/vers d'autres agents.

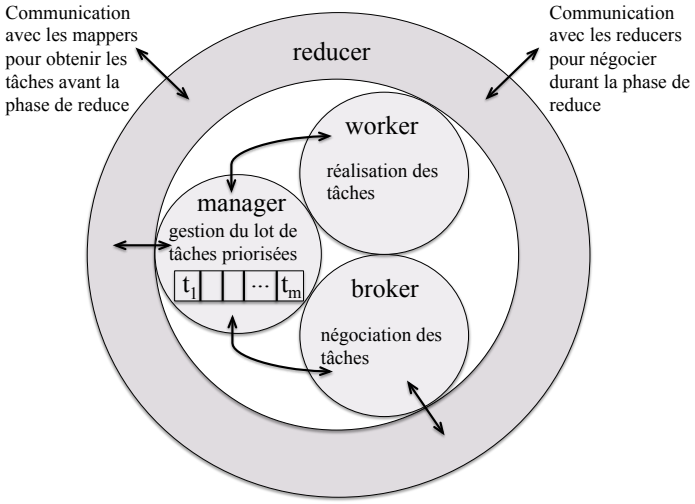


Figure 4. L'agent reducer crée trois agents : le manager, le broker et le worker

4.4. Protocoles

Nous présentons ici les protocoles qui régulent les interactions entre les sous-agents du même reducer et nous appliquons le *Contrat Net Protocol* (Smith, 1980) afin de déléguer des tâches.

Au sein du même reducer, le manager interagit avec le worker pour exécuter localement certaines tâches (voir figure 5a). Le manager attribue une tâche au worker via le message `Perform`. Lorsque la tâche est exécutée, le worker répond par `WorkerDone`, puis le manager peut émettre une nouvelle tâche.

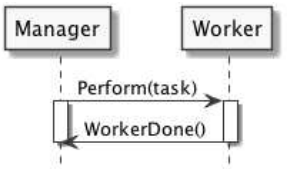
Le manager interagit avec le broker de différentes manières selon le rôle de ce dernier dans la négociation.

– Soit le broker agit en tant qu'enchérisseur (voir figure 5b). Il doit alors connaître la contribution locale pour répondre à un `Cfp`. Dans ce but, le broker envoie un `QueryContribution` au manager qui répond avec `Inform`.

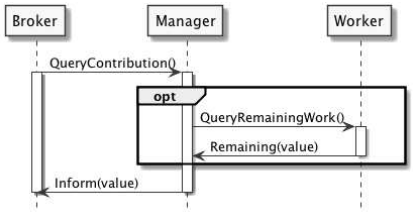
– Soit le broker agit en tant que commissaire-priseur (voir figure 5c). Pour déléguer une tâche, le manager envoie un `Submit`. Si le broker ne trouve aucun sous-traitant potentiel avant la date butoir, il répond au manager avec un `BrokerDeny`. Si toutes les accointances déclinent la délégation de tâche, le broker répond au manager avec un `CFPDeclinedByAll`. Le broker répond par un `BrokerReady` quand il a trouvé un contractant. Simultanément, la tâche peut avoir été exécutée par le

worker. Pour cette raison, le manager peut annuler la délégation de tâches. Sinon, le manager envoie un `Approve` et la délégation réussie se termine par un message `BrokerFinish`.

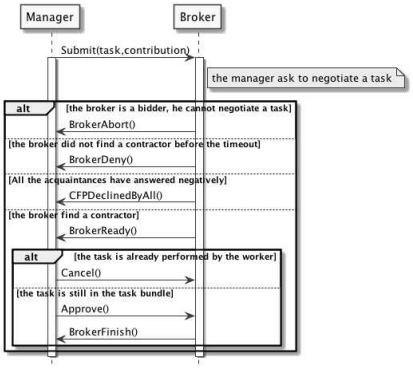
Il est important de noter que ces protocoles qui régulent les interactions entre les sous-agents permet de réaliser simultanément la délégation de tâches par l'intermédiaire du broker et l'exécution de tâches par l'intermédiaire du worker. Même si un broker ne peut être impliqué que dans une seule négociation à la fois, soit comme commissaire-priseur soit comme enchérisseur, plusieurs enchères impliquant des reducers distincts peuvent se produire simultanément.



(a) Protocole pour l'exécution d'une tâche



(b) Protocole d'interrogation de la contribution



(c) Protocole de soumission d'une tâche

Figure 5. Protocoles régulant les interactions entre le manager, le worker et le broker du même reducer

Comme représenté dans la figure 6, une enchère est initiée par un broker avec un `Cfp` qui contient le coût de la tâche déléguée et sa propre contribution. La contribution de l'initiateur est communiquée au manager avec un `InformContribution`. Selon son propre critère d'acceptabilité (voir définition 2) chacun des m participants peut soit décliner (`Decline`) soit accepter le cfp. Dans ce dernier cas, s'il n'est pas déjà impliqué dans une autre enchère, le participant envoie un `Propose` contenant sa contribution. Seule la proposition avec la plus petite contribution est sélectionnée comme la gagnante de l'enchère (cf. définition 3). Les autres propositions sont rejettées alors que le gagnant reçoit un `Accept` avec la tâche déléguée et doit alors accuser réception de la délégation avec un `Confirm`. Il convient de noter que, commme les reducers peuvent être distribués dans un *cluster*, les messages sont délivrés au plus une seule fois. Comme un message peut être perdu, le protocole d'enchères inclut les accusés de réception.

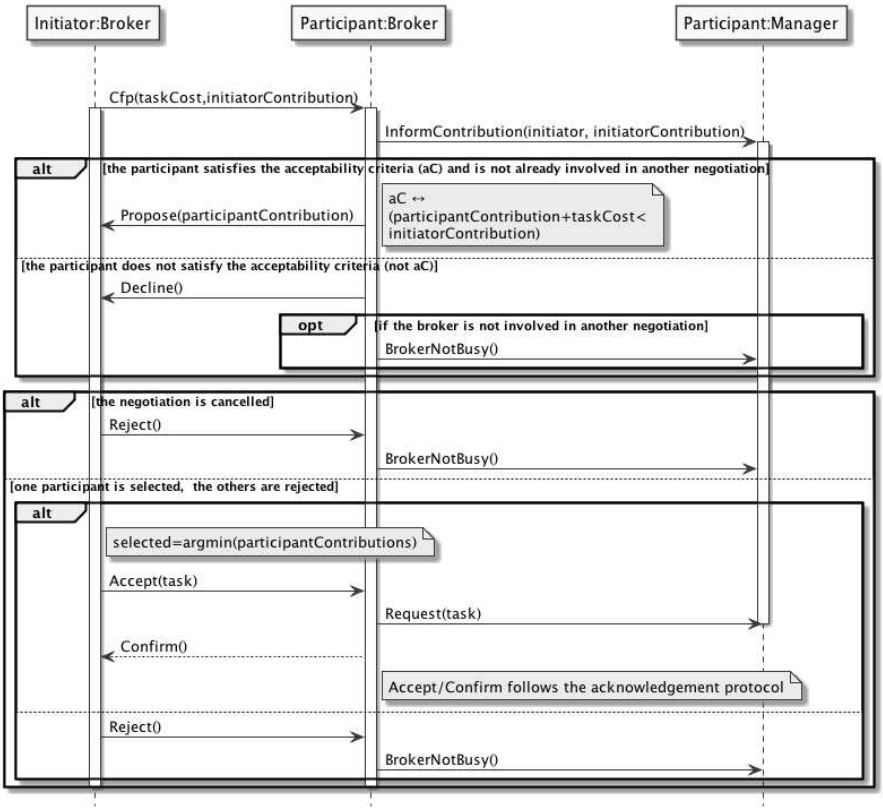


Figure 6. Protocole de négociation

4.5. Comportements

Nous esquissons ici les comportements des sous-agents qui sont conformes avec les protocoles précédents.

Manager. Cet agent gère le lot de tâches et coordonne les activités du worker et du broker. Le manager fournit certaines tâches au worker et amorce le broker pour initier les enchères. Au début, le lot de tâches est rempli par les mappers et complété par le broker lorsqu'il gagne des enchères. Afin de vider le lot de tâches, le manager donne la priorité au worker. Dès que le worker est libre, le manager lui assigne une nouvelle tâche. En fait, une tâche n'est déléguée que si le worker est occupé et le manager s'assure que le broker participe à au plus une enchère. La stratégie pour la gestion du lot de tâches doit déterminer les tâches effectuées par le worker et les tâches négociées. Nous aborderons cette problématique dans la section 4.6.
En outre, le manager interagit avec le superviseur pour détecter la terminaison de la phase reduce. Le manager est inactif lorsque le worker et le broker sont libres et quand le lot de tâches est vide. Le manager est réactivé quand il reçoit un `Request` de la

part de son broker qui a remporté une enchère. Dans ce cas, le manager est responsable de la tâche déléguée par un autre reducer.

Worker. Cet agent, qui est initialement libre, devient occupé dès qu'il reçoit un `Perform`. Lorsque la tâche a été effectuée, le worker informe le manager et il devient libre. Pendant son exécution, un worker peut être interrompu pour informer le manager du coût estimé restant de la tâche en cours d'exécution.

Broker. Le broker peut agir en tant qu'enchérisseur ou en tant que commissaire-priseur.
Broker en tant qu'enchérisseur. Quand le broker reçoit un `Cfp` d'un autre broker (c'est-à-dire un autre reducer), il demande au manager sa propre contribution afin de participer à l'enchère. Si le critère d'acceptabilité (voir définition 2) n'est pas rempli, le broker décline. Sinon, soit il ne répond pas car il participe déjà à une autre enchère, soit une proposition est envoyée. Quand la phase d'appel d'offres est clôturée : (i) soit il gagne l'enchère et il transmet la tâche à son manager puis confirme la délégation de tâche au commissaire-priseur ; (ii) soit l'enchérisseur ne gagne pas (i.e la date butoir est atteinte ou `Reject` est reçu) et le broker informe son manager qu'il est libre.
Broker en tant que commissaire-priseur. Quand le manager déclenche le broker (`Submit`), ce dernier démarre une enchère (`Cfp`). Chaque réponse est enregistrée, qu'il s'agisse de `Propose` ou de `Decline`. Quand toutes les réponses sont reçues ou si le délai est atteint, la meilleure proposition est choisie (voir définition 3). Nous remarquons que la négociation est annulée et le broker envoie une alerte (`CFPDeclinedByAll` ou `BrokerDeny`) à son manager si aucune proposition n'est reçue. Sinon, le broker doit accepter la proposition gagnante et refuser les autres. Il notifie (`Ready`) le manager qu'il a trouvé un contractant. Ensuite, le manager indique si la tâche est toujours disponible (`Approve`) ou non (`Cancel`). Si la tâche négociée n'est plus disponible (la tâche a été entre-temps confiée au worker), l'offre gagnante est rejetée. Autrement, la tâche est envoyée au gagnant et une confirmation est attendue.

On peut noter qu'un reducer peut espérer déléguer une tâche seulement si au moins un reducer a une contribution plus faible. Sinon, le déclenchement d'une enchère est inutile car la négociation va échouer. Nous avons affiné le comportement du manager pour ne pas initier de telles enchères. Quand le commissaire-priseur reçoit des messages `Decline` de toutes ses accointances, le manager correspondant entre en état de pause. Dans cet état, le broker peut être déclenché comme un enchérisseur, mais il n'est plus déclencé par le manager pour initier une enchère. Le manager ne peut quitter cet état que si :

– soit ce reducer s'occupe d'une nouvelle tâche et donc sa contribution augmente ;

– soit ce reducer est informé qu'une autre contribution a diminué en raison d'une délégation de tâche ou d'une variation de performance.

En d'autres termes, le manager quitte l'état de pause si le critère d'acceptabilité peut être rempli par au moins l'une de ses accointances. Afin d'estimer les critères d'ac-

ceptabilité des autres reducers, le manager stocke et met à jour les informations sur les autres contributions communiquées dans les messages `Cfp`.

Nous démontrons dans la section 4.7 que l'état de pause est atteint par tous les agents après un nombre fini d'enchères (voir proposition 9) et qu'aucun transfert de tâche ne peut produire une meilleure allocation des tâches dans un tel cas (voir proposition 11). Ce comportement permet de coordonner le processus de négociation et le traitement des données avec des informations locales afin d'éviter des enchères inutiles et donc diminuer le surcoût communicationnel. Il est important de noter que l'allocation de tâches est dynamique et adaptative car les négociations sont itérées. Si la tâche en cours est exécutée plus lentement qu'espéré par le worker, alors une allocation déséquilibrée apparaîtra et donc le manager peut être réactivé par la diminution des autres contributions pour déléguer les tâches qui sont encore dans son lot.

4.6. Gestion du lot de tâches

Le manager est responsable de la gestion du lot de tâches. Sa stratégie doit déterminer les tâches exécutées par le worker et celles déléguées par négociation. Dans ce but, nous présentons ici deux heuristiques qui seront expérimentalement comparées dans la section 5.

La stratégie naïve. Intuitivement, moins la tâche à déléguer est chère plus il est vraisemblable que l'enchère correspondante aboutisse. Conformément à ce principe, notre première stratégie consiste à exécuter localement autant que possible les tâches coûteuses. Selon cette stratégie naïve, le manager donne au worker les tâches les plus onéreuses et les moins coûteuses au broker. Cependant, avec cette approche, les tâches les plus coûteuses seront les dernières à être négociées et elles restent dans le lot jusqu'à ce qu'elle soient exécutées.

La stratégie k-éligible. Notre deuxième stratégie vise à diminuer le surcoût des communications. La tâche la moins onéreuse est donnée au worker car cette tâche a proportionnellement le surcoût communicationnel le plus élevé. Nous remarquons que, pour diminuer la contribution maximale des reducers, les tâches à déléguer ne sont pas nécessairement les moins chères. En fait, des tâches plus coûteuses peuvent être déléguées. De plus, le processus de négociation est itératif et les contributions sont communiquées dans le `Cfp` des différentes enchères. De cette façon, un commissaire-priseur dispose de croyances sur les autres contributions sans ajouter de messages au protocole et il peut affiner ses croyances au cours du processus de négociation. Un reducer peut estimer quelles tâches sont susceptibles d'être acceptées par ses accointances.

Une tâche τ est *k-éligible* si le critère d'acceptabilité pour τ est susceptible d'être rempli par au moins k accointances.

DÉFINITION 4 (tâche k-éligible). — *Soit $\Omega = \{1, \ldots, n\}$ une population de n reducers et $\mathcal{A}$ une allocation à l'instant t. On dénote $(c_{ij}^{\mathcal{A}}(t))_{j \in \Omega \setminus \{i\}}$ les croyances de*

l'agent i à propos des autres contributions.
La tâche $\tau \in \mathcal{T}_i$ est k-éligible *(avec $k < n$) pour l'agent i à l'instant t ssi :*

$$\exists \Omega' \subseteq \Omega \setminus \{i\} \ (card(\Omega') \geq k \wedge \forall j \in \Omega' \ c_{ij}^{\mathcal{A}}(t) + c_\tau < c_i^{\mathcal{A}}(t)) \tag{5}$$

Plus sont nombreux les reducers qui vérifient leur critère d'acceptabilité pour une tâche, plus il est vraisemblable que l'enchère correspondante aboutisse.

En outre, un commissaire-priseur sélectionne la tâche k-éligible qui minimise la contribution maximale après la délégation de tâches. En d'autres termes, le reducer i sélectionne la tâche $\tau \in \mathcal{T}_i$ telle que τ est une tâche k-éligible et que cette délégation de tâche au reducer ω_τ minimise la contribution maximale, même si ω_τ est le sous-traitant potentiel le plus chargé. Ce calcul est effectué par l'algorithme 7. À cette intention, nous introduisons le paramètre k_{max} qui représente le nombre maximum de sous-traitants potentiels attendus parmi $\Omega \setminus \{i\}$ ($1 \leq k_{max} < n$). Quand $k = 1$, toutes les tâches qui peuvent être acceptées par au moins un agent sont (1-)éligibles, même si elles sont coûteuses. Si $k = n - 1$, les tâches qui peuvent être acceptées par tous les autres reducers sont k-éligibles. Plus k_{max} est grand, moins les tâches k-éligibles sont susceptibles d'être chères. k_{max} évite de déclencher des enchères pour les tâches bon marché. De cette façon, nous souhaitons réduire le surcoût communicationnel lié à la délégation de tâches bon marché. Quand ω_τ est initialisé (ligne 7), nous considérons le reducer le plus chargé qui peut faire une proposition pour τ. De cette manière nous considérons le pire des cas. La tâche τ^* minimise la contribution maximum si l'enchère aboutit (ligne 8). On peut remarquer que $k = 0$ signifie qu'aucun autre reducer ne peut faire une proposition. Dans ce cas, le reducer i passe dans l'état de pause.

En résumé, la stratégie k-éligible donne la tâche la moins onéreuse au worker et elle donne au broker la tâche k-éligible qui minimise la contribution maximale si l'enchère aboutit.

4.7. Résultats théoriques

En premier lieu, on peut remarquer qu'une négociation améliore l'équité qui mesure si le traitement est réalisé au dépens de l'agent le plus chargé. Les tâches sont réparties de manière plus égalitaire après une négociation.

PROPRIÉTÉ 5. — *La variance des contributions des reducers diminue strictement après une enchère réussie.*

PREUVE 6. — *Soit $\Omega = \{1, \ldots, n\}$ une population de n reducers. Nous considérons ici une enchère réussie initialisée par le reducer* 1. *Par simplification, on note :*

- $(c_i)_{i \in \Omega}$, *les contributions des agents avant l'enchère ;*
- $(c'_i)_{i \in \Omega}$, *les contributions des agents après l'enchère ;*
- $\bar{c} = \frac{1}{n}\Sigma_{i=1}^{n} c_i = \frac{1}{n}\Sigma_{i=1}^{n} c'_i$ *la moyenne des contributions* [3] *;*

3. Il est important de noter que la délégation de tâche est conservative.

Données : Le nombre maximum de sous-traitants potentiels requis k_{max}
Entrées : le lot de tâches $\mathcal{T}_i$,
les croyances du reducer i à propos des autres contributions $(c_{ij})_{j\in\Omega\setminus\{i\}}$
Sorties : la tâche à déléguer τ^*

$k \leftarrow k_{max}$;
tant que $k > 0$ **faire**
  $\mathcal{T}_k \leftarrow \{\tau \in \mathcal{T}_i \mid \tau$ est k-éligible pour le reducer $i\}$;
  si $\mathcal{T}_k \neq \emptyset$ **alors**
    pour chaque $\tau \in \mathcal{T}_k$ **faire**
      /* Identifier les reducers potentiels susceptibles de valider le critère d'acceptabilité */
      $\Omega_\tau \leftarrow \{j \in \Omega \setminus \{i\} \mid c_{ij} + c_\tau < c_i\}$;
      /* Choisir le pire reducer potentiel, i.e. avec la contribution maximale */
      $\omega_\tau \leftarrow \underset{j\in\Omega_\tau}{\operatorname{argmax}}(c_{ij})$;
    /* Sélectionner la tâche qui minimise la contribution maximale dans le pire cas */
    $\tau^* \leftarrow \underset{\omega_\tau\in\mathcal{T}_k}{\operatorname{argmin}}(max(c_i - c_\tau, c_{\omega_\tau} + c_\tau))$;
    retourner τ^*
  sinon
    $k \leftarrow k - 1$;

Figure 7. Sélection par le reducer i de la tâche à déléguer

- $Var = \Sigma_{i=1}^{n}(c_i - \bar{c})^2$ *la variance des contributions avant l'enchère ;*
- $Var' = \Sigma_{i=1}^{n}(c'_i - \bar{c})^2$ *la variance des contributions après l'enchère.*

Soit $c > 0$ le coût de la tâche négociée et k le reducer qui gagne l'enchère. Grâce au critère d'acceptabilité du participant k (voir définition 2), $c_k + c < c_1$, et donc $c_k + c - c_1 < 0$. Ainsi, $Var' - Var < 0$.

On peut noter que le processus complet de négociation améliore également l'équité.

PROPRIÉTÉ 7. — *La variance des contributions décroît strictement suite à des enchères itératives et concurrentes réussies.*

PREUVE 8. — *Le comportement du manager garantit qu'un reducer peut être simultanément impliqué dans au plus une enchère (voir section 4.5). En fait, les rôles d'enchérisseur et de commissaire-priseur sont mutuellement exclusifs. Quand le broker joue le rôle d'un commissaire-priseur, il ne répond pas aux autres enchères en tant qu'enchérisseur. Réciproquement, un broker ne peut pas initier une enchère en tant que commissaire-priseur quand il est impliqué comme enchérisseur dans une autre enchère. Les enchères simultanées concernent des groupes disjoints de reducers. Il*

en résulte que chaque enchère est indépendante. En d'autres termes, les résultats des enchères sont indépendants. Selon la proposition 5, chaque enchère réussie diminue strictement la variance des contributions. Par conséquent, l'exécution itérative et simultanée de ces enchères diminue aussi strictement la variance des contributions.

Le processus de négociation globale se termine.

PROPRIÉTÉ 9. — *L'exécution itérative et simultanée des enchères se termine.*

PREUVE 10. — *D'après la proposition 7, la variance est positive et elle décroit strictement au cours du processus globale de négociation. En outre, le nombre de tâches est fini. Par conséquent, après un nombre fini d'enchères, la variance ne peut plus décroître et donc aucune négociation réussie n'est plus possible.*

Le processus de négociation est correct. Quand il s'arrête, aucun autre transfert de tâche ne peut alléger le reducer le plus chargé.

PROPRIÉTÉ 11. — *Quand le processus de négociation globale se termine, il n'existe aucun transfert de tâche qui peut diminuer la contribution du plus chargé des agents.*

PREUVE 12. — *Soit $\Omega = \{1, \ldots, n\}$ une population de n reducers et $\mathcal{A}$ une allocation de tâches à l'instant t. Soit j le plus chargé des reducers et τ et la moins onéreuse des tâches dans $\mathcal{T}_j$. Nous supposons qu'il existe un reducer i qui peut alléger j, i.e. $c_i + c_\tau < c_j$. Dans ce cas, le critère d'acceptabilité pour le reducer i est vérifié. L'enchère correspondante serait réussie, ce qui est une contradiction.*

Ces propriétés sont validées expérimentalement dans la section suivante.

5. Expérimentations

Nos expériences visent à : (i) comparer nos deux stratégies pour la gestion du lot de tâches ; (ii) évaluer notre proposition par rapport à la classique mise en œuvre distribuée du modèle de programmation MapReduce.

Nous avons implémenté notre prototype avec le langage de programmation Scala [4] et la boîte à outil Akka [5]. Akka, basé sur le modèle d'acteur (Hewitt *et al*., 1973), permet de combler l'écart entre les spécifications du comportement des agents (voir section 4.5) et leur implémentation.

Nous analysons un jeu de données réelles qui contient plus de 3 millions d'enregistrements météorologiques (identifiant de station, horodatage, température, pluviométrie, ...) provenant de 62 stations au cours des 20 dernières années [6]. Nous considérons ici deux jobs différents :

4. http://www.scala-lang.org/
5. http://akka.io
6. https://donneespubliques.meteofrance.fr/?fond=produit&id_produit=90&id_rubrique=32g

1. le premier `EnrPrTemp` (pour « enregistrement par température ») compte le nombre d'enregistrements par demi-degré de température. Ce job est réalisé par 10 mappers et 20 reducers. Certaines tâches reduce sont petites, d'autres sont importantes. L'allocation de tâches effectuée par le partitionnement par défaut d'Hadoop est inéquitable (voir figure 8) ;

2. le second `PluPrStat` (pour « pluviométrie par station ») compte les précipitations accumulées par station. Ce job est effectué par 10 mappers et 10 reducers. La taille des tâches reduce est homogène et le partitionnement des tâches par défaut est presque équitable (voir figure 9).

Par souci de simplicité, les reducers sont totalement connectés. Commme ils fonctionnent sur un seul ordinateur multi-cœur, nous pouvons supposer que les performances de nos reducers sont homogènes dans le temps.

5.1. Gestion du lot de tâches

Nous supposons que la stratégie k-éligible a un coût communicationnel inférieur à celui de la stratégie naïve (voire section 4.6). Afin de valider empiriquement cette hypothèse, nous mesurons le nombre d'enchères qui induisent un coût communicationnel. De toute évidence, nous nous attendons à ce que l'allocation de tâches finale atteinte par la stratégie k-éligible soit aussi bonne que celle atteinte par la stratégie naïve. À cette fin, nous considérons l'équité, c'est-à-dire le rapport entre la contribution du reducer le plus chargé et celle du reducer le moins chargé. Cette métrique indique si le traitement est effectué au détriment du plus mauvais reducer. Si la mesure est proche de 1, l'allocation est équitable. Pour chaque ensemble de paramètres, nous effectuons 5 exécutions. Comme l'écart-type dû au non-déterminisme de l'ordonnanceur est faible, nous ne montrons que les moyennes des mesures sur les différentes exécutions.

Le tableau 2 présente les résultats empiriques pour le job `EnrPrTemp`. Afin d'obtenir suffisamment d'enchères réussies, nous fixons le nombre maximal de sous-traitants potentiels requis, à savoir $k_{max} = 4$. En d'autres termes, une tâche est négociée si au moins $20\ \%$ des reducers peuvent faire une proposition. On constate que la stratégie k-éligible mène à presque cinq fois moins d'enchères que la stratégie naïve et l'allocation des tâches est légèrement plus équitable.

Tableau 2. Comparaison des stratégies naïve et k-éligible ave le job `EnrPrTemp`

	Stratégie naïve	**Stratégie k-éligible**
Nombre d'enchères	2695	572
Pourcentage d'enchères réussies	$22\ \%$	$26\ \%$
Équité	0.76	0.81

Le tableau 3 montre les résultats empiriques pour le job `PluPrStat`. Nous fixons comme précédemment $k_{max} = 2$ pour des raisons similaires. Comme l'allocation initiale des tâches correspondant au partitionnement par défaut d'Hadoop est presque

équitable, nous observons que beaucoup moins d'enchères sont nécessaires que pour le job `EnrPrTemp`. Toutefois, la stratégie k-éligible diminue encore significativement le nombre d'enchères. Contrairement au job précédent, la stratégie k-éligible conduit à une allocation de tâches légèrement moins équitable. Cela est dû à la différence entre ces stratégies pour la gestion des tâches les moins onéreuses. Tandis que la stratégie k-éligible donne ces tâches au worker pour être exécutées au plus tôt, la stratégie naïve négocie ces tâches ce qui permet d'équilibrer la charge de travail jusqu'à la fin du traitement des données.

Tableau 3. Comparaison des stratégies naïve et k-éligible ave le job `PluPrStat`

	Stratégie naïve	**Stratégie k-éligible**
Nombre d'enchères	107	77
Pourcentage d'enchères réussies	16 %	23 %
Équité	0.91	0.87

Pour les deux jobs envisagés, le pourcentage d'enchères réussies est peu élevé quelque soit la stratégie. En effet, peu de reducers ayant une faible contribution peuvent agir comme enchérisseurs dans les différentes enchères. Malheureusement, ils ne peuvent pas être impliqués simultanément dans plusieurs enchères. C'est pourquoi le nombre d'enchères reste élevé. Néanmoins, notre stratégie k-éligible nécessite beaucoup moins d'enchères pour équilibrer la charge de travail. Nous avons validé empiriquement notre hypothèse. De plus, l'adoption de la stratégie k-éligible ne pénalise pas l'équité de l'allocation des tâches. De petites déviations peuvent être expliquées par la forme du jeu de données. Ce sont les raisons pour lesquelles nous sélectionnons la stratégie k-éligible afin de comparer notre SMA adaptatif avec la classique mise en œuvre distribuée du modèle de programmation MapReduce.

5.2. Réallocation de tâches

Afin d'évaluer notre proposition, nous avons réimplémenté la classique mise en œuvre distribuée du modèle de programmation MapReduce (Dean, Ghemawat, 2004). Nous la comparons avec notre SMA. Rappelons que l'allocation initiale des tâches pour notre SMA correspond au partitionnement par défaut d'Hadoop.

La comparaison est faite sur l'équité (le rapport entre la contribution du reducer le plus chargé et celle du moins chargé) et le taux d'accélération (le rapport entre le temps d'exécution du MapReduce classique et celui de notre SMA). De plus, nous considérons l'allocation quasi-optimale des tâches obtenue par la règle LPT (*Longest Processing Time*) selon laquelle à $t = 0$, les n tâches les plus coûteuses sont affectées aux n reducers puis, chaque fois qu'un reducer est libre, il lui est assigné la tâche la plus coûteuse parmi celles restantes (Graham, 1969). Il est important de noter que, sans connaissance préalable sur les données, cette dernière est une limite théorique qui ne peut être calculée que rétroactivement.

La figure 8 (resp. figure 9) montre les contributions des reducers avec le MapReduce classique et notre SMA pour le job `EnrPrTemp` (resp. `PluPrStat`). À

Tableau 4. Comparaison de l'équité entre le MapReduce classique et notre SMA pour les jobs `EnrPrTemp` *et* `PluPrStat`

	Partition par défaut	**Allocation SMA**	**Allocation opt.**
`EnrPrTemp`	0	0.82	0.98
`PluPrStat`	0.23	0.87	0.97

gauche, le partitionnement par défaut d'Hadoop donne une allocation de tâches inéquitable. En particulier, peu de reducers exécutent des tâches avec cette partition pour le job `EnrPrTemp` (voir figure 8). Nous pouvons observer que la négociation équilibre la charge de travail entre les reducers car les tâches sont dynamiquement ré-allouées au cours du processus. Les reducers surchargés délèguent certaines tâches aux reducers qui sont moins chargés ou inoccupés.

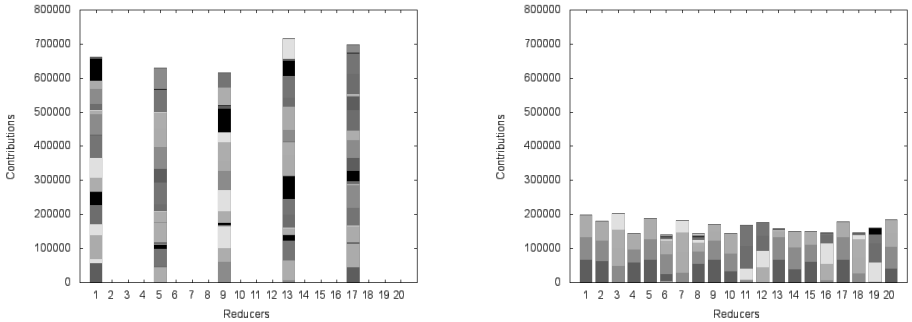


Figure 8. Les contributions des reducers pour le MapReduce classique (à gauche) et pour notre SMA (à droite) avec le job `EnrPrTemp`

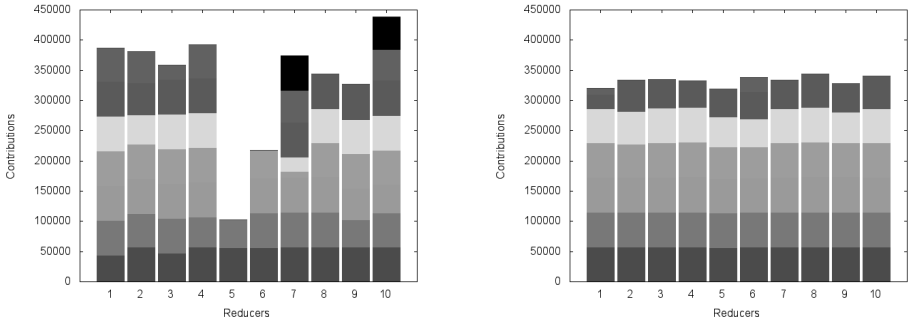


Figure 9. Les contributions des reducers pour le MapReduce classique (à gauche) et pour notre SMA (à droite) avec le job `PluPrStat`

Le tableau 4 montre que le processus de négociation conduit à une allocation de tâches équitable. Dans notre proposition, la phase de négociation et la phase reduce sont simultanées. Par conséquent, les tâches qui sont données au worker ne peuvent plus être négociées. C'est la raison pour laquelle il existe un écart entre

l'équité de notre allocation dynamique de tâches et l'allocation quasi-optimale. Grâce à une meilleure répartition du travail, le taux d'accélération de la phase reduce réalisée par notre SMA est de $3,67$ (respectivement $1,24$) pour le job `EnrPrTemp` (resp. `PluPrStat`). Même si l'allocation des tâches est presque équilibrée pour le job `PluPrStat`, la phase reduce réalisée par notre SMA est plus rapide que celle réalisée par le partitionnement classique des tâches. Ceci est dû au fait que notre SMA est toujours en mesure de produire une allocation de tâches plus équitable et que le surcoût lié à la négociation est faible grâce à l'état de pause.

En résumé, notre proposition réduit le temps d'exécution de la phase reduce pour ces deux jobs.

6. Conclusion

Les applications MapReduce sont complexes à optimiser car elles s'appuient sur des opérations définies par l'utilisateur et qui nécessitent que le développeur d'application comprenne l'implémentation du framework qu'il utilise (Hadoop par exemple) mais aussi qu'il ait une connaissance *a priori* des propriétés des données ainsi que de l'environnement physique. En particulier, des biais dans les données peuvent entraîner une répartition inégale des charges due à un partitionnement fixé et statique. Notre SMA constitue un modèle de calcul qui est intrinsèquement adaptatif.

Dans cet article, nous avons mis en œuvre une distribution du modèle de programmation MapReduce où l'allocation des tâches est le résultat de négociations multi-agents au cours de la phase reduce. Cette réallocation dynamique des tâches s'appuie sur des décisions locales des reducers incarnés par des agents, sans orchestration ni prétraitement des données et sans paramètre dépendant des données ou de l'environnement informatique. Plus précisément, chaque reducer est composé de trois sous-agents concurrents : un worker qui effectue localement des tâches ; un broker qui les négocie ; et un manager qui coordonne la négociation et le traitement local des données. Afin d'équilibrer la charge de travail, ces agents négocient des tâches en fonction de leurs contributions pour diminuer celle des reducers les moins performants, c'est-à-dire ceux qui retardent la phase reduce. Nous avons prouvé que notre processus de négociation améliore l'équité de l'allocation des tâches. Nos expériences ont confirmé que notre proposition diminue le temps d'exécution de la phase reduce. De plus, nous avons étudié deux stratégies pour la gestion du lot de tâches. Nous avons sélectionné la stratégie k-éligible, car nos expériences ont montré que cette stratégie nécessite moins d'enchères pour équilibrer la charge de travail sans que cela pénalise l'équité de l'allocation.

Plusieurs perspectives sont envisageables pour ce travail. Dans des travaux plus récents (Baert *et al.*, 2017), nous avons évalué empiriquement notre prototype dans un *cluster* de PC. À travers des jeux de données plus massifs, nous avons observé l'efficacité de notre proposition malgré les communications entre les nœuds liées aux négociations. Nous avons également abordé le problème du groupe de clés coûteux. Pour y parvenir nous considérons les tâches comme divisibles. Si une tâche coûteuse

est divisée en sous-tâches plus légères, la négociation de ces dernières permet de parvenir à une répartition des tâches plus équitable.

Nous envisageons désormais d'étendre notre proposition afin qu'un reducer puisse être enchérisseur dans plusieurs enchères simultanées. Cette extension doit préserver la correction et la terminaison du processus de négociation, mais elle devrait également améliorer le taux de réussite des enchères. Par conséquent, la surcharge communicationnelle devrait diminuer. De plus, nous souhaitons étudier plus en profondeur l'adaptation de notre approche dans un réseau hétérogène.

Remerciements

Ce travail s'inscrit dans le défi CNRS MASTODONS 2017. Nous remercions le comité de programme des JFSMA ainsi que les relecteurs RIA qui, par leurs remarques, nous ont permis d'améliorer cet article.

Bibliographie

Baert Q., Caron A.-C., Morge M., Routier J.-C. (2016). Allocation équitable de tâches pour l'analyse de données massives. In *Actes des journées francophones sur les systèmes multi-agents (JFSMA)*, p. 55–64. Saint-Martin-du-Vivier (Rouen), France, Cépaudès.

Baert Q., Caron A.-C., Morge M., Routier J.-C. (2017). Stratégie de découpe de tâche pour le traitement de données massives. In *Actes des journées francophones sur les systèmes multi-agents (JFSMA)*. Caen, France, Cépaudès. (À paraître)

Brandt F., Conitzer V., Endriss U., Lang J., Procaccia A. D. (2016). *Handbook of computational social choice*. Cambridge University Press.

Chen Q., Zhang D., Guo M., Deng Q., Guo S. (2010). SAMR: A self-adaptive MapReduce scheduling algorithm in heterogeneous environment. In *Proc. of 10th international conference on computer and information technology (CIT)*, p. 2736–2743.

Clinger W. D. (1981). *Foundations of actor semantics*. Thèse de doctorat non publiée, Massachusetts Institute of Technology.

Damamme A., Beynier A., Chevaleyre Y., Maudet N. (2015, May). The power of swap deals in distributed resource allocation. In *Proc. of the 14th international conference on autonomous agents and multiagent systems (AAMAS)*, p. 625-633. Istanbul, Turkey.

Dean J., Ghemawat S. (2004). MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th symposium on operating system design and implementation*, p. 137-150.

DeCandia G., Hastorun D., Jampani M., Kakulapati G., Lakshman A., Pilchin A. *et al*. (2007). Dynamo: Amazon's highly available key-value store. In *Proc. of the 21st ACM SIGOPS symposium on operating systems principles*, p. 205-220.

Graham R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, vol. 17, n° 2, p. 416–429.

Hewitt C., Bishop P., Steiger R. (1973). A universal modular ACTOR formalism for artificial intelligence. In *Proc. of the 3rd international joint conference on artificial intelligence*, p. 235-245.

Kwon Y., Balazinska M., Howe B., Rolia J. (2012). SkewTune: Mitigating skew in MapReduce applications. In *Proc. of the ACM SIGMOD international conference on management of data*, p. 25-36.

Lama P., Zhou X. (2012). AROMA: Automated resource allocation and configuration of MapReduce environment in the cloud. In *Proc. of the 9th internatinal conference on autonomic computing*, p. 63-72.

Liroz-Gistau M., Akbarinia R., Valduriez P. (2015). FP-Hadoop: efficient execution of parallel jobs over skewed data. *Proc. of the VLDB Endowment*, vol. 8, n° 12, p. 1856–1859.

Morge M., Stathis K., Vercouter L. (2008). Argumentation sur les motivations propres dans l'architecture V3A pour des agents auto-adaptatifs (présentation courte). In *Actes des journées francophones sur les systèmes multi-agents systèmes multi-agents*, p. 149-158. Brest, France, Cépaduès.

Smith R. (1980). The contract net protocol: Highlevel communication and control in a distributed problem solver. *IEEE Trans. on Computers, C*, vol. 29, p. 12.

Verma A., Cherkasova L., Campbell R. H. (2011). Aria: Automatic resource inference and allocation for MapReduce environments. In *Proc. of the 8th internatinal conference on autonomic computing*, p. 235-244.

Vernica R., Balmin A., Beyer K. S., Ercegovac V. (2012). Adaptive MapReduce using situation-aware mappers. In *Proc. of the 15th international conference on extending database technology*, p. 420-431.

White T. (2015). *Hadoop: The definitive guide, 4th edition*. O'Reilly Media.

Zaharia M., Chowdhury M., Das T., Dave A., Ma J., McCauley M. *et al*. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the 9th usenix conference on networked systems design and implementation*, p. 15-28.