Allocation équitable de tâches pour l'analyse de données massives

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

Allocation équitable de tâches pour l'analyse de données massives

Quentin Baert
quentin.baert@etudiant.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

De nombreuses entreprises utilisent l'application MapReduce pour le traitement de données massives. L'optimisation statique de telles applications est complexe car elles reposent sur des opérations définies par l'utilisateur, appelées map et reduce, ce qui empêche une optimisation algébrique. 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 de la tâche. Cependant, ces systèmes ne sont pas efficaces durant la phase d'apprentissage ou lorsqu'un nouveau type de tâches ou de données apparait. Dans ce papier, nous présentons un système multi-agents adaptatif pour l'analyse de données massives avec MapReduce. Nous ne pré-traitons pas les données et adoptons une approche dynamique où les agents reducers interagissent durant l'exécution. Nous proposons une ré-allocation des tâches basée sur la négociation pour parvenir à faire décroitre la charge de travail du plus chargé des agents reducers et ainsi réduire le temps d'exécution.

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

Abstract

Many companies are using MapReduce applications to process very large amounts of data. Static optimization of such applications is complex because they are based on user-defined operations, called map and reduce, which prevents some algebraic optimization. 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 type 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 and 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 execution time - we propose a task re-allocation based on negotiation.

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

Introduction

La science des données (ou *Data Science*) vise à traiter de grands volumes de données pour y extraire de nouvelles connaissances (*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 [2] 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 mais de nombreuses autres implémentations existent comme le *framework* Spark [9] ou la base de données NoSQL distribuée Riak construite par Amazon Dynamo [3]. 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 répartition efficace des tâches (c.à.d. la distribution 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 sont particulièrement appropriés pour s'adapter à des données inconnues, à des flux qui évoluent constamment ou à un environnement informatique dynamique.

Dans cet article, nous proposons un système multi-agents pour mettre en œuvre le patron de conception MapReduce afin d'analyser des données [1]. 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. À cette intention, les *reducers* sont impliqués dans de multiples enchères simultanées. 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 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-agents à des données réelles et nos observations confirment la plus-value des négociations.

Cet article est structuré comme suit. La section 1 présente des travaux connexes pertinents et introduit le patron de programmation MapReduce. La section 2 décrit le cœur de notre proposition. Puis nous présentons nos résultats expérimentaux dans la section 3. Finalement, la section 4 conclut.

1 Travaux connexes

Dans [2], les auteurs présentent le modèle de programmation MapReduce et son implémentation pour le traitement de données massives. Dans ce modèle, alors que la fonction *map* filtre les données, la fonction *reduce* les agrège. 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 (voir figure 1). Cela permet aux développeurs, sans aucune expérience avec la programmation parallèle et distribuée, d'utiliser facilement un tel système. Par exemple, le nombre de processus *map/reduce* peut être déterminé automatiquement en fonction de la taille des données ($card(mapper) = |data|/64Mo$) et de

1. Cet article est une version traduite et étendue de [1] notamment en termes d'expérimentation.

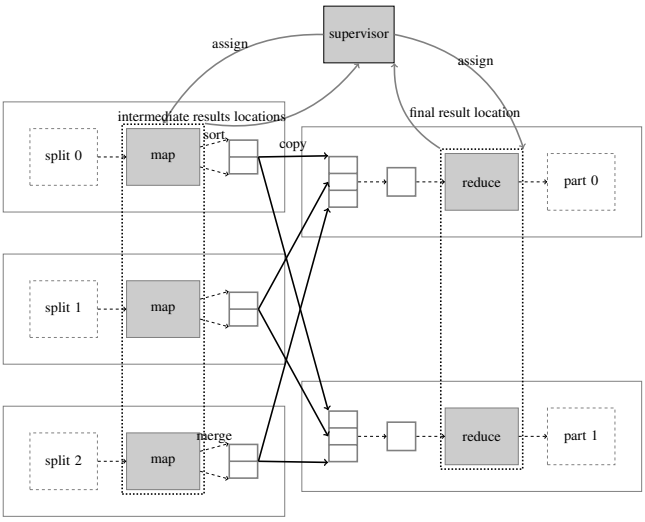


FIGURE 1 – Flot de données dans l'implémentation Hadoop de MapReduce.

la configuration matérielle ($card(reducer) \gg nbMachines$).

Le modèle de programmation MapReduce fait appel à deux fonctions, données par l'utilisateur, qui possèdent les signatures suivantes :

$$map : (K1, V1) \rightarrow List[(K2, V2)]$$
$$reduce : (K2, List[V2]) \rightarrow List[(K3, V3)]$$

À la sortie de chaque *mapper*, un *partitioner* prend les paires (*clé :K1*, *valeur :V1*) intermédiaires et les divise en sous-ensembles, un par *reducer*, tels que toutes les valeurs associées à la même clé (de type $K2$) sont groupées et affectées au même *reducer*. Par exemple, le *partitioner* de Hadoop utilise par défaut la fonction de hachage (`key.hashCode() % nombreDeReducers`), pour déterminer le reducer auquel est affectée la clef `key`. Le *partitioner* peut aussi être paramétré a priori par le développeur grâce à son expertise pour spécifier quelles clés doivent être traitées par quel *reducer*. Par la suite, chaque *reducer* prend les paires de la forme $(K2, list[V2])$ et applique la fonction *reduce* pour chaque groupe de valeurs associées à une clé. Une fois cette opération terminée, les paires (*clé :K3*, *valeur :V3*) finales sont écrites dans un fichier.

Que la fonction de répartition utilisée soit celle par défaut ou non, le partitionnement est fixé a priori. En d'autres termes, cette fonction ne dépend pas des données, et donc la charge de travail n'est pas nécessairement distribuée uniformément parmi les *reducers*. Le temps de calcul est déterminé par le *reducer* le plus chargé.

Dans cet article, nous nous concentrons sur la réduction de la charge de travail du *reducer* le plus chargé : une approche égalitaire de l'équité. Plusieurs systèmes ont étudié l'optimisation de la phase de *reduce*, par exemple en prédisant la performance grâce à l'étude du *job* ou grâce à la collecte de données lors des exécutions précédentes (voir [6] ou [8]). Notre solution ne pré-traite pas les données par une phase d'apprentissage automatique en utilisant des échantillons de données. Elle adopte une approche dynamique où des *reducers* adaptatifs prennent des décisions locales et interagissent pendant le déroulement du *job*. Dans [5], les auteurs étudient des situations non équilibrées entre *mappers* ou entre *reducers*. Ils y décrivent leur système nommé SkewTune, qui atténue deux types de biais : celui dû à une distribution des données non équitable et celui dû au fait que certaines données prennent plus de temps à traiter que d'autres. Quand une ressource (un *reducer*) est disponible car sa tâche est terminée, le *supervisor* identifie le *reducer* le plus lent et répartit à nouveau ses données d'entrées non traitées. Notre approche est similaire mais nous souhaitons que le partitionnement soit un choix collectif des *reducers* et non centralisé. De plus, nous cherchons également à gérer les ensembles de valeurs associées à une même clé et pas seulement les ensembles de clés.

Dans notre approche, l'allocation dynamique des tâches est basée sur une négociation entre *reducers*. La théorie du choix social offre des méthodes pour concevoir et analyser des décisions collectives qui combinent des préférences ou des bien-êtres individuels. Le calcul de 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. Cependant, une telle approche comportent des limitations qui nous semblent trop fortes : i) il se peut que regrouper toutes les données sur une même machine soit trop coûteux ; ii) si les données évoluent durant le processus, ce dernier doit recommencer afin de prendre les nouvelles données en compte ; iii) il est nécessaire que tous les agents soient connectés les uns aux autres et qu'ils puissent tous communiquer sans aucune restriction. Classiquement, dans un système distribué, le coût de la communication dépend de la topologie du réseau, c.à.d. des contraintes physiques. Nous avons plutôt choisi de considérer des enchères distribuées.

2 Proposition

Nous proposons ici un système multi-agents pour exécuter de manière distribuée le patron de conception MapReduce. Le paradigme multi-agents permet de s'adapter dynamiquement tant à l'environnement informatique [2] (i.e. la configuration matérielle) qu'aux biais des données d'entrée (c.-à-d. des clés mal réparties ou des clés associées à un grand nombre de valeurs). Afin de réduire le temps d'exécution de la phase de *reduce* et a fortiori le traitement des données, nous souhaitons réduire la charge totale du plus chargé des *reducers*. Dans ce but, l'agent *supervisor* ne centralise pas le contrôle du *job* MapReduce mais il est chargé du déploiement du système. De même, nous associons à chacun des processus (i.e *map/reduce*) un agent. Alors que les agents *mapper* répartissent a priori les tâches aux agents *reducer*, les agents *reducer* prennent des décisions locales pour redistribuer dynamiquement les tâches en fonction de leurs propres charges et de celles de leurs interlocuteurs. Dans une première approche, nous ferons l'hypothèse que les *reducers* déployés par le *supervisor* sont totalement connectés.

Dans cette section, nous présentons le cœur de notre proposition. Premièrement, nous survolons la proposition. Deuxièmement, nous présentons l'architecture de notre agent *reducer*. Troisièmement, nous introduisons les différents protocoles d'interaction dans lesquels les *reducers* sont impliqués. Finalement, nous présentons quelques résultats formels à propos de notre processus de ré-allocation.

2.1 Vue d'ensemble

Notre contribution cherche à fournir un partitionnement de tâches équilibré. Dans ce but, nous proposons une ré-allocation 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 (caractéristique 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*

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.

$\tau_1, \ldots, \tau_m$ *avec leurs coûts associés* $c_{\tau_1}, \ldots, c_{\tau_m}$ *et une population* $\Omega = \{1, \ldots, n\}$ *de n agents reducers, une* allocation *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 *:*

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

La contribution *d'un agent i au temps t dans l'allocation A est définie comme telle :*

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

où $w_i(t)$ *est le coût estimé ("reste à faire") de la tâche en cours d'exécution par l'agent i au temps t. Avant que la phase de reduce ne commence,* $w_i(0) = 0$*.*

Dans notre proposition, la phase de *map* ne diffère pas de celle du modèle MapReduce classique. Les *mappers* produisent les couples (*clé*,*valeurs*) intermédiaires qui sont envoyés au *reducers*. Cependant, pour chaque (*clé*,*valeurs*), les *mappers* ajoutent l'information sur le coût de la tâche pour ces valeurs (partielles). La méthode de répartition par défaut est ensuite utilisée pour effectuer la première distribution vers les *reducers*.

Alors que les *reducers* commencent la phase de *reduce*, ils initient simultanément une phase de négociation dans le but de faire baisser la contribution du *reducer* le plus chargé et ainsi de réduire le temps d'exécution global du *job*. Les agents *reducers* communiquent les uns avec les autres pour négocier la délégation d'une tâche au travers de messages *cfp* (*call for proposal*) dans le but d'alléger leur contribution. Un *reducer* qui envoie un message *cfp* devient un initiateur. Un message *cfp* contient le coût de la tâche à négocier ainsi que la contribution actuelle de l'initiateur.

Un *reducer* qui reçoit un *cfp* devient enchérisseur. Un enchérisseur accepte une tâche afin de faire baisser la contribution de l'initiateur sous réserve que cette tâche ne le rende pas plus chargé que l'initiateur. Formellement, sa décision est basée sur le critère local suivant :

Définition 2 (Critère d'acceptabilité)**.** *Soit* A *une allocation de tâches sur* n *agents à un instant* t*. L'agent* j *acceptera le transfert d'une tâche* $\tau \in \mathcal{T}_i$ *provenant d'un autre agent* i *si et seulement si :*

$$c_j^A(t) + c_\tau < c_i^A(t)$$

Autrement dit, un enchérisseur accepte une tâche si et seulement si, en cas de négociation réussie, sa nouvelle contribution sera strictement plus petite que la contribution originale de l'initiateur. Ainsi, pour les deux agents impliqués, la plus grande des contributions après la négociation est plus petite que celle avant la négociation.

Réciproquement, l'initiateur d'une négociation peut potentiellement recevoir plusieurs propositions en réponse à son *cfp*. Une proposition contient la contribution de l'exécutant potentiel. L'initiateur sélectionne alors parmi les enchérisseur l'agent qui possède la plus petite contribution. Formellement,

Définition 3 (Critère de sélection)**.** *Soit* A *une allocation de tâches entre* n *agents dans* Ω *à un temps* t*. Si l'agent* i *souhaite déléguer la tâche* τ *et a reçu des propositions des agents de* $\Omega' \subset \Omega$*, il sélectionne :*

$$\text{argmin}(\{c_j^A(t) \mid j \in \Omega'\})$$

De cette façon, le transfert de la tâche permet de charger le *reducer* le moins chargé dans le but d'équilibrer la charge de travail. On peut noter que l'évaluation du critère de décision pour le transfert d'une tâche requiert uniquement l'utilisation d'informations locales, une fois les contributions communiquées.

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 être allégé. Un *reducer* reprend l'envoi de *cfp* lorsqu'il acquiert l'information qu'un ou plusieurs agents de son réseau d'accointances est susceptible de l'accepter.

2.2 Architecture de l'agent *reducer*

Inspirés par [4], nous considérons qu'un agent : i) possède un identifiant unique ; ii) est activé par des messages déposés dans sa boite de réception ; iii) peut créer d'autres agents.

Afin de réduire la complexité liée à la conception de l'agent *reducer*, nous optons pour une architecture récursive d'agent. Cette approche modulaire permet non seulement de séparer les préoccupations, de paralléliser les traitements

mais également de rendre plus lisible le comportement des agents composants. Ainsi, un agent *reducer* crée trois sous-agents :

1. un agent *worker* qui exécute localement les tâches ;
2. un agent *broker* qui négocie les tâches en tant qu'initiateur ou enchérisseur ;
3. un agent *manager* qui orchestre les négociations menés par le *broker* avec les tâches réalisées par le *worker*. Le *manager* est responsable de l'ensemble de tâches à distribuer au *worker* et au *broker*. L'ensemble de tâches est ordonné suivant le coût des tâches. Afin de trouver plus facilement un repreneur, le *manager* essaie de déléguer la tâche la moins coûteuse. Au contraire, la tâche la plus coûteuse est exécutée localement par le *worker*.

Alors que les *reducers* sont voués à être déployés sur des machines distinctes, les agents composants d'un même *reducer* doivent se situer sur la même machine même s'ils sont exécutés par des processeurs différents. En fait, l'agent *reducer* joue le rôle de *proxy* et fait suivre les messages provenant de ou allant vers d'autres *reducers*. Contrairement aux communications locales, nous ne faisons pas d'hypothèses concernant la garantie de livraison des messages entre *reducers*. Les différents *brokers* communiquent entre eux via leur *reducer*. Il en est de même pour les communications entre les *managers* et les *mappers*/*supervisor*. À l'inverse, le *worker* communique uniquement avec son *manager*.

2.3 Protocoles

Nous passons en revue ici les interactions intra-*reducer* puis les interactions entre *reducers*.

Le *manager* interagit avec le *worker* afin d'effectuer des tâches (voir figure 2a). Le *manager* assigne une tâche au *worker* au travers d'un message `Perform` et le *worker* répond avec un message `WorkerDone` lorsque la tâche est effectuée. Ensuite, le *manager* est capable d'envoyer une nouvelle tâche. Afin d'estimer le coût du travail en cours, le *manager* peut aussi envoyer un message `QueryRemainingWork` au *worker*, auquel le *worker* répond avec un message `Remaining` (voir figure 2c).

Le *manager* interagit avec le *broker* de deux manières différentes en fonction de son rôle dans la négociation : un *broker* peut être initiateur d'une négociation ou il peut être enchérisseur.

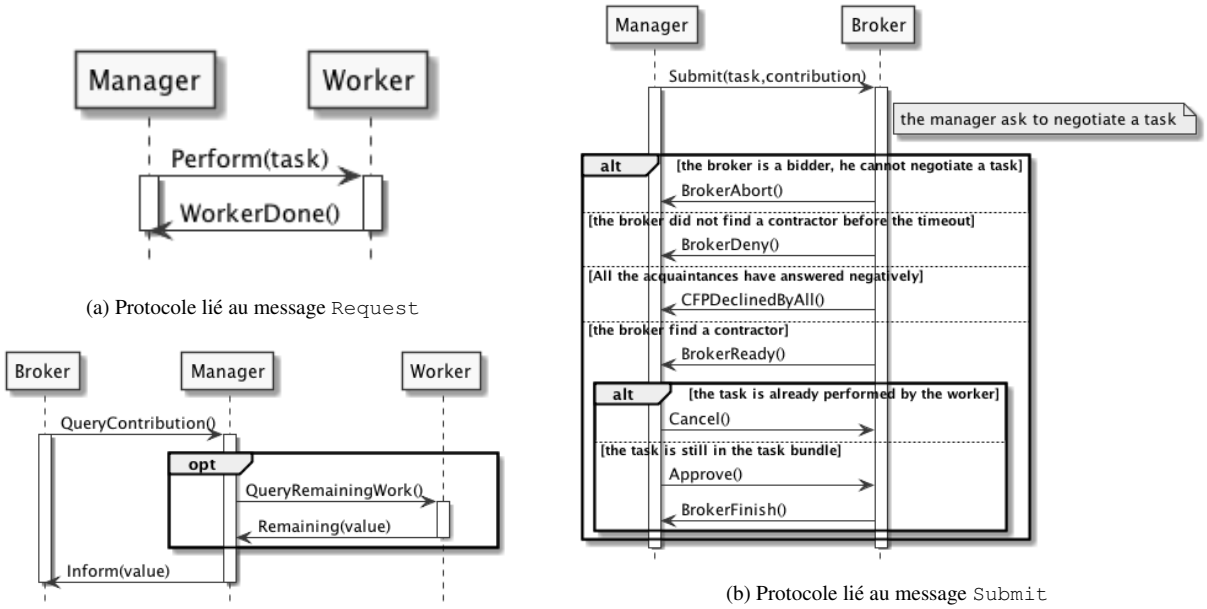
Si le *broker* agit comme un enchérisseur il a besoin de connaître la contribution locale afin de répondre à un `Cfp`. Dans ce but (voir figure 2c), le *broker* envoie un message `QueryContribution` au *manager* qui répond avec un `Inform`. Éventuellement, le *broker* peut demander au *manager* d'ajouter une tâche qu'il a remportée aux enchères à l'ensemble des tâches du *reducer* avec un message `Request`.

Pour déléguer une tâche, le *manager* envoie un message `Submit`, puis le *broker* initie une négociation (voir figure 2b). Si le *broker* ne trouve aucun repreneur, il répond au *manager* à l'aide d'un `BrokerDeny`. Autrement, le *broker* répond avec un message `BrokerReady`. Dans le dernier cas, il est toujours possible, bien que le *manager* ait donné la tâche à négocier au *broker*, que le *worker* soit finalement prêt à l'exécuter. Dans ce cas le *manager* fournit la tâche au *worker* et envoie un message `Cancel` au *broker*. Sinon, le *manager* envoie un message `Approve` au *broker* qui confirme la délégation de tâche avec un message `BrokerFinish`. Si ce n'est pas le cas, le *manager* reçoit un message `UnfinishTask` et la tâche retourne dans l'ensemble de tâches du *reducer*.

Finalement, les *brokers* des différents *reducers* peuvent négocier la délégation d'une tâche via un protocole d'enchère inspiré de [7] (voir figure 3). Une telle négociation est initiée par un *broker* avec un message `Cfp` qui contient le coût de la tâche à déléguer et sa propre contribution. En fonction de son propre critère d'acceptabilité (voir Définition 2) chacun des m participants peut soit décliner (`Decline`), soit enchérir en envoyant un `Propose` avec sa contribution. La proposition avec la plus petite contribution est sélectionnée et remporte l'enchère (voir Définition 3). Les autres enchérisseurs sont notifiés par un `Reject` alors que le gagnant reçoit un `Accept` et doit définitivement valider la délégation par un `Confirm`.

2.4 Comportements

Manager. Le *manager* coordonne les activités du *worker* et du *broker* : il fournit des tâches au *worker* et demande au *broker* d'émettre les `Cfp`. Cette coordination est basée sur les principes suivants : i) le *manager* donne la priorité au *worker* : une tâche est déléguée uniquement si le *worker* est occupé. Dès que le *worker* est libre, le *manager* lui donne une nouvelle tâche à exécuter ; ii) le *manager* assure que le *broker* est engagé dans au plus une négociation ; iii) l'en-



(a) Protocole lié au message `Request`

(b) Protocole lié au message `Submit`

(c) Protocole lié au message `QueryContribution`

FIGURE 2 – Protocole de régulations des interactions entre le *manager*, le *worker* et le *broker* au sein d'un même agent *reducer*

semble des tâches peut être assimilé à une file de priorités : le *manager* donne au *worker* la tâche la plus coûteuse et essaie de déléguer la tâche la moins coûteuse grâce au *broker*. Cet ensemble de tâches est initialement rempli par les *mappers*, puis par le *broker* quand il accepte un `Cfp` proposé par un autre *reducer*. De plus, le *manager* interagit avec le *supervisor* pour détecter la fin du traitement des données. Le *manager* est inactif lorsque le *worker* et le *broker* sont libres et que l'ensemble de tâches est vide. Le *manager* est réactivé quand il reçoit un message `Request` du *broker*.

Worker. Le *worker*, qui est initialement libre, devient occupé dès qu'il reçoit un `Perform`. Quand la tâche est réalisée, le *worker* en informe le *manager* et redevient libre. Pendant qu'il exécute une tâche, un *worker* peut donner une estimation du coût restant du travail en cours.

Broker. Le *broker* peut agir en tant qu'enchérisseur ou en tant qu'initiateur dans une négociation.

***Broker* en tant qu'enchérisseur.** Quand le *broker* reçoit un `Cfp`, il demande la contribution locale au *manager* dans le but de participer aux enchères. Si le critère d'acceptabilité (voir Définition 2) est rempli, une proposition est envoyée. Si ce n'est pas le cas, le *broker* refuse de participer aux enchères et informe le *manager* qu'il est libre. Du fait qu'un *broker* ne peut être enchérisseur que dans une seule négociation à la fois, il décline tous les autres `Cfp`. Quand l'enchérisseur est informé (ou pas) de la décision de l'initiateur de la négociation : i) soit l'enchérisseur gagne l'enchère, demande au *manager* d'ajouter la tâche à l'ensemble des tâches à traiter et confirme à l'initiateur que la tâche est bien prise en compte ; ii) soit l'enchérisseur ne gagne pas l'enchère (la date butoir [3] est atteinte ou un message `Reject` est reçu).

***Broker* en tant qu'initiateur.** Quand le *broker* reçoit un `Submit` du *manager*, il envoie un `Cfp` aux autres *brokers*. Chaque réponse, qu'elle soit une proposition ou un rejet, est mémorisée. Quand toutes les réponses ont été reçues (ou que la date butoir est atteinte), la meilleure proposition (celle avec la contribution la plus basse) est sélectionnée. Évidemment, si aucune proposition n'est reçue, la négociation est annulée et le *broker* envoie un message `BrokerDeny` au *manager*. Autrement, le *broker* sélectionne l'enchère gagnante et rejette les perdantes. Il notifie (à l'aide d'un message

3. La date butoir peut être fixée a priori voire s'adapter à la configuration matérielle si le nombre de réponses s'avère insuffisant.

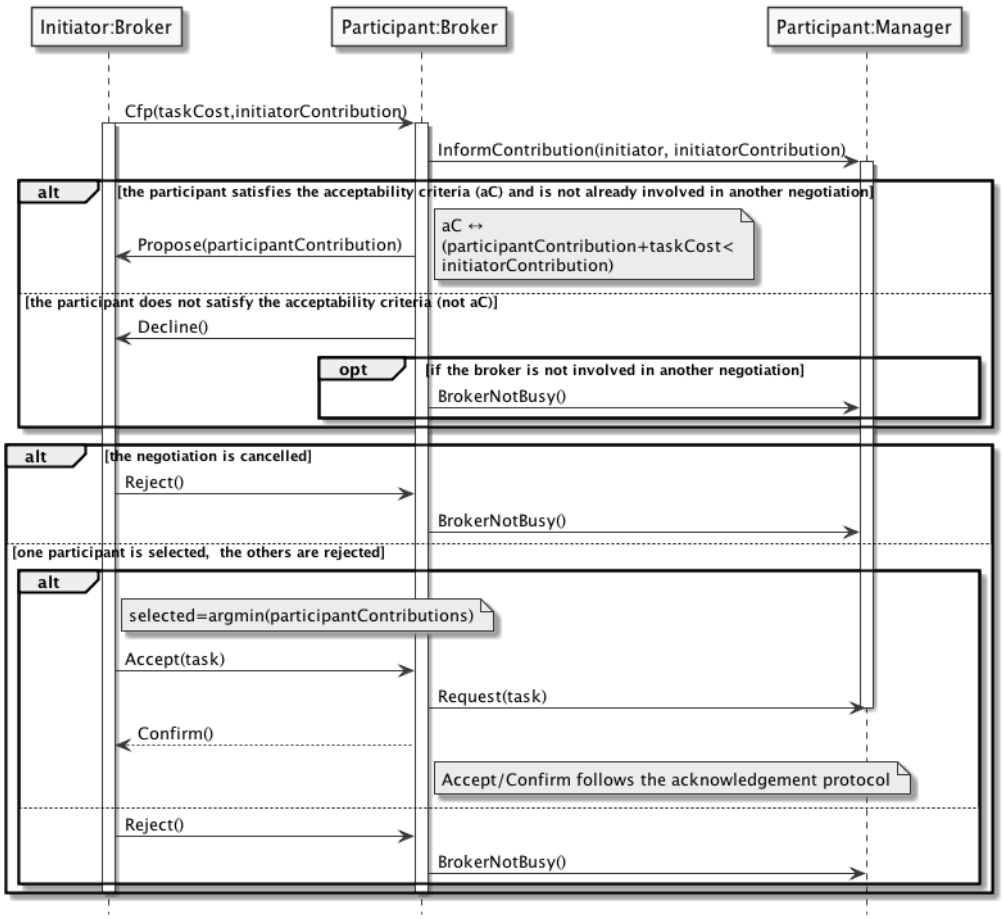


FIGURE 3 – Protocole de négociation

`BrokerReady`) le *manager* qu'il a trouvé un repreneur. En retour, le *manager* annonce si la tâche est toujours disponible (`Approve`) ou pas (`Cancel`). Si la tâche n'est plus disponible, la négociation est annulée et l'enchère gagnante est rejetée. Autrement l'enchère est acceptée et une confirmation est attendue.

Arrêter les *cfp*. Quand un *reducer* reçoit un message `Decline` de tous ses interlocuteurs en réponse à un de ses `Cfp`, il est inutile d'émettre à nouveau ce `Cfp` si le contexte reste le même. Le *reducer* peut alors entrer dans un état de pause pour prévenir l'envoi de `Cfp` inutile. Dans cet état le *reducer* continue à répondre aux autres `Cfp`. Il ne sort de cet état que si le contexte change. Le principe est le suivant : le contexte change lorsqu'un critère d'acceptabilité non satisfait peut le devenir. Les seules possibilités sont : i) une nouvelle tâche est ajoutée dans l'ensemble des tâches, faisant augmenter la contribution du *reducer* ; ii) le *reducer* est informé que la contribution d'une de ses accointances a diminué (une tâche a été déléguée ou un *worker* a terminé le traitement d'une tâche). Même si l'un de ces événements se produit, il n'y a aucune garantie que le critère d'acceptabilité soit rempli. Cependant, le *reducer* peut estimer s'il y a une chance ou non que cela se produise en gardant une trace des contributions de ses accointances. Cela est fait en stockant (et en mettant à jour) des informations sur les contributions reçues au travers des `Cfp` de ses accointances. Ainsi, le *reducer* peut estimer la possibilité que le critère d'acceptabilité soit rempli pour une de ses accointances et donc pour son `Cfp` d'être un succès. Quand c'est le cas, le *reducer* quitte l'*état de pause*. Nous verrons en 2.5 que la propriété 3 affirme qu'après un nombre fini de négociations, chaque agent est en *état de pause*. La propriété 4 affirme que cela se produit uniquement quand aucun transfert ne peut produire une meilleure répartition des tâches.

Comme les *workers* accomplissent leurs tâches, le contexte change et de nouveaux transferts de tâches deviennent possibles. Par exemple, cela peut se produire si un des *workers* travaille plus lentement que prévu au regard de l'estimation initiale du coût de la tâche. Dans cette situation,

les agents sortent de leur *état de pause* et commencent une nouvelle phase de négociation qui produira une meilleure distribution des tâches, c.à.d. une distribution qui permet de finir le traitement des données plus rapidement.

2.5 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.

Propriété 1. *La variance des contributions des reducers décroit après une négociation réussie.*

Preuve 1. *Soit $\Omega = \{1, \ldots, n\}$ un ensemble de n agents reducers. Considérons une négociation réussie menée par l'agent 1. On note :*

- *$(c_i)_{i\in\Omega}$, les contributions des agents avant la négociation ;*
- *$(c'_i)_{i\in\Omega}$, les contributions des agents après la négociation ;*
- *$\bar{c} = \frac{1}{n}\sum_{i=1}^{n} c_i = \frac{1}{n}\sum_{i=1}^{n} c'_i$ la contribution moyenne*[4] *;*
- *$Var = \sum_{i=1}^{n}(c'_i - \bar{c})^2$ la variance des contributions avant la négociation ;*
- *$Var = \sum_{i=1}^{n}(c'_i - \bar{c})^2$ la variance des contributions après la négociation.*

Soit $c > 0$ le coût de la tâche négociée et k l'agent reducer qui a gagné la négociation. Grâce au critère d'acceptabilité du participant k, on sait que $c_k + c < c_1$ et donc que $c_k + c - c_1 < 0$. Donc $Var' - Var < 0$.

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

Propriété 2. *La succession de négociations réussies fait décroitre la variance des contributions.*

Preuve 2. *Le protocole assure qu'aucun agent n'est impliqué simultanément dans plusieurs négociations : enchérisseur et initiateur sont deux rôles mutuellement exclusifs pour un agent broker ; engagé comme enchérisseur dans une négociation, l'agent broker ne répond pas aux cfps.*
Chaque négociation est donc indépendante, ce qui implique que le résultat d'une négociation n'a pas d'impact sur les autres. D'après la propriété 1, chaque négociation réussie fait décroître la variance des contributions, indépendamment des autres négociations, donc durant les négociations successives, la variance décroit.

Finalement, le processus de négociation termine.

Propriété 3. *La succession de négociations réussies termine.*

Preuve 3. *D'après la propriété 2, la variance décroit strictement (et est positive) durant la succession de négociations réussies. Le nombre de tâches étant fini, la variance ne peut plus décroître après un nombre fini de négociations réussies et donc aucune négociation réussie n'est plus possible.*

Le processus de négociation est correct :

Propriété 4. *Quand le processus de négociation termine, il n'existe plus de transfert de tâche qui pourrait réduire la contribution de l'agent le plus chargé.*

Preuve 4. *Soit l'agent j le plus chargé et τ la plus petite tâche de l'agent j. Supposons qu'il existe un agent i, avec une contribution c_i, tel que si i accepte le transfert de la tâche τ, cela fasse baisser la contribution la plus haute. Cela implique que $c_i + c_\tau < c_j$.*
Seulement, i aurait fait une proposition à un cfp venant de j pour la tâche τ. Ce cfp aurait mené à un succès ce qui est une contradiction.

3 Expérimentations et résultats pratiques

Afin d'évaluer notre proposition, nous avons implémenté deux versions de MapReduce : la version classique en utilisant la fonction de répartition par défaut de Hadoop et la version adaptative de notre système multi-agents. Ces deux implémentations ont été réalisées avec le langage de programmation Scala[5] moins verbeux que Java et la boîte à outils Akka[6] qui facilite le déploiement du modèle d'acteur sur une grappe de PCs.

Nous avons mené plusieurs expériences qui ont toutes donné lieu à une meilleure répartition des tâches par notre système adaptatif. Les expériences présentées ici concernent les données météo françaises depuis 1996[7]. Par

4. On note également que la négociation est conservative.

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

souci de simplicité, les *reducers* sont complètement connectés. De plus, ils sont tous situés sur une seule machine. La variabilité des résultats d'exécution liée au non-déterminisme de l'ordonnanceur est faible. Nous présentons donc une seule exécution par expérience. Les données représentent plus de 3 millions de relevés faits dans 62 stations (800 Mo).

Nous cherchons dans un premier temps à compter le nombre d'observations par demi-degré de température. Notre système de test contient 10 *mappers* et 20 *reducers*. La figure 4 montre la contribution des *reducers* ainsi que le coût de chaque tâche dans les deux versions de MapReduce. Chaque rectangle grisé représente une tâche dont le coût est proportionnel à la hauteur. Sur la gauche, la répartition par défaut mène à une distribution non équitable des tâches où seulement quelques *reducers* sont actifs. Sur la droite on observe que le système multi-agents, qui redistribue les tâches de manière dynamique au cours du processus, équilibre la charge de travail entre tous les *reducers*. Les *reducers* surchargés délèguent certaines de leurs tâches aux agents les moins chargés. La contribution de l'agent le plus chargé est alors réduite de 72 %, et l'équité qui mesure si le traitement est effectué à la charge de l'agent le plus chargé est améliorée en proportion. De plus, le ratio entre les contributions du moins chargé et du plus chargé des *reducers* passe de 0 à $0,7$.

Une seconde expérience, toujours sur les données météo, consiste à calculer la quantité de précipitations enregistrée dans chacune des stations. L'exécution concerne dans cette expérience 10 mappers et 10 reducers. On peut rappeler que la répartition initiale dans la version SMA correspond à celle de la version classique. En observant les contributions présentées dans la figure 5, nous constatons que la répartition est de nouveau équilibrée par notre système multi-agents. Dans cette expérience, la répartition atteinte par la version adaptative est proche d'une répartition optimale. Alors que la contribution maximale de l'allocation initiale est $\sim$ $439\,000$, celle de l'allocation multi-agents est $\sim$ $344\,000$. La contribution maximale d'une allocation idéale est $\sim$ $340\,000$. Il est important de noter que cette dernière est une borne théorique qui ne peut être calculée que *ex post*.

4 Conclusion

Les applications de MapReduce sont complexes à optimiser car elles sont basées sur des opérations définies par l'utilisateur 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, il est difficile de gérer la répartition de la charge entre les *reducers* car celle-ci est fixée indépendamment des données. Cela peut mener à une distribution du calcul qui est déséquilibrée. Un système multi-agents (SMA) est un modèle de calcul distribué et adaptatif. C'est pourquoi nous avons implémenté une version de MapReduce où l'allocation des tâches n'est pas établie par un utilisateur expert mais le résultat de négociations entre agents pendant la phase de *reduce*. Sans recourir à un aucun prétraitement ni expertise de l'utilisateur, l'adaptation du système est réalisée sans contrôle globale grâce à l'autonomie de décision locale des agents *reducers*. Plus précisément, notre modèle est basé sur des *reducers* composés de trois sous-agents coordonnés : le *manager*, le *broker* et le *worker*. Afin d'équilibrer la charge de travail, ces agents *reducers* négocient les tâches en se basant sur leur contribution individuelle dans le but de réduire la contribution de l'agent le plus chargé, c.à.d. celui qui retarde le traitement. Nos premières expérimentations sur des données réelles confirment qu'un SMA est approprié pour mettre en œuvre une telle allocation adaptative.

Ces résultats préliminaires mais prometteurs nous amènent à considérer plusieurs perspectives. Nous travaillons actuellement au déploiement du *framework* sur une grappe de PCs. Nous serons alors en mesure de nous comparer avec d'autres travaux cités dans cet article ainsi que de mesurer le surcoût lié aux communications.

Remerciements. Nous remercions les relecteurs pour leurs remarques. Ce travail s'inscrit dans le projet de recherche PartENS soutenu par le programme chercheur-citoyen de la région Nord Pas de Calais.

Références

[1] Baert, Q., Caron, A.-C., Morge, M., and Routier, J.-C. (2016). Fair multi-agent task allocation for large data sets analysis. In *Proc. of PAAMS, Sevilla, Spain.*, volume 9662 of *LNAI*, pages 24–35. Springer.

[2] Dean, J. and Ghemawat, S. (2004). MapReduce : Simplified data processing on large clusters. In *Sixth Symposium on Operating*

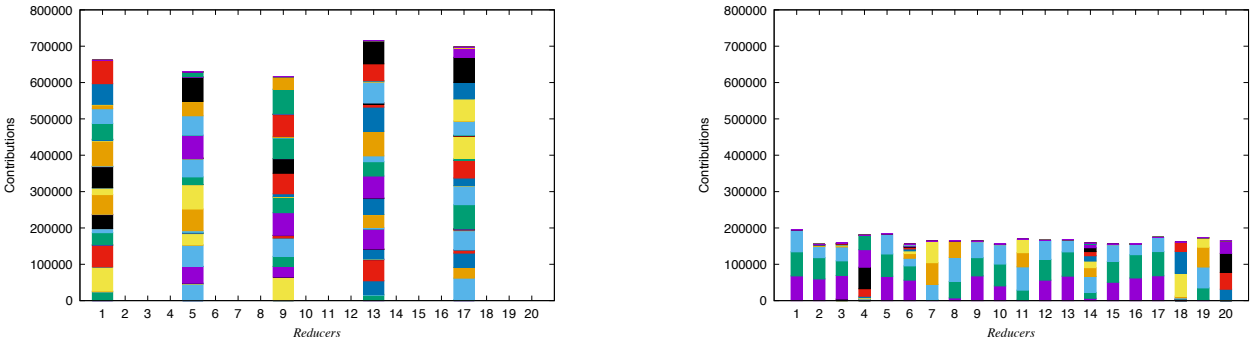


FIGURE 4 – Contributions des *reducers* pour la version classique de MapReduce où la répartition est fixée a priori (à gauche) et pour notre système multi-agents où la répartition est dynamique (à droite).

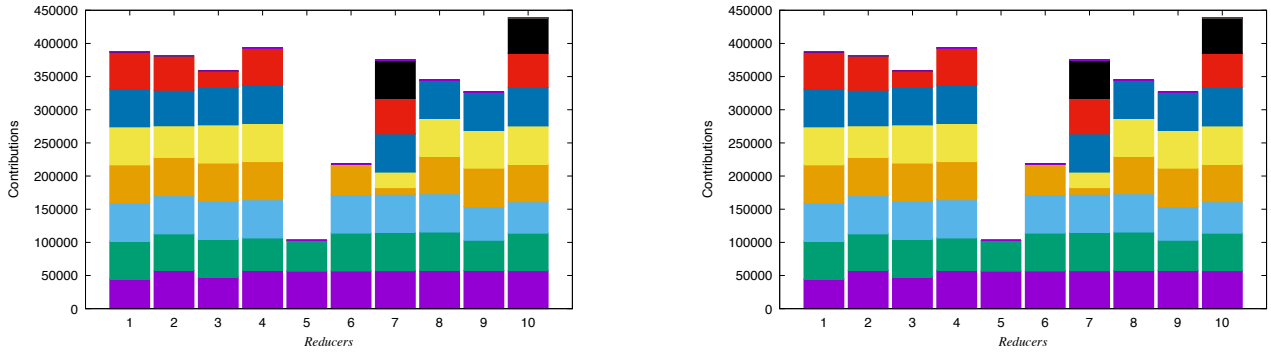


FIGURE 5 – Contributions des *reducers* pour la version classique de MapReduce (à gauche) et pour notre système multi-agents (à droite).

System Design and Implementation, pages 137–150.

[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo : Amazon's highly available key-value store. In *21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220.

[4] Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence*, pages 235–245.

[5] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. (2012). Skewtune : Mitigating skew in mapreduce applications. In *ACM SIGMOD International Conference on Management of Data*, pages 25–36.

[6] Lama, P. and Zhou, X. (2012). Aroma : Automated resource allocation and configuration of mapreduce environment in the cloud. In *9th International Conference on Autonomic Computing*, pages 63–72.

[7] Smith, R. G. (1980). The contract net protocol : High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, 29(12) :1104–1113.

[8] Verma, A., Cherkasova, L., and Campbell, R. (2011). Aria : Automatic resource inference and allocation for mapreduce environments. In *8th International Conference on Autonomic Computing*, pages 235–244.

[9] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets : A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX conference on Networked Systems Design and Implementation*, pages 15–28.