Negotiation Strategy of Divisible Tasks for Large Dataset Processing

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

Negotiation Strategy of Divisible Tasks for Large Dataset Processing *

Quentin Baert✉, Anne-Cécile Caron, Maxime Morge, and 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
{quentin.baert, anne-cecile.caron, maxime.morge, jean-christophe.routier}@univ-lille1.fr

Abstract. MapReduce is a design pattern for processing large datasets on a cluster. Its performances depend on some data skews and on the runtime environment. In order to tackle these problems, we propose an adaptive multiagent system. The agents interact during the data processing and the dynamic task allocation is the outcome of negotiations. These negotiations aim at improving the workload partition among the nodes within a cluster and so decrease the runtime of the whole process. Moreover, since the negotiations are iterative the system is responsive in case of node performance variations. In this paper, we show how, when a task is divisible, an agent may split it in order to negotiate its subtasks.

Keywords: Application of MAS, Automated negotiation, Adaptation

1 Introduction

The processing of large datasets requires to distribute data on multiple machines, typically a cluster of PC. Processing such distributed data is the purpose of the MapReduce design pattern [1]. A MapReduce application is composed of a map function which filters the data in order to build key-value couples and a reduce function which aggregates them. Many implementations of MapReduce exist, but the most popular is the open-source implementation Hadoop.

The user of a distributed application based on the MapReduce design pattern (e.g. Hadoop) must know its implementation, the input data and the runtime environment in order to configure beforehand the job. Nevertheless, even with a finely tuned configuration, data skews and heterogeneous runtime environments challenge the implementation choices.

In [2], the authors identify two common data skews during the reduce phase: (i) the partitioning skew occurs when a reducer processes a larger number of keys than others; (ii) the expensive key groups occurs when few keys are associated with a large number of values. As stated in [3], these two skews are

* This project is supported by the CNRS Challenge Mastodons.

widespread in todays applications and, they lead to an unbalanced workload of the reducers. Since the job ends when all the reducers have finished their work, the process is penalized by the most loaded reducer or (the slowest one). In this paper, we propose a multiagent system which implements the distributed MapReduce pattern where reducer agents negotiate divisible tasks while they process a job in order to improve the workload partition and so the runtime. The adaptivity of the MAS allows us to tackle both the reduce phase data skews and a heterogeneous runtime environment without data preprocessing. In [4], we have discussed the formal properties of our MAS and we have shown the advantages of a MAS architecture. Moreover, we have addressed the partitioning skew thanks to negotiations between reducers. Here, we address the expensive key groups skew with a negotiation strategy of divisible tasks. When the task delegation is socially irrational, our agents may split that task and negotiate the subtasks in order to reach a fairer task allocation. A reducer negotiates and splits tasks using its local beliefs about its peers workloads. These beliefs are updated during the reduce phase using the informations exchanged through the negotiations. Since the nodes may have heterogeneous performances, the process constantly adapts the distribution of the computations to the dynamics of the job processing. Furthermore, in order to improve the responsiveness of the MAS, we extend the negotiation process such that the agents can simultaneously bid in concurrent auctions. Finally, we show through several experiments that the workload balancing speeds up the data processing.

Section 2 presents the MapReduce design pattern and the related works. In Section 3, we shortly present our negotiation process with an illustrative example and we present our negotiation strategy of divisible task. Section 4 presents our experiments which highlight the added value of our adaptive multiagent system. Finally, Section 5 concludes and presents future works.

2 Motivation

MapReduce jobs consist of two sets of tasks, i.e. the *map* tasks and the *reduce* tasks, which are distributed among nodes within a cluster. A node which performs a map task (resp. a reduce task) is called a mapper (resp. a reducer). In order to perform such tasks, the nodes need these two functions given by the user:

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

Figure 1 illustrates the MapReduce data flow as implemented in Hadoop:

1. the supervisor shares input data by giving a slot to each mapper;
2. the mappers apply the map function over their slots and build the intermediate key-value pairs $(key : K2, value : V2)$;
3. a partitioning function is applied over the output of the mappers in order to split them in subsets, i.e. one subset per reducer such that the couples with

the same key are sent to the same reducer. In this way, a reducer process all the values of the same key (for data consistency);
4. the reducers aggregate the intermediate key-value to build the couples $(K2, list[V2])$. They apply the reduce function over the groups of values associated to each key;
5. the final key-value couples $(K3, V3)$ are written in the distributed file system.

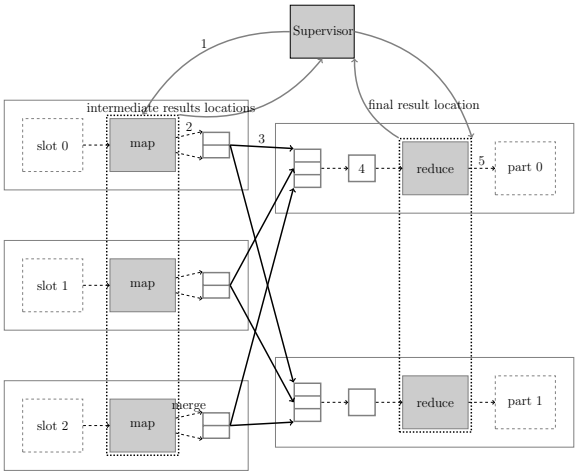


Fig. 1: MapReduce data flow.

Several criteria must be considered to compare our proposal to others[1]:

1. Prior knowledge about the data. Since the datasets are large, analyzing data beforehand is not realistic;
2. Data skew. We aim at addressing the data skews which lead to an unbalanced workload;
3. Self-adaptivity. We want the system to autonomously and constantly balance the workload.
4. Decentralization. A decentralized process is more responsive, robust and efficient than a centralized one.
5. Weak parametrization. Setting the parameters of a job requires to know the input data, the runtime environment and the MapReduce implementation. We aim at providing a solution which adapts the task allocation without expertise about the data and the computation environment.

The distribution of the MapReduce pattern needs to tackle the data skews which penalize the efficiency of the computation. [5] and [6] predict the performance with job profiling by collecting data during the previous runs. We do not

[1] Fault tolerance is out of the scope of our study.

want to preprocess data due to its computational cost, in particular for large datasets.

The partitioning skew leads to an unbalanced key allocation to the reducers. Without any prior knowledge about the data, the partitioning function cannot warrant a fair key allocation, and so a fair task allocation among the reducers. This data skew is tackled in [2, 7, 8] using centralized solutions with prior knowledge about the data and the environment or parametrized system. In [4], we also address it with a dynamic task allocation which is the outcome of concurrent negotiations between reducer agents all along the reduce phase. Unlike the other works, our proposal is decentralized and it does not require any configuration.

The data skew of expensive key groups is due to the fact that few keys are associated with a large number of values. For instance, it happens when the data can be approximated with a Zipfian distribution [9], i.e. the number of values for a key is inversely proportional to its rank in the frequency table. The consequent congestion phenomenon in the reduce phase is studied in [10]. Most of the time, this problem cannot be solved by a different key allocation to the reducers since a reducer is overloaded as soon as it is responsible for an expensive task. Even if this data skew is raised in [2], no solution is given. In [11], the authors also highlight this data skew. Since they claim that MapReduce requires that each key must be processed by a single reducer, they consider that the possibilities of fitting the system for this skew are very small. Then, their proposal is restricted to a user alert and ad-hoc solutions. In [8], the authors concretely tackle this issue. They propose to split the outputs of the mappers into blocks whose size must be *a priori* set up by the user. Thus, the values associated with the same key can be distributed in several blocks. The authors introduce the notion of *intermediate reduce* tasks which are applied to a subset of the values to produce *intermediate results*. Since the size of these tasks are parameterized, it is easier to balance the workload. The intermediate results for the same key are aggregated during the *final reduce phase*. This approach is centralized since the master node gathers all the information about the intermediate tasks and it orchestrates the reduce tasks allocation. Moreover, this proposal is based on some parameters which must be defined *a priori* (e.g. the size of non-divisible tasks). Finally, the task split is systematic. In this paper, we adopt a similar approach by splitting the tasks corresponding to the keys with a large number of values. The resulting subtasks can be negotiated and so dynamically allocated to the reducers in order to reach a balanced workload. However, unlike [8], the task split is only performed if necessary. Moreover, this mechanism does not require any predefined parameter to setup the size of the subtasks.

More generally, the multiagent approach for distributed problem solving, which encompasses distributed constraint solving [12] and negotiation [13], is suitable for adapting to unknown data and dynamic computing environments. It is worth noticing that, in most of the works on negotiation, agents give priority to their own goals. Conversely, in our context of distributed problem solving, agents have a common goal which has the priority over the individual ones. Contrary to [13], our mechanism does not allocate resources based on agents'

preferences once for all, but it iterates several task negotiations based on a local estimation of the remaining tasks to perform. In [12], the authors consider the problem of parallelizing heterogeneous computational tasks where each agent may not be able to perform all the tasks and may have different computational speeds. Let us note that this work addresses problems where the interactions between the tasks and machines are sparse, i.e. each agent can execute a small number of tasks and each task can be executed by a small number of agents. This is not the case in MapReduce applications. To our best knowledge, there are few works linking MAS and MapReduce frameworks. [14] presents a MapReduce pattern implementation based on mobile agents to replicate code and data for fault tolerance. However, this work does not apply self-organization techniques in order to adapt the MAS to the input data or the runtime environment. For this purpose, we adopt multiagent negotiation techniques.

3 Proposal

Our MAS addresses the two following data skews: the partitioning skew and the expensive key groups.

In order to address the partitioning skew, our reducers negotiate tasks using the Contract Net protocol. Doing so, they balance their workloads (called contributions) while they perform tasks during the reduce phase. It is important to note that task processing and negotiations simultaneously occur. Adaptation of the workload is then continuous. This process was presented in [4]. Here we extend it in order to allow agents to simultaneously bid in concurrent auctions, then to address the data skew of expensive key groups, we introduce a task split mechanism which allows reducers to partially process a task. This process leads to subtasks negotiations which refine the workload balancing.

3.1 Negotiation process

Let us recall the basic principles of the negotiation process through an example. We consider here a particular auction for a single MapReduce job.

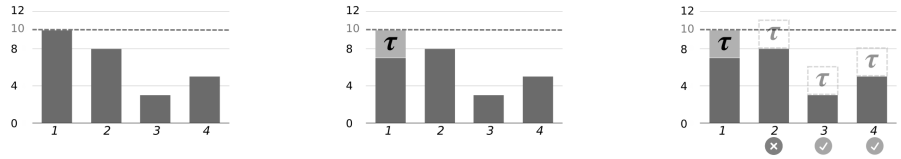
A reduce task represents a key and all its associated values. The cost of a task is defined by the number of values it contains. Thereby a task has the same cost for all the reducers of the system. We call *contribution* of a reducer the sum of costs for the remaining tasks it must perform.

In our example, we assume that the mapper phase has been completed and that the reduce tasks are initially allocated to four reducers, $\Omega = \{1, 2, 3, 4\}$. We focus on the task allocation at time t such that the individual contributions are $c_1(t) = 10$, $c_2(t) = 8$, $c_3(t) = 3$ and $c_4(t) = 5$ where $c_i(t)$ is the contribution of the agent i at time t (see Figure 2a). Each reducer has beliefs concerning the contributions of others. These beliefs are updated through information carried by the negotiation messages.

In order to decrease their contribution all the reducers initiate auctions. In particular, reducer #1 initiates an auction about the task τ with $c_\tau = 3$ through

a call for proposal (cfp) sent to the peers (see Figure 2b). A cfp contains the contribution of the initiator ($c_1(t)$) and the cost of the task to negotiate (c_τ). In order to decide if it can manage the task τ at time $t+1$, reducer i ($i \in \Omega \backslash \{1\}$) must satisfy the following acceptability criterion: $c_i(t)+c_\tau < c_1(t)$. The reducers which satisfy this criterion will improve the workload partition since they may take the responsibility of the task τ. Therefore, they make a proposal for τ while the others refuse the task delegation. For instance, reducer #2 does not take the responsibility of τ. Otherwise, its contribution $c_2(t) + c_\tau$ would be higher than $c_1(t)$. Meanwhile, reducer #3 and reducer #4 make some proposals for τ by sending their contributions to reducer #1 (see Figure 2c).
Reducer #1 receives the proposals of the agents $\Omega' = \{3, 4\}$. It updates its belief contributions about the other for future decisions. Then, it chooses to delegate τ to the least loaded bidder by applying the following selection criterion: $\underset{j \in \Omega'}{\text{argmin}}(c_j(t))$. In this way, reducer #1 accepts the proposal of reducer #3 and it rejects the one of reducer #4 (see Figure 2d).

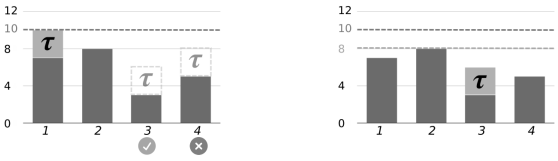
After the negotiation (at time $t + 1$), we observe that the task τ belongs to reducer #3. The new contributions are $c_1(t+1) = 7$, $c_2(t+1) = 8$, $c_3(t+1) = 6$ and $c_4(t+1) = 5$. Negotiation leads to a more efficient configuration, i.e. a fairer task allocation (see Figure 2e).



(a) Initial allocation: #1 is the most loaded one

(b) #1 proposes the task τ

(c) Acceptability criterion



(d) #1 applies the selection criterion

(e) Re-allocation

Fig. 2: Step-by-step negotiation process: reducer #1 delegates the task τ.

As it will be illustrated in our experiments (cf. Section 4), the task allocation is dynamic and adaptive since negotiations are repeated. For instance, if

a reducer is slowed down for some reasons, then an unbalanced allocation will appear, and so a negotiation will be triggered in order to delegate another task and decrease the current contribution of the delayed reducer.

Several reducers can simultaneously initiate negotiations but a reducer is either initiator or bidder at time t. In the following section, we will allow a bidder to be involved in more than one auction at a time.

3.2 Multi-auctions

We consider here a concurrent multi-auction process which allows agent to bid in several simultaneous auctions. In this way, the gap between the most loaded reducer and the least loaded one is filled faster and so, the responsiveness of the MAS is improved.

In order to manage several concurrent auctions, a bidder maintains an overhead of its current contribution. This overhead is the sum of all the task costs for which the bidder has made a proposal. In other words, a bidder records the overhead corresponding to the win of all the auctions in which it is involved. In this way, a bidder can make relevant proposals in several auctions and bids only if it may fulfill the acceptability criterion with its expected contribution (which includes its current contribution and its overhead).

Let c be the contribution of the reducer and o its current overhead. Initially, $o = 0$. When the reducer receives a cfp about a task τ with the cost c_τ from an initiator i with the contribution c_i, the bidder adopts the following behaviour:

- either $c + c_\tau \geq c_i$, the bidder does not fulfill the acceptability criterion and it declines the cfp;
- or $c+o+c_\tau < c_i$, the bidder makes a proposal with its expected contribution (i.e. $c+o$) since it fulfils the acceptability criterion whatever is the outcome of the pending auctions. Then, $o \leftarrow o + c_\tau$;
- or $c + o + c_\tau \geq c_i$ and $c + c_\tau < c_i$, its reply depends on the outcome of the pending auctions, so it stores the cfp in order to re-evaluate the corresponding acceptability criteria later.

When an auction ends, the bidder:

1. updates the overhead, i.e. $o \leftarrow o - c_\tau$, and possibly its contribution $c \leftarrow c + c_\tau$ if it wins the bid;
2. re-evaluates the stored cfps with its current contribution and the updated overhead. According to the previous behaviour, the bidder can reply to the stored cfps by an acceptance, a refusal, or postponing them.

Finally, the bidder only keeps the last cfp for each peer in order to avoid replying to a closed auction.

3.3 Task split: principle and bootstrapping

The skew of expensive key groups can be tackled neither with a static partition nor with our negotiation process. In order to allow reducers with expensive tasks to decrease their contributions by negotiation, we consider that the tasks are divisible. Therefore, we propose to split the expensive tasks into cheaper subtasks which are negotiable. In order to decrease the communicational and computational overhead of negotiations, tasks are split only if required.

As explained in Section 2, the tasks split requires to slightly modify the design pattern by introducing an intermediate reduce phase and a final reduce phase. Similarly to [8], we define the three following functions:

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

The user must decompose the reduce function R in an *intermediate reduce* function (IR) and a *final reduce* function (FR) such that for each key k and its values S, $R(k, S) = FR(k, < IR(k, S_1); \cdots ; IR(k, S_n) >)$ whatever is the partitioning of the values $S = S_1 \cup \cdots \cup S_n$.

The reduce function is an aggregation function. In [15], the authors identify three families of those functions: (i) the distributive functions where the reduce, the intermediate reduce and the final reduce are the same function (e.g. `sum`, `min`, `max`, `count`); (ii) the algebraic functions which can be decomposed using a bounded number of distributive functions (e.g. `avg` is decomposed by using an intermediate reduce function computing a `sum` and `count` the number of values, the division of the `sum` by the number of values is performed by the final reduce); and (iii) the holistic functions which are neither distributive nor algebraic (such as the median). It is difficult to find a relavante (IR, FR) decomposition for such functions since it may require too much intermediate data.

When a task is split into subtasks, IR is applied on each sub-task, producing an intermediate result, and FR is applied on all these intermediate results to produce the final result. Subtasks are considered as any other tasks, so they can be split and negotiated. The intermediate results have to be processed by the same reducer to compute the final result. So when a reducer initially splits a task, it collects the intermediate results and apply the final reduce function.

Let us study an example. The task τ is allocated to reducer i. This reducer splits τ in $\{\tau_1, \tau_2, \tau_3\}$. It processes τ_1 with IR function and it delegates τ_2 and τ_3 to reducers j and l, respectively. Reducer j splits τ_2 in τ_{21} and τ_{22} in order to delegate τ_{22} to a fourth reducer. The results of the application of IR on $\{\tau_1, \tau_{21}, \tau_{22}, \tau_3\}$ are sent to reducer i, i.e. the reducer which has split the initial task τ. Thereafter, i applies the final reduce function FR on these intermediates results.

In order to illustrate the negotiation strategy of subtasks, let us consider a population of n reducers. In order to bootstrap a task split, reducer i must fulfill the following split conditions:

1. there exist m reducers ($1 \leq m \leq n-1$) which are less loaded than i according to its beliefs;
2. i cannot delegate any task according to its beliefs;

Reducer i aims at decreasing its contribution. For this purpose, it splits its most expensive task in $k+1$ subtasks of the same cost with $1 \leq k \leq m$. The allocation of the k subtasks is conventionally negotiated with the peers.

3.4 Task split process

The task split heuristic is based on the beliefs of the reducer about the other reducers contributions. Let's remember that each reducer receives call for proposal from its pairs during negotiations. These messages contain the contributions of the auctioneers and allow the initiator to keep its beliefs up to date.

Definition 1 (Delta of contribution) *Let $\Omega = \{1, \ldots, n\}$ be a population of n reducers. At time t, each reducer i with the contribution $c_i(t)$ has a vector $\boldsymbol{r}_i = < r_{i_1}, \ldots, r_{i_{n-1}} > \in \Omega^{n-1}$ of its peers by increasing order of contributions. Let $c_{i_k}(t)$ be the estimated contribution of reducer r_{i_k} (i.e. the belief of reducer i about the contribution of the k^{th} reducer in $\boldsymbol{r}_i$). For each $r_{i_k} \in \boldsymbol{r}_i$, we define the delta of contributions as:*

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

According to the split conditions, if agent i can split a task, then there are m reducers which are less loaded than i. None of its tasks are negotiable, especially its biggest task τ and so we have: $\forall k \in [1; m],\ c_\tau \geq \Delta_i^k$. Thus, the split of the task τ aims at delegating k subtasks with the same cost. This delegation allows the reducers to decrease its contribution as much as possible.

Reducer i computes k such that:

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

This leads to the building of $k+1$ subtasks $\tau_1, \ldots, \tau_{k+1}$ with $c_{\tau_1} = \ldots = c_{\tau_k} = \frac{\Delta_i^k}{k+1}$ and $c_{\tau_{k+1}} = c_\tau - \frac{k\Delta_i^k}{k+1}$.

The following example illustrates how k is chosen and the impact it has on the contributions after negotiations.

Example Let $\Omega = \{1, 2, 3, 4\}$ be a set of four reducers with the contributions $c_1(t) = 80$, $c_2(t) = 20$, $c_3(t) = 40$ and $c_4(t) = 30$. Reducer #1 has two tasks: τ and μ. Since μ is the current running task, reducer #1 can only initiate an auction about the task τ which is not negotiable (see Figure 3). Therefore, there exist $m = 3$ reducers which are less loaded than reducer #1. The latter can split the task τ to decrease its contribution. We observe:

- $\boldsymbol{r}_1 =< 2, 4, 3 >$ (i.e. the peers by increasing order of contributions);

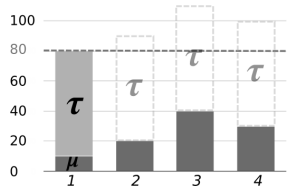


Fig. 3: Initial configuration where reducer #1 cannot negotiate the task τ.

– $\Delta_1^1 = c_1(t) - c_2(t) = 60$, $\Delta_1^2 = c_1(t) - c_4(t) = 50$, $\Delta_1^3 = c_1(t) - c_3(t) = 40$.

The number of subtasks modifies the resulting contributions. It is not always the case that $k = m$ gives the lowest contribution among the peers to the initiator.

If reducer #1 shares the task τ with a single reducer ($k = 1$), it builds the subtasks in order to balance its contribution with the peers. The best split to balance c_1 and c_2 consists in only considering Δ_1^1 and splitting it into two subtasks with the same cost. Therefore, the subtasks τ_1 and τ_2 are built from τ such that $c_{\tau_1} = \frac{\Delta_1^1}{2}$ and $c_{\tau_2} = c_\tau - c_{\tau_1} = c_\tau - \frac{\Delta_1^1}{2}$. In this way, reducer #2 may accept the task τ_1 which leads to a configuration where $c_1(t+1) = c_2(t+1) = 50$ (see Figure 4).

In the same way, the configurations with $k = 2$ (see Figure 4) is such that $c_1(t+1) = c_4(t+1) = 46$ and $c_1(t+1) = c_3(t+1) = 50$ in the configuration with $k = 3$ (see Figure 4).

More generally, if reducer #1 delegates k subtasks with the cost $\frac{\Delta_1^k}{k+1}$, its new contribution is $c_1(t+1) = c_1(t) - \frac{k\Delta_1^k}{k+1}$. We can observe that there is a value for k (here $k = 2$) which minimizes $c_1(t+1)$:

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

Due to lack of place, we only have presented a continuous tasks split. In reality, the tasks are composed of indivisible chunks of data (previously produced by the mappers). The actual task split process, which is similar to this continuous heuristic, take these chunks into account.

4 Experiments

Our experiments compare our proposal to the classical MapReduce distribution. Moreover, they evaluate the added-value of the negotiation of divisible tasks and of the multi-auction process. In other words, we compare our MAS with the one previously proposed in [4] using these metrics:

– the runtime of the reduce phase;
– the contribution fairness, i.e. the ratio between the minimum and the maximum contributions of the system at time t;

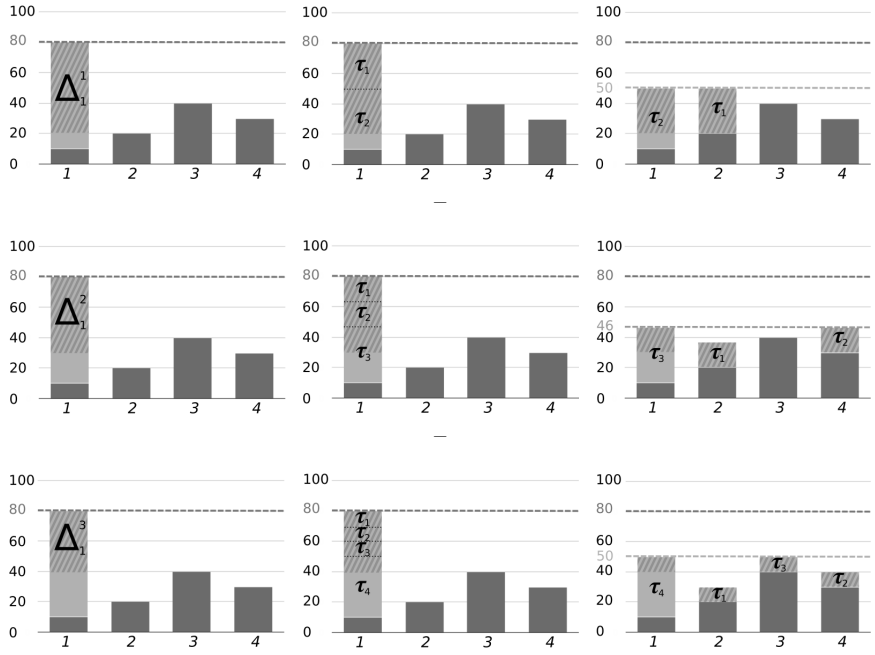


Fig. 4: Split of the task τ between reducers #1 and #2 (top), between reducers #1, #2 and #4 (center) and between reducers #1, #2, #3 and #4 (bottom).

– the runtime fairness, i.e. the ratio between the runtime of the slowest reducer and the runtime of the fastest one.

We have implemented our prototype with the programming language Scala and the Akka toolkit. The latter, based on the actor model, helps to fill the gap between the specification of the MAS and its implementation. Moreover, the deployment on a cluster of PCs is straightforward.

Previous experiments [16] have shown that our MAS is not penalized by the communicational and computational overhead due to the negotiation tasks since (i) if there are divisible tasks, the task split is performed only if required; and (ii) the reducers actually perform tasks while they negotiate. In this way, we do not increase the runtime of the reduce phase even if the workload distribution does not need to be improved.

Here, we make the assumption that the negotiation of divisible tasks may decrease the runtime of the reduce phase and that it helps the system to adapt itself to the heterogeneous performances of nodes.

Our experiments are based on a dataset[2] representing a snapshot of the Yahoo! Music community's preferences for various songs. The dataset contains over 717 million 5-star ratings of 136 thousand songs given by 1.8 million users of Yahoo! Music services. The data collected between 2002 and 2006 represents 10 Go. The job we consider counts the number of n-star ratings, with $n \in [1, 5]$. This dataset contains 4 "expensive" keys and 1 "cheap" one.

We compare the runtime of the reduce phase in the classical distribution of MapReduce with the MAS proposed in [4] and with our MAS which split tasks. We perform the job with one reducer per node. Since a reducer can process data from a mapper on another machine, the reduce phase is penalized by the non-locality of the data. This is the reason why we deploy the mappers on different machines than reducers in our experiments.

Figure 5 shows the runtimes according to the number of machines used, i.e. Intel (R) Core (TM) i5 3.30GHz PCs with 4 cores and 8GB of RAM. For each set of parameters, we perform 5 runs. Since the standard deviation due to the non-determinism of the scheduler is weak, we only show the averages on the different runs. The classical approach and the negotiation of indivisible tasks have the same performance since the 5 keys are not negotiable. They are not suitable for expensive key groups and so they are penalized by this data skew. By contrast, the negotiation of divisible tasks ends earlier since the available resources are better used. The runtime fairness reached by the negotiation of divisible tasks is about 0.99 while the runtime fairness of the classical approach is closed to 0 when at least one reducer does not perform any task and 0.36 with 5 reducers.

Figure 6 illustrates the dynamic of the contributions during the reduce phase due to the negotiation of divisible tasks between 12 agents. This mechanism quickly fills the gap between the most loaded reducer and the least loaded one during the whole process. Indeed, the negotiation of divisible tasks occurs simul-

[2] http://webscope.sandbox.yahoo.com/

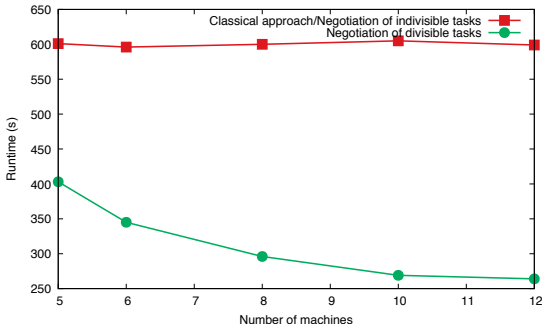


Fig. 5: Runtimes of the reduce phases

taneously to task processing, this makes it possible to dynamically and continuously allocate the tasks to the least loaded reducers. Additionally, we compare the concurrent multi-auction process which allows agent to bid in several simultaneous auctions to a single auction process where each reducer can be involved as an auctioneer in at most one auction at a time. While the single-auction process requires $58s$ to reach a contribution fairness greater than 0.70, the multi-auction process only needs $3s$[3]. Since the multi-auction process improves the responsiveness, the reduce phase with a multi-auction process is faster by $33s$ (around 12 %).

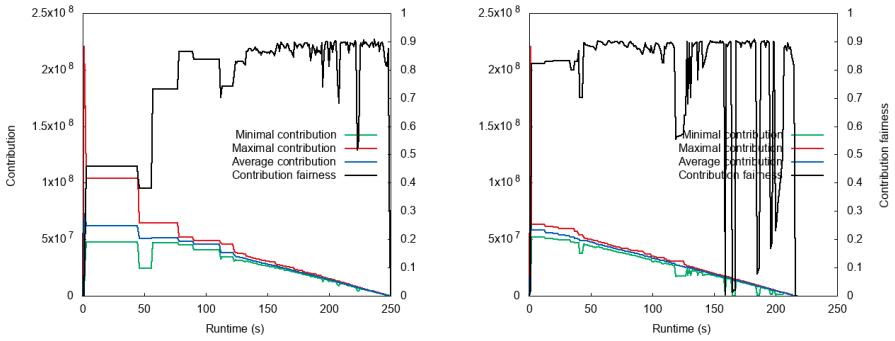


Fig. 6: Evolution of the contributions with a single-auction process (at left) and a multi-auction process (at right). The contribution fairness is defined in $[0; 1]$.

Finally, we have performed the same job on 7 machines. The first 6 reducers run alone on one computer and the 6 other reducers run together on a single computer and are then penalized. In our approach, the task allocation is adapted

[3] It is worth noticing that the negotiations and the data processing are not sequential but concurrent.

to the heterogeneous performance of nodes (cf. right of Figure 7). We can see that the system has distributed the tasks such that the reducers end their work at the same time according to the performances of their nodes. The dynamic and continuous task reallocation loads more the first 6 reducers which run alone on a computer until a runtime fairness of 0.99, i.e. the reducers finish their work almost simultaneously. Indeed, the negotiation of divisible tasks makes it possible to dynamically and continuously allocate the tasks to the fastest reducers. For comparison, left of Figure 7 shows the amount of work done by each reducer at the end of the job with an homogeneous environment.

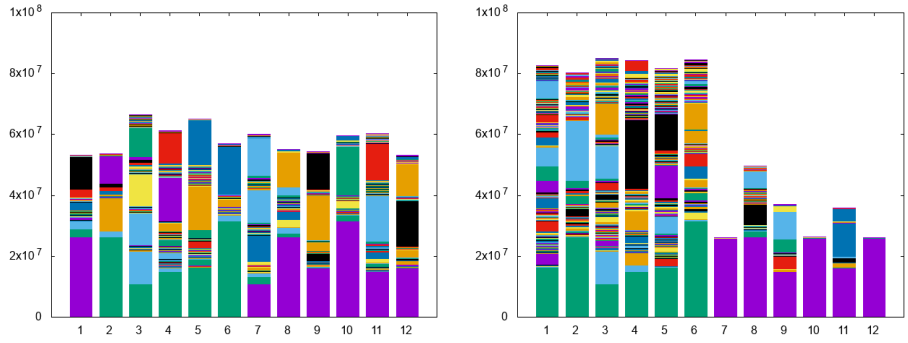


Fig. 7: The amount of work done by each reducer at the end of the job in a homogeneous runtime environment (at left) and in an heterogeneous runtime environment (at right).

These experiments show that our MAS benefits from the parallelism more than the other implementations. In particular, the negotiation of divisible tasks decreases the runtime of the reduce phase by improving the workload partition. Moreover, the multi-auction process improve the responsiveness.

5 Conclusion

In this paper we have shown how the deployment of the MapReduce design pattern using a MAS can address the data skews which penalize the reduce phase, in particular the expensive key groups. Our system requires neither preprocessing nor parametrization depending on the data but it addresses most of the practical applications. In our MAS implementation, the reducer agents split "non-negotiable" tasks and negotiate the task allocation while they process the data. Their decisions are based on the remaining data to process, i.e. their contributions. This continuous decision-making process is local and it requires no centralization of information. In order to improve the responsiveness, we have proposed a multi-auction process which allows agent to bid in several simultaneous auctions. It is worth noticing that if the runtime environment is

heterogeneous, the system is self-adaptive. Our experiments have shown that the negotiation of divisible tasks can decrease the runtime of the reduce phase by iteratively relieving the most loaded (or slowest) reducers.

In future works, we will integrate a data locality criterion in the collective decision-making process in order to limit the data transfer cost. For this purpose, we plan to abstract away from our practical application to consider the general problem of dynamic task re-allocation between heterogeneous machines.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: SOSDI. (2004) 137–150
2. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.: Skewtune: Mitigating skew in MapReduce applications. In: ACM SIGMOD ICMD. (2012) 25–36
3. Kwon, Y., Ren, K., Balazinska, M., Howe, B.: Managing skew in Hadoop. IEEE Data Eng. Bull. **36**(1) (2013) 24–33
4. Baert, Q., Caron, A.C., Morge, M., Routier, J.C.: Fair multi-agent task allocation for large datasets analysis. KAIS (2017) 10.1007/s10115–017–1087–4
5. Lama, P., Zhou, X.: Aroma: Automated resource allocation and configuration of MapReduce environment in the cloud. In: ICAC. (2012) 63–72
6. Verma, A., Cherkasova, L., Campbell, R.: Aria: Automatic resource inference and allocation for MapReduce environments. In: ICAC. (2011) 235–244
7. Chen, Q., Zhang, D., Guo, M., Deng, Q., Guo, S.: SAMR: A self-adaptive MapReduce scheduling algorithm in heterogeneous environment. In: ICCIT, IEEE (2010) 2736–2743
8. Liroz-Gistau, M., Akbarinia, R., Valduriez, P.: FP-Hadoop: efficient execution of parallel jobs over skewed data. VLDB Endowment **8**(12) (2015) 1856–1859
9. Li, W.: Random texts exhibit Zipf's-law-like word frequency distribution. IEEE Transactions on Information Theory **38**(6) (1992) 1842–1845
10. Lin, J.: The curse of Zipf and limits to parallelization: A look at the stragglers problem in MapReduce. In: Workshop on Large-Scale Distributed Systems for Information Retrieval. (2009) 2009
11. Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Handling data skew in MapReduce. In: ICCCSS. (2011) 574–583
12. Vinyals, M., Macarthur, K.S., Farinelli, A., Ramchurn, S.D., Jennings, N.R.: A message-passing approach to decentralized parallel machine scheduling. The Computer Journal **57**(6) (2014) 856–874
13. Nongaillard, A., Mathieu, P.: Egalitarian negotiations in agent societies. AAI **25**(9) (2011) 799–821
14. Essa, Y.M., Attiya, G., El-Sayed, A.: Mobile agent based new framework for improving big data analysis. IJACSA **5**(3) (2014) 25–32
15. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Mining and Knowledge Discovery **1**(1) (1997) 29–53
16. Baert, Q., Caron, A.C., Morge, M., Routier, J.C.: Stratégie de découpe de tâche pour le traitement de données massives. In Garbay, C., Bonnet, G., eds.: Journées Francophones sur les Systèmes Multi-Agents. Cohésion : fondement ou propriété émergente, Caen, France, Cépaudès édition (July 2017) 65–75