Distributed computing algorithm with high performance for big data

Randriamampiandry Rudy – Rakotomanana René – Randimbindrainibe Falimanana

Ecole Supérieure Polytechnique Antananarivo (ESPA) - Université d'Antananarivo
BP 1500, Ankatso – Antananarivo 101 – Madagascar

[1] *rudrandria@gmail.com,* [2] *reneheli@yahoo.fr,* [3] *falimanana@mail.ru*

Abstract

Many fields of science are now facing a deluge of data. One of the proposed approaches to enable the processing of such volumes is the MapReduce programming paradigm introduced by Google. This very simple execution scheme consists of two phases, map and reduce between which takes place a phase of massive data exchange between the machines executing the application. In this article, we propose a linear system defining a partitioning of the data to be processed and a dynamic transfer scheduling algorithm in order to optimize this intermediate phase.

Keywords

MapReduce, distributed algorithm, distributed applications, big data

---

1.Introduction

Data is growing at an explosive rate, entering the enterprise from different domains and in a myriad of formats. Social media, sensor data, spatial coordinates, and external data resource providers are just a few of the new data vectors businesses now need to tackle. Big Data is a set of data of such Volume, Velocity and Variety that it cannot be processed in traditional relational database management systems (RDBMS), Veracity is also an essential element of Big Data, it refers to the reliability of the data and the last element is the Value, it refers to the fact that each data must bring added value to the company.

Meeting the need for big data management requires fundamental changes in the architecture of data management systems. Among them are the highly distributed workflow processing systems that are at the heart of managing massive volumes of data.

This data can be an input to an application or an intermediate output that needs to be stored and managed. Some such applications include high-performance scientific data processing techniques and real-time streaming applications. These applications are subject to a series of calculation phases. Workflow Frameworks integrate and coordinate multiple tasks that may contain multiple collaborative tasks. Some of these tasks are executed sequentially but others can be executed in parallel on a distributed platform. A large amount of data is generated daily from these workflow processing systems which are extremely valuable with a wide diversity of types, it becomes difficult to process and store. Other applications process a massive data workflow using the MapReduce paradigm adopted and integrated by large companies like Google, Facebook, Amazon and LinkedIn. Such an application ecosystem requires a flexible composition of workflow tasks supporting different processing phases.

In this article, we identify the limitations of the algorithm proposed by Berlinska and Drozdowski [1] and propose an original algorithm for scheduling data transfers during the shuffle phase of a MapReduce application. The main objectives of this algorithm are to guarantee the non-congestion of the interconnection network, to limit the inactivity times of the computing nodes during this phase of communication, and finally to minimize the completion time of the application.

The remainder of this article is organized as follows. Section 2 presents various previous works related to this issue. In section 3 we detail the application and platform models that we used to design scheduling and partitioning algorithms optimizing data transfers during the shuffle phase. Finally, we will conclude this article and present our future work in section 5.

2.Related works

Many works have focused on the problem of the cost of data transfers within MapReduce applications. Most of them deal with data locality during the mapping phase. One of the proposed algorithms [8] improves this locality by introducing a delay before migrating a task to another node, if the preferred node is not available. The BAR algorithm [4] aims to approach the optimal data distribution given an initial configuration that will be dynamically adapted.

LEEN [3] is an intermediate key partitioning algorithm that aims to balance the duration of reduce while trying to reduce the bandwidth consumption during the shuffle. This algorithm is based on statistics of frequencies of appearance of intermediate keys in an attempt to create balanced partitions and optimize data transfers.

The HMPR algorithm [5] proposes a pre-shuffling which tends to reduce the quantity of data to be transferred as well as the number of transfers. For this, it predicts the partition in which the data will be generated at the output of the map and has the piece of data processed by the node which will execute the reduce of this partition if possible.

The Ussop execution environment [6], targeting computing grids, adapts the amount of data to be processed by each map according to the computing power of the machine that runs it. In addition, this tool tends to reduce intermediate data transfers by locally executing the reduce on the machine that generated the most intermediate keys.

A MapReduce application can be considered as a set of divisible tasks since the data to be processed can be distributed indifferently between map instances. It is therefore possible to apply results from the theory of divisible tasks [7] to this type of application. This is the approach that was followed by Berlinska and Drozdowski [1]. In this article, the authors consider an execution environment whose number of computing nodes is greater than the number of communications that can take place simultaneously without causing contention. To avoid the appearance of this phenomenon, they propose to model the execution of a MapReduce application by a linear program which generates a distribution of the data and a static scheduling by phases of the communications. If this approach is interesting, the use of a linear program makes it inapplicable for instances involving more than a few hundred maps because the resolution time can sometimes exceed several minutes. Also, sometimes the linear program solver fails for some instances. The scheduling by phases induces, moreover, a large number of inactivity times on the machines and the network during the shuffle. In the following section, we propose original solutions to these problems in a similar framework of use.

3.Contribution

3.1 Models

The platform and application models used in this article are similar to those used by Berlinska and Drozdowski [1] who, in particular, expressed throughputs in seconds per byte so as to avoid splits in their linear programs. We use these units here so as to remain consistent with their work. We therefore consider a cluster of machines interconnected by a single switch. The network thus formed is therefore star-shaped. The links connecting the machines to the switch are homogeneous, without latency and with speed C expressed in seconds per byte. The main factor limiting the performance of a MapReduce application is the capacity of the switch. We consider that this is much lower than the sum of the bandwidths of the links connecting each machine to the switch.

For simplicity, we define the switch throughput as a multiple of the link throughput: $\sigma = C/l$. In other words, the switch is able to serve l communications simultaneously without inducing contention. Beyond that, it becomes a bottleneck and the performance of all communications in progress is degraded. Finally, the processing capacity of the machines is A_i seconds per byte.

A MapReduce application is mainly represented by the amount of data α_i to be processed by each map. The total amount of data to be processed is called V, and is equal to the sum of the α_i. We also define γ as being the ratio between the amount of data passed as a parameter of a map and the volume of intermediate data produced. During the shuffle phase, each map will therefore have to distribute $\gamma \times \alpha_i$ bytes between different reduce tasks. Finally, our model introduces a delay when starting map tasks. Each node is started sequentially with a delay S between each start. This can be due, for example, to the loading time of the application code on the nodes. For the sake of simplicity, we consider that each node executes only one mapper or reducer process.

3.2 Berlinska and Drozdowski's approach

In their article [1], Berlinska and Drozdowski attempt a global optimization of partitioning and ordering. In order to avoid sharing the bandwidth of the links and to avoid contention, they opt for a phased arrangement of simultaneous and ordered transfers. The order relation between transfers can be defined as follows.

$$start(i;j) > end(i;j-1) \qquad for\ i \in 1 \dots m\ ;j \in 2 \dots r \qquad (1)$$
$$start(i;j) > end(i-1;j) \qquad for\ i\ \in 1 \dots m;j \in 2 \dots r \qquad (2)$$

With $start(a;\ b)$ and $end(a;\ b)$ giving respectively the start and end dates of the transfer from node a to node b, and m and r represent respectively the number of nodes executing map tasks and tasks reduce.

Each map process therefore transfers its data first to reduce process 1, then to reduce 2, etc. Each map process must wait for the previous map process to finish its transfer to reduce *j* to begin its transfer to reduce *j* itself.

$$itv(i,j) = \left(\left\lfloor \frac{j}{l} \right\rfloor - 1\right)m + i(j-1)\ mod\ l \qquad for\ i \in 1 \dots m\ ;j \in 2 \dots r \qquad (3)$$
$$vti(i) = a|itv(a,b) = i \qquad b \in 1 \dots r \qquad (4)$$

Equation (3) defines the function itv which gives the number of the time interval during which mapper i will transfer to reducer j. Equation (4) defines vti as the reciprocal application of itv. For a given interval i, it matches the set of mappers that will transfer during that interval. Finally $itv(r;m)$ corresponds to the last transfer that will be performed.

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

The above linear program (5)– (8) attempts to minimize the end date of the last transfer. Constraint (6) causes the first transfer to begin when the computation ends. Inequality (7) indicates that the size of an interval must be large enough to fit all the expected transfers into it. And finally the sum (8) ensures that we process all the data.

3.3 Partitionning

the first transfer of a mapper i ends when the calculation of the mapper $i\ +\ 1$ also ends, in other words:

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

This is verified whenever $\alpha_i < \alpha_{i+1}$. Moreover, when the platform is homogeneous (all A_i are equal), then $\alpha_i < \alpha_{i+1}$ is equivalent to:

$$S \times r \times m < \gamma \times V \times C \qquad (9)$$

However, the start-up time S (of the order of a few seconds) is generally much lower than the transfer time of all the data to be processed (of the order of a terabyte). Under these conditions, we therefore propose to calculate the partitioning by the following linear system.

$$iS + \alpha_i A_i + \frac{\alpha_i \gamma C}{r} = (i+1)S + \alpha_{i+1}A_{i+1} \qquad for\ i = 1, \dots, m-1 \qquad (11)$$
$$\sum_{i=1}^{n} \alpha_i = V \qquad (12)$$

This linear system can be solved in $O(m)$ time and $O(m)$ space. Experience shows that this linear system calculates α_i similar to the linear program (5) – (8) except for rounding errors.

3.4 Scheduler

In the chosen platform model, it is necessary to avoid contention on the switch and the communication links. For this, we keep the constraints of order (1) – (2). Nevertheless, the global limit of simultaneous transfers imposes to choose the transfers to be carried out at a given moment. The heuristic we propose is to keep $i + j$ constant for each pair of $mapper$ process i and $reducer$ process j. In practice, this means that when $mapper$ 1 is forwarding to $reducer$ 5, $mapper$ 2, if not forwarding to $reducer$ 4, will be encouraged to do so.Algorithm 1 presents the strategy we propose. In this algorithm, node.state contains the representation of the node's current activity, and target node contains the identifier of the reduce to which node is transferring or wants to perform its next transfer. When the transfer from a mapper i to a reducer j ends, for each inactive mapper i′ and its next target reducer j', we determine $p_{i'} = i' + j'$. Then we select the nodes which minimize $p_{i'}$. These nodes are considered the most delayed and their transfers must begin as soon as possible. This favors the maximization of the use of the available bandwidth and makes it possible not to violate the constraint (2) without making it explicit in the algorithm. The `ON_COMPUTE_END` procedure is called as soon as a map task finishes processing all of its data. It calls the `REQUEST_TRANSFER` procedure which will start the requested transfer if this does not violate the constraints on the use of the bandwidth. The `ON_TRANSFER_END` procedure is called when a transfer ends. It begins by initiating the transfer with the highest priority if it exists. Then it initiates the next transfer of the node whose transfer has just ended, if possible. This algorithm respects the constraints imposed on the use of the network while occupying it to the maximum at all times. Indeed, if the bandwidth of the switch is already saturated, then only the transfer initiated by the call to `REQUEST_TRANSFER` on line 28 will be effective, the termination of a communication only allowing the start of a single new transfer. On the other hand, if the order constraints (1) and (2) have prevented transfers from starting, the switch is not saturated. The termination of a transfer then only allows to start at most two new transfers, which is done lines 28 and 32.

Algorithm 1 Transfer scheduling algorithm

```
1: procedure REQUEST_TRANSFER(node)
2:   if the network link of the target reducer is busy or the limit of the switch has been reached then
3:     node.state ← IDLE
4:   else
5:     node.state ← TRANSFER
6:     START¯TRANSFER(node, node.target)
7:   end if
8: end procedure
9: procedure ON_COMPUTE_END(node)
10:  REQUEST_TRANSFER(node)
11: end procedure
12: function NODE_TO_WAKE
13:   for all N node in IDLE state do
14:     if target reducer's network link is busy then
15:       continue with next node
16:     endif
17:     p[N] ← number of N + number of N.cible
18:   end for
19:   if p is empty then
20:     return undefined value
21:   else
22:     return N for which p[N] is the lowest
23:   end if
24: end function
25: procedure ON_TRANSFER_END(node)
26:   n ← NODE_TO_WAKE
27:   if n is not undefined then then
```

```
28:     REQUEST_TRANSFER(n)
29:   endif
30:   if node hasn't done every transfers then
31:     node.target ← next node
32:     REQUEST_TRANSFER(node)
33:   else
34:     node.state ← TERMINATED
35:   endif
36: endprocedure
```

4.Conclusion

In this article, we focused on optimizing the shuffle phase of a MapReduce application. For this we have proposed a linear system coupled to a dynamic transfer scheduler. In the future, we plan to apply these algorithms and compare the results of the experiment to those of Berlinska and Drozdowski based on a linear program and a static scheduler per phase, we hope for better runtimes, faster, more stable, and fail-safe schedule construction, and better scalability.

5.Bibliography

1. Berlinska (J.) et Drozdowski (M.). – Scheduling Divisible MapReduce Computations. Journal of Parallel and Distributed Computing, vol. 71, n3, mars 2010, pp. 450–459.

2. Dean (J.) et Ghemawat (S.). – MapReduce : Simplified Data Processing on Large Clusters. In : Proc. Of he 6th Symposium on Operating Systems Design & Implementation (OSDI). pp. 137–150. – San Francisco, CA, décembre 2004.

3. Ibrahim (S.), Jin (H.), Lu (L.), Wu (S.), He (B.) et Qi (L.). – LEEN : Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. In : Proc. of the Second IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 17–24. – Indianapolis, IN, novembre 2010.

4. Jin (J.), Luo (J.), Song (A.), Dong (F.) et Xiong (R.). – BAR : An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing. In : Proc. of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). pp. 295–304. – Newport Beach, CA, mai 2011.

5. Seo (S.), Jang (I.), Woo (K.), Kim (I.), Kim (J.-S.) et Maeng (S.). – HPMR : Prefetching and Pre-shuffling in Shared MapReduce Computation Environment. In : Proc. of the 2009 IEEE International Conference on Cluster Computing (Cluster). – New Orleans, LA, septembre 2009.

6. Su (Y.-L.), Chen (P.-C.), Chang (J.-B.) et Shieh (C.-K.). – Variable-Sized Map and Locality-Aware Reduce on Public-Resource Grids. FGCS, vol. 27, n6, juin 2011, pp. 843–849.

7. Veeravalli (B.), Ghose (D.), Mani (V.) et Robertazzi (T.). – Scheduling Divisible Loads in Parallel and Distributed Systems. – IEEE Computer Society Press, 1996, 292p.

8. Zaharia (M.), Borthakur (D.), Sarma (J. S.), Elmeleegy (K.), Shenker (S.) et Stoica (I.). – Job Scheduling for Multi-User MapReduce Clusters. – Rapport technique n UCB/EECS-2009-55, EECS Department, University of California, Berkeley, avril 2009.