On Optimizing Data Locality in Hadoop

Thibaut Marty Joris Dugueperoux Timothée Haudebourg Grégoire Bonin

May 9, 2016

Abstract

Nowadays, computing over big data becomes more and more important. In this context, the use of the Map-Reduce paradigm is preponderant in cloud computing.

The first step of this paradigm, the map phase, will perform map tasks on data stored on each node. Hence, it becomes important to optimize the storage of data through the nodes, such that each node can perform tasks locally without interruption.

In order to maximize data locality, we propose various approaches based on adaptation of the scheduling policy and of the data distribution.

1 Introduction

To compute over big data, the Map-Reduce paradigm was developed by Google in [2], and then implemented in other frameworks such as Hadoop.

In this article we will deal with data locality and scheduling policy, and the ways in which they can improve computing effectiveness in Hadoop. In Hadoop, the data is divided into data **chunks** which are then distributed onto several nodes to be computed in the map tasks. The reduce tasks then aggregate the results of map tasks to compute the final result. We have studied and introduced chunks distribution algorithms and scheduling policies, and conducted experiments over iHadoop (a framework based on SimGrid [1] to simulate Hadoop behaviour) to test whether they were better or not than the state-of-the-art algorithms.

Currently, even if it sounds intuitive that data locality implies better performances and if there are many theoretical argument to converge in this direction, it is hard to determine exactly the real gain and the value of such an improvement.

For that reason, we decided to first try to evaluate the real impact of data locality by analysing the log files generated in Hadoop.

To do so, we used the Hadoop log dataset released by Kai Ren and Garth Gibson, and first exploited in their publication [4].

First, we computed the average length of a map task, whether they are locally computed or not. As a result, we determined that in April 2010 and on the collected log datasets, the average length of a locally executed map task was $57.4s$, against $98.7s$ for tasks in which the data first have to be moved, which means that the average gain on a map task is 41.8%.

Furthermore, we can also show in various cases how interesting it is to apply data locality. Indeed, the figure 1 shows, in two map heavy profiles of jobs, the average duration of a job according to the percentage of data locality.

From these figures, we can hence understand the interest of data locality, and then our motivation to optimize it as much as possible. Indeed, these graphs represent the computation time of two

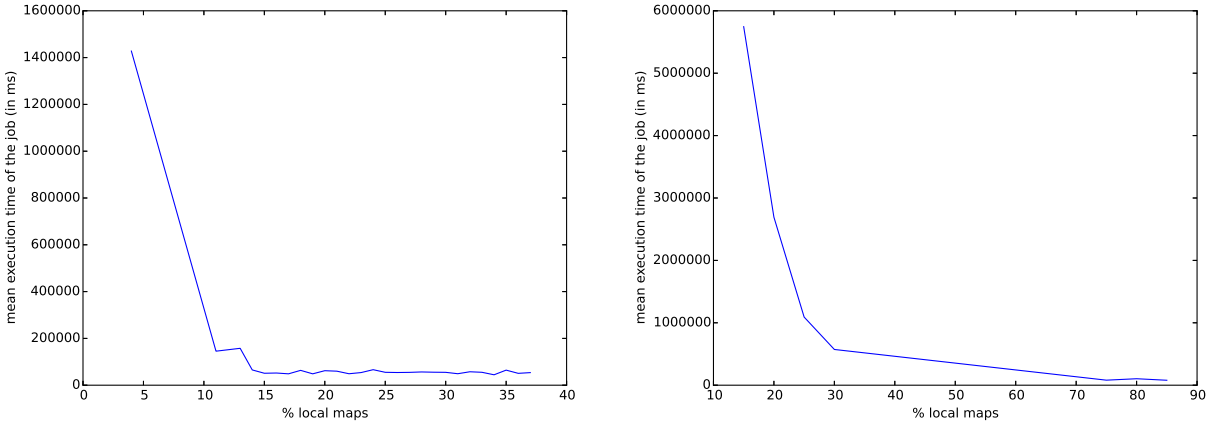

Figure 1: Average duration of a job in accordance with the percentage of local data for jobs which have on the left between 95 and 100 map and 40 and 50 reduce tasks, and on the right between 500 and 1000 map and between 0 and 50 reduce tasks.

profiles of map-centered jobs. They both show that a low data locality highly increases the computation time, even if the advantage given by data locality quickly becomes stable. However, the limit beyond which the interest of data locality decreases seems to depend on the category of jobs, which incites us to maximize it regardless to this stability.

2 Related Work

2.1 Chunks Distribution

Data Locality is important to achieve a good execution time. We present here several policies to distribute data chunks over the nodes, which adopt several views on data locality and resistance over failure.

2.1.1 Kutlu

Kutlu et al. introduce in their article [5] chunks distribution policy to make data as fault tolerant as possible by not having more than one common data chunk between every two nodes. This is possible thanks to the introduction of a shift in the replicates distribution. For example, the first replica is shifted by one node per row, and the second replica is shifted by two nodes per raw.

The algorithm works as follows: Having S data chunks to compute per node, and a replication factor of R, with a number n of nodes (where $n = R \times p$), then we have $S \times R \times p$ data chunks per block (a block is a group of p nodes) to place. For each block, we place the data chunks according to the shift induced by its block number and row number: the shift is equal to $blockNumber \times RowNumber$. There are $S \times R$ rows per block. In figure 2, there are 3 nodes per block, a replication factor of 3, and $S = 1$. Hence, we organize data in 3 rows and the data chunks are distributed as presented.

There are two conditions to have no more than one common data chunk between every couple of nodes: $S \times R$ must be lower than p and p must be a prime number.

Figure 2: Example of a distribution of chunks according to Kutlu's policy

2.1.2 Xie

Xie et al. introduce in their article [6] an other chunks distribution policy in a heterogeneous cluster. It begins by estimating the computation power of the nodes and then attributes data chunks to the nodes proportionally to its estimated power.

The algorithm maintains two lists during the execution: a list of overloaded nodes and a list of underloaded nodes and tries to balance the nodes by moving data chunks from overloaded nodes to underloaded nodes.

This method is mainly useful for nodes or data addition and suppression, to balance the workload between nodes, but It is useless in our case, because we have a homogeneous clusters.

2.2 Scheduling Policy

An other way to achieve data locality is to work on the scheduler side, which assigns map tasks to nodes, instead of working on the chunk distribution policy. In their article, Guo et al. [3] studies the importance of data locality in Map-Reduce, describe a scheduler base on the Linear Sum Assignment Problem (LSAP).

When a job is coming, tasks assignment problem is transformed into a simple LSAP problem (between tasks and nodes) and a LSAP resolution algorithm, such as the Hungarian Algorithm, is applied.

3 Experiments and results

3.1 Chunk distribution

The chunks distribution algorithms introduced in Section 2 are not designed to work in our context, with Hadoop. The following section discuss the adaptation of these algorithm in iHadoop, and the associated experiments.

3.1.1 Kutlu

The original Kutlu algorithm splits each data in different "data blocs". In our case, the term "data" refers to a HDFS chunk. Those chunks are not splittable, and the algorithm is simplified.

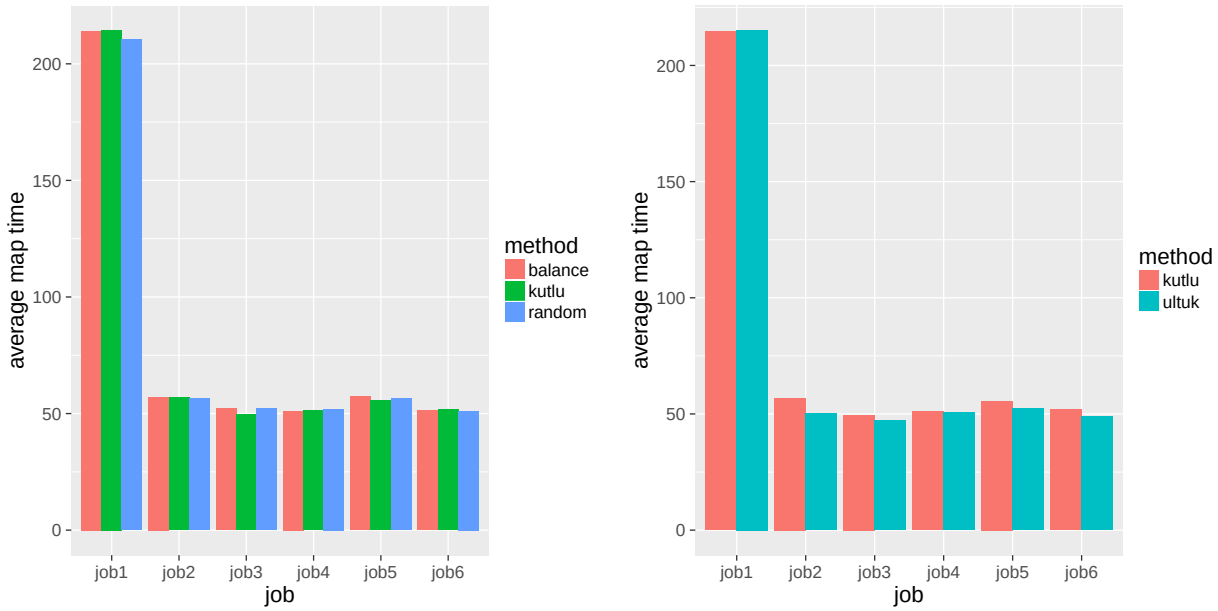

Figure 3: Average map time with different distribution methods for different job types

Algorithm 1: Hadoop version of Kutlu's algorithm

Input: Chunks, the list of chunks to distribute. Nodes, the list of nodes. R, the replicas factor.

```
// How many nodes per class
n ← Nodes.count/R;
forall c_i ∈ Chunks do
    line ← i/n;
    column ← i mod n;
    for r := 0 to R-1 do
        offset ← r × line;
        destination ← column + offset;
        send(c_i, n_destination ∈ Nodes);
```

The Algorithm 1 corresponds to our implementation of Kutlu's algorithm in the Hadoop context.

We compared the performances of Kutlu with different distribution methods already implemented in iHadoop. We first tried to evaluate the performance of Kutlu in terms of data locality.

Figure 3 shows the results of our experiment for different types of jobs.

It seems difficult to conclude that Kutlu improve either data locality or the average map duration.

It can be noticed that, with Kutlu (and other distribution algorithms), there is always a way to schedule map tasks in a way that forbid the execution of local map tasks for at least one node. For example, in a pool of k nodes, selecting tasks for the $k-1$ firsts nodes that use all the chunks of the k^{th} node, refusing any local task for the node k.

Distributing chunks in a way that minimize the overlap between chunks seems not to be a good idea to improve data locality.

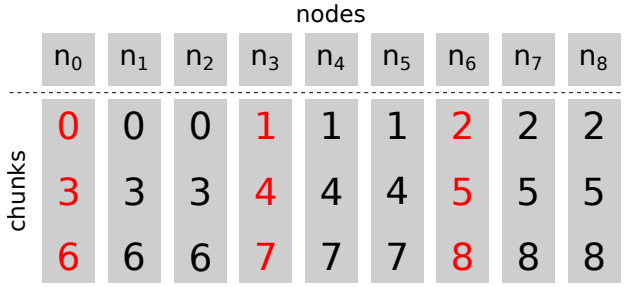

Figure 4: 100% Locality distribution by maximizing overlap between chunks

3.1.2 Ultuk

We made the hypothesis that the best way to achieve 100% locality is not by minimizing overlap between chunks (Kutlu's way), but by maximizing it.

Figure 4 presents a way to reach 100% locality with the same example as seen in section 2.1.1. This time, chunks are grouped by 3. Replicas of the same group will be in the same nodes.

Algorithm 2: Our "Ultuk" distribution algorithm

Input: Chunks, the list of chunks to distribute. Nodes, the list of nodes. R, the replicas factor.

forall $c_i \in Chunks$ **do**
  $node = R \times \lfloor (i \mod Nodes.count)/R \rfloor$;
  for $r := 0$ **to** $R\text{-}1$ **do**
    $destination \leftarrow node + r$;
    send(c_i, $n_{destination} \in Nodes$);

We implemented this idea in iHadoop by using the algorithm 2.

Figure 5 shows the locality of map tasks for different types of jobs. This result is surprising. We though that the Ultuk algorithm would achieve 100% locality.

3.2 Scheduling Policy

The LSAP-based scheduling policy have been implemented, but not tested on more than one type of job. For now, this scheduler shows no significant improvement compared to the default FIFO scheduler.

Conclusion

Hence, after our evaluation of the great interest of data locality, we have implemented and tested several state-of-the art ways to distribute the chunks over the nodes. We have also designed and implemented our own distribution algorithm, Ultuk. Thanks to these comparison between Kutlu, Utluk and the default random policy of hadoop on iHadoop, it appears that none of these policies seems to be significantly better than the other. If this result is quite understandable for the Kutlu

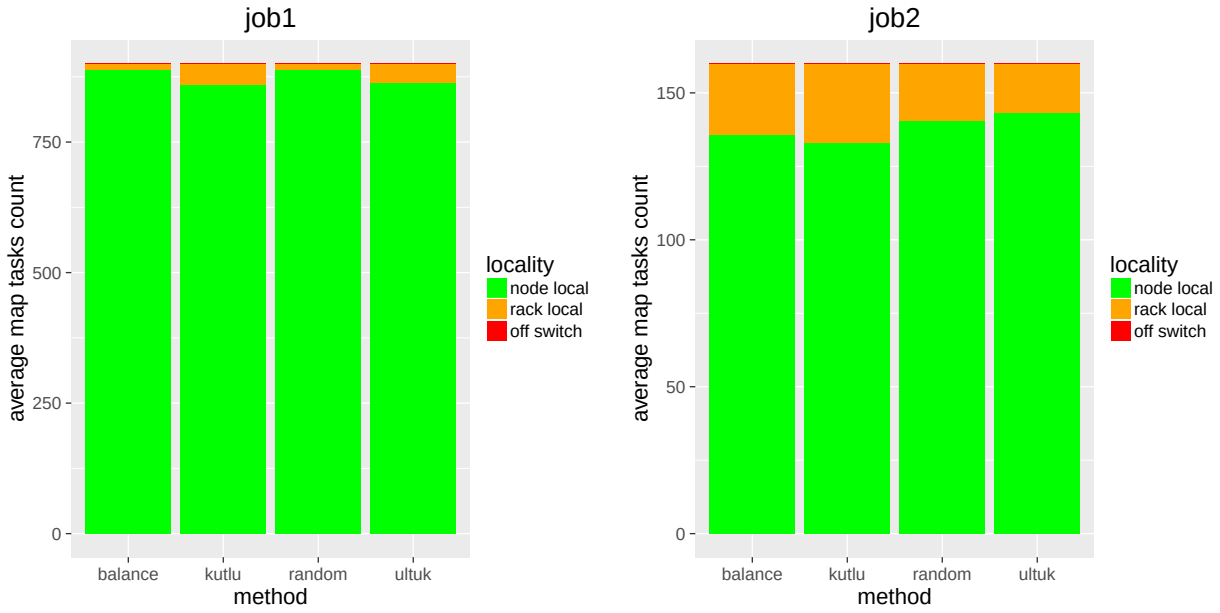

Figure 5

distribution, it is quite more surprising for the Utluk one, which is supposed to optimize data locality.

Similarly, we have also compared two different scheduling policies: the default FIFO scheduler, and the LSAP-based scheduler. However, on our tests, the data locality were already high, the improvement of the LSAP-based scheduler was not significant either.

In the future, it would probably be interesting to go further into detail to understand more precisely the flaws of the Utluk distribution, and to see to what extent the LSAP-based scheduler improve the default FIFO policy in cases where data locality is not easily accessible.

References

[1] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[2] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing, January 19 2010. US Patent 7,650,331.

[3] Zhenhua Guo, Geoffrey Fox, and Mo Zhou. Investigation of data locality in mapreduce. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 419–426. IEEE Computer Society, 2012.

[4] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[5] Mucahid Kutlu, Gagan Agrawal, and Orhan Kurt. Fault tolerant parallel data-intensive algorithms. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10. IEEE, 2012.

[6] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–9. IEEE, 2010.