Scheduling/Data Management Heuristics

Frédéric Desprez, Sylvain Gault, Frédéric Suter

MapReduce ANR-10-SEGI-001
Scalable data management for
Map/Reduce-based data-intensive applications
on clouds and hybrid infrastructures.

D3.1 Scheduling/Data Management Heuristics

Frédéric DESPREZ	Inria Grenoble Rhône-Alpes	frederic.desprez@inria.fr
Sylvain GAULT	Inria Grenoble Rhône-Alpes	sylvain.gault@inria.fr
Frédéric SUTER	CC IN2P3	frederic.suter@cc.in2p3.fr

Abstract

Data volume produced by scientific applications increase at a high speed. Some are expected to produce several petabyte per year. In order to process this amount of data, the computing power of several hundreds or thousands of machines have to be used at the same time. Regarding this, one of the biggest challenge is : how to program these machines in order to make them to collaborate for the same computation ? One answer brought by Google is the MapReduce paradigm [7]. MapReduce has the advantage of being quite simple to program for the user and handle on its own the repetitive or complex tasks like the data transfers between nodes, task scheduling or handling node failure. These automatic tasks have to be handled in an optimized way in order to make the framework fast and scalable. This report presents our first studies towards an efficient scheduling of MapReduce operations. More specifically, we focused on the scheduling of the data transfers together with the tasks. We present here an interesting work around this topic and our algorithm which improves their results.

1 Introduction

Since the dawn of computers, the volume of data to be processed by applications has continuously increased. This makes the processing requirement always bigger. Nowadays, we have reached the point where applications may have hundreds of terabytes to petabytes of data to process. For instance, the Large Synoptic Survey Telescope (LSST) is expected to produce 30 terabyte per night [9], and the Large Hadron Collider (LHC) is expected to produce around 15 petabytes per year [5]. Web indexing has also big requirement on computing power since a lare amount of new data is produced and updated daily.

Processing huge amount of data is a real challenge by itself since the data have to be spread over several nodes just as the computation. In order to avoid the pain of reinventing the wheel for every processing task, some library already exists to help programmers in the production of parallel and distributed applications, such as MPI [10] which make the usage of the network simpler. Some framework have also been proposed to provide higher level of abstraction of the distributed computation such as DIET [4], Dryad [6], BOINC [1] or MapReduce [7]. Providing a high level of abstraction to the user require the software to be smart. Specifically, for data-intensive computation, data management and scheduling are the biggest challenges.

Scheduling and resource management has been one of the main research subjects around Grids and Clouds. The MapReduce programming paradigm introduces new problems and it raises several research issues linked to data management as well as task scheduling. Below we mention some preliminary efforts concerning Cloud infrastructures. For Desktop Grids, no clear research results can be mentioned at this point for scheduling for MapReduce applications, since simply enabling MapReduce on Desktop Grids is still at an embryonic stage.

2 MapReduce

MapReduce is a framework for distributed computation introduced by Google in 2004. It aims at performing computation on a large number of nodes (several thousands) for processing large amount of data. Although the Google's implementation is not available to the public, some free implementations exists, especially Hadoop which has been reported to be used by several organizations such as Yahoo! [11] and Facebook [12].

MapReduce framework is based on the functions *Map* and *Reduce* which are commonly used in functional programming. In functional programming languages map is a high order function, it usually take a function and a list as argument. Then, map apply the function on every element of the list, and construct a new list. For instance, map could be use to multiply by 2 every element of a list. The reduce function (also known as fold) is another high order function which usually take a function and a list as argument. But unlike map, argument function must take two arguments, one is the result of a previous call of this function, the other is an element of the list. This function can be used to sum every element a list.

Within MapReduce, the spirit of these function is still there, but things are a bit different. It works in three steps as follow as seen on Figure 1.

- The input data is split into records composed of a key and a value. These key/value pairs are processed by the map function which produce 0, 1 or several key/value pairs. All the nodes which have been assigned a map task will process concurrently several parts of the input data. This is called the map phase.
- The data created by the map will then be physically grouped by key. This means that every node may send some data to every other node. This step is called the shuffle.
- Once the data have been grouped by key, for each key, the reduce function is applied on the values associated with that key and produce a new set of values for that key. This new

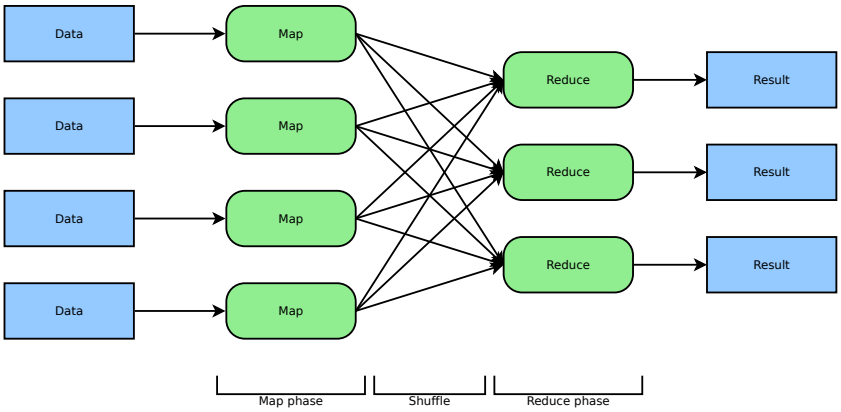


FIGURE 1: MapReduce Workflow.

set is usually reduced to one single value. Every node which have been assigned a reduce task will process the data it has received from the mappers. This is called, the reduce phase.

One classical example of MapReduce application is the *wordcount* where the user wants to count the number of occurrence of each word in a large set of documents. For this, the user could write a map and a reduce function like the pseudo code given in Figure 2.

```
Map (String key, String value) {
        /* key is the document name
         * value is the document contents */
        for each word in value {
                output the pair (word, 1);
        }
}

Reduce (String key, ListOfInteger list) {
        Integer result = 0;
        for each value in list {
                result += value;
        }

        output the list (result);
}
```

FIGURE 2: Word Count Example.

In this pseudo code, the map function will output one pair (word, 1) per word in the document, which means that several pairs will be produced if the same word appear several time. Then the reduce function just sum the values up, thus counting the number of occurrence for each word. The final result will be a list of pairs (word, count) which hold the wanted result.

As one may see, it is quite inefficient to produce one pair for each word when a word appear several times. This issue is addressed by the combiners which is out of the scope of this report.

More information can be found in the Google's publication.

But to make MapReduce work so easily from the user's point of view, the framework has to be optimized on several points. Indeed, it has to be fault tolerant, it should avoid data transfers between nodes for map computation, and it should schedule the transfers between the Map and the Reduce functions correctly in order to avoid that two nodes try to transfer the data to the same other node at the same time. This would increase the transfer time while better solutions exists.

3 Related Work

There already are some work around scheduling in MapReduce. Most of them focus on locality of data for the Map computation. For instance, a delay scheduler [13] has been introduced to improve the data locality by scheduling tasks on the fly. It works by delaying tasks which cannot be run on a node holding a copy of the data and will allow gradually the data transfer to be first rack-local, and finally completely remote.

Another approach for this data locality issue is BAR [8] which starts from an initial allocation of tasks to the nodes for a given data distribution and then tune it to reach a better locality access. It takes into account various parameters like node workload and network usage to allow more or less data transfers between nodes.

The divisible load theory [3] is a methodology involving continuous models for dividing workload over several processing nodes. Using continuous models usually allow simpler solving of problems than a discreet one, while still giving solutions close to optimal. This divisible load theory has been applied to MapReduce scheduling by Berlinska and Drozdowski [2]. This is the work we took as the basis.

4 Description of Previous Work by Berlinska and Drozdowski

4.1 Approach

4.1.1 Overview

In their paper [2], Berlinska and Drozdowski model a whole MapReduce job execution and aim at optimizing the whole schedule length. This is done by assigning the right amount of data for each node to process. Moreover, their model include a bandwidth limitation which is taken into account by carefully and strictly scheduling data transfers between mappers and reducers.

4.1.2 Model

B & D described two views of the same model. The first one describes the MapReduce jobs in terms of meaningful and actually measurable variables like communication startup time or computing rate. This is called "microscopic view" in their paper. As this view is quite hard to manipulate, a "macroscopic view" is derived from the previous one where variables or constant may actually be composed of several constants.

Table 1 is a list of useful variables used in the model.

In their model there are m nodes performing Map computations over a total of V bytes, and r nodes performing Reduce operations. For the sake of simplicity they are considered as distinct sets of machines, thought they could be the same with some transfers shorter than others because of localhost data transfers.

Map nodes are considered to start in a sequential way. That means that node 1 will spend S seconds initializing before node 2 can start its own initialization. This initialization step may be

V	Total amount of data to process
m	Number of Map nodes
r	Number of Reduce nodes
S	Sequential startup time in seconds
C	Transfer rate between Map nodes and Reduce nodes in seconds per byte
l	Maximum number of simultaneous transfers
A_i	Computing rate for i-th node in second per byte
α_i	Amount of data processed by i-th node
γ	Data ratio between mapper output and input

TABLE 1: Main Variables of the Berlinska and Drozdowski's Model.

related to the time taken to upload the code to the nodes. For simplicity purpose, S is considered to be the same among all the nodes. Although not mentioned, most of the results of this report would still hold for initialization time distinct for each node as long as the startup order of nodes is the same.

In this model, the bandwidth is limited for each node and is named C (in seconds per byte). Since the data are supposed to be already present on the computing nodes (as usual in MapReduce) there is no need to transfer the data to the computing nodes. Only a rebalancing may be necessary but the time to transfer data for this, is not part of the model.

In addition to the bandwidth limitation, there is a bisection width limitation. That means that only l transfers at the same time can occur between the mappers and the reducers. This can happen in real situation in the case of a switch connecting more nodes that it can handle data transfers at full speed at the same time, or in the case of a tree-like network for which only the worst case is taken into account.

Every node i has a computing rate A_i expressed in seconds per byte. This is a macroscopic variable that hides a computation overhead and a data transfer (to read from the disk) in addition to the actual pure-computing rate.

The α_i variable is the amount of data (in bytes) to be processed by the i-th node. These variables are actually computed by the algorithms in the paper.

Eventually, the γ variable is the ratio between the amount of input data to the mappers, and the amount of output data of the mappers. Its value depends on the kind of computation performed by the Map operation. It is computed as follows.

$$\gamma = \frac{output\_size}{input\_size}$$

Simplifications and constraints. As a simplification, this paper considers that every mapper will start to transfer its data to mappers only **after** the end of the Map computation. Moreover, each mapper will send the same amount of data to every reducer. This is a strong assumption but it should statistically be met. This last simplification has the implication that every reducer will get the same total amount of data (one chunk from every mapper). Thus, the reducer computation time is not worth mentioning.

Between the Map and Reduce phases there are $m * n$ transfers to perform. The network model follows the one-port model on both mapper and reducer side. This constraint, in addition to the bandwidth limitation l, implies that the transfers should be carefully scheduled.

4.1.3 Resolution

The goal is to reduce the completion time. As the previous simplifications say, the computation time of the reducers is not taken into account. Thus, only the mapper computation time and the transfer time are to be computed. In order to compute the transfer time, the transfer scheduling has to be known and deterministic. That's why Berlinska and Drozdowski chosed a time-interval based algorithm.

In this algorithm, a transfer from one mapper to one reducer cannot be interrupted. The length of the time intervals is determined by the largest transfer to do during this interval. And finally, the l mappers allowed to transfer at a given time are chosed by a simple round-robin.

This simple algorithm allows to express the completion time as a formula depending on the previously given parameters. Especially, the time to perform all the transfers depends on the largest transfer from a mapper to a reducer that occurs in every time-interval. That is why the α_i have to be computed by a linear program.

4.1.4 Issues

Because the number of constraints and variables of the linear program can be quite large. The number of variables is $O(mr/l)$ and the number of constraint is $O(mr)$. That makes the scheduler take up to several minutes or hours for a few hundred nodes with LP-solve on an Intel quadcore at 2.4GHz. This time is not acceptable for a real-life MapReduce framework.

Furthermore, as the parameters of the model may vary, some others parameters are hard to predict. In particular, the A_i (computing throughput) may depend on the data itself. And the γ parameter (data ratio of the mappers) is just impossible to predict in most cases. A way to solve this is to continuously update the schedule with respect to the measurements reported by the nodes. But this can only be useful if the schedule can be recomputed fast enough.

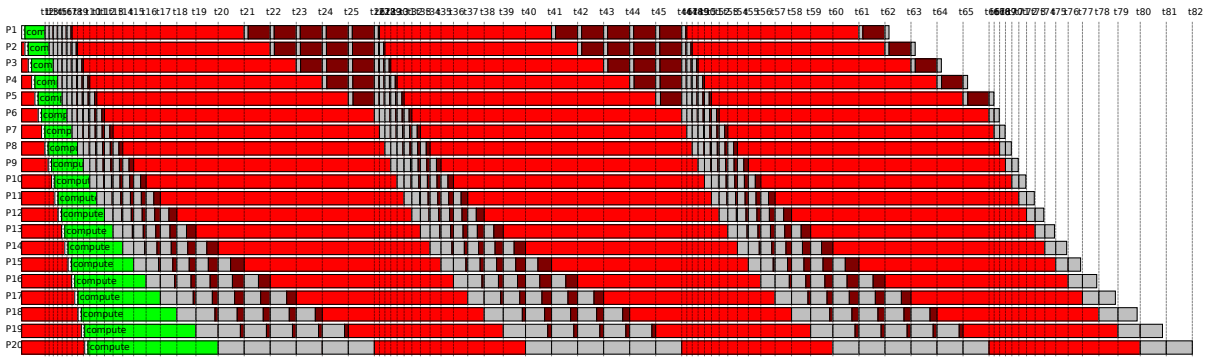


FIGURE 3: Schedule example as computed by the algorithm of Berlinska and Drozdowski.

And the chosen transfer scheduling algorithm, thought easy to write as a formula, uses inefficiently the network bandwidth. Figure 3 is a Gantt chart of occupation of mapper nodes as computed by the algorithm from Berlinska and Drozdowski. In light red is the time where the nodes are idle, and dark red is the time spent waiting for some transfers to finish. The white part (just before the green one) is the sequential startup time. In green is the time spent on data processing, and gray is the time taken to transfer processed data. And what can be seen in this figure, is that for each time interval, the nodes with a small amount of data to transfer will wait for the node with the biggest one to finish. Hence, the network usage will drop to 1 transfer at time towards the end of every time-interval.

5 Description of Our Algorithms

As previously said, the algorithm has to be made more dynamic because of the imprecision of the forecast that would be made about the computing time. But this dynamicity imply that the schedule has to be computed fast enough in order to apply its modifications to the physical system before the computation is done.

The main reason why the problem has to be expressed as a linear program which takes so much time to solve, is that it the transfer scheduler imply a dependence between every α_i for every time-interval.

5.1 Using a Linear System

One thing that can be remarked with the linear program, is that, sometime, it computes the α_i such that one transfer from the i-th machine will end just at the same time when α_{i+1} ends its computation. Formally, this means that α_i and α_{i+1} are computed such that :

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

This especially happens when every A_i are equals and $Srm < \gamma VC$, which occurs quite often since S is usually very small (a second or less) and V is usually quite big (in the order of TB). When every A_i are equals, the above formula imply that $\alpha_i < \alpha_{i+1}$.

In a such case, the α_i can be computed by a linear system as follows :

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

for $i < m$

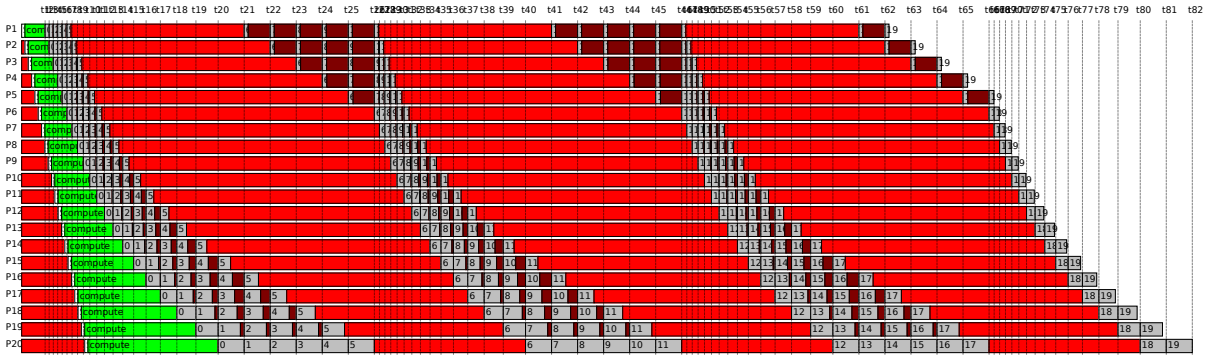
$$\sum_{i=1}^{m} \alpha_i = V$$

This system can be solved in $O(m)$ time and $O(m)$ space.

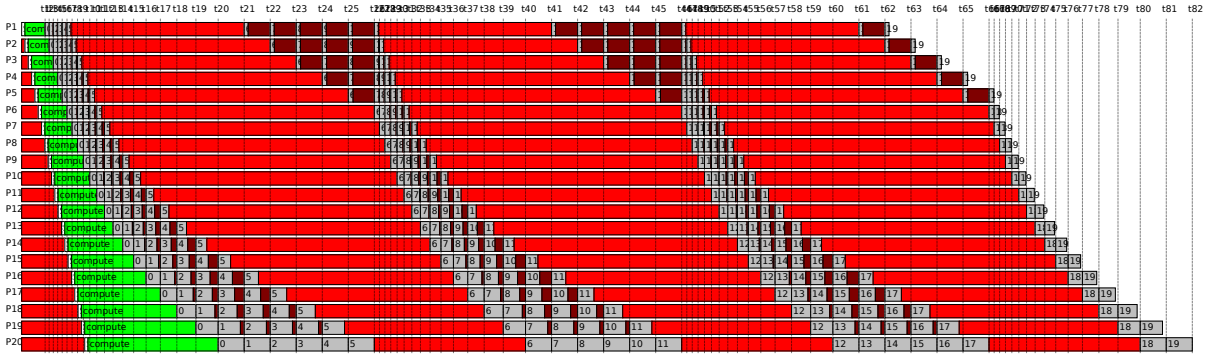
Experiments show that in such a case the linear system compute the same α_i as the linear program except for the float rounding errors. Figure 4(a) shows the schedule computed by the linear program (correcting some minor bugs from the original algorithm from Berlinska and Drozdowski), the numbers on the gray transfers are the target reducers for the transfers. Figure 4(b) shows the schedule computed by the linear system above. As it can be seen, there is no noticeable difference between the results as long as $\alpha_i < \alpha_{i+1}$ holds.

When the condition $Srm < \gamma VC$ does not hold, the results of the linear system are very different from the ones computed by the linear program. Figure 5(a) shows the behavior of the linear program on another set of parameters, and Figure 5(b) shows the results of the linear system. It can be noticed that the completion time is greater with the linear system than with the linear program. This is mainly due to the gap between the end of computation and the start of the first transfer. But another transfer scheduler exhibits better performance (see Section 5.2).

It should also be noticed that less nodes are used by the linear system than the linear program. This is because the computation of the amount of data to process α_i may output negative results. As a result some nodes are eliminated from the computation. This node-elimination implies that the linear system has to be solved $\log m$ times with different number of nodes. The overall complexity is then $O(m \log m)$.

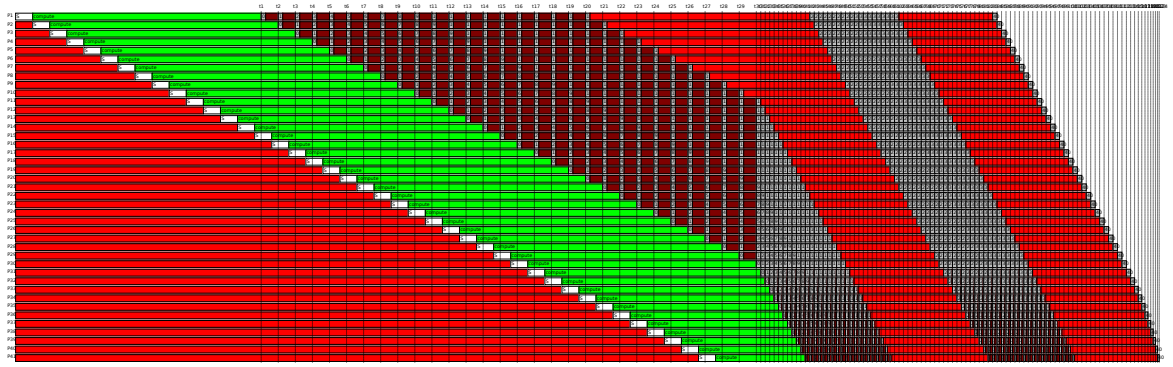


(a) linear program

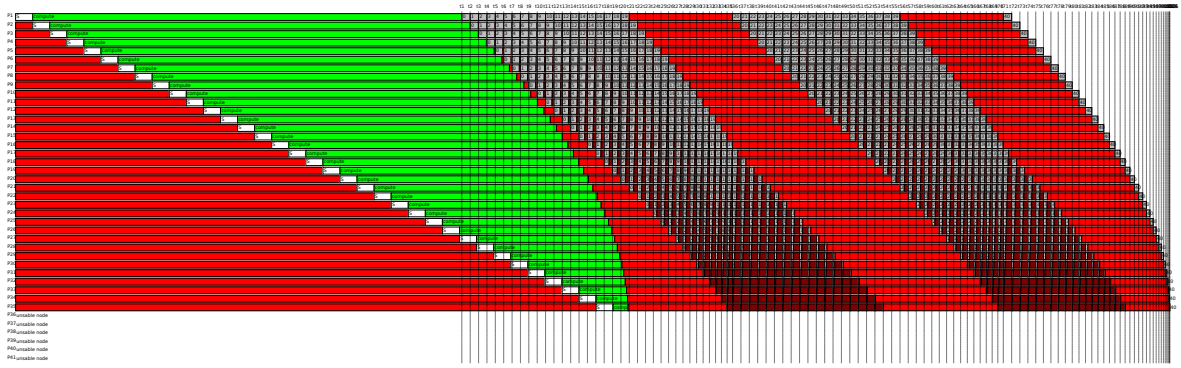


(b) linear system

FIGURE 4: Schedule examples with $\alpha_i < \alpha_{i+1}$.

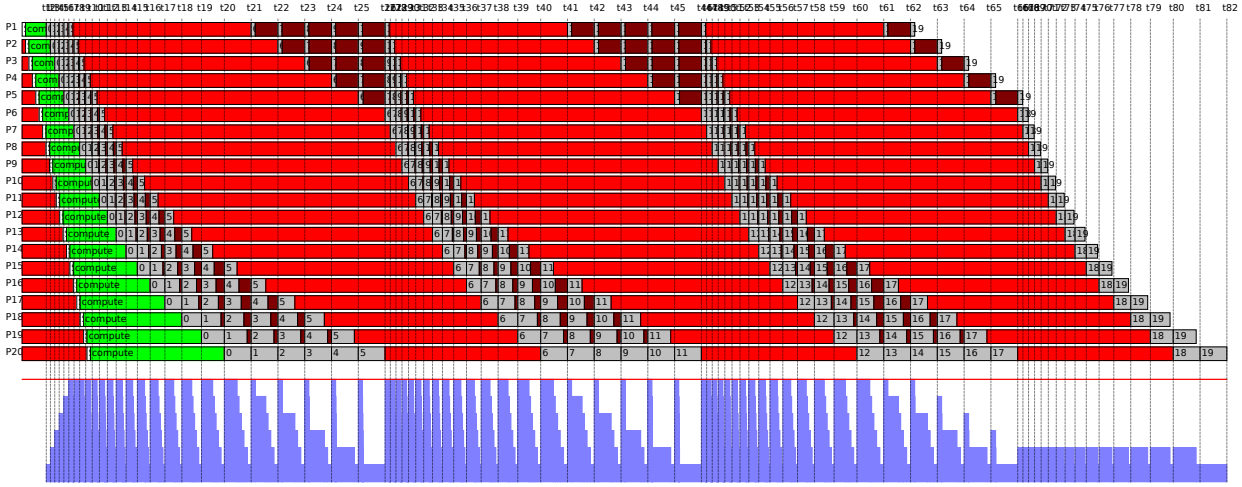


(a) linear program

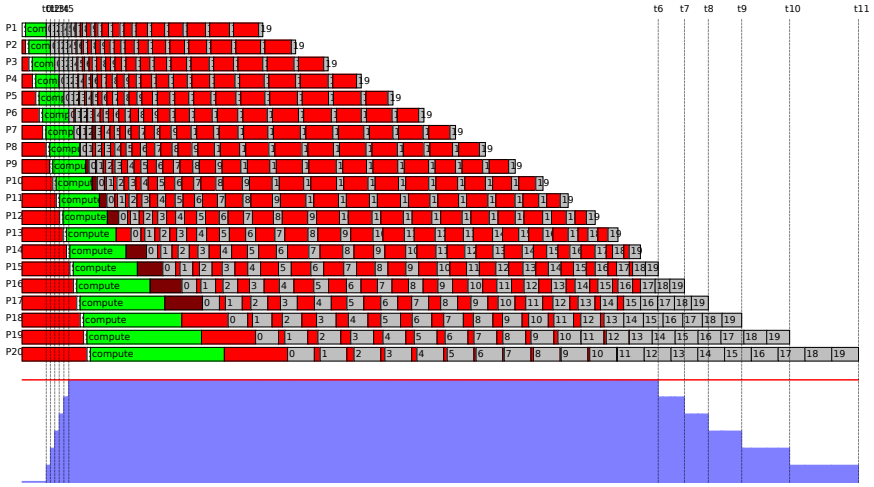


(b) linear system

FIGURE 5: Schedule example with $\alpha_i >= \alpha_{i+1}$.



(a) original scheduler



(b) new scheduler

FIGURE 6: Schedule examples with $\alpha_i < \alpha_{i+1}$ plus bandwidth usage.

5.2 Transfer Scheduling

Figure 6(a) shows the bandwidth usage over the time just under the classic Gantt schedule for the time-interval based scheduler. This chart shows a lot of "holes" which could be filled with a better transfer scheduling algorithm. As a simplification purpose, every mapper does its transfers in the same order, starting from the reducer 1 to the reducer r. And a transfer from mapper i to reducer j cannot start before the transfer from mapper $i-1$ to reducer j has finished. The last constraint is an easy way to enforce the constraint that two mappers should not transfer to the same reducer at the same time while reducing the complexity of scheduling.

In order to avoid a bad bandwidth usage and still enforce the above constraints, a first heuristic could be to try to perform the transfer of the "latest" nodes as soon as possible, which means, try to make every node transfer to the same reducer at the same time. But given the constraints, an implementation of this heuristic could easily lead to a situation where the first mapper has been delayed and when it starts its transfers, every other node has to wait for it. That's why a better heuristic is to try to keep the quantity $i+j$ among all the nodes constant, with i the mapper number and j the target reducer for the transfer last started. Which means that when mapper 1 perform its transfers to reducer 5, mapper 2 would transfer to reducer 4.

Sketch of the algorithm : When a transfer from mapper i to reducer j ends, for every idle i' mapper and its next target reducer j' compute $p_i = i' + j'$. Then select the nodes with the lowest previously computed p_i. These nodes are considered the most *late* and their transfers should start as soon as possible. If several nodes are equally late, then the one with the biggest α_i is chosen because, among all these selected nodes, the ones with the biggest amount of data to transfer will be most probably the ones to delay the end of all transfers.

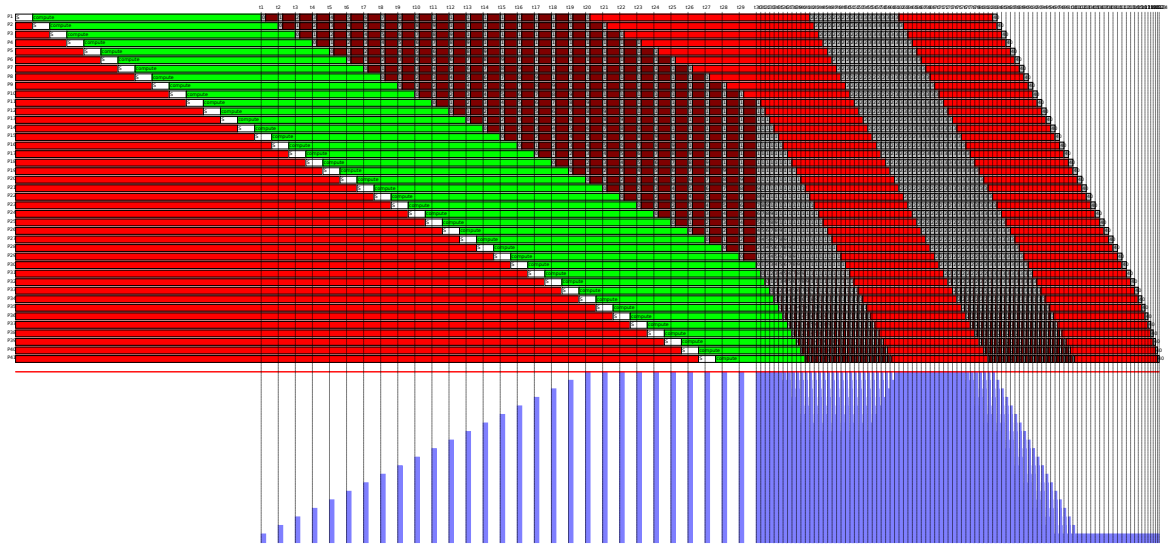
This second heuristic should limit the case when dependencies prevent a full bandwidth usage. And actually, the case does not happen often within the simulations. The result of this second heuristic can be seen in Figure 6(b) for the case with $\alpha_i < \alpha_{i+1}$. In this example, a 30% improvement has been achieved and the bandwidth usage show no hole and is at its maximum capacity during the main part of its transfers.

When $\alpha_i >= \alpha_{i+1}$ different things happens. First, when the amount of data to be processed by each node α_i is computed by the linear program, it may happen that the maximum network bandwidth cannot be reached. Figure 7(a) show the original schedule along with its implied bandwidth usage and Figure 7(b) show the behavior of the new transfer scheduler when the α_i are yet computed by the linear program. It can be seen on the later that the maximum bandwidth shown by the red line is never reached. Actually, the α_i have been computed for the first time-interval based transfer scheduler. That's why we obtain such a result. Still, in this example, it achieves a 16% improvment over the completion time.

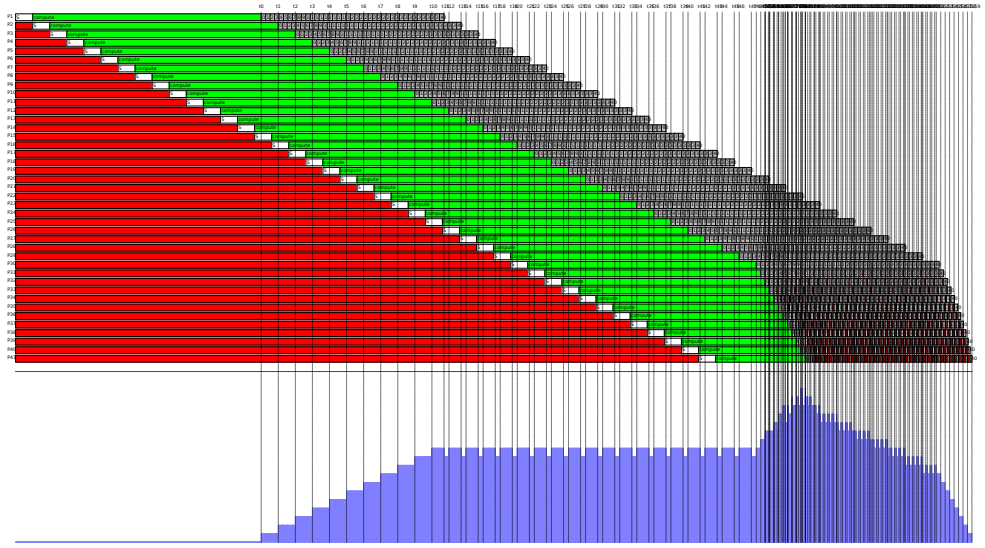
When the α_i are computed by the linear system and the transfers are scheduled by the original time-interval based one, the completion time is not much different from the original transfer scheduler with the α_i computed by the linear program. The result can be seen in Figure 8(a), and shows a performance decrease of less than 1%. Just like the reverse hybrid case (shown in Figure 7(b)) the computation of the α_i has not been designed for this transfer scheduler. And finally, Figure 8(b) show the result when the amount of data per node α_i is computed by the linear system and the transfers are scheduled by the new algorithm. It achieve an improvment of 17% as compared to the original algorithm.

5.3 Transfer Preemption

With the previous approach, it appears sometimes that a low priority transfer starts just before a previously idle node ask for a high priority transfer. This implies that a low priority transfer prevents a high priority transfer to occur. This is a priority inversion.

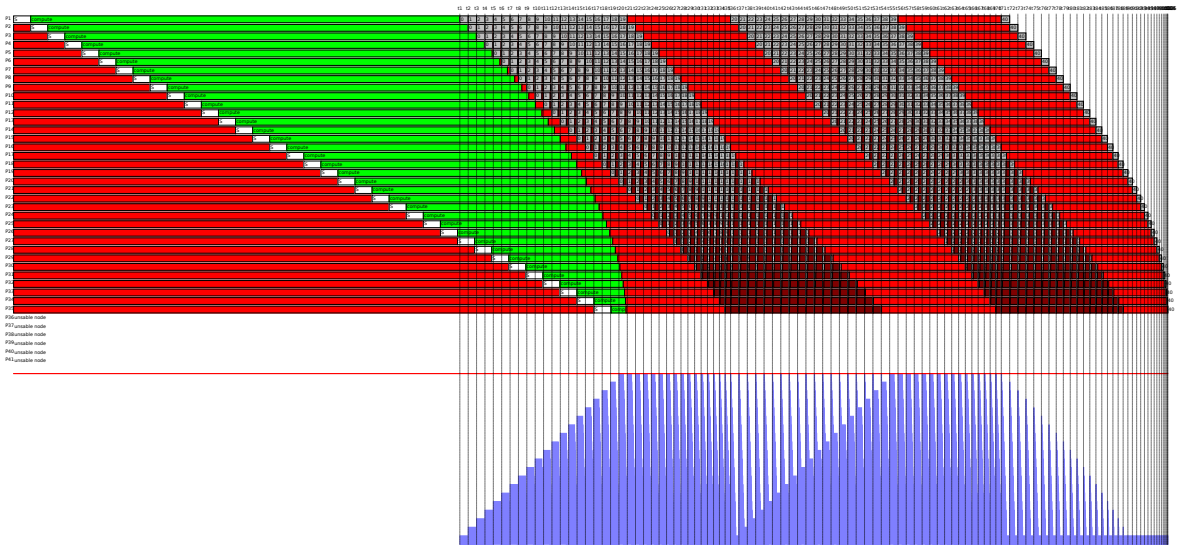


(a) original scheduler

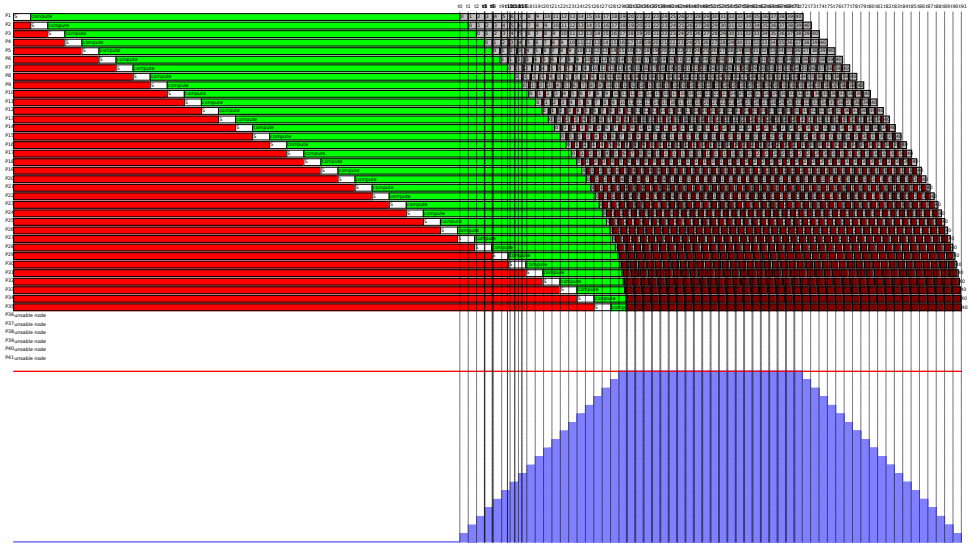


(b) new transfer scheduler

FIGURE 7: Schedule examples for the original compute scheduler with $\alpha_i >= \alpha_{i+1}$ plus bandwidth usage.



(a) original transfer scheduler



(b) new scheduler

FIGURE 8: Schedule examples for the new compute scheduler with $\alpha_i >= \alpha_{i+1}$ plus bandwidth usage.

One way to fix this priority inversion is to allow transfer preemption. This means that when a node wants to start a transfer, the scheduler will start it if there is currently less than l transfers occuring. If the transfer count limit is already reached, then the scheduler will search for a transfer with a lower priority that will be suspended. This transfer suspension can actually happen in two cases. The first case is when a mapper finish its computation and want to start its first transfer, a transfer with a lower priority can be preempted. The second case is when a mapper i with a large α_i cannot transfer its data to reducer j because the mapper $i-1$ is already transferring to the same reducer j. Then the scheduler starts a transfer with a lower priority from a mapper k, and when the transfer from $i-1$ to j is done, then the mapper i can start its transfer to reducer j **and** mapper $i-1$ can start its transfer to reducer $j+1$. since transfers from i and $i-1$ have a higher priority than those from k, the transfer from k can be suspended.

In the current model, this algorithm with transfer preemption cannot perform worse than the previous one. Indeed, the above-mentioned case where preemption happen are times where a node with big amount of data would wait for a node with a small one. Keeping in mind the constraints on the order of the transfers, the last transfer will always be the one from mapper m to reducer r. This implies that the only thing that can happen is the big transfers start earlier thus making the completion time better.

On Figures 9(a) and 9(b), it can be seen that the algorithm with preemption performs slightly better since the bandwidth usage is maintained at is maximum capacity a bit longer. This represent a bit less than 2% improvement. This is, indeed, negligible but cannot be worse that not preempting transfers in the current model. Actually the model does not take latency into account, thus, the gain of preempting unimportant transfers may be mitigated by the cost of the preemption itself.

Moreover, on Figure 9(b), it can be seen that the first transfers do not start just after the computation as finished whereas the α_i have been computed such that the computation would end just when its first transfer can actually occur. This is due to the fact that the network has already reached its maximal capacity and is used by transfer with higher or equal priority.

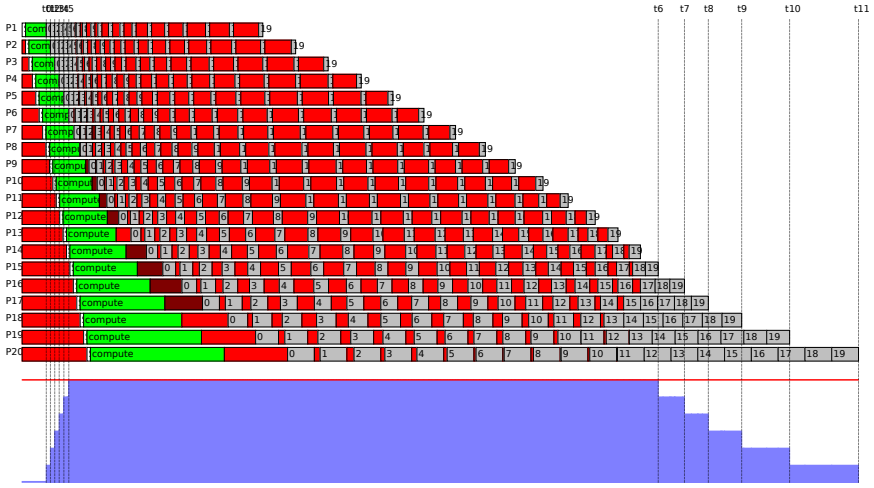
In order to deal with this problem and make the big transfers start as soon as the computation finishes, we tried to relax the constraints on the order of transfers between nodes. Which means that when a big transfer is ready, it is allowed to interrupt any other transfer. The resulting schedule is shown in Figure 10(b). This actually show performance quite similar to the previous algorithms whose result has been repeated in Figure 10(a) for comparison.

6 Conclusion and Future Work

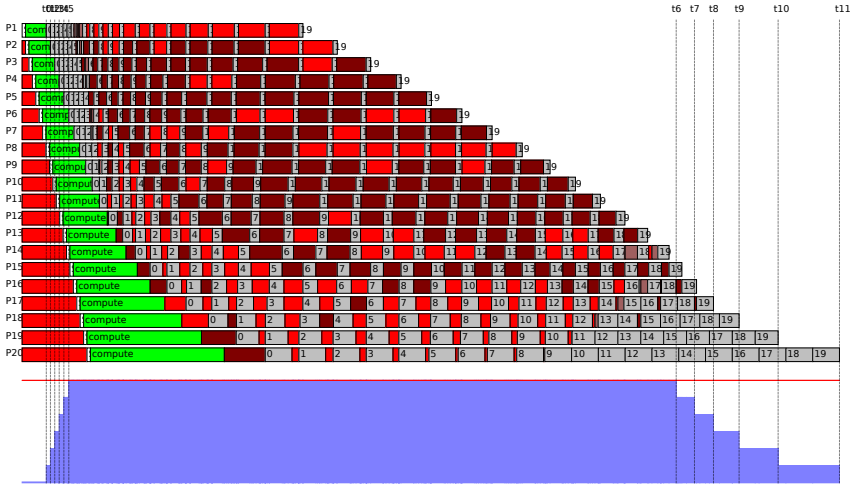
To conclude, these algorithms for either computing the α_i parameters and for scheduling the transfers outperform the ones from Berlinska and Drozdowski on several points. The computation of the α_i is faster since it relies on a linear system solved in time $O(m)$ instead of a linear program. Although in some cases the results are quite different, this does not affect much the overall performance.

The transfer scheduler also performs better than the one from Berlinska and Drozdowski since it does as much as possible to maximize bandwidth usage. Moreover, it does not involve synchronization barriers which would be quite costly in term of latency. On the other side, it is hard to predict anything and proving anything about it looks non-trivial.

Thus, as a future work, we plan to evaluate in a more in-depth manner the gains and limits of the transfer algorithms. We also plan to implement these algorithms in a real MapReduce software and confront the model used to the real world. The main issue we expect is the latency of the network which is not part of the model. The latency may negligible compared to the time

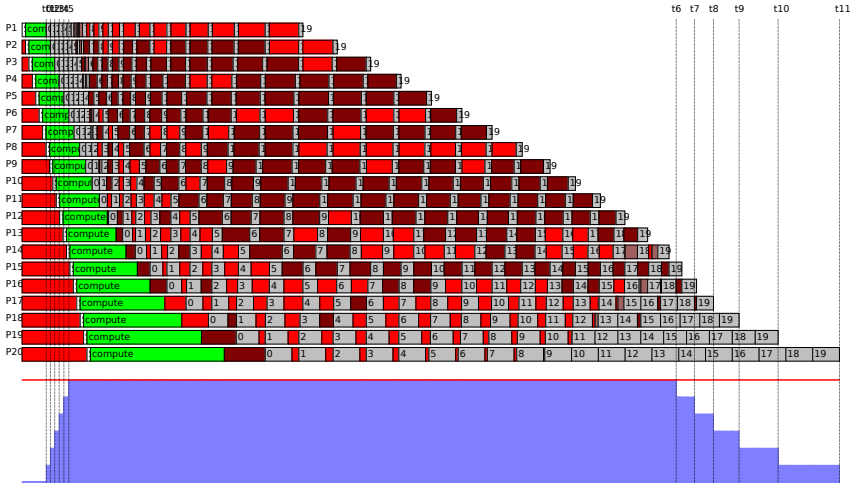


(a) without transfer preemption

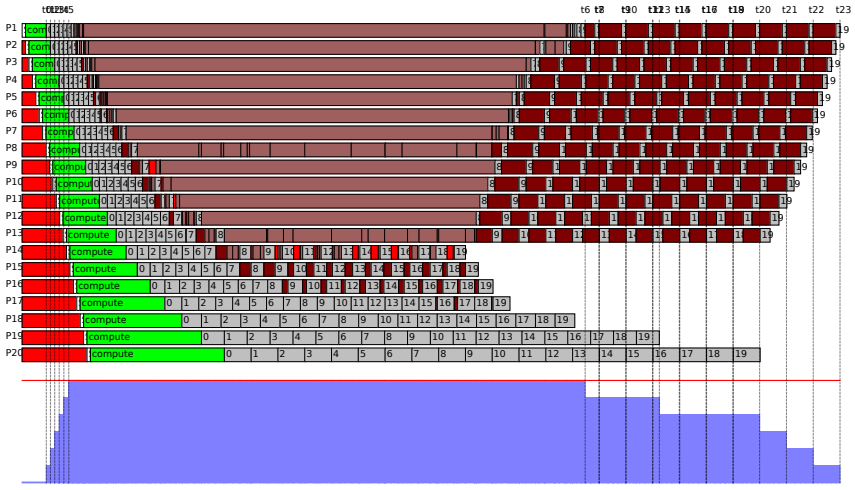


(b) with transfer preemption

FIGURE 9: Schedule examples for the transfer scheduler with and without preemption.



(a) previous priority policy



(b) transfer size priority policy

FIGURE 10: Schedule examples for the transfer scheduler with different priority policies.

needed to do the actual transfers, but the synchronization between the mapper nodes and the master node may induce an unknown latency. Then we plan on implementing these algorithms with an iterative schedule recomputation. Indeed, as some parameters are unknown and/or platform dependent getting them as we get feedback from the computing nodes and adjusting the forecasted schedule and its platform counterpart. Starting from this, it could be quite easy to extend the model and software to handle iterative MapReduce computation where the next iteration starts as soon as possible.

Références

[1] D.P. Anderson. BOINC : A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

[2] J. Berlinska and M. Drozdowski. Scheduling divisible MapReduce computations. *Journal of Parallel and Distributed Computing*, 71(3) :450–459, March 2010.

[3] Veeravalli Bharadwaj. *Scheduling divisible loads in parallel and distributed systems*. Wiley-IEEE Computer Society Pr, Los Alamitos-Washington-Brussels-Tokyo, 1996.

[4] Eddy Caron and Frédéric Desprez. Diet : A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3) :335, 2006.

[5] CERN. Worldwide LHC Computing Grid, 2011.

[6] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad : distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72. ACM, 2007.

[7] Dean Jeffrey and Ghemawat Sanjay. MapReduce : Simplified data processing on large clusters. In *In OSDI'04 : Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, volume 51. USENIX Association, 2004.

[8] Jiahui Jin, Junzhou Luo, Aibo Song, Fang Dong, and Runqun Xiong. BAR : An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 295–304. IEEE, 2011.

[9] LSST. Large Synoptic Survey Telescope.

[10] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1 : The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revi edition, 1998.

[11] Yahoo ! Hadoop at Yahoo !, 2012.

[12] Paul Yang. Facebook uses Hadoop, 2011.

[13] Matei Zaharia, Dhruba Borthakur, J.S. Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55, Apr*, pages 2009–55, 2009.