Dynamic Scheduling of MapReduce Shuffle under Bandwidth Constraints

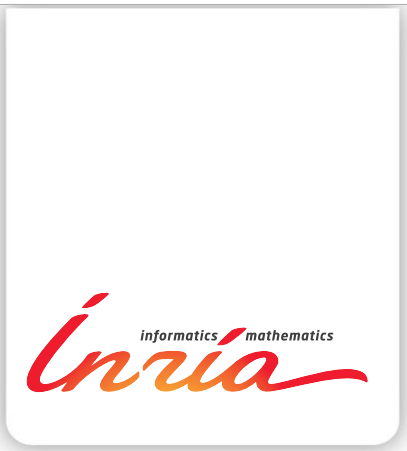
Sylvain Gault, Frédéric Desprez

Dynamic Scheduling of MapReduce Shuffle Under Bandwidth Constraints

Sylvain Gault, Frédéric Desprez

RESEARCH REPORT
N° 8574
August 1, 2014
Project-Team Avalon

ISSN 0249-6399 ISRN INRIA/RR--8574--FR+ENG



Dynamic Scheduling of MapReduce Shuffle Under Bandwidth Constraints

Sylvain Gault, Frédéric Desprez

Project-Team Avalon

Research Report n° 8574 — August 1, 2014 — 35 pages

Abstract: Whether it is for e-science or business, the amount of data produced every year is growing at a high rate. Managing and processing those data raises new challenges. MapReduce is one answer to the need for scalable tools able to handle the amount of data. It imposes a general structure of computation and let the implementation perform its optimizations. During the computation, there is a phase called *Shuffle* where every node sends a possibly large amount of data to every other node. This report proposes and evaluates six algorithms to improve data transfers during the *Shuffle* phase under bandwidth constraints.

Key-words: Big Data, MapReduce, shuffle, scheduling, network, contention, bandwidth, regulation

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Ordonnancement Dynamique du Shuffle MapReduce sous Contrainte de Bande Passante

Résumé : Que ce soit pour l'e-science ou pour les affaires, la quantité de données produites chaque année augmente à une vitesse vertigineuse. Gérer et traiter ces données soulève de nouveaux défis. MapReduce est l'une des réponses aux besoins d'outils qui passent à l'échelle et capables de gérer ces volumes de données. Il impose une structure générale de calcul et laisse l'implémentation effectuer ses optimisations. Durant l'une des phases du calcul appellée *Shuffle*, tous les nœuds envoient des données potentiellement grosses à tous les autres nœuds. Ce rapport propose et évalue six algorithmes pour améliorer le transfert des données durant cette phase de *Shuffle* sous des contraintes de bande passante.

Mots-clés : Big Data, MapReduce, shuffle, ordonnancement, réseau, contention, bande passante, régulation

Contents

1 Introduction

In the past decades, the amount of data produced by scientific applications has never stopped growing. Scientific instruments still make the rate of data production to continuously grow as well. DNA sequencers in biology, the Large Hadron Collider (LHC), the Large Synoptic Survey Telescope (LSST) in physics, high-resolution scanners in medical imaging, and digitalisation of archives in Humanities are few examples of scientific tools producing data at a high rate. Sensor Networks become quite common, while web indexing and social network analysis are hot research topics.

With new orders of magnitude in data production come new challenges related to storage and computation. Depending on the kind of processing needed for these Big Data, several approaches can be considered. The users of the LHC are gathered into large international collaborations to analyze the data and perform long lasting simulation based on these data. Another approach is to rely on an easily scalable programming model. Google proposed to use MapReduce [1] for this purpose in order to handle the web indexing problems in its own data-centers. This paradigm, inspired by functional programming, distributes computations on many nodes that can access the whole data through a shared file system. MapReduce system users usually only write a MapReduce application by providing a *Map* and a *Reduce* function.

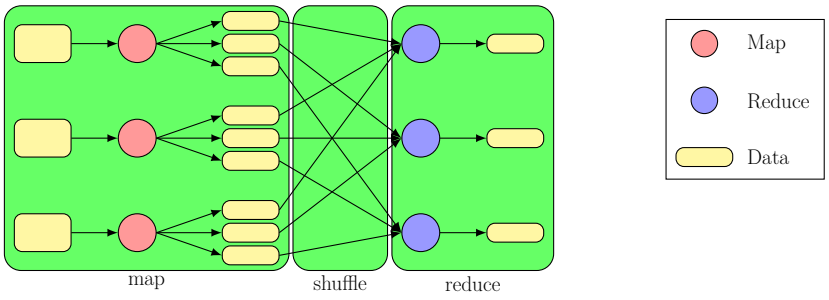


Figure 1: Typical structure of a MapReduce application.

The global process of a MapReduce application mainly consists in 3 steps, *Map*, *Shuffle* and *Reduce* as shown on Figure 1. During the *Map* phase, every *Mapper* process reads a chunk of data and applies the *Map* function on every record of that chunk and produces a number of key-value pairs. All those key-value pairs make up what is called the *intermediate data*. Within every *Mapper* process, the intermediate data is split into *partitions*. A partition represents the set of data to be sent to a given *Reducer* process, and the partition number needs to be a deterministic function of the key. Then during the *Shuffle* phase, all the pairs with an equal key are gathered in a single *Reducer* to build pairs made of a key and a list of values. The *Reducer* process can thus run the *Reduce* function on every pair and produce one result per intermediate key. Every *Mapper* has potentially some data to send to every *Reducer*.

Among the 3 phases of MapReduce, the *Reduce* phase usually takes a negligible amount of time compared to the two others. Most works [2, 3, 4, 5, 6] that try to optimize MapReduce have mainly focused their efforts on the *Map* phase, improving the data-computation locality, computation balance between node, or fault tolerance. But, despite being an important phase, the *Shuffle* has been largely forgotten.

In order to optimize the performance / cost ratio, most MapReduce platforms run on moderately high-end commodity hardware. Common hard disk drives in 2013 can achieve a throughput of more than 170 MB/s on a 7200 rpm HDD. RAID configurations and SSD drives can lead to much higher throughput. The CPU can also achieve a high data throughput. For instance, a simple `md5sum` reading from a shell pipe can achieve a throughput of 372 Mo/s. As most *Map*

tasks involve a much simpler computation than a md5, it can be assumed that the CPU is not the bottleneck of a MapReduce application.

However, the network throughput is usually bound to 1 Gbps (or 125 MB/s) on commodity hardware. From this, the time taken by the *Map* phase is expected to be more or less or equivalent to that of the *Shuffle* phase if the *Map* operation is optimized enough and generates an amount of data of the same order of magnitude as its input data.

Network bandwidth can become a scarce resource on some platforms. Indeed, most non high-end network equipments cannot guaranty that the overall sustained throughput would be equal to the sum of the throughput of all its connected ports. Some previous work [7] showed that when contention occurs in a LAN, the overall throughput drops because of the delay needed by TCP to detect and retransmit the lost packets. Moreover, the well-known Cisco System corporation also sells, for instance, 10 Gb Ethernet switches that do not provide a backplane bandwidth equal to the sum of the bandwidth of all the ports. Thus it is asserted that, in general, the overall bandwidth is limited.

In a MapReduce application, it is quite common that the *Mappers* do not process the same amount of data and that the *Map* processes do not terminate at the same time. Thus, sharing the bandwidth equally among the mappers (as the network stack would do by default) may lead to a suboptimal bandwidth usage due to the mappers that finished later and those with more intermediate data. How should the *Shuffle* phase be managed to make it as quick as possible?

This report proposes and compares several algorithms to optimize the *Shuffle* phase. Section 2 review some works that try to optimize the *Shuffle* phase in MapReduce. The models proposed in Section 3 are then analyzed in Section 4.1 to produce a lower bound on the duration of the *Shuffle*. Then, 6 algorithms are presented in Section 4.2. In Section 4.3 an algorithm to regulate the bandwidth is presented, this part is actually used by 3 of the algorithms. Section 5 evaluate the accuracy of the platform simulation, the regulation algorithms presented before and all the 6 algorithms under various conditions. And finally Section 6 concludes this work and proposes some ideas for further research.

2 Related Work

2.1 Shuffle Optimisation

A few works have investigated the problem of the transfer cost during the *shuffle* phase of a MapReduce application. Most of them focus on optimizing the task and data placement during (or just after) the *Map* phase.

Balanced Reduce Partitions The LEEN [8] (locality-aware and fairness-aware key partitioning) algorithm tries to balance the duration of the reduce tasks while minimizing the bandwidth usage during the *Shuffle* phase. This algorithm relies on statistics about the frequency of occurrences of the intermediate keys to get to create balanced data partitions. This approach is complementary to that presented here.

Shuffling Less Data Another complementary approach is the HPMR [9] algorithm. It proposes a *pre-shuffling* phase that leads to a reduced amount of transfered data as well as the overall number of transfers. To achieve this, it tries to predict in which partition the data go into after the *Map* phase and tries to place this *Map* task on the node that will run the *Reduce* task for this partition.

Conversely, the Ussop [10] runtime, targeting heterogeneous computing grids, adapts the amount of data to be processed by a node with respect to its processing power. Moreover

it tends to reduce the intermediate amount of intermediate data to transfer by running the *Reduce* task on the node that hold most of the data to be reduced. This method can also be used together with our algorithms.

Using Divisible Load Theory A MapReduce application can be seen as a set of divisible tasks since the data to be processed can be distributed indifferently on the *Map* tasks. It is then possible to apply the results from the divisible load theory [11]. This is the approach followed by Berlińska and Drozdowski [12]. The authors assume they have access to a runtime environment in which the bandwidth of the network switch is less than the maximum bandwidth that could be used during the *Shuffle* phase, thus inducing contention. To avoid this contention, they propose to model the execution of a MapReduce application as a linear program that generates a static partitioning and a static schedule of the communications based on a set of communication steps. While interesting, we showed in a previous work [13] that this approach is hardly scalable and that the chosen communication pattern is clearly suboptimal.

2.2 Platform and Performance Models

Infrastructure Models Despite the fact that the cloud tries to hide the details of the hardware infrastructure, the performance may depend on it. It thus need to be modeled. A cloud platform is usually modeled as a 2 or 3 levels tree network [14, 15]. At the lowest level, the nodes are grouped into racks with a switch for each one. Then the racks are connected together with another switch to make a cluster. The clusters are connected together forming another level. Even though the network tree may look like a fat-tree [16, 17], it is common that the switch to switch links are undersized regarding the actual number of nodes they connect. This practice is called oversubscription. Those models may, however, need to be simplified to solve a given problem.

Network Models On a more specific point, some researchers [7] modeled the effect of contention in an all-to-all operation. Those models are represented by affine functions. In this document, contention does not have to be modeled since it is actually avoided.

Coarse MapReduce Models Based on those cloud models, it is possible to build other models to try to predict the duration of applications in the cloud. Some work [18] take the oversubscription into account that may create contention. Others [19] model applications processing jobs and ignore the internal network as they assume the usage of high-throughput low-latency network. Those models are very coarse grain and do not take into account the specificities of MapReduce.

Detailed MapReduce Models On the opposite side, some work model the execution of a MapReduce job in too much details to be easily usable. A research report [20] provides extensive details in modeling the execution of a job in Hadoop MapReduce. It includes 5 steps on the *Mapper* side: *read*, *map*, *collect*, *spill*, and *merge*. The amount of intermediate data produce is proportional to the amount of input data. On the *Reducer* side there are 4 steps *shuffle*, *merge*, *Reduce*, and *write*. Most of those 9 phases are proportional to the amount of data. Moreover, some parts of the models are quite Hadoop-specific and could be simplified. It could also be noted that the network model used does not take contention or latency into account and thus it needs no assumption on the network topology.

Simulation-Based Modeling MRPerf [21] and Starfish [22] with its *What-If* engine [23] both relies on a simulation to predict the time taken by a MapReduce job. The models implemented in the simulator are very fine grain. They notably take into account the scheduling policy and the data location which are usually unpractical to manipulate as a model.

Specific Models In ARIA [24] an unusual approach is taken to model a MapReduce application when the *Map*, *Shuffle*, and *Reduce* phases are split into several waves of tasks. Its goal is to allocate the resource to meet a soft deadline as a Service-Level Objective. For each phase, the minimum, maximum, and average time are used as they may have a different impact on the total duration of the job. It also make a special case of the first wave of *Shuffle* tasks. Here the models also use the average input size of the *Map* tasks and two constants that model the ratio of data volume between the input and output of the *Map* phase, and between the input and output of the *Reduce* phase.

Another approach [25] to model the duration time of the tasks of a MapReduce job is to use a stochastic model to take into account of the variability of the duration time. It can thus produce a probability function of the duration of the *Map* and one for the *Reduce* tasks. This approach is augmented with the date of arrival of the workers and the duration distribution of the *Shuffle* phase. This approach is most useful to simulate the execution of a MapReduce job as long as nothing changes the distribution function of the duration of the tasks. This makes the models hard to use when it comes to experiment with new scheduling algorithms.

Some work [26] models the execution of a MapReduce job into the cloud taking into account the first upload from the permanent storage to the cloud storage. The *Shuffle* phase seems to be left out, or, at least, not detailed enough. The models use a discrete time interval during which the computation and / or the data transfers occurs. The models used are all linear or affine functions. The goal is to be able to use linear programming to determine an optimal scheduling strategy on several cloud services. However, no follow-up publication actually doing it could be found.

History-Based Models SkewTune [5] and HAT (History-based Auto-Tuning MapReduce) [27] estimate remaining time of the tasks to find stragglers. To this end, they use historical information to estimate the progress of a task to find the slowest tasks and handle them separately as they probably would slow down the whole jobs execution.

Another interesting approach that uses historical information is to use statistics tools [28]. This work does not actually explicitly models the execution of a MapReduce job. Instead, it uses a statistical framework [29] based on KCCA (Kernel Canonical Correlation Analysis) [30] to estimate the time required to execute a give MapReduce job based on previous runs. To do this, it takes into account two sets of parameters. The job configuration and data characteristics on one side, and several performance metrics on the other side. The algorithm is trained with some measures and then tries to predict the performance metrics based on the job configuration and data characteristics. This can also be used as a workload generator. The first downside of this method is that it requires several runs before giving out accurate results. The second one is that this method is complex to implement and manipulate.

Reliability Models On a side note, none of those models take into account the dependability issue which may occur in large scale platforms or on low cost hardware. Some work explore this issue on the hardware [31] and virtual machine level [32]. The reliability has also been taken into account into broader cloud models [33, 34].

3 Defined Models

3.1 Platform Models

The considered target platform is a cluster connected by a single switch, forming a star-shaped communication network. Every link connecting a node to the switch has a capacity of C bytes per second and the switch has a bandwidth of σ bytes per second. The network links are assumed to be full-duplex[1]. σ is supposed to be an integer multiple of the link bandwidth. Thus $\sigma = l \times C$. This restriction is only important for some algorithms presented in Sections 4.2.2 and 4.2.3. Moreover, the overall bandwidth is assumed to be limited. Meaning that if there are n nodes connected to the switch, then $l \leq n$. Above l concurrent transfers, the communications will suffer from contention, thus degrading the performance.

As the focus is on the throughput of the *Shuffle* phase, the data are supposed to be large enough and so that the time to transfer a chunk of data is linear with the size of the chunk. More formally, this means: $time = \frac{datasize}{bandwidth}$.

Thus, the model ignores any latency as well as any mechanism of the network stack that could make the actual bandwidth lower than expected for a short amount of time, such as the TCP slow-start. This network model also ignores any acknowledgment mechanism of the underlying network protocols that can consume some bandwidth and any interaction between CPU usage and bandwidth usage. Therefore, in order to make these assumptions real, we choose to map one *Mapper* or *Reducer* process per physical node when this matters.

3.2 Application Models

A MapReduce application is represented here by the number of *Mapper* processes m, and the number of *Reducer* processes r. A *Map* task i will transfer $\alpha_{i,j}$ bytes to the *Reduce* task j. The amount of data a *Mapper* process i will have to send is called $\alpha_i = \sum \alpha_{i,j}$. And $V = \sum \alpha_i$ is the total volume of intermediate data.

It is also assumed that the intermediate data generated by a given *Mapper* process cannot start being sent to the *Reducers* before the end of the computation. And a *Mapper* i finishes its computation S_i seconds after the first mapper has finished its computation. For the sake of simplicity, it is assumed that the *Mappers* are ordered by the date of termination of the computation. Thus, $S_i < S_{i+1}$ for $1 \leq i < m$ and $S_1 = 0$. Figure 2 shows the Gantt chart of a possible execution of a MapReduce application following this model. The computation time is green, the transfer time is grey and in red is the idle time.

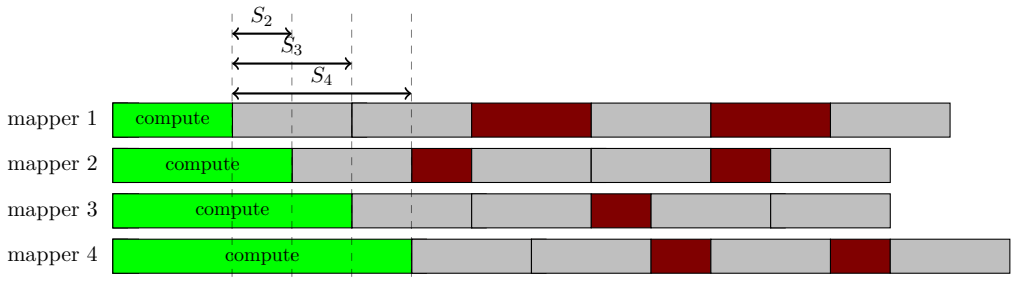


Figure 2: Gantt chart of a possible execution following the application model.

[1] A full-duplex network link can send and receive a frame at the same time.

4 Shuffle Optimization

4.1 Model Analysis

Variable	Unit	Description
σ	B/s	Switch bandwidth.
C	B/s	Link bandwidth.
l		$l = \sigma/C$ Ratio between switch and link bandwidth.
m		Number of *Mapper* processes.
r		Number of *Reducer* processes.
S_i	s	Delay between the end of the first and the i-th *Mapper*.
α_i	B	Amount of data produced by *Mapper* i.
V	B	$V = \sum \alpha_i$ Total volume of data to transfer.
$\alpha_{i,j}$	B	$\alpha_i = \sum \alpha_{i,j}$ The amount of data to be sent from *Mapper* i to *Reducer* j.

Table 1: Summary of the variables of the model.

From the models given in the previous section some properties can be derived which may prove useful regarding the optimization of the duration of the *Shuffle* phase. Namely, a sufficient condition of optimality for a transfer scheduling algorithm can be defined, as well as a lower bound for the duration of the *Shuffle* phase. As a reminder, Table 1 summarizes the notations and variables of the models used.

4.1.1 Sufficient Conditions of Optimality and Lower Bound

As a metric, what we try to optimize is the time between the start of the first transfer and the end of the last transfer. Indeed, in the general case, a *Reduce* task cannot start before all the data are available. Thus, all the *Reduce* tasks will start almost at the same time, which corresponds to the end of the *Shuffle* phase.

Sufficient condition From the afformentioned models, we can derive a few properties of an optimal algorithm. It would be trivial to prove that an algorithm that uses all the available bandwidth from the beginning to the end of the *Shuffle* phase would be optimal, which can be achieved if all the transfers would end at the same time. This is not a necessary condition for an algorithm to be optimal since, in some cases, these requirement cannot be met. But this is sufficient to make an algorithm optimal. See Figure 3 for a representation of the bandwidth usage through time of the case that fulfill the sufficient condition.

Lower bound This sufficient condition allows to compute a lower bound of the *shuffle* duration. This lower bound is divided in two parts. The first part t_1 is when the overall bandwidth is limited by the number of nodes transferring with a bandwidth C. The second part t_2 is the period when the bandwidth is actually limited by the switch to σ bytes per second.

$$
\begin{aligned}
t &= t_1 + t_2 \\
t_1 &= S_l \\
t_2 &= \frac{V - C\sum_{i=1}^{l-1}(S_l - S_i)}{\sigma}
\end{aligned}
$$

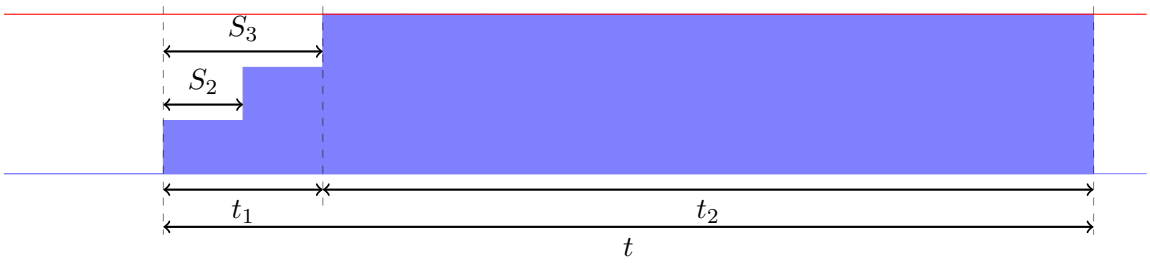


Figure 3: Lower bound calculation based on the bandwidth usage.

As a special case when $S_{i+1} = S_i + \Delta_S \ \forall i \in [1..l-1]$, t_2 can be simplified further. This case is the one we had in the experiments presented in Sections 5.4 and 5.6 and the following formula is the one used as lower bound for comparison.

$$\begin{aligned}
t_1 &= \Delta_S(l-1) \\
t_2 &= \frac{V - C\sum_{i=1}^{l-1} i \times \Delta_S}{\sigma} \\
&= \frac{V - C \times \Delta_S \sum_{i=1}^{l-1} i}{\sigma} \\
&= \frac{V}{\sigma} - \frac{C \times \Delta_S \times l(l-1)}{2\sigma} \\
&= \frac{V}{\sigma} - \frac{C \times \Delta_S \times l(l-1)}{2 \times l \times C} \\
&= \frac{V}{\sigma} - \frac{\Delta_S(l-1)}{2}
\end{aligned}$$

Then the expression of t simplifies also.

$$\begin{aligned}
t &= t_1 + t_2 \\
&= \Delta_S(l-1) + \frac{V}{\sigma} - \frac{\Delta_S(l-1)}{2} \\
&= \frac{V}{\sigma} + \frac{\Delta_S(l-1)}{2}
\end{aligned}$$

This lower bound can be interpreted as the optimal time to transfer all the data through the switch plus a term that represents the non-full usage of the bandwidth during the start of the *Shuffle*. It should be noted that this lower bound does not depend on the individual intermediate data size α_i. It only depends on the total amount of intermediate data and on the network characteristics.

4.2 Algorithms

In order to try to maximize bandwidth usage during the *Shuffle* phase, several algorithms are evaluated. The first one is the simplest algorithm we could imagine, and probably the one

implemented in every framework. It just starts every transfer as soon as the intermediate data are available. Then two discrete algorithms that are either based on a partial order of the transfers or on two ordered lists are presented. And eventually, three algorithms based on bandwidth regulation are presented. They only differ on the way the bandwidth is allocated to every transfer.

4.2.1 Start-ASAP Algorithm

As reference, the simplest algorithm considered is that which consists in starting every transfer as soon as the intermediate data are available. It thus relies on the operating system to share the bandwidth between every transfer from a single node. On average, the system shares the bandwidth equally among the transfers. It also relies on the switch to share the bandwidth fairly among the nodes it interconnects in case of contention.

4.2.2 Order-Based Algorithms

The second algorithm we propose is derived from the work of Berlińska and Drozdowski [12]. In order to reduce the complexity of the algorithm, they propose to always transfer the data in the same order. One transfer is defined by a source *Mapper* and a target *Reducer*. The *Mappers* are ordered by time of arrival and numbered from 1 to m, and the *Reducer* are also numbered from 1 to r.

Order constraints The partial order of the transfer they propose is that every *Mapper* will transfer its intermediate data to the *Reducers* in the order they were assigned (1). This ensures that the *Mapper* private link will never be overloaded. The second constraint (2) is symmetrical, it says that every *Reducer* will receive its intermediate data from the *Mappers* in the order that they were assigned. This ensures that the reducers link will never be overloaded. A transfer is said to be *ready* if and only if those both constraints are met.

$$start(i,j) > end(i,j-1) \qquad \forall i \in 1..m, \forall j \in 2..r \qquad (1)$$

$$start(i,j) > end(i-1,j) \qquad \forall i \in 2..m, \forall j \in 1..r \qquad (2)$$

In our algorithm, we keep that partial order for its property of non-contention of private links. However, unlike Berlińska and Drozdowski the transfers are not run by phases here. Instead, the transfers are run as soon as Constraints (1) and (2) are met and that the bandwidth of the switch is not saturated.

Algorithm The previous constraints only force a partial order on the transfers. A scheduler still have to choose which transfers to start when several transfers are ready. The intuition would say it is better to keep a maximum number of active transfers at the same time in order to maximize the network usage and minimize the completion time. However, Constraints (1) and (2) force a delay of one transfer between the sender transfers. That's why the heuristic proposed here is to keep the active transfers on a diagonal as shown on Figure 4.

More formally, a *reverse priority* ρ is assigned to every transfer $i \to j$ such that $\rho_{i\to j} = i + j$. The lower the value of ρ, the higher the priority. However this priority only comes into play when the dependencies of a transfer are met.

Algorithm 1 present the proposed strategy. In this algorithm, *node.state* hold the representation of the current activity of the node, and *node.target* hold the identifier of the *reduce* to which *node* is transferring or will perform its next transfer. When the transfer from a *mapper*

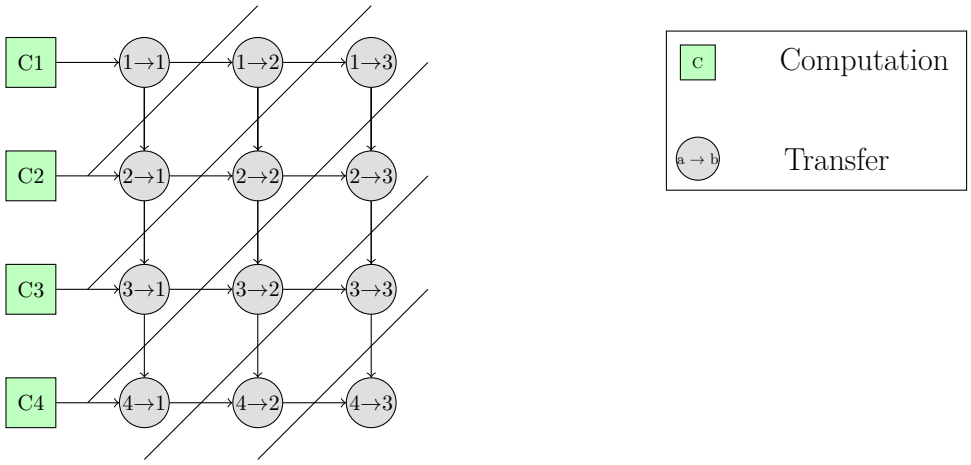


Figure 4: Dependency graph between the transfers with diagonals.

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:        end if
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] est 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
28:        REQUEST_TRANSFER(n)
29:    end if
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:    end if
36: end procedure
```

i to a *reducer* j ends, then for each idle *mapper* i' and its next target *reducer* j', the priority $p_{i'} = i' + j'$ is computed. Then the nodes that minimize $p_{i'}$ are selected. These nodes are considered to be the *most late* and their transfers have to start as soon as possible. This promotes maximization of bandwidth usage and allows not to break Constraint (2) without stating it explicitly in the algorithm.

Procedure ON_COMPUTE_END is called as soon as a *map* task finishes to process its data. It calls the procedure REQUEST_TRANSFER that will start the requested transfer if that does not violate the bandwidth constraints. The procedure ON_TRANSFER_END is called when a transfer ends. It starts by launching the transfer with higher priority if it exists. Then it starts the next transfer of the node that has just terminated if that is possible.

This algorithm enforces the constraints on the network usage while using it at its maximum at any time. Indeed, if the bandwidth of the switch was already fully used, then, the only call to REQUEST_TRANSFER that will actually start a transfer is the one on Line 28. Furthermore, if the order defined by Constraints (1) and (2) prevented a transfer to start, then the switch is not fully used and the termination of one transfer may start 2 new transfers at most. That's what the calls to REQUEST_TRANSFER on Line 28 and Line 32 do.

Limitations The main limitations of this algorithm is that the start up is slow and needs that at least $l - 1$ transfers are finished before enough transfers can run at the same time and fully utilize the bandwidth of the switch. And the symmetric situation happens at the end when there is not enough transfers remaining to run in parallel to saturate the bandwidth.

4.2.3 List-Based Algorithm

In order to overcome the limitations of the order-based algorithm, it has been decided to break the dependencies defined by Constraints (1) and (2) between transfers, and allow to reorder them. However, constraints remain that there must never be two transfers at the same time from a single *Mapper* (3) or toward a single *Reducer* (4) as this would create some contention and degrade the performance.

$$start(i,j) \geq end(i,j') \vee end(i,j) \leq end(i,j') \; \forall i \in [1..m], j, j' \in [1..r], j \neq j' \tag{3}$$

$$start(i,j) \geq end(i',j) \vee end(i,j) \leq end(i',j) \; \forall i, i' \in [1..m], j \in [1..r], i \neq i' \tag{4}$$

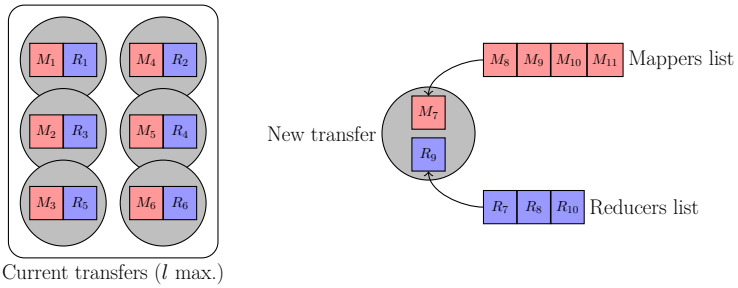


Figure 5: Overview of the list-based algorithm.

Algorithm In order to fulfill Constraints (3) and (4) this algorithm handles the *Reducers* in a list from which a *Reducer* will be taken from and associated to a *Mapper* to form a couple that represents a transfer. Figure 5 shows an overview of the way this algorithm works. When a *Mapper* is ready to run a transfer, the algorithm iterates through the list and takes the first

Reducer that the current *Mapper* has not transfered its data to, yet. Once found, the reducer is removed from the list. This *Reducer* will be put back to the end of the list when the transfer is finished, allowing another *Mapper* to transfer its data to this *Reducer*. Thus, this list of *Reducers* will be kept ordered by the time their last transfer finished.

It may happen that for a given *Mapper* there is no *Reducer* it has not already transfered its data. In that case, another *Mapper* may be waiting for a transfer, and it is given a chance to start a transfer with the same method. Thus the *Mappers* are handled in another list. When there are less than l transfers running at the same time, the first element of the *Mappers* list is taken and the algorithm search for a *Reducer* to run a transfer to from this mapper as previously described. If no *Reducer* can be found, the next *Mapper* in the mapper list is taken, and so on. When a *Mapper* has been found, it is removed from the list. And when a *Mapper* finished its transfer, it is put back in the list if it has not finished all its transfers. But, it is inserted in the list so that the list is ordered by the number of transfers left to be started for every *Mapper*. The more a *Mapper* has transfers to run, the earliest it is in the list.

More formally, it means that there are two lists named ml and rl. ml is empty in the beginning and will contain only the id of the *Mappers* that finished their *Map* computation and still have some intermediate data to transfer. ml is assumed to be automatically ordered by the number of remaining transfers, like a priority queue. rl contains the id of every *Reducer* at the beginning, and is only ordered by the fact that the *Reducer* id will be always be enqueued at the end. Algorithm 2 defines a function TRANSFER_CHOICE that returns a pair of *Mapper* and *Reducer* representing the transfer to be started. This procedure makes explicit the way a transfer is chosen among the remaining transfers.

Limitations Although we expect this algorithm to perform better than the order-based algorithm, some corner cases may still remain. Indeed, it may happen a *Mapper* finishes a transfer and no other transfer can start because all the *Reducers* that have data to receive are already managing another transfer. Thus leading to a suboptimal usage of the bandwidth. We can expect that this situation is more common when l is almost as large as r. Indeed, when l is close to r, most *Reducers* will be involved in a transfer at any given time, thus reducing the possible choice for a target *Reducer*.

4.2.4 Per-Process Bandwidth Regulation

Unlike the previous discrete algorithms, we also propose three algorithms based on the bandwidth regulation of every data transfer. The idea behind these algorithms comes from the sufficient condition of an optimal algorithm that either uses all the bandwidth of the switch or all the bandwidth it can use on the links and that makes all the transfers finish at the exact same time.

For this, we assume that for a given data transfer, a given bandwidth can be maintained. We also assume that this bandwidth can be modified dynamically. The way we achieved this is explained in Section 4.3. We first propose an algorithm based on a per-process bandwidth regulation.

This first algorithm divides the switch's bandwidth σ among the *Mapper* processes. The amount of bandwidth allocated to a given mapper is computed so that its transfer will finish at the same time as the others.

Model addition For this algorithm, the models described in Section 3 and summarized in Table 1 need to be completed. Every *Mapper* is assumed to have the same amount of data to transfer to every *Reducer* process.

Algorithm 2 List-based transfer scheduling algorithm.

```
1: function REDUCER__CHOICE(i)
2:     for all j ∈ rl do
3:         if i has some data to transfer to j then
4:             rl ← rl \ {j}
5:             return j
6:         end if
7:     end for
8:     return ∅
9: end function
10: function TRANSFER__CHOICE
11:     for all i ∈ ml do
12:         j ←REDUCER__CHOICE(i)
13:         if j ≠ ∅ then
14:             ml ← ml \ {i}
15:             return (i, j)
16:         end if
17:     end for
18:     return ∅
19: end function
20: procedure START__TRANSFERS
21:     t ←TRANSFER__CHOICE
22:     while t ≠ ∅ and the number of concurrent transfers < l do
23:         start transfer t
24:         t ←TRANSFER__CHOICE
25:     end while
26: end procedure
27: when Mapper p finishes its computation
28:     ml ← ml ∪ {p}
29:     START__TRANSFERS
30: end when
31: when Transfer p → q finish
32:     if p still have some data to transfer then
33:         ml ← ml ∪ {p}
34:     end if
35:     rl ← rl ∪ {q}
36:     START__TRANSFERS
37: end when
```

$$\alpha_{i,j} = \frac{\alpha_i}{r} \; \forall j \in [1..r] \tag{5}$$

$ready(t)$ is the set of *Mapper* processes that have finished their computations and have not yet finished to transfer their intermediate data at a date t. $\beta_{i,j}(t)$ is the bandwidth allocated to the transfer from *Mapper* i to *Reducer* j at a date t, and $\beta_i(t) = \sum \beta_{i,j}(t)$ the bandwidth allocated to a given *Mapper* process i. And because of assertion (5), we also have $\beta_{i,j}(t) = \beta_i(t)/r$. It also defines $\alpha_i(t)$ the amount of intermediate data a ready *Mapper* i still have to transfer to the reducers at a date t.

Algorithm $\beta_i(t)$ can be computed by solving the following system.

$$\frac{\alpha_i(t)}{\beta_i(t)} = T \qquad \forall i \in ready(t) \tag{6}$$

$$\sum_{i \in ready(t)} \beta_i(t) = \sigma \tag{7}$$

The system (6) – (7) can be reformulated as follows.

$$\frac{\alpha_i(t)}{\beta_i(t)} = \frac{\alpha_j(t)}{\beta_j(t)} \qquad \forall i \in ready(t) \wedge \text{ any } j \in ready(t) \wedge i \neq j$$

$$\sum_{i \in ready(t)} \beta_i(t) = \sigma$$

And then as:

$$\alpha_i(t)\beta_j(t) = \alpha_j(t)\beta_i(t) \qquad \forall i \in ready(t) \wedge \text{ any } j \in ready(t) \wedge i \neq j \tag{8}$$

$$\sum_{i \in ready(t)} \beta_i(t) = \sigma \tag{9}$$

The later is clearly a linear system that can be solved in linear time with the number of *Mappers* $\mathcal{O}(m)$. However, a trivial solution exists to this system. Let's write $V(t) = \sum_{i \in ready(t)} \alpha_i(t)$ as a shorthand. Then a solution is:

$$\beta_i(t) = \sigma \frac{\alpha_i(t)}{V(t)} \; \forall i \in ready(t)$$

The verification of this solution is quite simple. The solution for $\beta_i(t)$ can be replaced in Equation (6) and it can be seen that the value does not depend on i.

$$\begin{aligned} \frac{\alpha_i(t)}{\beta_i(t)} &= \frac{\alpha_i(t)}{\sigma \frac{\alpha_i(t)}{V(t)}} \\ &= \frac{\alpha_i(t) V(t)}{\sigma \alpha_i(t)} \\ &= \frac{V(t)}{\sigma} \end{aligned}$$

Similarly, the value for $\beta_i(t)$ can be replaced in the left hand side of Equation (7) and it can be seen that it is equal to σ as expected.

$$\begin{aligned}\sum_{i \in ready(t)} \beta_i(t) &= \sum_{i \in ready(t)} \sigma \frac{\alpha_i(t)}{V(t)} \\ &= \frac{\sigma}{V(t)} \sum_{i \in ready(t)} \alpha_i(t) \\ &= \frac{\sigma}{V(t)} V(t) \\ &= \sigma\end{aligned}$$

Solving the linear system (8) – (9) may lead to a process bandwidth being greater than the link bandwidth $\beta_i(t) > C$, which could not be supported by the hardware. It is then possible to truncate the bandwidth to C and redistribute the remaining bandwidth among the remaining ready *Mapper* processes. This is more precisely described in Algorithm 3.

Algorithm 3 Per process bandwidth distribution.

1: $E \leftarrow ready(t)$
2: $\tau \leftarrow \sigma$
3: **repeat**
4: $\quad \gamma_i \leftarrow \tau \frac{\alpha_i(t)}{V(t)} \; \forall i \in E$
5: $\quad E' \leftarrow \{i | i \in E \wedge \gamma_i > C\}$
6: $\quad \beta_i(t) \leftarrow C \; \forall i \in E'$
7: $\quad E \leftarrow E \setminus E'$
8: $\quad \tau \leftarrow \tau - C\,|E'|$
9: **until** $E' = \emptyset$
10: $\beta_i(t) \leftarrow \gamma_i \; \forall i \in E$

In this algorithm, E, E', τ, and γ_i are only used as temporary storage. $ready(t)$ and σ are the input, and $\beta_i(t)$ is the output. As previously said, every iteration of this algorithm runs in time $\mathcal{O}(m)$. This can be seen in Algorithm 3 on Lines 4 to 7. And since the set E holds a maximum of m elements in the beginning and is reduced by at least one element on every iteration, that means that a maximum of m iterations may be needed. Thus this algorithm has an overall worst-case complexity of $\mathcal{O}(m^2)$. It should be noted that when the bandwidth of a *Mapper* process i is reduced to C, it means that this process will not be able to complete its transfers at the same time of the others. In this case, this algorithm may not be optimal. A sufficient condition for this algorithm to be optimal can thus be determined.

Sufficient condition of optimality. From the sufficient condition of optimality presented in Section 4.1.1, this algorithm is optimal as long as it can make all the transfers end at the same time. It will make the transfers end at the same time from the point there is enough potential contention for the bandwidth of all the *Mapper* process to be less than C. A necessary condition for this to happen is that every transfer is long enough to not terminate before the last transfer starts, which means $\frac{\alpha_i}{C} > S_m - S_i$. And that at the date $t = S_m$ the remaining intermediate data must be distributed so that no process bandwidth is greater than C.

$$\sigma \frac{\alpha_i(S_m)}{V(S_m)} < C \; \forall i \in [1..m]$$

Limitations As this algorithm assumes that a given *Mapper* process has the same amount of intermediate data to transfer to every *Reducer* (5), the bandwidth computed for every process is divided evenly among the transfers $\beta_{i,j} = \beta_i/r$. However, in reality, assertion (5) will rarely be met, and despite the bandwidth control, the transfers from the same *Mapper* may not all progress at the same speed, thus introducing more imbalance and degrading the performance.

4.2.5 Per-Transfer Bandwidth Regulation

The second algorithm is very similar to the previous one. The main difference is that it tries to address the limitation of setting the same bandwidth to every transfer from a given process by computing a bandwidth for every transfer, thus removing Assertion (5).

Model addition The model and notation here are mostly the same as for the previous algorithm. $ready(t)$ is the set of couple (i, j), i being a *Mapper* which finished its computation, and j being a *Reducer*. $\alpha_{i,j}(t)$ is the amount of intermediate data the ready *Mapper* i has to transfer to the *Reducer* j at a date t. And $\beta_{i,j}(t)$ the bandwidth allocated to the transfer from the ready *Mapper* i to the *Reducer* j at a date t.

Algorithm The bandwidth of transfers $\beta_{i,j}(t)$ can be computed by solving the following system.

$$\frac{\alpha_{i,j}(t)}{\beta_{i,j}(t)} = T \qquad \forall (i,j) \in ready(t) \tag{10}$$

$$\sum_{(i,j)\in ready(t)} \beta_{i,j}(t) = \sigma \tag{11}$$

The system (10) – (11) can be reformulated as follows.

$$\alpha_{i,j}(t)\beta_{k,l}(t) = \alpha_{k,l}(t)\beta_{i,j}(t) \quad \forall (i,j) \in ready(t) \wedge \text{ any } (k,l) \in ready(t) \wedge (i,j) \neq (k,l)$$
$$\sum_{(i,j)\in ready(t)} \beta_{i,j}(t) = \sigma$$

This linear system can be solved in time $\mathcal{O}(m \times r)$. The trivial solution to the linear system of the previous algorithm can be adapted to this one. Let's redefine a shorthand $V(t) = \sum_{(i,j)\in ready(t)} \alpha_{i,j}(t)$. Then a solution is:

$$\beta_{i,j}(t) = \sigma \frac{\alpha_{i,j}(t)}{V(t)} \; \forall (i,j) \in ready(t)$$

This direct solution may, again lead to some bandwidth being greater than the link bandwidth. It can be truncated in a similar way as previously, as shown in Algorithm 4.

Since E may contain every couple of *Mapper* and *Reducer*, its maximal size is $m \times r$. Thus, every iteration runs in time $\mathcal{O}(m \times r)$ because of Lines 4 to 7. And because at least one couple is removed from E at every iteration, this whole algorithm has a time complexity $\mathcal{O}(m^2r^2)$ in the worst case.

Algorithm 4 Bandwidth calculation of the per-transfer bandwidth regulation algorithm.

1: $E \leftarrow ready(t)$
2: $\tau \leftarrow \sigma$
3: **repeat**
4: $\quad \gamma_{i,j} \leftarrow \tau \frac{\alpha_{i,j}(t)}{V(t)} \; \forall (i,j) \in E$
5: $\quad E' \leftarrow \{(i,j) | (i,j) \in E \wedge \gamma_{i,j} > C\}$
6: $\quad \beta_{i,j}(t) \leftarrow C \; \forall (i,j) \in E'$
7: $\quad E \leftarrow E \setminus E'$
8: $\quad \tau \leftarrow \tau - C \, |E'|$
9: **until** $E' = \emptyset$
10: $\beta_{i,j}(t) \leftarrow \gamma_{i,j} \; \forall i \in E$

Limitations This algorithm has two main limitations. The first one is that, although it takes into account the capacity of the network links, it does not take into account that several transfers occur at the same time from a given process or toward a given process. Thus, even if every transfer bandwidth $\beta_{i,j}(t)$ is less than the capacity of the link C, the sum of the bandwidth of all the transfers of one process may be greater than C. A consequence of this, is that the bandwidth of the switch σ may not be reached because the bandwidth throttled by the network interface cannot be allocated to the other transfers. The second limitation is the scalability of this algorithm. If $m = r$, the complexity of this algorithm is $\mathcal{O}(m^4)$ which is not acceptable.

4.2.6 Two Phases Per-Transfer Regulation

A mix of both above regulation-based algorithms should be able to overcome most of the limitations. The idea of this third algorithm is to compute the bandwidth with the per-process bandwidth regulation algorithm, and then distribute it according to the amount of intermediate data of every transfer, instead of allocating it evenly. This algorithm is expected to never allocate too much bandwidth to a given *Mapper* process and make all the transfers of all the *Mappers* finish at the same time.

Model addition Similarly to the previous algorithms $ready(t)$ is the set of *Mapper* processes that have finished their computation but not the transfer of their intermediate data. $\beta_{i,j}(t)$ is the bandwidth allocated to the transfer from *Mapper* i to *Reducer* j at a date t, and $\beta_i(t) = \sum \beta_{i,j}(t)$ the bandwidth allocated to a given *Mapper* process i. $\alpha_i(t)$ is also the amount of intermediate data a ready mapper i still have to transfer to the reducers at date t. And $\alpha_{i,j}(t)$ the amount of data still to be transfered from *Mapper* i to *Reducer* j.

Algorithm The first phase of this algorithm is exactly Algorithm 3 that computes values for $\beta_i(t)$. The second phase applies a very similar algorithm for every process in order to distribute the bandwidth among the transfers. Algorithm 5 shows whole algorithm. Lines 1 to 10 come from the per-process bandwidth regulation algorithm, it computes $\beta_i(t)$ from $\alpha_i(t)$, $V(t)$, σ and $ready(t)$. The second phase spans across Lines 11 to 13. It takes values for $\beta_i(t)$ as input and produce values for $\beta_{i,j}(t)$ as output.

The worst-case complexity of the first phase is $\mathcal{O}(m^2)$ just like the per-process regulation algorithm. The complexity of the second phase is $\mathcal{O}(m \times r)$ because the loop iterates m times, and every iteration has $\mathcal{O}(r)$ operations to perform. The complexity of the full two-phases algorithm is then $\mathcal{O}(m^2 + m \times r)$. The second phase can be distributed and every *Mapper* can

Algorithm 5 Bandwidth calculation of the two-phases bandwidth regulation algorithm.

$E \leftarrow ready(t)$
$\tau \leftarrow \sigma$
repeat
    $\gamma_i \leftarrow \tau \frac{\alpha_i(t)}{V(t)} \; \forall i \in E$
    $E' \leftarrow \{i | i \in E \wedge \gamma_i > C\}$
    $\beta_i(t) \leftarrow C \; \forall i \in E'$
    $E \leftarrow E \setminus E'$
    $\tau \leftarrow \tau - C \, |E'|$
until $E' = \emptyset$
$\beta_i(t) \leftarrow \gamma_i \; \forall i \in E$
for $i \in ready(t)$ **do**
    $\beta_{i,j}(t) \leftarrow \beta_i(t) \frac{\alpha_{i,j}(t)}{\alpha_i(t)} \; \forall j \in E$
end for

distribute its allocated bandwidth $\beta_i(t)$ on its own. Thus reducing the worst-case complexity of the whole algorithm to $\mathcal{O}(m^2 + r)$.

Limitations Although this algorithm prevents any contention on the switch or on the private links of the *Mappers*, contention may happen on the *Reducers* side. Indeed, nothing prevents several *Mappers* from sending some data to the same *Reducer* with a bit rate sum greater than the bandwidth of its private link.

4.3 Point-to-Point Bandwidth Regulation

The regulation-based algorithms assume that it is possible to regulate the bandwidth of every transfer and to change the target bandwidth at any moment. However, this is not an immediate task since, in the end, only packets can be sent over the network. Our regulation algorithms raise two main challenges regarding this issue. The first one is about actually limiting the average bandwidth to a given value. The second challenge is about dynamically modifying the bandwidth at any time and any given number of times while still guarantying that the average bandwidth is the one requested. Some work exists on the topic [35, 36, 37], however, most of them focus mainly on only maintaining an average bandwidth. Moreover, it was pretty straightforward to implement in our own framework.

4.3.1 Limiting the Average Bandwidth

The first step is to limit the average bandwidth. The algorithm we used for this is quite simple. The amount of data that has been sent is accumulated, and after a chunk of data has been sent, the execution is suspended for a certain duration.

The sleep duration is computed so that the average bandwidth from the beginning of the transfer to that date is exactly the average bandwidth wanted. So if t_0 is the date the whole transfer started, d is the total amount of data sent, and β is the wanted bandwidth, then, the execution will be suspended until the date $t_1 = t_0 + \frac{d}{\beta}$. Algorithm 6 shows a pseudo code for this algorithm. This algorithm is independent from the size of the chunk of data. However, as it is shown in Section 5.2.2 on a real computer, the size of the chunk may have an impact on the performance.

As long as every *send* happens at a bandwidth greater than β, it is a direct consequence that this algorithm will guaranty the average bandwidth (from t_0 to the end) to be β. However,

Algorithm 6 Simple regulation of a point to point bandwidth.

```
1: t_0 ← now()
2: d ← 0
3: for chunk ∈ data do
4:     send(chunk)
5:     d ← d + size(chunk)
6:     sleep_until(t_0 + d/β)
7: end for
```

it can also proved that between two iterations, the average bandwidth is also β. Indeed, let's name t_i the date at the end of the i-th iteration and d_i the value of d at the date t_i. Then an expression for the date t_i can be written and the time taken by one iteration can be computed.

$$
\begin{aligned}
t_i &= t_0 + \frac{d_i}{\beta} \\
t_{i+1} - t_i &= \left(t_0 + \frac{d_{i+1}}{\beta}\right) - \left(t_0 + \frac{d_i}{\beta}\right) \\
t_{i+1} - t_i &= \frac{d_{i+1}}{\beta} - \frac{d_i}{\beta} \\
t_{i+1} - t_i &= \frac{d_{i+1} - d_i}{\beta}
\end{aligned}
$$

Thus, the time taken by the $i+1$-th iteration is exactly the time that would be needed to send the $i+1$-th chunk of data at a bandwidth of β byte per second.

4.3.2 Run Time Setting of the Bandwidth

The second step is to make it work with β being variable through the time, making it a function of the time $\beta(t)$. The easiest solution of atomically resetting t_0 to $now()$ and d to 0 could produce undesirable effects. If it is chosen to not interrupt the *sleep_unitl*, then the average bandwidth would not be equal the average of $\beta(t)$ because Algorithm 6 may sleep too long if the target bandwidth has been suddenly raised. Conversely, interrupting the *sleep_until* call and continuing the execution would always produce another send to happen. Which can may compromise the regulation if the target bandwidth is often changed.

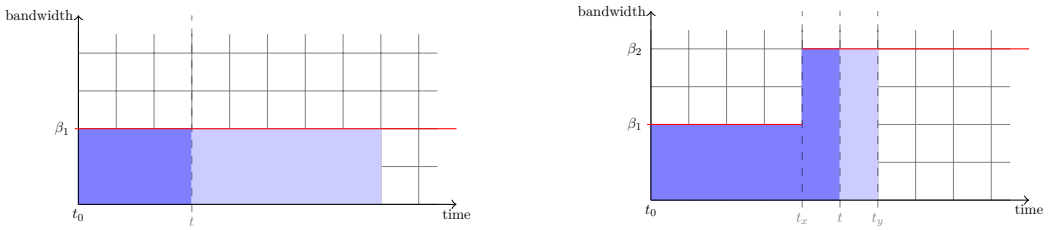
Another approach is to lie to Algorithm 6 by setting a fake value for t_0 named t'_0. A value that would lead it to make the average real bandwidth to be equal to the average value of $\beta(t)$ while still guarantying that when β is not modified, the target bandwidth is maintained.

Let's consider a scenario where from t_0 to t_x the target bandwidth is β_1 and from t_x to t_y the target bandwidth is β_2. d_x bytes are transfered between t_0 and t_x. And d_y bytes are transfered between t_0 and t_y. At t_x the value of t_0 change to be t'_0, hence $t_y = t'_0 + \frac{d_y}{\beta_2}$. Then, solve for t'_0 the equation between the average real bandwidth and the average of the wanted bandwidth.

$$\begin{aligned}
\frac{d_y}{t_y - t_0} &= \frac{\beta_1 (t_x - t_0) + \beta_2 (t_y - t_x)}{t_y - t_0} \\
d_y &= \beta_1 (t_x - t_0) + \beta_2 (t_y - t_x) \\
d_y &= \beta_1 (t_x - t_0) + \beta_2 \left(t'_0 + \frac{d_y}{\beta_2} - t_x \right) \\
d_y &= \beta_1 (t_x - t_0) + \beta_2 t'_0 + d_y - \beta_2 t_x \\
t'_0 &= \frac{\beta_1 (t_x - t_0) - \beta_2 t_x}{-\beta_2} \\
t'_0 &= t_x - (t_x - t_0) \frac{\beta_1}{\beta_2}
\end{aligned}$$

It is really interesting to remark that this value of t'_0 does not depend on any d_i nor on t_y. This means that this new value for t_0 can be computed at the date t_x when the target bandwidth is actually changed. Then, the *sleep* of Algorithm 6 can be interrupted and restarted with the new values for β and t_0.

This result can actually be intuitively understood by picturing a graph of the wanted bandwidth through time. Figure 6 shows how the past and foreseen bandwidth usage evolves during the time t. In dark blue is the past bandwidth usage as pictured by the model, and in light blue is how the bandwidth should be used if nothing changes in the future. In Figure 6(a) the bandwidth limit is set at β_1, while, in Figure 6(b) the bandwidth limit is set to β_2 at the date t_x. The total blue area (proportional to the amount of data) is the same in both cases.



(a) Before target bandwidth change.

(b) After target bandwidth change.

Figure 6: Schema of bandwidth usage past and foreseen.

When computing a fake value for t_0 what really happens is that a *fake shape* is with the same surface as before but, with a height (a bandwidth) equal to the new bandwidth. Figure 7 shows an example of what could happen. This allows to remove any reference to the previous bandwidth as well. Given this, it can easily be seen that this technique will work for more than one bandwidth update.

4.3.3 Precision Improvement

As the three regulation-based algorithms rely on a continuous data transfer while the data are actually sent by chucks. Providing them with the amount of data really transfered may lead to wrong bandwidth calculation. Indeed, reporting that a chunk of data has been fully transfered while a *sleep* is currently smoothing the consumed bandwidth, break the assumption that the transfer is continuous. This can lead the regulation-based algorithm to compute a wrong bandwidth.

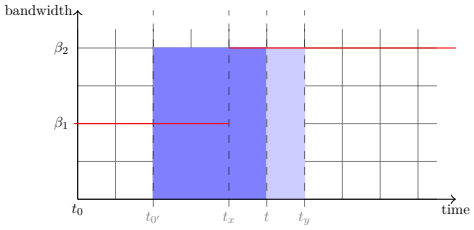


Figure 7: Schema of the calculated $t_{0'}$.

That is why the remaining data size reported is interpolated as follow. With t_x being the date when the remaining size is asked, and r the amount of data actually reported.

$$r = \alpha_{i,j} - \beta_{i,j}\,(t_x - t_0)$$

4.4 Implementation Details

All those algorithms are implemented in *HoMR*. *HoMR* is our Home-made MapReduce. It is written in C++ under the scope of the ANR project MapReduce. It is built from software components, based on the L^2C low-level component model. The low-level component assembly is generated from a high-level component model HLCM [38]. This allows to easily swap any component implementation with another variant. This is used to test several *Shuffle* schedulers and to replace the word reader with a word generator.

The transfer scheduler components are implemented in an event-driven way. Every time a computation or a transfer finishes, a method of the transfer scheduler is called. In every such event the three regulation-based schedulers recompute the new bandwidth allocated to the processes or to the transfers. In a perfect world, there is no need to recompute the bandwidth allocation out of these events. However, it does not cost anything more than an $\mathcal{O}(m \times r)$ packets exchange to recompute the bandwidth allocation when no event have been received during a few seconds. In our regulation-based transfer schedulers, the bandwidth allocation is recomputed after 5 seconds of idle time.

The bandwidth regulator presented in Section 4.3 uses 4 threads to send the data. This helps to improve the maximal bandwidth that can be reached. This number of threads has been determined by running a few tests by hand, increasing the number of threads until it no longer improves the maximal bandwidth. The same behavior has been observed with *iperf* which shows a maximal throughput for 4 client threads.

5 Experiments

In order to test these algorithms, a few experiments have been performed on the Grid'5000 experimental testbed, first to ensure the environment behave as expected, and second to compare the 6 algorithms presented in this document.

5.1 Platform Setup

The model assumes that the platform is a switched star-shaped network with a limited bandwidth on the switch. And the whole point of the algorithms is to control the bandwidth used during the *Shuffle* phase. Thus, in order to evaluate these algorithms in the case of several switch bandwidth limit configurations, a switch is simulated by the mean of a node dedicated to routing packets and all other nodes configured to route packets through this node. However,

as all the nodes are physically connected to a real switch, the *ICMP redirect* mechanism had to be disabled to ensure that every packet sent over the network really goes through the node designated as the router.

This may not be an optimal simulation of a switched network since the routing mechanism implies a *store-and-forward* method of packets forwarding, instead of a *cut-through* as most switches do. However, we believe that this does not have a significant effect on the measured throughput. This allow to easily control the overall bandwidth available on this routing node.

As all the packets have to go through the network interface of the router node twice, a fast network is needed in order to simulate a switch with a throughput greater than 1 Gbps. Thus, *InfiniBand 40G* interfaces are used with an *IP over InfiniBand* driver for the ease of use. The bandwidth of the router is controlled with the Linux `tc` tool. The performance behavior and limit of this setup has been tested and the results are shown in Section 5.2.1. As the network is based on fast network interface controllers (NIC), the bandwidth of the private links is also limited to 1 Gbps with `tc`. The `tc` rules used to limit the bandwidth are the same on the router and on the compute nodes. They are based on the Hierarchical Token Bucket (HTB) method to limit the outgoing bandwidth and on the default algorithm to limit the incoming bandwidth. The operating system of the nodes is Debian wheezy with Linux 2.6.32 as a kernel.

The hardware used is the Edel cluster on Grid'5000. Every node on this cluster has two quad-core CPUs Intel Xeon E5520 @2.27 GHz. Every node is equipped with 24 GB of memory, 1 Gigabit Ethernet and 1 InfiniBand 40G cards. On this hardware, a latency of 0.170 ms has been measured on average on a direct point to point ping using the Gigabit Ethernet NIC. A latency if 0.315 ms is also measured using the above-mentioned routed network setting on the InfiniBand NIC.

5.2 Preliminary Tests

5.2.1 `tc` Regulation on Router

In order to test whether the bandwidth can be limited correctly on the router, a setup with 2 nodes plus a router node is used. Only the router node has a limited bandwidth, and `iperf` is run on the two other nodes to measure the actual bandwidth.

The bandwidth limit is set with `tc` from 100 Mbps up to 12 Gbps by steps of 100 Mbps. And the actual bandwidth is measured with `iperf` with 4 parallel clients threads on the client side. Every experiment is run 5 times to estimate the variability of the measure.

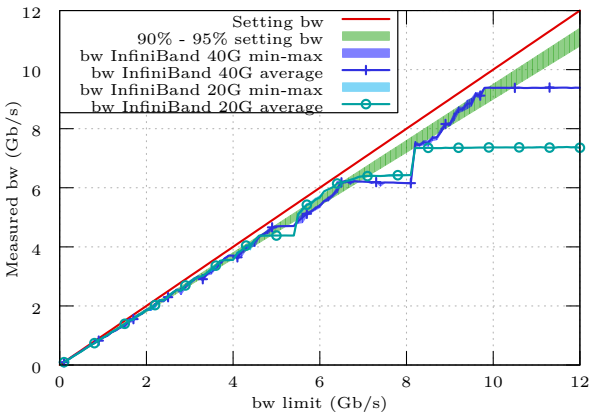


Figure 8: Bandwidth limitation on Linux + `tc` on InfiniBand.

Figure 8 shows the results of this experiment. Globally, it can be seen that the measured

bandwidth follows an almost linear trend which corresponds to roughly 90% to 95% of the target bandwidth. This trend continues until the maximal bandwidth the system can support is reached. Also, the measures are quite stable as the difference between the maximal and minimal measured bandwidth never exceeds 0.28 Gbps or 8% of the average bandwidth.

However, some steps are clearly distinguishable around 5 Gbps and from 6.5 to 8 Gbps. During these steps, increasing the bandwidth limit with `tc` does not increase the actual bandwidth. As the result is surprising, it has been re-executed on another cluster with *InfiniBand 20G* network adapters, and we see that the same result happen. We have no real explanation for that. The experiment has also been tried with a Linux 3.2.0 with a Debian Wheezy, the results (not shown here) are completely different and show a greater variability. Thus, we think that this is a performance bug in Linux.

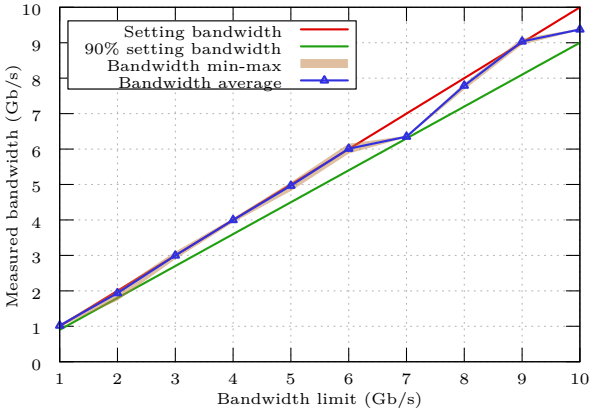


Figure 9: Bandwidth test of Linux + `tc` on InfiniBand with correction.

In order to still get the wanted bandwidth, the data from Figure 8 are used to find a setting bandwidth that would result in an actual bandwidth being close to the target one. For instance, to get an actual bandwidth of 5 Gbps, the bandwidth limit set with `tc` is 5.994 Gbps. Since for the real experiments the bandwidth limit will vary by 1 Gbps steps, the value for only 10 target bandwidth have to be found. Figure 9 shows the bandwidth measured with `iperf` when using to the corrected setting bandwidth. It can be noted that the average actual bandwidth is remarkably accurate with respect to the setting bandwidth. The only deviations that can be noted happen for a target bandwidth of 7, 8 and 10 Gbps. Those points corresponds to the biggest steps in Figure 8 and to the maximal reachable bandwidth. For those points, the drift could not be completely compensated. Those outliers show a bandwidth still greater than 90% of the target bandwidth. Except from those 3 points, all other points show an actual bandwidth between 96% and 103% of the bandwidth wanted.

5.2.2 Bandwidth Regulation

The second base block on which these algorithms rely on is the ability to regulate the bandwidth at the application level. The method used to obtain this behavior is described in Section 4.3.

To check whether this performs correctly, an experiment is set up with only two nodes interconnected by an *InfiniBand 40G* network. Then the size of the messages varied from 4 bytes to 64 MB and the target bandwidth from 1 KB/s to 1 GB/s and the overall average bandwidth is measured. Each measure is repeated 10 times.

Figure 10 shows a 3D plot of the results of this experiment. It shows the actual bandwidth with respect to the message size and to the desired bandwidth. Some points are missing in

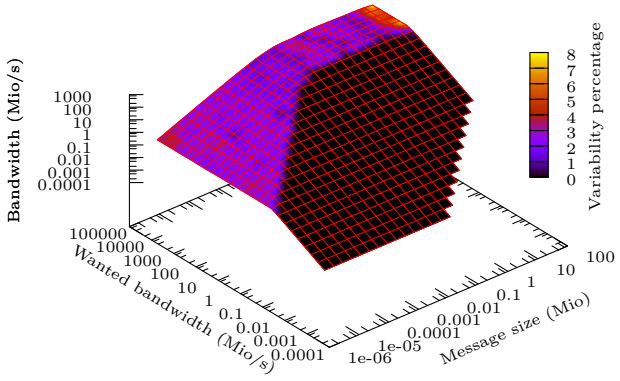


Figure 10: Bandwidth regulation test, 3D plot.

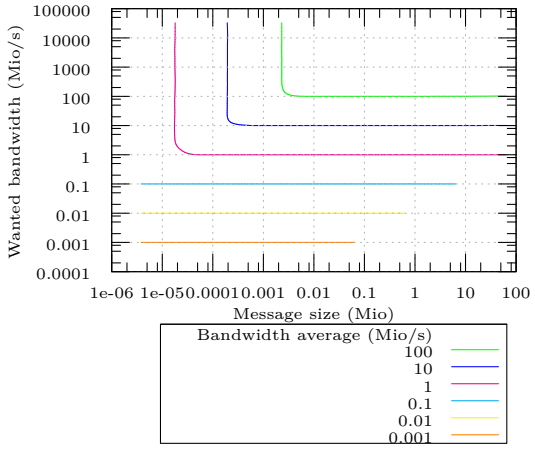


Figure 11: Bandwidth regulation test, contour plot.

the result because sending a large amount of data at a very low bandwidth would require too much time. The colors represent the percentage of variability. Figure 11 represents the same information under the form of a contour plot.

The black plan on Figure 10 shows that when the required bandwidth is small enough and the block size is big enough, the bandwidth regulation system is able to maintain the desired bandwidth with a pretty good accuracy. However, when the message size is too small, the system becomes CPU bound and cannot send enough data to reach the desired bandwidth. And because this case is CPU bound, a variability of 0.5% to 1% can be seen. Finally, when the messages are large enough and the required bandwidth is high enough, the system reaches the limit of the network interface and become IO bound. Thus our bandwidth regulation component works as intended.

5.3 Synchronous Transfer Start

The first experiment with our algorithms is simple and all other experiments are only variations of this one. The job that is run is a word count. However, for the sake of simplicity and control, the data are not read from a file, they are generated by a component *WordGenerator*. This

allows to control the amount of intermediate data produced. In order to control the time at which the *Map* computations end, an artificial synchronization barrier is added. This allows for an evaluation of the behavior of the *Shuffle* phase.

For this first experiment all the *Map* computations finish at the exact same time and every *Mapper* have the same amount of intermediate data. Every *Mapper* process generates 2.56 GB of intermediate data. The same amount of data has to be sent to every *Reducer*. The router's bandwidth is then varied from 1 Gb/s to 10 Gb/s and the time taken from the start of the first transfer to the end of the last transfer is measured. This duration is then compared to the lower bound. This experiment is run with 10 *Mappers* and 10 *Reducers*. Every configuration is run 5 times.

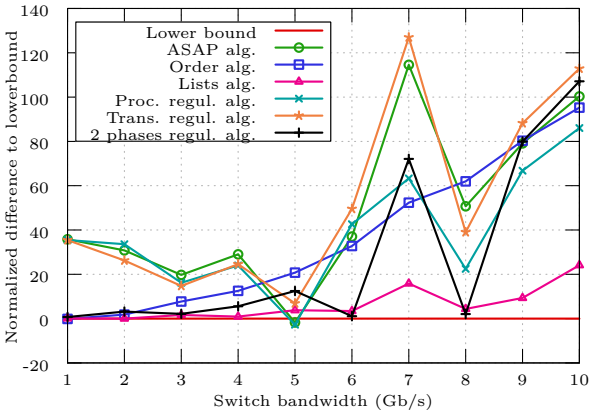


Figure 12: Median time taken by all the 6 algorithms under various bandwidth restrictions with same amount of data and synchronous start of transfers.

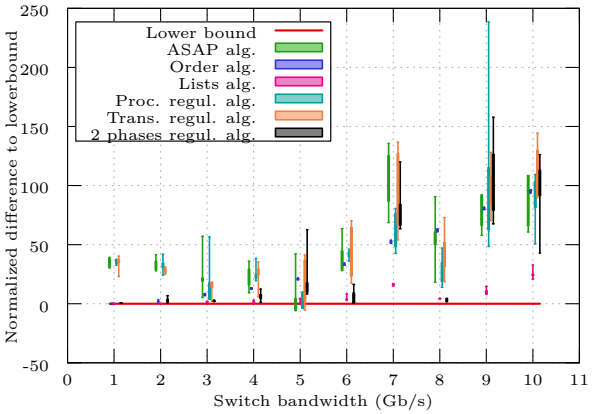


Figure 13: Variability of the time taken by all the 6 algorithms under various bandwidth restrictions with same amount of data and synchronous start of transfers.

Figure 12 shows the result of this experiment in terms of percentage to the lower bound. As every measure has been made 5 times, the median time is represented on this figure. Figure 13 shows the variability of the measures in candle sticks.

The result for the discrete algorithms (*Order alg.* and *Lists alg.* on the figure) is close to what was expected. The result for the order-based algorithm increases on Figure 12, not because it takes more time as the bandwidth of the switch increase, but because it does not

decrease as fast as the lower bound. The time needed before the maximal number of concurrent transfers can run simultaneously become less and less negligible as the overall allowed number of concurrent transfers increases. The lists-based algorithm shows a behavior close to the optimal. The only performance degradation occurs for a switch bandwidth of 7 Gb/s and 10 Gb/s. Those configuration, as of Figure 9 are known not to offer the actual bandwidth wanted.

The per-process regulation algorithm, per-transfer regulation algorithm, and the reference algorithm behave similarly for the same reasons. They create contention at some point, thus losing some packets creating a latency. The per-process regulation algorithm is not supposed to generate any contention for this experiment. The cause of this is unknown. However, the instantaneous egress bandwidth of the switch (not shown here) shows that those 3 algorithms can reach the bandwidth actually set with `tc` by running some transfers from several nodes at the same time. While the `tc` skew correction has been tested only for several transfers from the same node with `iperf`.

The two phases regulation algorithm shows a good behavior for a switch bandwidth less or equal to 8 Gb/s. Above that limit it creates contention and exhibits a behavior as bad as the reference algorithm. Also, for 7 Gb/s, this algorithm produces a peak of bad performance. This can be interpreted as a high sensitivity to the setting of the switch bandwidth. If the switch bandwidth is overestimated, the two phases algorithm creates contention which leads to a very bad performance. While if it is underestimated, the switch would never be used at its maximal capacity.

5.4 1 Second Steps Between Computation End

The second experiment is very similar to the previous one. Only a one-second delay between the end of every *Map* computation has been added, thus creating a slight imbalance among the *Mapper* process.

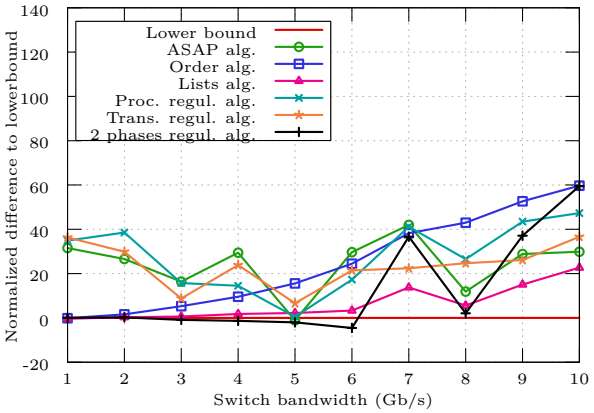


Figure 14: Median time taken by all the 6 algorithms under various bandwidth restrictions with same amount of data and 1 second step between transfer start.

Figure 14 shows the result of this experiments in terms of percentage to the lower bound. The discrete algorithms show a similar behavior as during the previous experiment. However, the order-based algorithm shows a slightly better behavior. Its distance to the lower bound increases slower. This is due to the time taken to gain parallelism among the transfers that is partially compensated by the 1 second steps between the computation end. The reference algorithm also shows a better performance. This is due to the fact that in the beginning and in the end, not all the *Mappers* are transferring data, thus there is less contention and less

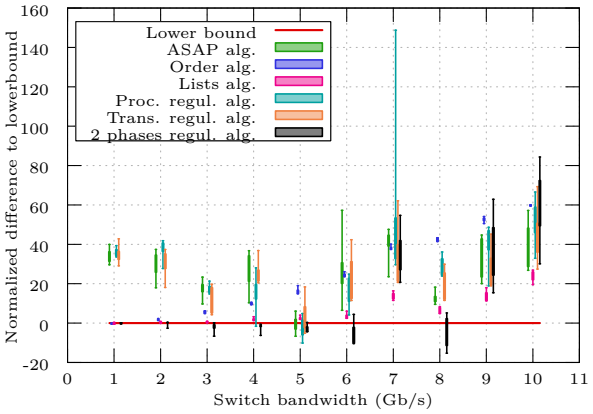


Figure 15: Variability of the time taken by all the 6 algorithms under various bandwidth restrictions with same amount of data and 1 second step between transfer start.

performance degradation. The global behavior of the list-based and two-phases algorithms remains the same. However the two-phases algorithm appears to be super-optimal by up to 5% for some configurations. The cause is not very clear. It is supposed to be caused by `tc` that is not very accurate to limit the bandwidth from several sources.

5.5 Synchronous Transfer Start with Heterogeneous Amount of Data

For the third experiment, all the *Mappers* finish their computation at the same time, but all the *Mapper* do not have the same amount of intermediate data. The amount of intermediate data for each *Mapper* is 2.56 GiB plus $(i-1) \times 64$ MiB, i being the id of the *Mapper* process, ranging from 1 to m. As previously, we expect the smarter algorithms to perform better.

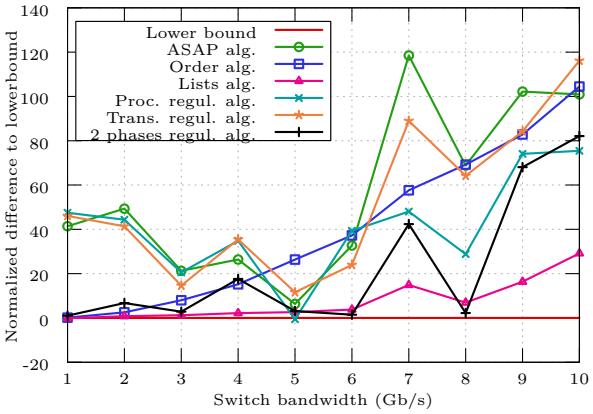


Figure 16: Median time taken by all the 6 algorithms under various bandwidth restrictions with various amount of data and synchronous start of transfers.

The results of this experiments presented in Figure 16 show similar results as the previous experiment. Variability is shown in Figure 17.

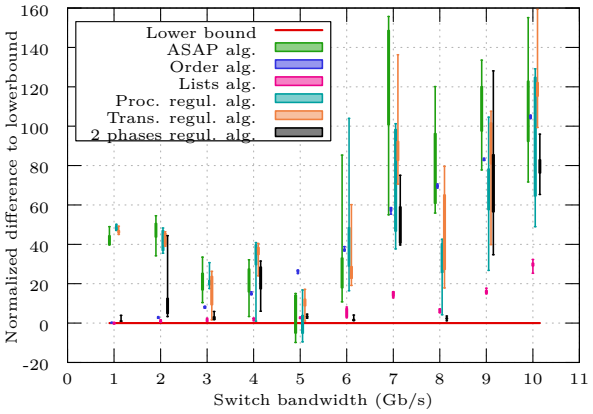


Figure 17: Variability of the time taken by all the 6 algorithms under various bandwidth restrictions with various amount of data and synchronous start of transfers.

5.6 1 Second Steps Between Transfer Start with Heterogeneous Amount of Data

And finally, the fourth experiment combines the delay before starting the transfers and the imbalance of the amount of intermediate data.

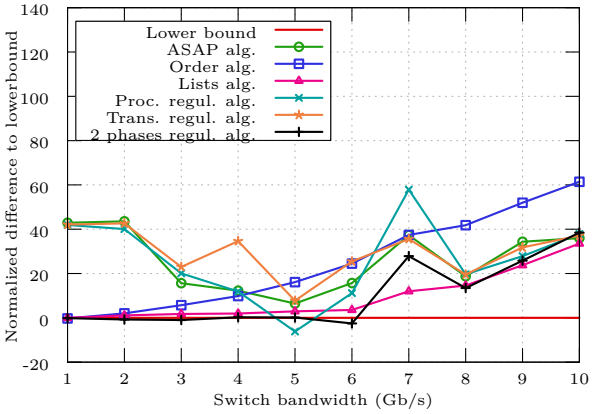


Figure 18: Median time taken by the 6 algorithms under various bandwidth restrictions, with an imbalanced amount of intermediate data and non-synchronous transfer start.

The results of this experiments presented in Figure 18 in terms of difference to lower bound normalized to the lower bound itself. The variability of the measures is presented in Figure 19 in the form of whiskers boxes.

On the results it can be seen that the discrete algorithms show a quite smooth behavior, the list-based algorithm being always better than the order-based algorithm. Both show a quite low variability, except one measure for the order-based algorithm that show an outlier for a router bandwidth of 5 Gb/s. During this measure, the bandwidth on the router node has dropped to 0 unexpectedly during 10 seconds. The order-based algorithm show a performance behavior that gets further and further from the lower bound, as we would expect because of slow start up and slow *stop down* as explained in Section 4.2.2. The same global behavior is observed for the lists-based algorithm, this time it is the heuristic for sorting the *Mappers* by priority that cannot catch up the imbalanced transfer progression brought by the 1 second delay of the

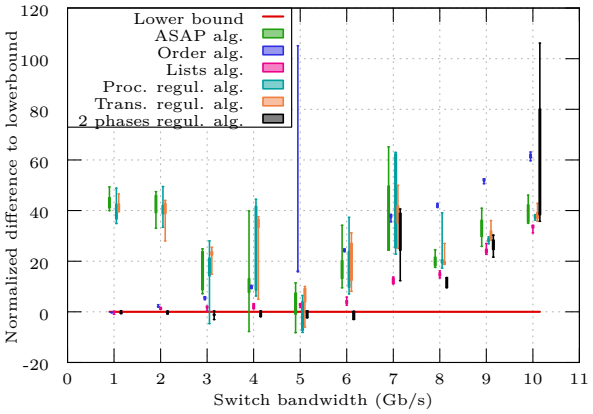


Figure 19: Variability of the time taken by the 6 algorithms under various bandwidth restrictions, with an imbalanced amount of intermediate data and non-synchronous transfer start.

transfer start and the 64 MiB steps of the amount of intermediate data.

The three algorithms ASAP, per-process regulation, and per-transfer regulation show a poor performance for a small router bandwidth. Indeed, those algorithms generates some contention either on the router or on the private links. This contention leads to packet loss and retransmission after a timeout. These algorithms also show a performance that is equivalent to that of the 2 phases regulation and lists-based, for a router bandwidth large enough. The per-process regulation algorithm show a super-optimal performance for a router bandwidth of 5 Gb/s. This is actually due to an overshoot of the bandwidth set with `tc`. Indeed, while the transfers from a single node to another does not exceed the wanted bandwidth on the router as seen on Figure 9, it appears that when several nodes transfer some data at the same time, the total bandwidth may exceed a bit the target bandwidth. Globally, the 3 algorithms ASAP, per-process regulation, and per-transfer regulation show a bowl-shaped performance curve centered around 5 Gb/s. The left part seems to be due to the decrease of the contention ratio, leading to an increase of the performance. And the right part seems to be due to the imbalance among the amount of intermediate data to transfer that makes the increase of the bandwidth of the router have only a slight impact in the actual performance.

The two phases regulation algorithm exhibits an optimal behavior for a router bandwidth less or equal to 6 Gb/s. It may even be slightly super-optimal for the same reason exposed before. However, as this algorithm regulates the bandwidth so that it never exceeds the bandwidth of the switch (here simulated by a router) the overshoot can never be very large. As before, the two phases regulation algorithm also shows a peak at 7 Gb/s due to the fact that the bandwidth of the router could not be set to exactly 7 Gb/s. Thus some contention appears and degrades the performance as it happens to the ASAP algorithm for instance. This behavior can be interpreted as a high sensitivity to contention thus making the parameter σ of this algorithm (the switch bandwidth) critical for a good performance.

6 Conclusion and Future Works

The *Shuffle* phase has been largely ignored by the academic work despite being a potentially important bottleneck. In this report we show that although a no-op algorithm performs well under perfect balance and synchronous conditions without contention, smarter algorithms are proven to be more efficient in all other cases. Especially the list-based algorithm and the two phases algorithms. The second one may perform optimally in some cases, but is quite sensitive

to the switch bandwidth parameter while the first one only needs to know how many concurrent transfers the switch can support. While those two algorithms periodically communicate with the centralized scheduler, the list-based algorithm induces an idle time between the transfers while the scheduler make a decision. This does not happen with the regulation algorithms since they just continue their transfer with the former bandwidth set while the scheduler computes the new one. The scalability of these algorithms still has to be tested.

However, we believe that the results could be better if the bandwidth of the router could be precisely limited with `tc`. Our attempt at mitigating the aberrations are a good start but it shows some limitations. A future direction could be to compare the behavior of Linux + `tc` with that of a real switch. This work assumed that a switched network with limited bandwidth could be simulated with a routed network. However, most current switches uses a *cut-through* method for forwarding the packets, while a routed network with a Linux system implies a *store-and-forward* method. It has not been proven that this difference does not have a sensible influence on the performance. We could make some experiments on a real switch. We could also try to map the *Mappers* and *Reducers* processes on the same nodes and check whether or not the same behavior is observed. Some parameters have to be known by the regulation algorithms such as the links bandwidth and the switch bandwidth. It would be a great improvement for the usability if those parameters could be determined automatically. Or even better, if they could be adjusted at run time if the bandwidth has to be shared with other applications. The platform model currently assumes that the network topology is switched star-shaped with an equivalent bandwidth on every private link. Extending both the models and algorithms to other network topologies and with less restriction is also interesting. Some of these algorithms could be extended to start the data transfer of the intermediate data before the computation is finished. Although extending the algorithms seems easy, their performance and the influence of the uncertainty about the data not produced yet bring new challenges.

7 Acknowledgment

This work is supported by the French National Research Agency (Agence Nationale de la Recherche) in the framework of the MapReduce project under Contract ANR-10-SEGI-001.

The experiments referenced in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners.

References

[1] 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.

[2] 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.

[3] Matei Zaharia and Dhruba Borthakur. Delay Scheduling: A simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[4] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

[5] YC Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating Skew in MapReduce Applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36, 2012.

[6] Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, and Song Guo. SAMR: A Self-Adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 2736–2743. Ieee, June 2010.

[7] Luiz Angelo Steffenel. Modeling Network Contention Effects on All-to-All Operations. *2006 IEEE International Conference on Cluster Computing*, 2006.

[8] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. 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)*, pages 17–24, Indianapolis, IN, November 2010. IEEE.

[9] Sangwon Seo, Ingook Jang, Kyungchang Woo, Inkyo Kim, Jin-Soo Kim, and Seungryoul Maeng. 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, September 2009.

[10] Yen-Liang Su, Po-Cheng Chen, Jyh-Biau Chang, and Ce-Kuen Shieh. Variable-Sized Map and Locality-Aware Reduce on Public-Resource Grids. *FGCS*, 27(6):843–849, June 2011.

[11] Bharadwaj Veeravalli, Debasish Ghose, Venkataraman Mani, and Thomas Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, 1996.

[12] Joanna Berlinska and Maciej Drozdowski. Scheduling Divisible MapReduce Computations. *Journal of Parallel and Distributed Computing*, 71(3):450–459, March 2010.

[13] Sylvain Gault. Ordonnancement Dynamique des Transferts dans MapReduce sous Contrainte de Bande Passante. In *ComPAS'13 / RenPar'21 - 21es Rencontres francophones du Parallélisme*, 2013.

[14] Luis André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2013.

[15] Albert Greenberg, Srikanth Kandula, David A. Maltz, James R. Hamilton, Changhoon Kim, Parveen Patel, Navendu Jain, Parantap Lahiri, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, pages 51–62, 2009.

[16] Charles E. Leiserson. Fat-trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, 1985.

[17] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, pages 63–74. ACM Press, 2008.

[18] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, page 242, New York, New York, USA, 2011.

[19] Nikolay Grozev and Rajkumar Buyya. Performance Modelling and Simulation of Three-Tier Applications in Cloud and Multi-Cloud Environments. *The Computer Journal*, 2013.

[20] Herodotos Herodotou. Hadoop Performance Models. Technical report, 2011.

[21] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. Using Realistic Simulation for Performance Analysis of MapReduce Setups. In *Large-scale system and application performance*, 2009.

[22] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Conference on Innovative Data Systems Research*, pages 261–272, 2011.

[23] Herodotos Herodotou and Shivnath Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. In *Proceedings of the very large Data Bases Endowment*, volume 4, 2011.

[24] Abhishek Verma, Ludmila Cherkasova, and RH Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In *8th ACM international conference on Autonomic computing*, pages 235–244, 2011.

[25] Gunho Lee. Resource Allocation and Scheduling in Heterogeneous Cloud Environments. Technical report, 2012.

[26] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, number 4, pages 408–409. ACM, 2010.

[27] Quan Chen, Minyi Guo, Qianni Deng, Long Zheng, Song Guo, and Yao Shen. HAT: History-Based Auto-Tuning MapReduce in Heterogeneous Environments. *The Journal of Supercomputing*, 64(3):1038–1054, September 2013.

[28] Archana Ganapathi, Yanpei Chen, Armando Fox, Randy Katz, and David Patterson. Statistics-Driven Workload Modeling for the Cloud. In *IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*, pages 87–92. IEEE, 2010.

[29] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *2009 IEEE 25th International Conference on Data Engineering*, pages 592–603, March 2009.

[30] Francis R. Bach and Michael I. Jordan. Kernel Independent Component Analysis. *The Journal of Machine Learning Research*, 3:1–48, 2003.

[31] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing Cloud Computing Hardware Reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 193–204, New York, New York, USA, 2010.

[32] Dong Seong Kim, Fumio Machida, and Kishor S. Trivedi. Availability Modeling and Analysis of a Virtualized System. In *International Symposium on Dependable Computing*, pages 365–371, November 2009.

[33] Haiyang Qian, Deep Medhi, and Kishor Trivedi. A Hierarchical Model to Evaluate Quality of Experience of Online Services hosted by Cloud Computing. In *International Symposium on Integrated Network Management*, pages 105–112, 2011.

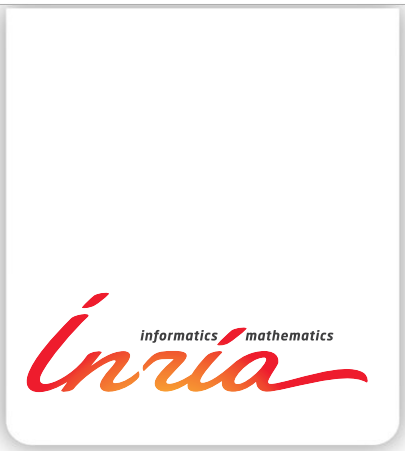
[34] Astrid Undheim, Ameen Chilwan, and Poul Heegaard. Differentiated Availability in Cloud Computing SLAs. pages 129–136, September 2011.

[35] M. Frank and P. Martini. Practical Experiences with a Transport Layer Extension for End-to-End Bandwidth Regulation. In *Local Computer Networks, 1997. Proceedings., 22nd Annual Conference on*, pages 337–346, Nov 1997.

[36] Ian F Akyildiz, Jörg Liebeherr, and Debapriya Sarkar. Bandwidth Regulation of Real-Time Traffic Classes in Internetworks. *Computer Networks and ISDN Systems*, 28(6):855–872, 1996.

[37] Matthias Frank and Peter Martini. Performance Analysis of an End-to-End Bandwidth Regulation Scheme. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1998. Proceedings. Sixth International Symposium on*, pages 133–138. IEEE, 1998.

[38] Julien Bigot and Christian Pérez. *High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives*, volume 20 of *Advances in Parallel Computing*, chapter On High Performance Composition Operators in Component Models, pages 182–201. IOS Press, 2011.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399