Dynamic Scheduling of MapReduce Shuffle under Bandwidth Constraints

Sylvain Gault and Christian Perez

Avalon Research Team
Inria / LIP, ENS Lyon, France
{sylvain.gault,christian.perez}@inria.fr

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 paper proposes and evaluates two algorithms to improve data transfers during the *shuffle* phase under bandwidth constraints.

Keywords: Big Data, MapReduce, shuffle, scheduling, network, contention, bandwidth, regulation.

1 Introduction

In the past decades, the amount of data produced by scientific applications has never stopped growing. Several solutions have been proposed to handle these new order of magnitude in data production. Among them Google proposed to use MapReduce [5] in order to handle the web indexing problems in its own data-centers. This paradigm, inspired by functional programming, distributes the computation on many nodes that can access the whole data through a distributed file system. MapReduce users usually implement an application by only providing a *map* and a *reduce* function.

The global process of a MapReduce application mainly consists in 3 steps, *map*, *shuffle* and *reduce* as shown in Fig. 1. During the *map* phase, every *mapper* process reads a chunk of data and applies the *map* function on every record of that chunk to produce 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 send to a given *reducer* process. Then during the *shuffle* phase, all the pairs

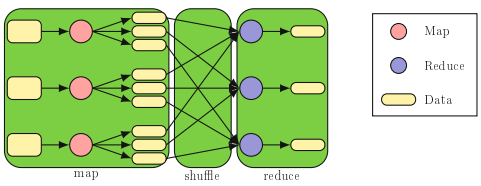


Fig. 1. Typical structure of a MapReduce application

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 117–128, 2014.

with an equal key are gathered into 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* may send some data to every *reducer*.

Most work that try to optimize MapReduce have mainly focused on the *map* phase [4,9,14,13]; the *shuffle* has been largely forgotten despite it might take a non-negligible amount of time.

To optimize the performance / cost ratio, most MapReduce platforms run on moderately high-end commodity hardware. Nowadays, a classical HDD can produce a throughput of more than 1 Gbps, and even more with RAID configurations, while the network remain bound to 1 Gbps. The time taken by the *map* phase is expected to be equivalent to that of the *shuffle*. Some works [11] show 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.

Problem statement. 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 would do the network stack by default) may lead to a suboptimal bandwidth usage due to the mappers that finished later and those with more intermediate data. How to improve the *shuffle* phase? This work proposes and compares several scheduling algorithms to optimize the *shuffle* phase.

This paper is structured as follow: Section 2 reviews some related works that optimize the *shuffle* phase. Section 3 describes the proposed algorithms while a few important implementation details are presented in Sect. 4. Experimental results are discussed in Sect. 5. Section 6 concludes the paper.

2 Related Work

Some 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.

The LEEN [8] (locality-aware and fairness-aware key partitioning) algorithm try to balance the duration of the reduce tasks while minimizing the bandwidth usage during the *shuffle*. 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 ours.

Another complementary approach is the HPMR [10] algorithm. It proposes a *pre-shuffling* phase that leads to reduce the amount of transferred data as well as the overall number of transfers. To achieve this, it tries to predict in which partition the data will 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 [12] 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.

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 [2]. This is the approach followed by Berlińska and Drozdowski [1]. They consider 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 [6] that this approach is hardly scalable and that the chosen communication pattern is clearly suboptimal.

3 Shuffle Optimization

3.1 Platform and Application Models

We consider as a platform model a cluster connected by a single switch, thus forming a star-shaped network. Every link connecting a node to the switch has a capacity of C byte/s and the switch have a bandwidth of σ byte/s. σ is supposed to be an integer multiple of the link bandwidth. Thus $\sigma = l \times C$. Above l concurrent transfers, communications will suffer from contention.

A MapReduce application is represented here by the number of *mapper* processes m, and the number of *reducer* processes r. There are more *reducers* than what the switch can support, meaning that $r \geq l$. A *map* task i will transfer $\alpha_{i,j}$ bytes to the *reduce* task j. We call $\alpha_i = \sum \alpha_{i,j}$ the amount of data a *mapper* process i will have to send. Let $V = \sum \alpha_i$ be the size of all intermediate data.

We also assume that a *mapper* cannot send its data before its computations are finished. A *mapper* i finishes its computation S_i seconds after the first *mapper*. For the sake of simplicity, the *mappers* are numbered by the date of termination of the computation. Thus, $S_i < S_{i+1}$ and $S_1 = 0$. Figure 2 show the Gantt chart of a possible execution of a MapReduce application following this model. The computation time is green (or dark gray), the transfer time is light gray and in red (or black) is the idle time.

As we mainly focus on the throughput, 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.

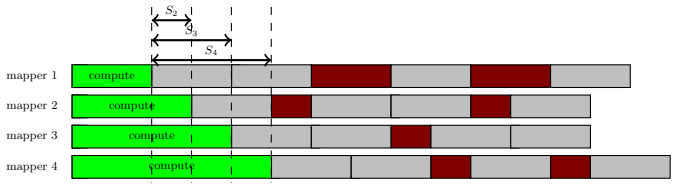


Fig. 2. Gantt of a possible execution following the application model

The transfer time of a chunk is proportional to its data size. This network model also ignores any acknowledgment mechanism of the underlying network protocols that can consume bandwidth and any interaction between CPU usage and bandwidth usage. Therefore, in order to make these assumptions realistic, we choose to map one *mapper* or *reducer* process per physical node.

3.2 Sufficient Conditions of Optimality and Lower Bound

We aim at optimizing 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 input 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 above 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. One way to achieve this, is by making all the transfers end at the exact same time. This is not a necessary condition for an algorithm to be optimal since, in some cases, these requirement cannot be met.

Lower bound. From this sufficient condition, a lower bound of the *shuffle* duration t that will be used in the analysis of the experiments can be computed [7] as

$$t = S_l + \frac{V - C\sum_{i=1}^{l-1}(S_l - S_i)}{\sigma}$$

3.3 Algorithms

To maximize the use of the bandwidth during the *shuffle* phase, we design and evaluate three algorithms. The first one is the simplest and probably the one implemented in every framework. The second algorithm is based on two ordered lists. The thrid algorithm is based on bandwidth regulation. Let describe them.

Start-ASAP Algorithm. As a reference algorithm, we consider the simplest algorithm which consist in starting every transfer as soon as the intermediate data are available. It thus relies on the operating system and on the network equipment to share the bandwidth between multiple transfers.

List-Based Algorithms. This algorithm enforces the constraints there must never be two transfers at the same time from a single *mapper* (1) or toward a single *reducer* (2) as this would create some contentions.

$$start(i,j) \geq end(i,j') \vee end(i,j) \leq end(i,j') \ \forall i \in [1..m], j, j' \in [1..r], j \neq j' \quad (1)$$
$$start(i,j) \geq end(i',j) \vee end(i,j) \leq end(i',j) \ \forall i, i' \in [1..m], j \in [1..r], i \neq i' \quad (2)$$

Algorithm. To fulfill the constraints (1) and (2), the list-based algorithm handles the *mappers* and *reducers* in a list from which a *reducer* is taken and associated to a *mapper* to form a couple that represents a transfer. When at least one *mapper* is ready to start a transfer, the first ready *mapper* is taken from the list of *mappers*. Then, the algorithm iterates through the list of *reducers* to take the first that the current *mapper* has not transferred its data to yet. Once found, the *reducer* and the *mapper* are removed from their lists. When the transfer is done, the *reducer* is put back at the end of the list of reducers thus keeping it ordered by date of last finished transfer. The *mapper* is inserted back into the list keeping the list ordered by the increasing number of remaining transfers. It may happen that for a given *mapper* there is no *reducer* it has not already transferred its data. In that case, the next entry in the list of *mappers* is considered.

More formally, that means that there are two lists named ml and rl. ml is empty in the beginning and it will contain only the id of the *mappers* that have 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* in the beginning, and it is ordered by the fact that the *reducer* id will be always be queued at the end.

The transfer to start is chosen as follow. While no *mapper* $i \in ml$ has been chosen, take the next *mapper* i from ml. Iterate through rl until a *reducer* j is found that i has some data to send to. Once this is found, break every loop, (i, j) is the transfer to start. As a last step, i is removed from ml and j is removed from rl. The full algorithm is described in [7].

Limitations. Although we expect this algorithm to perform better than the reference algorithm, some corner cases may still remain. Indeed, it may happen a *mapper* has some transfers left to do but cannot start them because all the reducers are already busy transferring. We can expect that this situation is more common when l is almost as large as r. Moreover, the scalability of this algorithm seems limited since the centralized scheduler has to be queried for every transfer, and their number grows with the number of nodes.

Two Phase Per-Transfer Regulation. The idea of this algorithm is based on the sufficient condition of an optimal algorithm that uses all the bandwidth of the switch and that terminates all the transfers at the same moment. For this, we assume that for a given data transfer, a given bandwidth can be maintained. We also assume that the bandwidth can be modified dynamically. The way we achieved this is explained in [7].

This algorithm computes the bandwidth to be allocated to every sending *mapper* process with respect to the amount of data to send. This bandwidth is then distributed among the transfers *mapper* $\rightarrow$ *reducer* inside the *mapper* process. This algorithm is expected to never allocate too much bandwidth to a given *mapper* process and to finish all the transfers at the same time.

Model addition. For this algorithm we name $ready(t)$ 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 has to transfer to the reducers at date t. And $\alpha_{i,j}(t)$ the amount of data still to be transferred from *mapper* i to *reducer* j.

Algorithm. The first phase computes the values for $\beta_i(t)$, the bandwidth allocated to *mapper* i. The second phase applies a very similar algorithm for every process in order to distribute the bandwidth among the transfers.

The first phase is done by computing $\beta_i(t) \leftarrow \sigma \frac{\alpha_i(t)}{V(t)}$ for every *mapper* i. If one value for $\beta_i(t)$ is larger than C, then it is set to C, and the remaining bandwidth is redistributed among the other *mappers*. It should be noted that when the bandwidth of a *mapper* 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. The second phase is done by computing $\beta_{i,j} \leftarrow \beta_i(t) \frac{\alpha_{i,j}(t)}{\alpha_i(t)}$ for every transfer (i, j). The full algorithm is presented in [7].

The complexity of one iteration of the first phase is $\mathcal{O}(m)$ because it has to compute $\beta_i(t)$ for every *mapper* i. This computation could be repeated a maximum of m times. Thus this first phase run in $\mathcal{O}(m^2)$ in the worst case. The complexity of the second phase is $\mathcal{O}(m \times r)$ because $\beta_{i,j}(t)$ has to be computed for every *mapper* i and every *reducer* j. The complexity of the algorithm is then $\mathcal{O}(m^2 + m \times r)$. This may sound large, but, the second phase can be distributed and every *mapper* i can share its allocated bandwidth itself and it can compute $\beta_{i,j}(t)$ from $\beta_i(t)$ 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. However, since $r \geq l$, this case should not occur very often.

4 Implementation Details

To evaluate these algorithms, we have implemented them in our own MapReduce framework HoMR (HOmemade MapReduce) which has been developed in the context of the French ANR MapReduce project. It is based on HLCM/L^{2}C [3], a software component model. HoMR is written in C++ and it relies on CORBA for inter-process communications.

The two phase algorithm recomputes the allocated bandwidths every time a transfer terminates, or every 5 seconds if nothing happened. This enables to avoid a slight imprecision in the bandwidth control to be compensated. The value of 5 seconds has been chosen arbitrarily. However, experiments have shown that the computed bandwidth for every *mapper* is always slightly different.

To reach the maximal bandwidth of a single link, every *mapper* uses 4 threads to send data per transfer. This value has been suggested by the experiment with `tc` presented in Sect. 5.2. It has been further confirmed by tests with HoMR.

5 Experiments

To evaluate these algorithms, we have performed some experiments on the Grid'5000 experimental testbed. First we ensure the environment behaves as expected, and we compare the 3 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. The central point of our algorithms is to control the bandwidth used during the *shuffle* phase. Thus, to evaluate our algorithms in the case of several switch bandwidth limit configurations, we simulated a switch by the mean of a node dedicated to routing packets.

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

As all the packets will have to traverse twice the network interface of the router node, we need a fast network to simulate a switch with a throughput greater than 1 Gbps. Thus, we used *InfiniBand 40G* interfaces 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 of this setup is tested in Sect. 5.2. As the network is based on fast network interface controllers (NIC), the bandwidth of the private links is configured to 1 Gbps. The `tc` rules used are based on the `htb` algorithm 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.

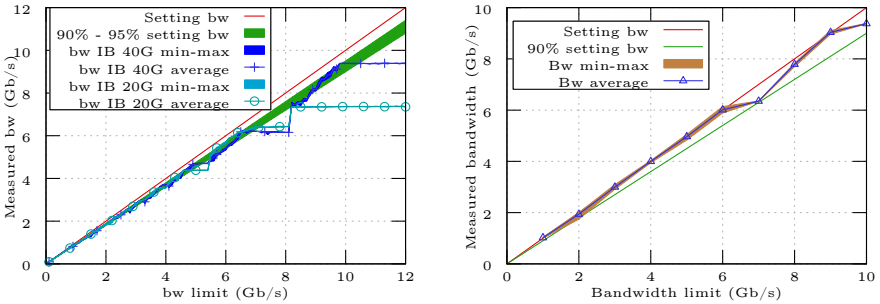
Experiments have been done on the Edel cluster of Grid'5000. Each node has 2 quad-core CPUs Intel Xeon E5520 @2.27 GHz, and it equipped with 24 GB of memory, 1 Gigabit Ethernet and 1 InfiniBand 40G cards.

5.2 Preliminary Tests

`tc` Regulation on Router. To test whether we can limit correctly the bandwidth on the router we used 2 nodes and a router node. Only the router node has a limited bandwidth. The limit is varied by steps of 100 Mbps up to 12 Gbps. The actual bandwidth is measured with `iperf` with 4 parallel clients threads on the client side. Every measure is run 5 times.

Figure 3(a) shows the results of this experiment. Globally, we observe that the measured bandwidth stay roughly between 90% and 95% of the target bandwidth until the maximal bandwidth of the system is reached. Also, the measures are quite stable as the difference between the maximal and minimal measured bandwidth never exceed 0.28 Gbps or 8% of the average bandwidth.

However, some steps are clearly distinguishable around 5 Gbps and 7 Gbps. Similar results have been obtained on another cluster with *InfiniBand 20G* network adapters. We have no real explanation for that. The results on a newer



(a) Without correction (b) With correction

Fig. 3. Bandwidth limitation on Linux + `tc` on InfiniBand

version of Linux (not shown here) are completely different. Thus, we think that this is a performance bug in Linux.

To still get the targeted bandwidth, we built a lookup table to set the bandwidth limit that would produce the bandwidth actually targeted. Figure 3(b) show the measured bandwidth with respect to the corrected setting bandwidth. It can be noted that the larger step from Fig. 3(a) could not be completely corrected and that the bandwidth of 10 Gbps cannot be reached. Except from those outliers, the measured bandwidth is always between 96% and 103% of the target bandwidth. However, the lookup table has been tested only when one node emit the data and one node receive them. It is not impossible that when several nodes send data at the same time the bandwidth drift shown in Fig. 3(a) is not the same. This would make the correction applied inaccurate.

Bandwidth Regulation. The two phase algorithm relies on the ability to regulate the bandwidth. The method we used for this is described in [7]. To check whether it performs correctly, we set up an experiment with only two nodes interconnected by an *InfiniBand 40G* network. We then vary the size of the messages from 4 bytes to 64 MB and the target bandwidth from 1 KB/s to 1 GB/s and measure the overall average bandwidth. Each measure is repeated 10 times.

Figure 4 shows a 3D plot of the results this experiment. It shows the actual bandwidth with respect to the message size and to the targeted bandwidth. Some points are missing in the results because the execution time required for them would have been too long. The colors (or gray scale) represent the percentage of variability.

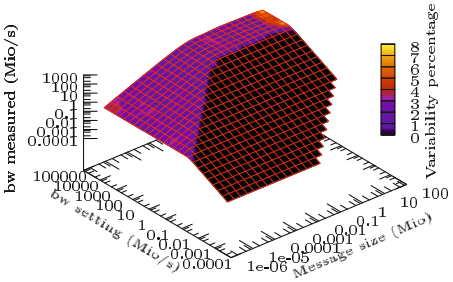


Fig. 4. Bandwidth regulation test

This figure shows 3 distinct areas. A black plan for small targeted bandwidth and large enough message size, a purple (or gray) plan for small message size and high targeted bandwidth, and a *roof* for large message size and large targeted bandwidth. The black plan is the area where the regulation algorithm works perfectly. The purple plan is the area where the system is CPU-bound. And the *roof* is the area where the system is bound by the network bandwidth.

5.3 Synchronous Transfer Start

The first experiment with our algorithms is simple and all other experiments are only variations of this one. The MapReduce job is a word count application. However, for the sake of simplicity and control, the data are not read from a file, they are generated by a component *WordGenerator*. This enables 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. In the next experiment, an imbalance is simulated by adding a sleep after this barrier. This enables to evaluate 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 varied from 1 Gb/s to 10 Gb/s. The time taken from the start of the first transfer to the end of the last transfer is measured and compared to the lower bound. This experiment is run with 10 *mappers* and 10 *reducers*. Every configuration is run 5 times.

Figure 5 displays the results of this experiment in terms of percentage with respect to the lower bound. As every measure has been made 5 times, the median time is represented on this figure.

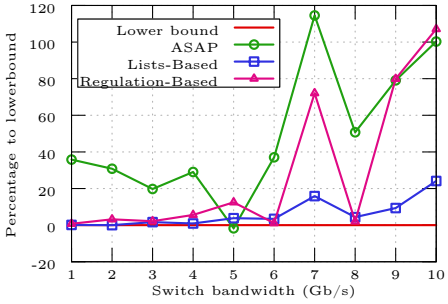


Fig. 5. Median time taken by all the 3 algorithms under various bandwidth restriction with same amount of data and synchronous start of transfers

The results for the list-based algorithm are close to what was expected. The lists-based algorithm has a behavior close to the optimal. Performance degradations occur for a switch bandwidth of 7 Gb/s and 10 Gb/s. Those configurations, as shown in Fig. 3(b), are known not to offer the actual bandwidth targeted.

The two phase regulation algorithm shows a good behavior for a switch bandwidth less or equal to 8 Gb/s. Above that limit it creates contentions and it 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 overestimation of the switch bandwidth.

Regarding the variability of the performance, the list-based algorithm has a good stability. The standard deviation of the time of the shuffle phase is no more than 3%. The two other algorithms have a variability between 0% and 17%.

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 6 displays the results of this experiments in terms of percentage with respect to the lower bound.

The list-based algorithm has a similar behavior as in the previous experiment. The reference algorithm also has 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 performance degradation. The global behavior of the list-based and two phase algorithms remain the same. However the two phase algorithm appears to be super-optimal by up to 5% for some configurations. The cause is not very clear. It is supposed that raising the limit bandwidth as done in Fig. 3(b) is only valid for a single source and single destination network.

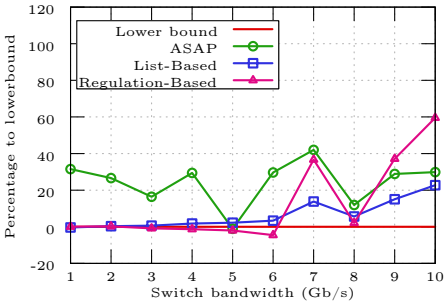


Fig. 6. Median time taken by all the 3 algorithms under various bandwidth restriction with same amount of data and 1 second step between transfer start

5.5 1 Second Step between Transfer Start with Heterogeneous Amount of Data

The third and final experiment combines the delay before starting the transfers and imbalance of the amount of intermediate data.

The results of this experiment are displayed in Fig. 7 in terms of percentage with respect to the lower bound.

The results shows that the list-based algorithm has a more regular behavior. It produces a performance variability of less than 2%. As previously, the reference algorithm has poor performance for small bandwidth as it generates some contentions. It also has a variability

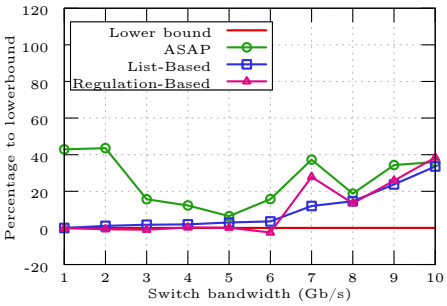


Fig. 7. Median time taken by the 3 algorithms under various bandwidth restriction, with an imbalanced amount of intermediate data and non-synchronous transfer start

between 2% and 14%. Its performance is equivalent to that of two phase and lists-based algorithms for a router bandwidth which is large enough. Globally, the reference algorithm has 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; 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 has only a slight impact in the actual performance. Except for a bandwidth limit of 7 Gbps, the two phase algorithm achieves performance close or better than the list-based algorithm. It also has a variability usually less than 3%, except for a bandwidth limit of 7 and 10 Gbps where some contentions occur.

6 Conclusion and Future Work

The two phase algorithm performs equally or better than the list-based one when the switch bandwidth is precisely known, but it is quite sensitive to an over-estimation of this parameter. The impact of contention on the *reducers*' side has been ignored here and should be investigated. Taking the bandwidth regulation from the *reducers*'s side may be beneficial. The list-based may hardly scale and it requires the switch bandwidth to be an integer multiple of the link bandwidth.

Several assumptions have been made in this work, and removing them would be a step forward. On the experiment part, the most important assumption is that Linux + `tc` mimic the behavior of a real switch, which could be investigated further.

On the model part, some simplifications have been made which may not always be realistic. This is, however, a crucial step towards a more general solution. Forcing the *mappers* and *reducers* to be located on distinct nodes each is a strong requirement that could be addressed by allowing the list-based algorithm to pick only one of the collocated process, and by adding an intermediate phase to the two phase algorithm that would distribute the bandwidth allocated to a node among the processes.

Assuming a single-switch network is also a strong constraint. However, it could be mitigated by discriminating the same-switch transfers from the others and tweaking the algorithm parameters accordingly.

Automatically determining the parameters of the algorithms, such as link and switch bandwidths, would remove this burden from the user. Making the estimated bandwidth dynamic would also make those algorithms more robust against the disturbances that may be caused by the other users of a cloud infrastructure. Also, network topology discovery would be very important to ease their usage.

In the end, the *shuffle* phase has been largely ignored by the academic work despite being a potentially important bottleneck. This has showed that a no-op algorithm lead to bad performance in case of network contention and smarter algorithms are proven to be more efficient.

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. Berlinska, J., Drozdowski, M.: Scheduling Divisible MapReduce Computations. Journal of Parallel and Distributed Computing 71(3), 450–459 (2010)
2. Bharadwaj, V., Robertazzi, T.G., Ghose, D.: Scheduling Divisible Loads in Parallel and Distributed Systems. IEEE Computer Society Press (1996)
3. Bigot, J., Pérez, C.: On High Performance Composition Operators in Component Models. In: High Performance Scientific Computing with special emphasis on Current Capabilities and Future Perspectives, Advances in Parallel Computing, vol. 20, pp. 182–201. IOS Press (2011)
4. Chen, Q., Zhang, D., Guo, M., Deng, Q., Guo, S.: Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In: 2010 IEEE 10th Int. Conf. on. Computer and Information Technology (CIT), pp. 2736–2743 (June 2010)
5. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Proc. of the 6th Conf. on Symposium on Opearting Systems Design & Implementation, OSDI 2004, vol. 6, p. 10. USENIX Association (2004)
6. Gault, S.: Ordonnancement dynamique des transferts dans MapReduce sous contrainte de bande passante. In: ComPAS 2013 / RenPar'21 - 21eme Rencontres Francophones du Parallélisme (January 2013)
7. Gault, S., Desprez, F.: Dynamic Scheduling of MapReduce Shuffle Under Bandwidth Constraints. Tech. Rep. Inria/RR-8574
8. Ibrahim, S., Jin, H., Lu, L., Wu, S., He, B., Qi, L.: LEEN: Locality/fairness-aware key partitioning for mapreduce in the cloud. In: 2010 IEEE Second Int. Conf. on Cloud Computing Technology and Science (CloudCom), pp. 17–24 (November 2010)
9. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.: Skewtune: Mitigating skew in mapreduce applications. In: Proc. of the 2012 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD 2012, pp. 25–36. ACM (2012)
10. Seo, S., Jang, I., Woo, K., Kim, I., Kim, J.S., Maeng, S.: HPMR: Prefetching and pre-shuffling in shared mapreduce computation environment. In: IEEE Int. Conf. on Cluster Computing and Workshops, CLUSTER 2009, pp. 1–8 (August 2009)
11. Steffenel, L.: Modeling network contention effects on all-to-all operations. In: 2006 IEEE Int. Conf. on Cluster Computing, pp. 1–10 (September 2006)
12. Su, Y.L., Chen, P.C., Chang, J.B., Shieh, C.K.: Variable-sized map and locality-aware reduce on public-resource grids. Future Generation Computer Systems 27(6), 843–849 (2011)
13. Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Job scheduling for multi-user mapreduce clusters. Tech. Rep. UCB/EECS-2009-55, EECS Department, University of California, Berkeley (April 2009)
14. Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In: Proc. of the 5th European Conf. on Computer Systems, EuroSys 2010, pp. 265–278. ACM (2010)