Byzantine Fault-Tolerant MapReduce: Faults Are Not Just Crashes

Pedro Costa[1], Marcelo Pasin[1], Alysson N. Bessani[1], Miguel Correia[2]
[1]Universidade de Lisboa, Faculdade de Ciências, LASIGE – Lisboa, Portugal
[2]Instituto Superior Técnico, Technical University of Lisbon, INESC-ID – Lisboa, Portugal

Abstract

MapReduce is often used to run critical jobs such as scientific data analysis. However, evidence in the literature shows that arbitrary faults do occur and can probably corrupt the results of MapReduce jobs. MapReduce runtimes like Hadoop tolerate crash faults, but not arbitrary or Byzantine faults. We present a MapReduce algorithm and prototype that tolerate these faults. An experimental evaluation shows that the execution of a job with our algorithms uses twice the resources of the original Hadoop, instead of the 3 or 4 times more that would be achieved with the direct application of common Byzantine fault-tolerance paradigms. We believe this cost is acceptable for critical applications that require that level of fault tolerance.

1. Introduction

MapReduce is a framework developed by Google for processing large data sets [8]. It is composed by a programming model and a runtime system, being used extensively by Google in its datacenters to support core business functions such as index processing for its web search engine. Google's implementation is not openly available, but an open source version called Hadoop[1] [34] is used by many cloud computing companies, including Amazon, EBay, Facebook, IBM, LinkedIn, RackSpace, Twitter, and Yahoo!.[2] MapReduce is also a promising approach for scientific computing [10, 15]. A final argument in favor of the importance of MapReduce is the appearance of commercial versions like Windows Azure's MapReduce and Amazon Elastic MapReduce.

MapReduce was designed to be fault-tolerant because at scales of thousands of computers and hundreds of other devices like network switches, routers and power units, component failures are frequent. For instance, Dean reported that in the first year of a cluster at Google there were 1000 individual machine failures and thousands of hard drive failures [7]. Google and Hadoop MapReduce mostly tolerate crashes of map and reduce tasks. If one of these tasks stops before its conclusion, this is detected and a new instance of the same task is created. Additionally, data is stored in disk together with checksums, which allow its corruption to be detected [14, 23, 34].

Although it is crucial to tolerate crashes of tasks and data corruptions in disk, other faults that can affect the *correctness of the results* of MapReduce are known to happen and will probably happen more often in the future [31]. A recent study of DRAM errors in a large number of servers in Google datacenters for 2.5 years concluded that these errors are more prevalent than previously believed, with more than 8% DIMMs affected by errors yearly, even if protected by error correcting codes (ECC) [32]. A Microsoft study of 1 million consumer PCs shown that CPU and core chipset faults are also frequent [25].

The fault tolerance mechanisms of current MapReduce implementations, namely Hadoop, cannot deal with such *accidental arbitrary faults* or *accidental Byzantine faults* [1] (we do not consider malicious faults). They cannot be detected using checksums and often do not crash the task they affect, so they can silently corrupt the result of a task. They have to be detected and their effects masked by *executing each task more than once*. This basic idea was proposed in the context of volunteer computing to tolerate malicious volunteers, that return false results of the tasks they were supposed to execute [29]. That work, however, considered bag-of-tasks applications, which are simpler than MapReduce jobs. A similar but more generic solution is Byzantine fault-tolerant state machine approach, in which a set of programs are executed in parallel by different servers that execute commands in the same order [30, 4, 5, 33]. This approach, however, is not directly applicable to the replication of MapReduce tasks, only of a service that follows the client-server model (e.g., a file server). A naive solution for MapReduce would be to execute each job twice and re-execute it if the results do not match, but its cost is excessive in case there is a fault.

In this paper we present a Byzantine fault-tolerant (BFT) MapReduce runtime system. This system tolerates faults that corrupt the results of computation of tasks, such as the above-mentioned cases of DRAM and CPU errors/faults. Our BFT MapReduce follows the approach of executing each task more than once, similarly to the works mentioned above. The chal-

[1]Hadoop is an Apache open source project with many components. We use the term Hadoop to mean its MapReduce runtime system.

[2]http://wiki.apache.org/hadoop/PoweredBy

lenge was to do this *efficiently*, e.g., by running only 2 copies of each task when there are no faults. Notice that, for instance, the state machine approach requires $3f + 1$ replicas to tolerate at most f faulty replicas, which gives a minimum of 4 copies of each task [4, 5]. We use several mechanisms to minimize both the number of copies of tasks executed and the time needed to executing them. In case there is a fault, the cost of our solution is close to the cost of executing the job twice, instead of 3 times as the naive solution proposed above; if there is more than one fault, the gap is even larger.

Our solution is more expensive than using the original MapReduce runtime or Hadoop. A typical configuration will require that each task is executed twice, which is a considerable overhead in terms of resources used and possibly of execution time. However, we believe this cost is acceptable for critical applications that require a high degree of certainty of the correctness of the results obtained. A large set of scientific computing applications will fall in this category [10, 15].

The main contribution of the paper is an algorithm to execute MapReduce jobs tolerating arbitrary faults. We implemented our BFT MapReduce by modifying Hadoop and measured its performance using the Hadoop's Gridmix benchmark.[3] These experiments confirmed that indeed it is possible to run a Byzantine fault-tolerant Hadoop using twice the resources of Hadoop.

2. MapReduce and Hadoop

MapReduce comprises a programming model based on map and reduce functions as found in functional programming (with a somewhat modified meaning), and an execution environment using a large number of computers as found in clusters and datacenters. Programmers specify map and reduce functions: the former is used to process an input file and generate key-value pairs, the latter is used to merge several such pairs (with the same key) into a single key-value pair. The running environment first splits the input file, then feeds several instances of the map function with those splits. The multiple map outputs are then sorted key-wise and split again, now fed to multiple reduce functions, in a phase known as shuffle. Multiple reduce outputs are finally concatenated in an output file. According to Dean and Ghemawat, it is possible to express many real world tasks using this model [8].

Hadoop is an implementation of MapReduce made from scratch, freely available through the Apache license [34]. It is not only a framework to implement and run MapReduce algorithms in Java, but also a handy tool for developing alternative, improved, systems for MapReduce, such as the one presented in this paper.

Hadoop users submit jobs to it, containing the implemented map and reduce functions and an input file reference. The input file must have been previously stored in the Hadoop file system (HDFS), which breaks the file in smaller replicated pieces, called splits. The splits are homogeneously stored in the same nodes available for running Hadoop jobs.

HDFS is a file system tailored for Hadoop. It manages a file namespace and allows user data to be stored in files split into multiple, distributed blocks. It replicates data blocks of files on multiple hosts, default is three times: two in the same rack and one in another rack. Although it does not implement the POSIX semantics, its performance is tuned for data throughput and large files (blocks in the range of 64 MB). HDFS is implemented using a single NameNode, the master server that manages the file namespace operations (open, close, rename) and regulates access to files by clients. In addition, there are a number of DataNode slave servers, usually one per node in the cluster, which manage storage attached to the nodes that they run on, and serve block operations (create, read, write, remove, replicate). DataNodes communicate to move blocks around, for load balancing and to keep the replication level on failures.

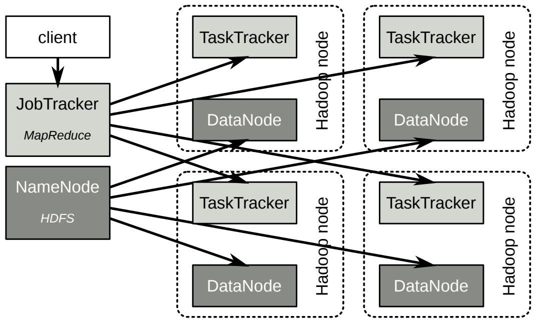

FIGURE 1: Hadoop architecture

The architecture of Hadoop is presented in Figure 1. Jobs are submitted to and taken care of by a service called JobTracker. The JobTracker, currently implemented as a centralized service, breaks a job into pieces, called tasks. It creates one map task per input split, and a predefined number of reduce tasks. Tasks are assigned to nodes based on the task queue size within the nodes. Also, when assigning a map task to a node, preference is given to nodes containing the input split, then to the nodes in the same rack, then those in the same cluster, then datacenter, and so on.

Each node available to run Hadoop jobs hosts a service, called TaskTracker. This service runs the tasks and sends heartbeat messages to the JobTracker. Heartbeats carry information on the percentage of the input split already processed by local tasks (if any is running), and signal task ends and error conditions. Heartbeat replies may carry new tasks to run in the node.

Hadoop has been built with some level of faults tolerance. Heartbeat messages, or the lack of them, allow the JobTracker to figure out that a task stalled or failed. Using different nodes, the JobTracker runs extra, speculative, tasks for those lagging behind and restarts the failed ones. Nevertheless, this model only supports crashes, not arbitrary faults. In this pa-

[3]http://hadoop.apache.org/mapreduce/docs/current/gridmix.html

per we develop and describe the implementation of a more elaborated algorithm that provides Byzantine fault-tolerance.

3. Byzantine Fault-Tolerant MapReduce

This section present the BFT Hadoop MapReduce system. We start by presenting the system model, then the algorithm and the techniques we use.

System model. The system is composed by a set of distributed *processes*: the *clients* that request the execution of jobs composed by map and reduce tasks, the *JobTracker* that manages the execution of a job as explained, a set of *TaskTrackers* that execute tasks, the *NameNode* that manages access to data stored in HDFS, and a set of *DataNodes* where HDFS stores file blocks. We say that a process is *correct* if it follows the algorithm, otherwise we say it is *faulty*. We also use these two words to denominate a task (map or reduce) that, respectively, returns the result that corresponds to an execution in a correct TaskTracker (correct) or not (faulty). Processes run in *servers* in a datacenter.

We assume that clients are always correct, because if they are not there is no point in worrying about the correctness of the job's output. We also assume that the JobTracker is always correct, which is the same assumption that Hadoop does [34]. It would be possible to remove this assumption by replicating the JobTracker, but it would complicate the design considerably and this component does much less work than the TaskTrackers. The TaskTrackers can be correct or faulty, so they can arbitrarily deviate from the algorithm and return corrupted results of the tasks they execute. We assume that HDFS (NameNode and DataNodes) is Byzantine fault-tolerant, so it is always correct. We do not study this aspect because there is already one BFT HDFS implementation in the literature, based on the UpRight library [5].

The system is asynchronous, i.e., we make no assumptions about bounds on processing and communication delays for our mechanisms, although the original Hadoop mechanisms make assumptions about such times for termination. We assume that the processes are connected by reliable channels, so no messages are lost, duplicated or corrupted. In practice this is provided by TCP/IP connections. We assume the existence of a hash function that is collisions-resistant, i.e., for which it is infeasible to find two inputs that produce the same output (e.g., SHA-1).

Our algorithm is parameterized with f. Consider that n replicas of task T are executed with the same input I. f is the maximum number of faults that can affect the replicas of T in such a way that the result of the faulty tasks is the same. In other words, if T is a function and $\{T_1, T_2, ..., T_n\}$ the faulty replicas executed, f is the maximum value that n takes in such a way that $T_1(I) = T_2(I) = ... = T_n(I)$. Notice that the meaning of f is different from the meaning of f or t used typically in fault tolerance, which is the maximum number of faulty replicas [4, 5, 33, 20, 22, 3]. It is much more probable that n replicas fail, than that n replicas fail without crashing and return the same result. Our parameter f, like the other f or t, has a probabilistic meaning (hard to quantify precisely): it means that the probability of more than f faulty replicas of the same task returning the same output is negligible. We expect $f = 1$ to be a realistic value.

The algorithm. A simplistic solution to make MapReduce Byzantine fault-tolerant given the system model would be the following. First, the JobTracker starts $2f + 1$ replicas of each map task in different servers and TaskTrackers. Second, the JobTracker starts also $2f + 1$ replicas of each reduce task. Each reduce task fetches the output from all map replicas, picks the most voted results, processes them and stores its output in HDFS. In the end, either the client or a special task must make the vote of the outputs to pick the most voted. An even more simplistic solution would be to run a consensus, or Byzantine agreement between each set of map task replicas and reduce task replicas. This would involve even more replicas (typically $3f + 1$) and more messages exchanged.

The first simplistic solution is very expensive because it replicates everything $2f + 1$ times: task execution, map task inputs reading, communication of map task outputs, and storage of reduce task outputs. We use a set of techniques to avoid these costs:

Deferred execution. Crash faults are detected by the previously existing Hadoop mechanisms, and arbitrary faults are uncommon, so there is no point in always executing $2f + 1$ replicas to usually obtain the same result. The JobTracker starts only $f + 1$ replicas of the same task, and the reduce tasks check if they all return the same result. If a timeout elapses, or some returned results do not match, more replicas (up to f) are started, until there are $f + 1$ matching replies.

Tentative reduce execution. Waiting for $f + 1$ matching map results before starting a reduce task can put a burden on end-to-end latency for the job completion. A better way to deal with the problem is for the JobTracker to start executing the reduce tasks just after receiving the first copies of the required map outputs, and then, while the reduce is still running, validate the input used as the map replicas outputs are produced. If at some point it is detected that the input used is not correct, the reduce task can be restarted with the correct input. This point will be implemented in the future to improve the performance of the application.

Digest outputs. $f + 1$ matching outputs of maps or reduces have to be received to be considered correct. These outputs tend to be large, so it is useful to fetch one output from some task replica and compare just digests (hashes). This way, it is still possible to validate the output without generating much additional network traffic.

Tight storage replication. We can write the output of both map and reduce tasks to HDFS with a replication factor of f (typically 1), instead of 3 (the default value). We are already

Algorithm 1 BFT MapReduce algorithm

```
1: client stores input data in HDFS and submits the job to JobTracker;
2: JobTracker inserts $f + 1$ replicas of each map task in the task queue;
3: JobTracker assigns each map tasks to one TaskTracker that has one execution slot free and that, preferably, is located in the same server as the DataNode that contain the input for that task;
4: when a map task starts, it reads the input from a DataNode;
5: during the execution of a map task, every TaskTracker periodically sends heartbeat messages to the JobTracker; if a TaskTracker stops sending heartbeats for a certain task, the JobTracker inserts the task again in the queue; if a TaskTracker detects that a task stopped, it sends a message to the JobTracker that does the same;
6: when a map task finishes, the TaskTracker sends a heartbeat with the digest of the result to the JobTracker; if the JobTracker has digests of $f + 1$ or more replicas of a map task and there are no $f + 1$ that match, then it starts another replica of the same map;
7: when a certain percentage of the map tasks are concluded (i.e., have $f + 1$ matching digests), the JobTracker inserts $f + 1$ replicas of each reduce task in the queue and gives it the digests of the map outputs;
8: when a reduce task starts, it gets its input from the map tasks that generated it; for each map, it gets the complete output from one of the TaskTrackers that executed it and provided a digest that matches;
9: during the execution of a reduce task, TaskTracker and JobTracker do the same heartbeat management as done for a map task;
10: when a reduce task finishes, it stores the output and a hash in HDFS (without replication) and sends a heartbeat with the digest of the result to the JobTracker;
11: when all reduces are concluded (i.e., have $f + 1$ matching digests), the JobTracker informs the client;
12: client picks a copy of the output that matches the digest.
```

replicating the tasks, and their outputs will be written on different locations, so we do not need to replicate these outputs even more.

The algorithm is presented in Algorithm 1. In the normal case, Byzantine faults do not occur, so, these mechanisms greatly reduce the overhead introduced by the basic scheme. Specifically, without Byzantine faults, only $f + 1$ replicas are executed in task trackers, the latency is similar to the one without replication, the overhead in terms of communication is small, and the storage overhead is minimal. Similarly to the original Hadoop, our BFT version can run *speculative* tasks for those lagging behind.

Notice that our algorithm tolerates any number of arbitrary faults during the execution of a job, because map and reduce tasks can be re-executed until $f + 1$ outputs match. The limit f is only on the number of faulty replicas of a task that return the same output.

4. The Prototype

The prototype of our system was implemented by modifying the original Hadoop 0.20.0 source code. Hadoop is written in Java so we describe the modifications made per class. The modifications essentially implement Algorithm 1. HDFS was not modified for the reason already mentioned: there is already a BFT HDFS so we did not study this issue.

Most modifications were made in the `JobTracker` component and class. For a given job, this component stores in a queue one object per map and reduce task, from which they are removed and scheduled to be executed. We modified it to store in the queue $2f + 1$ replicas of each task (which slightly differs from how the algorithm is presented in the previous section), but we modified the scheduler in such a way that only $f + 1$ replicas of each task are executed initially (deferred execution). The format of the identifier of tasks (maps and reduces) was modified to include a replica number, so that they can be differentiated. A map task takes as input the path to a split of the input file. The JobTracker gives each map replica a path to a different replica of the split, stored in a different DataNode, whenever possible (i.e., as much as there are enough replicas of the split available). It tries to instantiate map tasks in the same server where the DataNode of the split is, so this usage of different split replicas forces the replicas of a map to be executed in different TaskTrackers, which improves fault tolerance.

The JobTracker stores information of a running job in a object of the `JobInProgress` class. TaskTrackers send heartbeat messages to the JobTracker periodically. We modified this process to include a digest (SHA-1) of the result in the heartbeat that signals the conclusion of a task (map or reduce). This digest is saved in the `JobInProgress` object, more precisely in an object of the class `VotingSystem`. When there are digests from the $f + 1$ replicas of a task, the JobTracker decides if they match or if it is necessary to launch another replica. If it is, it schedules for execution one of the replicas already in the queue (there were f extra); in the very unlikely case of the $2f + 1$ replicas are used and no matching result is obtained, one more replica is both inserted in the queue and scheduled. The `Heartbeat` class used to represent a heartbeat message, was modified to include the digest and task replica identifier.

5. Experimental Evaluation

In this section, we assess the performance of our BFT MapReduce in a cluster. Hadoop provides a benchmark application called Gridmix that is composed of a set of jobs: monsterquery, webdatascan, webdatasort, combiner, streamingsort, and javasort. We used it to evaluate our prototype, mostly comparing it to the non-modified Hadoop (denominated the *official* Hadoop MapReduce from now on). We run our experiments in Grid'5000, a French geographically distributed infrastructure used to study large-scale parallel and distributed systems.

The set of experiments was split in half and conducted in set of 10 machines from 2 different Grid'5000 sites, for the single purpose of reducing the time taken to run them. All executions of the same job were done in the same site, and each value shown in the graph is an average of 5 executions. We decided that the chosen machines to perform the tests had the same characteristics. The experiments ran for many hours and Grid'5000 reservations are limited in terms of $number\ of\ nodes \times duration$. Being the infrastructure shared, it is unlikely to get identical nodes when reserving them in large quantities. As a result, we limited the tests to the chosen set. We run the monsterquery, streamingsort and javasort experiments in the Paramount cluster whose machines are Dell PowerEdge 1950 (Intel Xeon 5148 LV 2.33Ghz 4 cores in 2 CPUs, 8GB RAM, Gigabit Ethernet, 2x300GB SATA). Webdatascan, webdatasort and combiner were executed in Chicon cluster whose machines are IBM eServer 326m (AMD Opteron 285 2.6GHz, 4 cores in 2 CPUs, 4GB RAM, Gigabit Ethernet, 80GB SATA).

In the experiments we considered only the case of $f = 1$. Recall the meaning of f in our system and that the probability of the corresponding assumption being violated is even lower than with BFT replication algorithms. Nevertheless, even for these algorithms usually only the case of $f = 1$ is evaluated [4, 5, 33]. The values we present are averages of 5 executions of each experiment. Each job was executed with different numbers of input splits, from 10 to 1000. In all our experiments, an input split consists of 64 MB of data that is stored in DataNodes. We used at most 1000 splits, so we generated 64 GB of data. The reason why we used 64 MB of data is that it is the default size of a block in HDFS. We did not use multiples of 64 MB because the data size would be very large and the experiments would take longer than we desired.

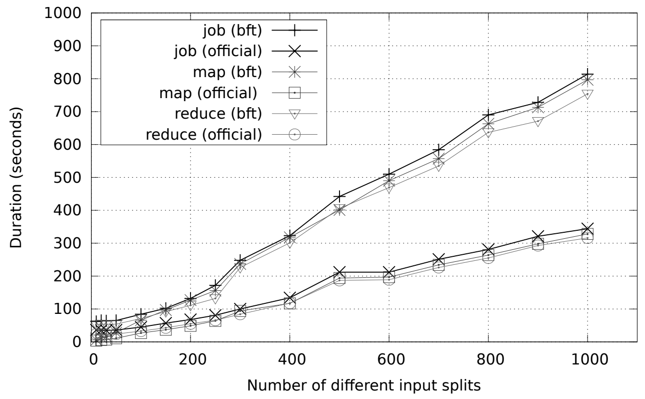

FIGURE 2: Webdatascan execution duration with different number of input splits

We start by comparing the duration of the execution of both versions. Figure 2 shows the duration of the executions with the different numbers of input splits for the webdatascan job, and the duration of map and reduce tasks specifically. This figure is similar to those of the other jobs, which we do not show for lack of space. Instead, Figure 3 shows the ratio of the durations of the BFT and the official versions for all jobs. Both figures show that the BFT version is on average two times slower than the official version. This is what was expected and is desirable, because the BFT version executes $f + 1 = 2$ times more map and reduce tasks, and the number of machines is much lower than the number of tasks. Without deferred execution and the same number of servers, the duration would be $2f + 1 = 3$ times. The figures show some oscillations in relation to the average of 2 times, due to effects like the execution of speculative tasks and variations in the duration of reduce tasks.

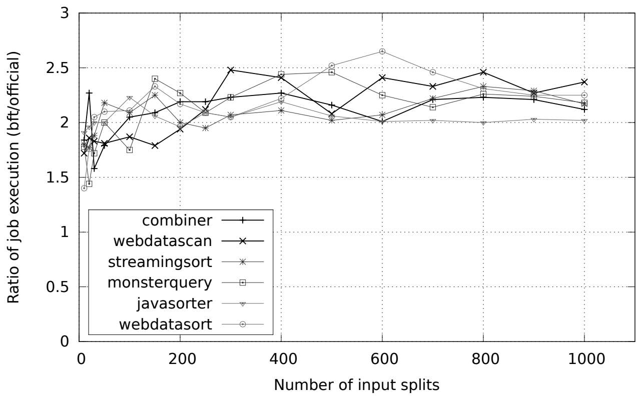

FIGURE 3: Ratios of job execution duration

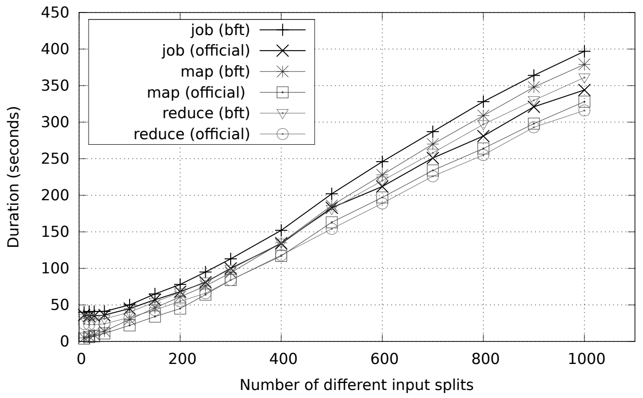

FIGURE 4: Webdatascan execution duration with the official version run in 10 nodes and the BFT version in 20 nodes

To show that the double of the duration corresponds roughly to twice the consumption of resources, we made a experiment with a setting that is an exception with respect to all others: we executed the BFT version in 20 nodes, while the official version was again executed in 10 nodes. The results of this experiment for the webdatascan job is shown in Figure 4. With this difference in the number of nodes, the duration of the BFT version is only slightly higher than the official version.

To better understand the factors that impact the duration of the execution of the two MapReduce versions we provide some additional graphics. Each map task reads 64 MB of data from HDFS (one input split). Figure 5 shows the total data read by map tasks in the combiner job. Figure 6 shows the ratio of bytes read between the BFT and the official version for all jobs. All these values were extracted from the actual execution logs, not computed by us. The main conclusion is that the fact that BFT version runs twice more map tasks, also makes it read twice more data from HDFS, with the corresponding cost in terms of performance.

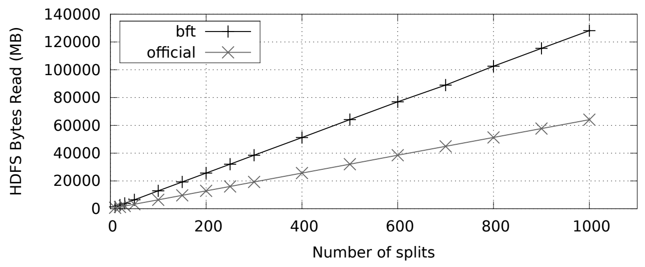

FIGURE 5: Data read from HDFS with the combiner job

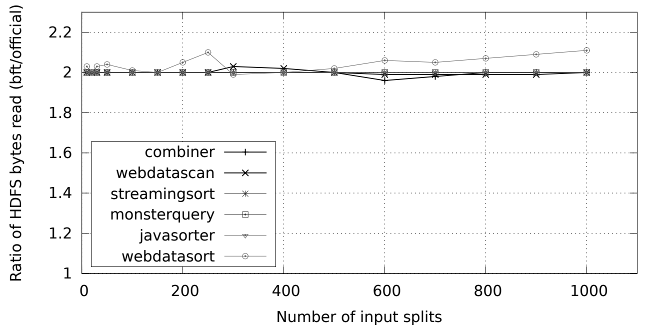

FIGURE 6: Ratio of data read from HDFS

We did a similar analysis for the amount of output data generated by map tasks. Figure 7 depicts the ratios between the BFT and the official version for all jobs. Again the BFT version runs twice more tasks, so it produces twice more data. In some cases more than twice of the data was produced, which shows that speculative map tasks were executed.

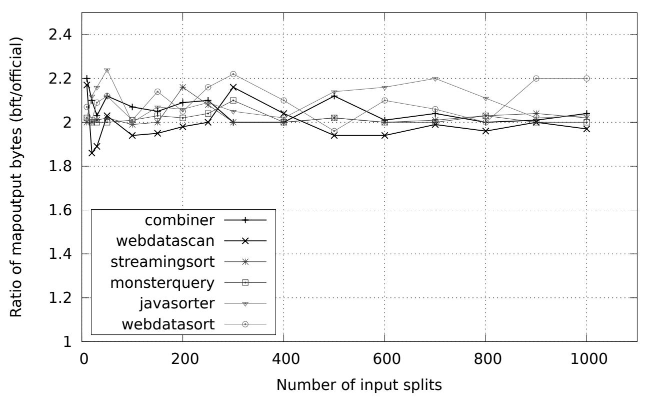

FIGURE 7: Ratio of map output size

The Hadoop MapReduce default scheduler (as also the original one [8]) tries to run each map task in the location where its input split is stored. When that is not possible, the input split, that is typically large, has to be transferred over the network, delaying the beginning of the task execution, using network bandwidth, etc., all factors that have a negative impact in the system performance. When a task is launched in the same location as where the split resides, this task is said to be *data local*. Figure 8 shows the percentage of data local tasks in the monsterquery job and Figure 9 shows the ratio for all jobs. The jobs executed with less input splits shows more irregular results, and sometimes, the percentage of data local tasks launched are between 80% and 90%. As the number of input splits increase, the results tend to be more regular and become near 100%. For monsterquery the percentage of data local tasks is similar for both versions. For the other jobs (except streamingsort) we can observe that the ratio is often higher than 2, meaning that the BFT version had better locality than the official version.

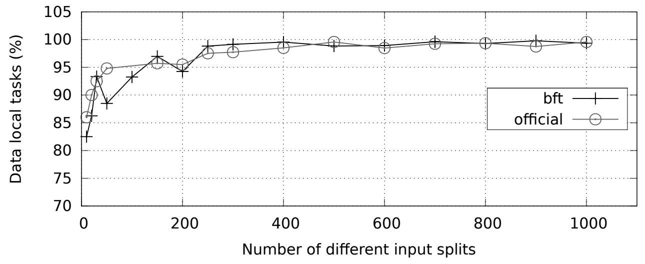

FIGURE 8: Percentage of data local map tasks launched in monsterquery

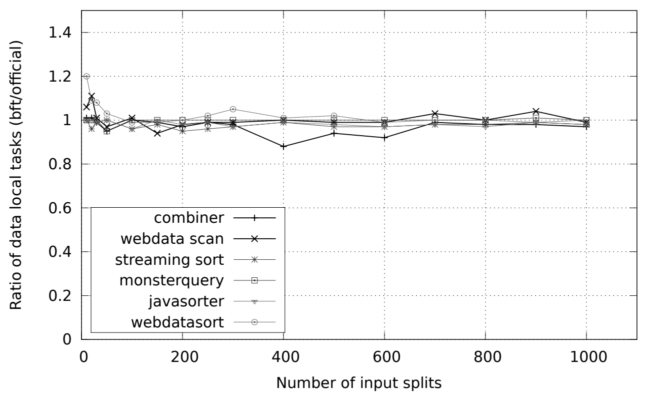

FIGURE 9: Ratio of data local map tasks

6. Related Work

MapReduce has been the subject of much research. Work has been done in adapting MapReduce to perform well in several environments and kinds of applications, such as multi-core and multiprocessor systems (Phoenix system) [28], heterogeneous environments as Amazon EC2 [35], dynamic peer-to-peer environments [21], high-latency eventual-consistent environments as Windows Azure (AzureMapReduce system) [15], iteractive applications (Twister system)

[9], and memory and CPU intensive applications (LEMO-MR system) [11]. Another important trend is research on using MapReduce for scientific computing, e.g., for running high energy physics data analysis and Kmeans clustering [10], and for the generation of digital elevation models [18]. Several systems are similar to MapReduce in the sense that they provide a programming model for processing large data sets, but that allow more complex interactions and/or provide a higher level of abstraction: Dryad [16], Pig Latin [26], Nephele [2]. All these works show the importance of the MapReduce programming model, but from the fault tolerance point of view they do not add to the original MapReduce.

Tolerance to arbitrary faults is a long trend in fault tolerance. Voting mechanisms for masking Byzantine faults in distributed systems were introduced in the early 1980s [27, 19]. State machine replication is a generic solution to make a service crash or Byzantine fault-tolerant [30]. It has been shown to be practical to implement efficient Byzantine fault-tolerant systems [4, 6] and a long line of work appeared, including libraries such as UpRight [5] and EBAWA [33]. As already pointed out, state machine replication is not adequate to make MapReduce Byzantine fault-tolerant. It would be possible to replicate the whole execution of MapReduce in several servers or sets of servers, but the cost would be high.

Byzantine quorum systems have been used to implement data stores with several concurrency semantics [20, 22], even in the cloud [3]. Although the voting techniques have something in common with what we do in our system, these solutions cannot be used to implement BFT MapReduce because it is not a storage service, but a system that does processing.

For volunteer computing and bag-of-tasks applications, Sarmenta proposed a mechanism for sabotage-tolerance based on voting [29]. Most of that work focus on scheduling the workers in a way that no more than a number of false results are obtained. Although we also use voting, we do not consider malicious behavior of workers, only accidental faults, so there is no point in doing complicated scheduling other than avoiding running twice the same task in the same node. Furthermore, much of the novelty of our work is on exploiting the two processing steps (map and reduce) and the (typical) large data size to improve the performance. This is completely different from what that paper does. Another work studies the same problem and presents optimal scheduling algorithms [13]. Fernández et al. also study the same problem, but focus on defining lower bounds on the work done based on a probabilistic analysis of the problem [12]. Again, our problem is different and this kind of analysis is not our objective with this paper.

Very recently, a similar work on volunteer computing but for MapReduce applications appeared [24]. Similarly to our work, the solution is based on voting. The main differences are that the work focus on a different environment (volunteer computing) and does not attempt to reduce the cost and improving the performance, so it does not introduce any of the optimizations that are the core of our work. That paper also presents a probabilistic model of the algorithm that allows assessing the probability of getting a wrong result, something that we do not present here.

The problem of tolerating faults in parallel programs executed in unreliable parallel machines was studied by Kedem et al. long ago [17]. However they proposed a solution based on auditing intermediate steps of the computation to detect faults. On the contrary, we assume that it is not practical to detect arbitrary faults in the execution of arbitrary programs, so comparing two or more executions of a task is the only possibility of detecting faulty processing.

7. Conclusions

We present a Byzantine fault-tolerant MapReduce algorithm and prototype, as well as its evaluation using Hadoop's Gridmix benchmark in the Grid'5000 testbed. The evaluation confirms what might be intuited from the algorithm: that with $f = 1$ the time to execute a job essentially doubles in a small cluster, which is equivalent to approximately the double CPU time. We argue that $f = 1$ is a realistic assumption because: (i) arbitrary faults are rare; (ii) it means that the probability of more than one faulty replicas of the same task returning the same output is negligible.

It is important to notice that our BFT MapReduce tolerates any number of faulty task executions at a low cost: the re-execution of that task. This is not what happens with a simplistic solution like executing a job more than once using the original Hadoop and comparing the outputs. If each execution was affected by one fault in any task, the job might be re-executed forever without any two outputs ever matching.

Acknowledgments This work was partially supported by FCT and EGIDE (Programme PESSOA) through FTH-Grid project, by the EC FP7 through project TCLOUDS (ICT-257243), by the FCT through project CloudFIT, the Multi-annual Program, the Carnegie Mellon Portugal Program, and the INESC-ID multi-annual PIDDAC Program funds. We warmly thank our colleagues of project FTH-Grid for many discussions on topics of this article: Luciana Arantes, Vinícius Cogo, Olivier Marin, Pierre Sens, Fabrício Silva, Julien Sopena. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Mar. 2004.

[2] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 119–130, 2010.

[3] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 31–46, Apr. 2011.

[4] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, Nov. 2002.

[5] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché. UpRight cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles – SOSP'09*, Oct. 2009.

[6] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design & Implementation*, Apr. 2009.

[7] J. Dean. Large-scale distributed systems at google: Current systems and future directions. Keynote speech at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS), Oct. 2009.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, Dec. 2004.

[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818, 2010.

[10] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for data intensive scientific analyses. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 277–284, 2008.

[11] Z. Fadika and M. Govindaraju. LEMO-MR: Low overhead and elastic MapReduce implementation optimized for memory and cpu-intensive applications. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 1–8, 2010.

[12] A. Fernandez, L. Lopez, A. Santos, and C. Georgiou. Reliably executing tasks in the presence of untrusted entities. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 39–50, 2006.

[13] L. Gao and G. Malewicz. Internet computing of tasks with dependencies using unreliable workers. In *Proceedings of the 8th International Conference on Principles of Distributed Systems*, pages 443–458, 2004.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[15] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox. MapReduce in the clouds for science. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 565–572, 2010.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, 2007.

[17] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. G. Spirakis. Combining tentative and definite executions for very fast dependable parallel computing. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 381–390, 1991.

[18] S. Krishnan, C. Baru, and C. Crosby. Evaluation of MapReduce for gridding LIDAR data. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 33–40, 2010.

[19] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[20] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.

[21] F. Marozzo, D. Talia, and P. Trunfio. Adapting MapReduce for dynamic environments using a peer-to-peer model. In *Proceedings of the 1st Workshop on Cloud Computing and its Applications*, Oct. 2008.

[22] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proc. of the 16th International Symposium on Distributed Computing - DISC 2002*, pages 311–325, Oct. 2002.

[23] K. McKusick and S. Quinlan. GFS: evolution on fast-forward. *Communications of the ACM*, 53:42–49, Mar. 2010.

[24] M. Moca, G. C. Silaghi, and G. Fedak. Distributed results checking for MapReduce in volunteer computing. In *Proceedings of the 5th Workshop on Desktop Grids and Volunteer Computing Systems*, May 2011.

[25] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs. In *Proceedings of the EuroSys 2011 Conference*, pages 343–356, 2011.

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.

[27] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of ACM*, 27(2):228–234, Apr. 1980.

[28] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[29] L. F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18:561–572, Mar. 2002.

[30] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[31] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1), 2007.

[32] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, pages 193–204, 2009.

[33] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Proceedings of the 12th IEEE International High Assurance Systems Engineering Symposium*, Nov. 2010.

[34] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.

[35] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on OSDI*, pages 29–42, 2008.