On the Performance of Byzantine Fault-Tolerant MapReduce

Pedro Costa, Marcelo Pasin, Alysson Bessani, Miguel Correia

Abstract—MapReduce is often used for critical data processing, e.g., in the context of scientific or financial simulation. However, there is evidence in the literature that there are arbitrary (or Byzantine) faults that may corrupt the results of MapReduce without being detected. We present a Byzantine fault-tolerant MapReduce framework that can run in two modes: non-speculative and speculative. We thoroughly evaluate experimentally the performance of these two versions of the framework, showing that they use around twice more resources than Hadoop MapReduce, instead of the three times more of alternative solutions. We believe this cost is acceptable for many critical applications.

Index Terms—Hadoop, MapReduce, Byzantine Fault Tolerance

1 Introduction

The MapReduce framework has been developed by Google for processing large data sets [2]. It is used extensively in its datacenters to support core functions such as the processing of indexes for its web search engine. Google's implementation is not openly available, but there is an open source version called Hadoop[1] [3] that is used by many cloud computing companies, including Amazon, EBay, Facebook, IBM, LinkedIn, RackSpace, Twitter, and Yahoo!.[2] Other versions are appearing, e.g., Microsoft's Daytona [4] and the Amazon Elastic MapReduce service [5].

The term MapReduce denominates both a programming model and the corresponding runtime environment. Programming in MapReduce involves developing two functions: a map and a reduce. Each input file of a job is first processed by the map function, then the outputs of these tasks are processed by the reduce function. According to Dean and Ghemawat, this model can express many real world applications [2].

Google's MapReduce platform was designed to be fault-tolerant, because at scales of thousands of computers and other devices (network switches and routers, power units), component failures are frequent. Dean has reported that there were thousands of individual machine, hard drive and memory failures in the first year of a cluster at a Google data center [6]. Both the original MapReduce and Hadoop use essentially two fault tolerance mechanisms: they monitor the execution of map and reduce tasks and reinitialize them if they stop; they add checksums to files with data, so that file corruptions can be detected [3], [7].

Although it is crucial to tolerate crashes of tasks and data corruptions in disk, other faults that can affect the *correctness of results* of MapReduce are known to happen and will probably happen more often in the future [8]. A recent 2.5-year long study of DRAM errors in a large number of servers in Google datacenters, 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) [9]. A Microsoft study of 1 million consumer PCs showed that CPU and core chipset faults are also frequent [10]. MapReduce is designed to work on large clusters and process large data, so errors will tend to occur.

The fault tolerance mechanisms of the original MapReduce and Hadoop cannot deal with such *arbitrary* or *Byzantine faults* [11], [12], even if considering only accidental faults, not malicious faults, as we do in this paper. These faults cannot be detected using file checksums, so they can silently corrupt the output of any map or reduce task, corrupting the result of the MapReduce job. This can be problematic for critical applications, such as scientific or financial simulation. However, it is possible to mask the effect of such faults by *executing each task more than once*, comparing the outputs of these executions, and disregarding the non-matching outputs. Sarmenta proposed a similar approach in the context of volunteer computing to tolerate malicious volunteers that returned false results of tasks they were supposed to execute [13]. However, he considered only bag-of-tasks applications, which are simpler than MapReduce jobs. A similar but more generic solution consists in using the *state machine replication* approach [14]. This approach is not directly applicable to the replication of MapReduce tasks, only to replicate the jobs, which is expensive. A cheaper and simpler solution, which we call *result comparison scheme*, would be to execute each job twice

Pedro Costa and Alysson Bessani are with LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal (pcosta@lasige.di.fc.ul.pt, bessani@di.fc.ul.pt). Marcelo Pasin is with the University of Neuchatel, Switzerland (marcelo.pasin@unine.ch). Miguel Correia is with INESC-ID, Instituto Superior Técnico, Technical University of Lisbon, Portugal (miguel.p.correia@ist.utl.pt). A preliminary version of this paper appeared in the IEEE CloudCom 2011 Conference [1].

[1] Hadoop is an Apache open source project with several components. We use the term Hadoop to mean its MapReduce runtime system, which is the main component of Hadoop.

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

and re-execute it if the results do not match, but the cost would be high in case there is a fault.

This paper presents a Byzantine fault-tolerant (BFT) MapReduce runtime system that tolerates arbitrary faults by executing each task more than once and comparing the outputs. The challenge was to do this *efficiently*, without the need of running $3f + 1$ replicas to tolerate at most f faulty, which would be the case with state machine replication (e.g., [15]–[17]). The system uses several techniques to reduce the overhead. With $f = 1$, it manages to run only two copies of each task when there are no faults plus one replica of a task per faulty replica, instead of a replica of the whole job as in the result comparison scheme.

In this paper we are especially interested in the performance of the BFT MapReduce system. Therefore, we designed it to work in two modes: *non-speculative* and *speculative*. What differentiates them is the moment when reduce tasks start to run. In non-speculative mode, $f+1$ replicas of all map tasks have to complete successfully for reduce tasks to be launched. In speculative execution, reduce tasks start after one replica of all map tasks finish. While the reduce tasks are running, it is necessary to validate the remaining map replicas' outputs. If at some point it is detected that the input used in the reduce tasks was not correct, the tasks will be restarted with the correct input.

We modeled analytically the performance of our BFT MapReduce and evaluated the system extensively in the Grid'5000 testbed[3] using Hadoop's GridMix benchmark [18]. The main conclusions are that our solution is indeed more efficient than the alternatives, using only twice as many resources as the original Hadoop, and that the speculative mode considerably accelerates the execution when there are no faults.

In summary, the main contributions of the paper are:

- an algorithm to execute MapReduce jobs that tolerates arbitrary faults and than can run in two modes, speculative and non-speculative;
- an extensive experimental evaluation of the system using Hadoop's GridMix benchmark in the Grid'5000 testbed.

The paper is organized as follows. Section 2 introduces the MapReduce framework and Hadoop. Section 3 presents the system model in which the algorithm presented in Section 4 is based. Section 5 presents the experimental evaluation. Section 6 discusses related work and Section 7 concludes the paper.

2 MAPREDUCE AND HADOOP

The MapReduce programming model is inspired in the map and reduce functions used in functional programming. In this model, the programmer has to write a pair of functions with these names: *map* and *reduce*. The framework essentially applies these functions in sequence to an input in the form of a set of files often called *splits*. Each split is processed by a task that runs an instance of the map function and saves its result — a sequence of key-value pairs — in the local filesystem. These map task outputs are sorted key-wise, partitioned and fed to tasks that run instances of the reduce function. This phase of sorting, partitioning and feeding is often called *shuffle*. The reduce tasks store their output in an output file.

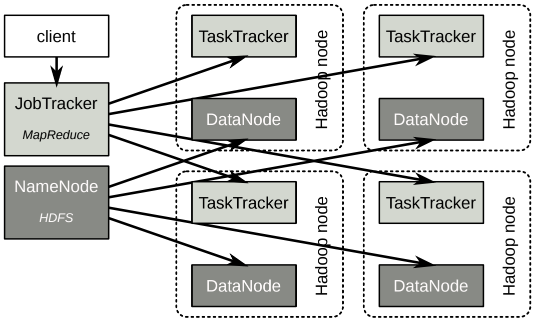


Fig. 1: Hadoop architecture.

Hadoop MapReduce is an implementation of the MapReduce programming model, freely available through the Apache license [3]. It is not only a framework to implement and run MapReduce algorithms, but also a handy tool for developing alternative and improved systems for MapReduce, such as the one presented in this paper.

Hadoop includes the Hadoop Distributed File System (HDFS), which is inspired in the Google File System [7]. HDFS is used to store the input splits and the final output of the job, but not the intermediate results (map outputs, reduce inputs) which are saved in local disc. HDFS is a distributed filesystem tailored for Hadoop. It breaks files in blocks that it replicates in several nodes for fault tolerance. By default it replicates files three times: two in the same rack and one in another rack. Although it does not implement the POSIX call semantics, its performance is tuned for data throughput and large files (blocks in the range of 64 MB) using commodity hardware. HDFS is implemented using a single *name node*, the master node that manages the file name space operations (open, close, rename) and controls access to files by clients. In addition, there are a number of *data nodes*, 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). Data nodes communicate to move blocks around, for load balancing and to keep the replication level on failures.

The Hadoop architecture is presented in Figure 1. MapReduce jobs are submitted to and managed by a centralized service called *job tracker*. This service creates one map task per input split and a predefined number of reduce tasks. It assigns a task to a node with the following priorities (from highest to lowest): to a node

[3] https://www.grid5000.fr

where the input split it has to process is stored; to a node in the same rack; to a node in the same cluster; to a node in the same datacenter. This notion of running the task in the node that has its input is called *locality*.

Each node available to run Hadoop tasks runs a software service called *task tracker* that launches the tasks. Hadoop has mechanisms to tolerate the crash of task trackers and the tasks they execute. A task tracker sends heartbeat messages to the job tracker periodically. These messages indicate that the task tracker has not stopped and, besides that, they carry updated information about the task tracker to the job tracker, as the percentage of the work assigned to the tracker already executed, list of finished tasks, and any error in a task execution. Heartbeat messages (or lack of them) allow the job tracker to figure out that a task tracker or a task stalled or stopped. Using different nodes, the job tracker runs speculative tasks for those lagging behind and restarts the failed ones. Nevertheless, this model only supports crashes, not arbitrary faults. In this paper we develop and describe the implementation of a more elaborated algorithm that provides Byzantine fault tolerance.

3 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 functions, the *job tracker* that manages the execution of a job, and a set of *task trackers* that launch map and reduce tasks. We do not consider the components of HDFS in the model, as the algorithm is to mostly orthogonal to it (and there is a Byzantine fault-tolerant HDFS in the literature [16]).

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 task tracker (correct) or not (faulty). We assume that clients are always correct, because they are not part of the MapReduce execution and if clients were faulty the job output would be necessarily incorrect. We also assume that the job tracker is always correct, which is the same assumption that Hadoop does [3]. It would be possible to remove this assumption by replicating the job tracker, but it would complicate the design considerably and this component does much less work than the task trackers, so we leave this as future work. The task trackers can be correct or faulty, so they can arbitrarily deviate from the algorithm and return corrupted results of the tasks they execute.

Our algorithm does not rely on assumptions about bounds on processing and communication delays. On the contrary, the original Hadoop mechanisms do make assumptions about such times for termination (e.g., they assume that heartbeat messages from correct task trackers do not take indefinitely to be received). 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 connections. We assume the existence of a hash function to produce message digests. This function is collision-resistant, i.e., it is infeasible to find two inputs that produce the same output (e.g., SHA-1 or SHA-3, recently chosen by the NIST).

Our algorithm is configured with a parameter f. In distributed fault-tolerant algorithms f is usually the maximum number of faulty replicas [15]–[17], [19]–[21], but in our case the meaning of f is different: *f is the maximum number of faulty replicas that can return the same output given the same input*. Consider a function $\mathcal{F}$, map or reduce, and that the algorithm executes several replicas of the function with the same input I, so all correct replicas return the same output O. Consider also the worst case in which there are f faulty replicas that execute $\mathcal{F}$ and $\mathcal{F}_1(I) = \mathcal{F}_2(I) = ... = \mathcal{F}_f(I) = O' \neq O$. The rationale is that f is the maximum number of replicas that can be faulty and still allow the system to find out that the correct result is O. If the system selects the correct output by picking the output returned by $f+1$ task replicas, it will never select O' because it is returned by at most f replicas. Similarly to the usual parameter f, our f 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.

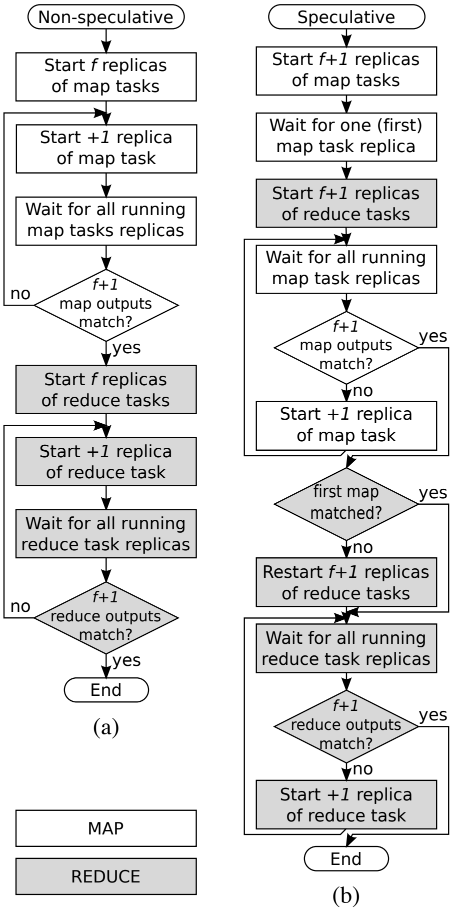
4 BFT MapReduce Algorithm

4.1 Overview

A simplistic solution to make MapReduce Byzantine fault-tolerant considering f the maximum number of faulty replicas is the following. First, the job tracker starts $2f+1$ replicas of each map task in different nodes and task trackers. Second, the job tracker 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 the output in HDFS. In the end, either the client or a special task must vote the outputs to pick the correct result. An even simpler 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 $3f+1$ [22]) 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. Starting from this solution, we propose a set of techniques to avoid these costs:

Deferred execution. Crash faults, which happen more often, are detected using Hadoop standard heartbeats, while arbitrary faults are dealt using replication and voting. Given the expected low probability of arbitrary faults [9], [10], there is no point in always executing $2f+1$ replicas to obtain the same result almost every time. Therefore, our job tracker starts only $f+1$ replicas of map and reduce tasks. After map tasks finish, the


Non-speculative
Start *f* replicas of map tasks
Start *+1* replica of map task
Wait for all running map tasks replicas
f+1 map outputs match?
no
yes
Start *f* replicas of reduce tasks
Start *+1* replica of reduce task
Wait for all running reduce task replicas
f+1 reduce outputs match?
no
yes
End
MAP
REDUCE
Speculative
Start *f+1* replicas of map tasks
Wait for one (first) map task replica
Start *f+1* replicas of reduce tasks
Wait for all running map task replicas
f+1 map outputs match?
yes
no
Start *+1* replica of map task
first map matched?
yes
no
Restart *f+1* replicas of reduce tasks
Wait for all running reduce task replicas
f+1 reduce outputs match?
yes
no
Start *+1* replica of reduce task
End

(a)

(b)

Fig. 2: Flowcharts of (a) non-speculative and (b) speculative executions.

reduce tasks check if all $f+1$ replicas of every map tasks produced the same output. If some outputs do not match, more replicas are started until there are $f+1$ matching replies. At the end of execution, the reduce output is also checked to see if it is necessary to launch more reduce replicas. This algorithm is represented as a flowchart in Figure 2(a).

Digest outputs. $f+1$ map outputs and $f+1$ reduce outputs must be matched to be considered correct. These outputs tend to be large, so it is useful to fetch only one output from some task replica and compare its digest with those of the remaining replicas. With this solution, we avoid transferring the same data several times causing additional network traffic, and we just transfer data from one replica and the digests from the rest.

Tight storage replication. We write the output of all reduce tasks to HDFS with a replication factor of 1, instead of 3 (the default value). We are already replicating the tasks, and their outputs will be written on different locations, so we do not need to replicate these outputs even more. A job starts reading replicated data from HDFS, but from this point forward, the data that is saved in the HDFS by each (replicated) task is no longer replicated.

Speculative execution. Waiting for $f+1$ matching map results before starting a reduce task can worsen the time for the job completion. A way to deal with the problem is for the job tracker to start executing the reduce tasks immediately after receiving the first copy of every map output (see Figure 2(b)). Whenever $f+1$ replicas of a map task finish, if the results do not match, another replica is executed. If $f+1$ replicas of a map finish with matching results but these results do not match the result of the first copy of the task, then the reduces are stopped and launched again with the correct inputs. For a job to complete, $f+1$ matching map and reduce results must be found for all tasks, and the reduces must have been executed with matching map outputs.

The difference between the non-speculative and speculative modes of operation (Figure 2) is that the latter uses these four techniques, whereas the former excludes speculative execution.

4.2 The Algorithm in detail

The algorithm is based on the operation of Hadoop MapReduce and follows its terminology. Recall that a client submits a job to the job tracker that distributes the work to the several task trackers. Therefore, the algorithm is divided in the part executed by the job tracker (Algorithms 1 and 2) and the part executed by the task tracker (Algorithm 3). The work itself is performed by the map and reduce tasks, whose code is part of the job specification.

The presentation of the algorithm follows a number of conventions. Function names are capitalized and variable names are in lowercase letters (separated by '_' if composed of multiple words). Comments are inside {...}. The operator |...| returns the number of elements in a set. There are a number of configuration constants and variables. The algorithm is a set of *event handlers* executed in different conditions.

The idea of the algorithm consists essentially in the job tracker inserting tasks in two *queues* – *q_maps, q_reduces* – and the task trackers executing these tasks. Several auxiliary functions are used to manipulate these queues:

- *Enqueue(queue, tuple)* – inserts the *tuple* describing a task in *queue*;
- *Dequeue(queue, tuple)* – removes a task described by *tuple* from *queue*;
- *FinishedReplicas(queue, task_id)* – searches in *queue* for replicas of a task identified by *task_id* and returns the number of these replicas that have finished;
- *MatchingReplicas(queue, task_id)* – searches in *queue* for finished replicas of a task identified by *task_id* and returns the maximum number of these replicas that have matching outputs;
- *MaxReplicaId(queue, task_id)* – searches in *queue* for replicas of a task identified by *task_id* and returns the highest *replica_id* among them;
- *ReplicaOutput(queue, task_id, replica_id)* – searches in *queue* for a finished task replica identified by *task_id* and *replica_id* and returns its output;

Algorithm 1: Job tracker — common part and non-speculative mode.

```
constants:
    mode            {non-speculative or speculative}
    f               {maximum number of faulty replicas that return the same output}
    nr_reduces      {number of reduce tasks used}
    splits          {split locations}

variables:
    q_maps          {queue of map tasks pending to be executed or being executed; initially empty}
    q_reduces       {queue of reduce tasks pending to be executed or being executed; initially empty}
    reduce_inputs   {identifiers of map replicas that gave inputs to the reduces; initially empty}
    reduces_started {indicates if reduces already started; initialized with false (speculative mode)}

{start execution}
upon job execution being requested do
    for replica_id := 1 to $f+1$ (
      for map_id := 1 to |splits|
        Enqueue(q_maps, (map_id, replica_id, splits[i], not_running));
    )

{restart a stopped task}
upon task (queue, task_id, replica_id) stopping do
    task := Dequeue(queue, (task_id, replica_id));
    task.running := not_running;
    Enqueue(queue, task);

{non-speculative mode: start extra map task replica, start reduce tasks}
upon map task (map_id, replica_id, splits) finishing and mode = non-speculative do
    if (FinishedReplicas(q_maps, map_id)$\geq f+1$ and MatchingReplicas(q_maps, map_id)$< f+1$) (
      new_replica_id := maximum replica_id+1;
      Enqueue(q_maps, (map_id, new_replica_id, splits, not_running));
    ) else (
      if (∀map_id : MatchingReplicas(q_maps, map_id)$\geq f+1$) (
        reduce_inputs := {(map_id, urls) tuples with the f+1 matching output urls for each map_id}
        for replica_id := 1 to $f+1$ (
          for reduce_id := 1 to nr_reduces
            Enqueue(q_reduces, (reduce_id, replica_id, reduce_inputs, not_running));
        )
        for all map_id, replica_id
          Dequeue(q_maps, (map_id, replica_id));
      )
    )

{non-speculative mode: start extra reduce task replica, finish job}
upon reduce task (reduce_id, replica_id, reduce_inputs) finishing and mode = non-speculative do
    if (FinishedReplicas(q_reduces, reduce_id)$\geq f+1$ and MatchingReplicas(q_reduces, reduce_id)$< f+1$) (
      new_replica_id := maximum replica_id+1;
      Enqueue(q_reduces, (reduce_id, new_replica_id, reduce_inputs, not_running));
    ) else (
      if (∀reduce_id : MatchingReplicas(q_reduces, reduce_id)$\geq f+1$) (
        for all reduce_id, replica_id
          Dequeue(q_reduces, (reduce_id, replica_id));
      )
    )

{send task to task tracker}
upon receiving a TASK_REQUEST (task_type, splits_stored_node) message from task tracker do
    queue = undefined;
    if (task_type = map and ∃(map_id, replica_id, split, not_running) ∈ q_maps) (
      queue := q_maps;
      funct := map;
      replica_id_ := replica_id;
      inputs := split;
      if (∃(map_id, replica_id, split, not_running) ∈ q_maps : split ∈ splits_stored_node) (
        task_id := map_id;
      ) else (
        task_id := map_id : (map_id, replica_id, split, not_running) ∈ q_maps;
      )
    ) else if (task_type = reduce and ∃(reduce_id, replica_id, split, not_running) ∈ q_reduces) (
      queue := q_reduces;
      funct := reduce;
      task_id := reduce_id;
      replica_id_ := replica_id;
      inputs := reduce_inputs;
    )
    if (queue = q_maps or queue = q_reduces) (
      task := Dequeue(queue, (task_id, replica_id_));
      task.running := running;
      Enqueue(queue, task);
      Send EXECUTE_TASK (funct, inputs, task_id, replica_id_) message to the task tracker;
    ) else (
      Send NO_TASK_AVAILABLE message to the task tracker;
    )
```

Algorithm 2: Job tracker — speculative mode.

```
{speculative mode: start extra map task replica, start/restart reduce tasks}
upon map task (map_id, replica_id, splits) finishing and mode = speculative do
    if (FinishedReplicas(q_maps, map_id)$\geq f+1$) (
      if (MatchingReplicas(q_maps, map_id)$< f+1$) (
        new_replica_id := MaxReplicaId(q_maps, map_id)+1;
        Enqueue(q_maps, (map_id, new_replica_id, splits, not_running));
      ) else (
        if ($\exists$(map_id, replica_id_, url_) $\in$ reduce_inputs :
            ReplicaOutput(q_maps, map_id, replica_id_)$\neq$ MatchingReplicasOutput(q_maps, map_id)) (
          for all reduce_id, replica_id
            Dequeue(q_reduces, (reduce_id, replica_id));
          reduce_inputs := {(map_id, replica_id, url) tuples with the output url of replica_id of a map_id
                            matching $f$ outputs from other different replicas}
          for replica_id := 1 to $f+1$ (
            for reduce_id := 1 to nr_reduces
              Enqueue(q_reduces, (reduce_id, replica_id, reduce_inputs, not_running));
          )
        )
        for all map_id, replica_id
          Dequeue(q_maps, (map_id, replica_id));
      )
    ) else (
      if (not reduces_started and $\forall$map_id : FinishedReplicas(q_maps, map_id)$\geq 1$) (
        reduces_started := true;
        reduce_inputs := {(map_id, replica_id, url) tuples with the output url of replica_id of map_id}
        for replica_id := 1 to $f+1$ (
          for reduce_id := 1 to nr_reduces
            Enqueue(q_reduces, (reduce_id, replica_id, reduce_inputs, not_running));
        )
      )
    )

{speculative mode: start extra reduce task replica}
upon reduce task (reduce_id, replica_id, reduce_inputs) finishing and mode = speculative do
    if (FinishedReplicas(q_reduces, reduce_id)$\geq f+1$ and MatchingReplicas(q_reduces, reduce_id)$< f+1$) (
      new_replica_id := MaxReplicaId(q_reduces, reduce_id)+1;
      Enqueue(q_reduces, (reduce_id, new_replica_id, reduce_inputs, not_running));
    )

{speculative mode: finish job}
upon $\forall$map_id : MatchingReplicas(q_maps, map_id)$\geq f+1$ and
     $\forall$reduce_id : MatchingReplicas(q_reduces, reduce_id)$\geq f+1$ and
     $\forall$(map_id, replica_id, url) $\in$ reduce_inputs :
     ReplicaOutput(q_maps, map_id, replica_id) = MatchingReplicasOutput(q_maps, map_id) and
     mode = speculative do
    for all reduce_id, replica_id
      Dequeue(q_reduces, (reduce_id, replica_id));
```

Algorithm 3: Task tracker.

```
constant:
    task_type           {type of task this task tracker executes, map or reduce}

variables:
    splits_stored_node  {splits currently stored in this node}
    executing_task      {indicates if a task is being executed; initialized with false (not executing)}

{every T units of time request a task for execution or send heartbeat}
upon timer expiring do
    if (executing_task = false) (
      Send TASK_REQUEST (task_type, splits_stored_node) message to the job tracker;
    ) else (
      Send HEARTBEAT message to the job tracker;
    )
    launch timer;

{execute a task}
upon receiving EXECUTE_TASK (funct, inputs, task_id, replica_id) message from the job tracker do
    if (executing_task = false) (
      executing_task := true;
      execute funct(inputs); {funct may be a map or a reduce}
      Send TASK_FINISHED (task_id, replica_id) message to the job tracker;
      executing_task := false;
    ) else (
      Send ANOTHER_TASK_EXECUTING message to the job tracker;
    )
```

- *MatchingReplicasOutput(queue, task_id)* – searches in *queue* for finished replicas of a task identified by *task_id* and returns the output of the maximum number of replicas that have matching outputs;

Let us first present the algorithm executed by the *job tracker* in non-speculative mode (Algorithm 1), then the changes to this algorithm when executed in speculative mode (Algorithm 2), and finally the algorithm executed by every task tracker (Algorithm 3).

Non-speculative job tracker. When the execution of a job is requested, the job tracker inserts $f+1$ replicas of every map task in the *q_maps* queue, which is the minimum number of replicas executed of every task (lines 13-18). Each map task is in charge of processing one input split (line 16). If any task (map or reduce) stops or stalls, it is dequeued and enqueued again to be re-executed (lines 21-24).

In the non-speculative mode, two things may happen when a map task finishes (lines 26-41). If $f+1$ or more replicas of a task have finished but there are no $f+1$ matching outputs, then a Byzantine fault happened and another replica is enqueued for execution (lines 28-30). If there are already $f+1$ matching outputs for every map task, then the map phase ends and the reduces are enqueued to be executed (lines 32-40). To be consistent with Hadoop's nomenclature, we use *urls* to indicate the locations of the map outputs passed to the reduces.

When a reduce task finishes in non-speculative mode, two things can happen (lines 43-53). Similarly to the map tasks, if $f+1$ or more replicas of a task have finished but there are no $f+1$ matching outputs, then there was a Byzantine fault and another replica is enqueued (lines 45-47). Otherwise, if there are already $f+1$ matching outputs for every reduce task, then the job execution finishes (lines 49-52).

The last event handler processes a request for a task coming from a task tracker (lines 55-82). If a map task is being requested, the job tracker gives priority to map tasks for which the input split exists in the node that requested the task. Otherwise, it assigns to the task tracker the next non-running map task in the queue. If a reduce is requested, the job tracker returns the next reduce task in the queue.

Speculative job tracker. Algorithm 2 contains the functions that change in the job tracker when the algorithm is executed in speculative mode. Similarly to what happens in non-speculative mode, if $f+1$ or more replicas of a map task have finished but there are no $f+1$ matching outputs, another replica is enqueued (lines 4-6). On the contrary to the other mode, only one replica of each map task must have finished for the reduces to be enqueued for execution (lines 23-30). Finally, in speculative mode there is an extra case: if there are $f+1$ matching outputs of a task but they differ from the one that was used to start executing the reduces, all the reduces have to be aborted and restarted (lines 8-21).

When a reduce task finishes in speculative mode, if there are $f+1$ outputs for that task but not $f+1$ with matching outputs, a new replica is enqueued (lines 33-38).

The event handler in lines 40-47 checks if the job can finish in speculative mode. It tests if there are enough matching map and reduce outputs (lines 41-42) and if the reduces were executed with correct input (line 43-44). If that is the case, the job finishes. This handler is exceptional in the sense that it is activated by the termination of both map and reduce tasks; its code might be part of the two previous handlers but we made it separate for clarity.

Task tracker. The presentation of the *task tracker* algorithm (Algorithm 3) was simplified by considering that a task tracker does not execute tasks in parallel and that it only executes maps or reduces (defined by the constant in line 2). In practice what happens are essentially N parallel executions of the algorithm, some for map tasks, others for reduce tasks (the number of each is configurable, see Section 5). Periodically every task tracker either requests a task to the job tracker when it is not executing one, or sends a heartbeat message reporting the status of the execution (lines 9-15). If it receives a task from the job tracker, it executes the task and signals termination to the job tracker (lines 18-26). The algorithm does not show the details about inputs/outputs but the idea is: the input (split) for a map task is read from HDFS; the inputs for a reduce task are obtained from the nodes that executed the map tasks; the outputs of a reduce task are stored in HDFS.

Discussion. Algorithms 1-3 show the implementation of only two of the four mechanisms used to improve the efficiency of the basic algorithm: deferred execution and speculative execution. The digest outputs mechanism is hidden in functions *MatchingReplicas* and *MatchingReplicasOutput*. The tight storage replication is implemented by modifying HDFS.

In the normal case, Byzantine faults do not occur, so the mechanisms used in the algorithm greatly reduce the overhead introduced by the basic scheme. Specifically, without Byzantine faults, only $f+1$ replicas of each task are executed and the storage overhead is minimal. Notice also that our algorithm tolerates any number of arbitrary faults during the execution of a job, as long as there are no more than f faulty replicas of a task that return the same (incorrect) output.

4.3 The Prototype

The prototype of the BFT MapReduce runtime was implemented by modifying the original Hadoop 0.20.0 source code. Hadoop is written in Java so we describe the modifications made in key classes. HDFS was almost not modified for two reasons. First, it is used only before the execution of map tasks and after the execution of reduces, therefore it barely interferes with the performance of MapReduce. Second, there is a Byzantine fault-tolerant HDFS in the literature [16] so we did not investigate this issue. The single modification was the setting of the replication factor to 1 to implement the tight storage replication mechanism (Section 4).

Most modifications were made in the JobTracker class in order to implement Algorithms 1-2. The constants of the algorithm (lines 1-5) are read from an XML file. 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 job tracker gives each map replica a path to a different replica of the split, stored in a different data node, 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 data node of the split is, so this usage of different split replicas forces the replicas of a map to be executed in different task trackers (TaskTracker class), which improves fault tolerance.

The JobTracker class stores information of a running job in an object of the JobInProgress class. The TaskTracker class sends heartbeat messages to job tracker 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). The digest is saved in a JobInProgress instance, more precisely in an object of the VotingSystem class. The Heartbeat class, used to represent a heartbeat message, was modified to include the digest and task replica identifier.

5 Evaluation

We did an extensive evaluation of our BFT MapReduce. Our purpose was mainly to answer the following questions: Is it possible to run BFT MapReduce without an excessive cost in comparison to the original Hadoop? Is there a considerable benefit in comparison to the use of common fault tolerance techniques such as state machine replication? Is there a considerable benefit in using the speculative mode in scenarios with and without faults? Is it possible to still achieve a high degree of locality in comparison to the original Hadoop?

Our experimental evaluation was based on a benchmark provided by Hadoop called GridMix [18]. More specifically, we used GridMix2, which is composed by the following jobs: monsterquery, webdatascan, webdatasort, combiner, streamingsort and javasort. The experiments were executed in the Grid'5000 environment, a French geographically distributed infrastructure used to study large-scale parallel and distributed systems, during several months.

5.1 Analytical Evaluation

This section models analytically the performance of the BFT MapReduce, the original MapReduce, and hypothetical BFT MapReduce systems based on state machine replication and the result comparison scheme. This performance is analyzed in terms of a single metric: *the total execution time* or *makespan*. Our objective is twofold: to do a comparison with systems that do not exist, so cannot be evaluated experimentally; to provide an expression that helps understanding the experimental results presented in the following sections.

The execution of a job is composed by serial and parallel phases. The job initialization, the shuffle (sending the map task outputs to the reduce tasks), and the finishing phase belong to the serial phase. We model these times as a single value Ts. The two parallel phases are the execution of map and reduce tasks. The time to execute the map (resp. reduce) tasks depends on maximum the number of map (resp. reduce) tasks that can be executed in parallel.

The total execution time of a job (makespan) without faults is obtained using Equation 1 for all considered versions of MapReduce. The versions are differentiated by the value of α, that corresponds to the number of replicas of map and reduce tasks executed (without faults). For the original MapReduce $\alpha = 1$ and for the BFT MapReduce $\alpha = f + 1$. In the two hypothetical schemes, the client issues the job to a set of replicas that process the job in parallel and return the results, which are compared by the client. Byzantine fault-tolerant state machine replication typically requires $3f+1$ replicas, but Yin et al. [23] have shown that only $2f+1$ have to execute the service code, therefore, for this version $\alpha = 2f + 1$. The result comparison scheme consists in executing the whole job $f + 1$ times; if all executions return the same result, that is the correct result, otherwise the job has to be re-executed one or more times until there are $f + 1$ matching results. Therefore, for this scheme $\alpha = f + 1$.

$$Tj = Ts + \alpha \cdot \left\lceil \frac{Nm}{Pm \cdot N} \right\rceil \cdot Tm + \alpha \cdot \left\lceil \frac{Nr}{Pr \cdot N} \right\rceil \cdot Tr - \Omega \quad (1)$$

In relation to the rest of the parameters, Nm and Nr are the number of map and reduce tasks executed. Pm and Pr are the number of tasks that can be executed in parallel per task tracker (in Hadoop by default $Pm = Pr = 2$) and N is the number of nodes (or task trackers). Tj is the time of a job execution, Tm is the average time that it takes to execute a map task, and Tr is the same for a reduce task. These values change from job to job and have to be obtained experimentally. Ω expresses an overlap that may exist in the execution of map and reduce tasks. For the original Hadoop we observed that Ω is essentially 0 because reduce tasks start by default after 95% of map tasks finish. So we use this value for the state machine replication and result comparison versions. For the BFT MapReduce we have two cases. In non-speculative mode, again $\Omega = 0$. In speculative mode, the reduces start processing data after the first replica of every map is executed, so there is an overlap and $\Omega > 0$.

To assist in the comparison among MapReduce systems, we introduce a parameter called the *non-replicated task processing time*, Tn. This parameter measures the time to process map and reduce tasks without replication, i.e., in the original Hadoop. It is obtained from Equation 1 by setting $Ts = 0$ (it considers only task processing time), $\alpha = 1$ (no replication), and $\Omega = 0$ (no overlap). The Tn parameter is defined in Equation 2.

$$Tn = \left\lceil \frac{Nm}{Pm \cdot N} \right\rceil \cdot Tm + \left\lceil \frac{Nr}{Pr \cdot N} \right\rceil \cdot Tr \qquad (2)$$

Table 1 compares the MapReduce systems when there are no faults: the original Hadoop, our BFT MapReduce in non-speculative (BFT-MR-ns) and speculative (BFT-MR-s) modes, the hypothetical BFT MapReduce based on state machine replication (BFT-MR-smr) and the BFT MapReduce based on the result comparison scheme (BFT-MR-cmp). The comparison is made in terms of Tj/Tn and assumes $Ts \ll Tn$, $Nm \gg Pm \cdot N$ and $Nr \gg Pr \cdot N$ (consider the contrary: if there are much more resources than tasks, executing some extra tasks does not impact the total execution time). It is shown in terms of a formula and by instantiating it with the first values of f. This table shows the benefit of our BFT MapReduce opposed to the state machine replication version and to the comparison scheme version.

System	Tj/Tn	$f = 1$	$f = 2$	$f = 3$
Hadoop	1	1	1	1
BFT-MR-ns	$f + 1$	2	3	4
BFT-MR-s	$f + 1 - \frac{\Omega}{Tn}$	$2 - \frac{\Omega}{Tn}$	$3 - \frac{\Omega}{Tn}$	$4 - \frac{\Omega}{Tn}$
BFT-MR-smr	$2f + 1$	3	5	7
BFT-MR-cmp	$f + 1$	2	3	4

TABLE 1: Analytical comparison between MapReduce systems without faults.

Table 2 shows the impact of a map task affected by a Byzantine fault in the makespan of a job. The impact would be similar with a faulty reduce task, but Pm would be substituted by Pr. The table shows that the impact in our BFT MapReduce is quite small, whereas for the result comparison scheme the makespan doubles.

System	Extra time if there is a faulty map task
Hadoop	cannot tolerate
BFT-MR-ns	$Tm/(Pm \cdot N)$ on average
BFT-MR-s	$Tm/(Pm \cdot N)$ on average
BFT-MR-smr	0
BFT-MR-cmp	Tj

TABLE 2: Effect of a single faulty map task in the makespan for all MapReduce systems.

Clearly the answer to one of the questions — if there was a clear benefit in using our scheme instead of state machine replication or the result comparison scheme — is positive.

5.2 Experimental Evaluation

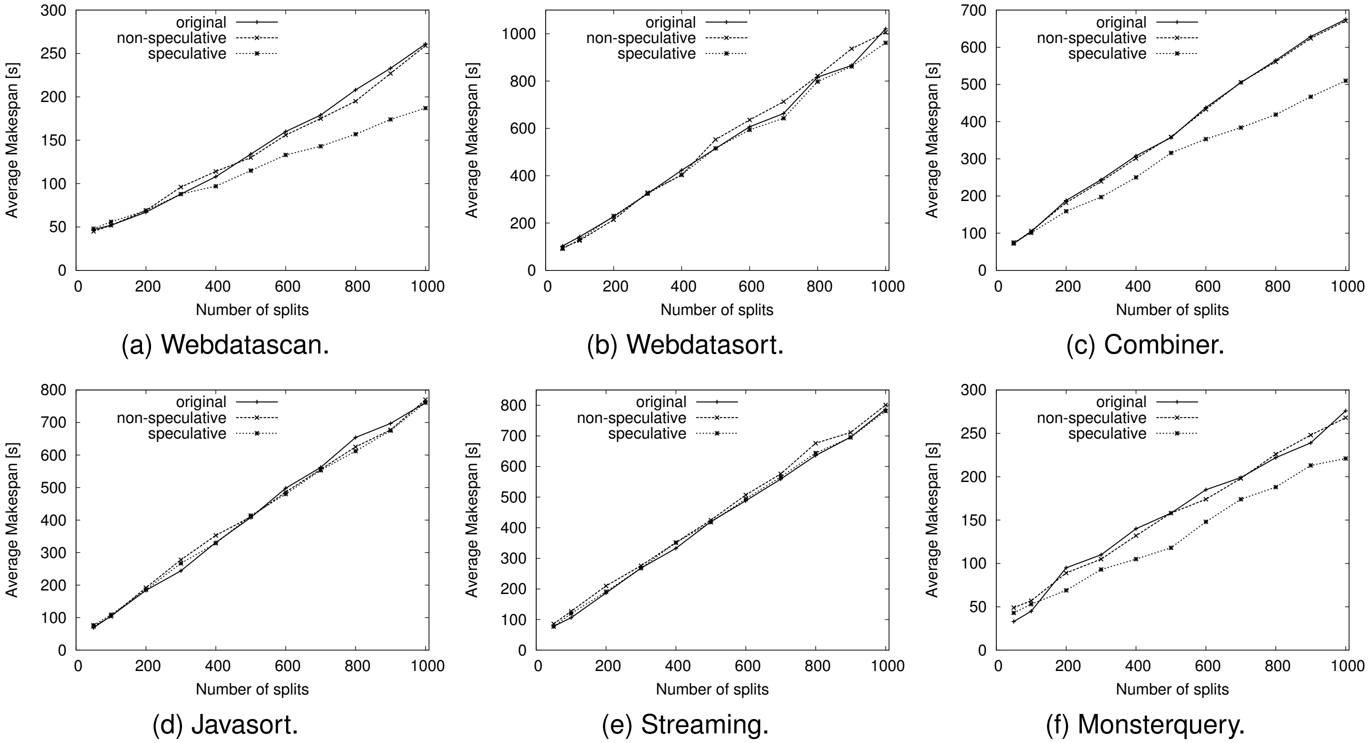
5.2.1 Makespan vs. Number of Input Splits

Figure 3 shows the makespan of the six GridMix2 benchmark applications for the original Hadoop and our BFT MapReduce in the non-speculative and speculative modes. We consider only the case of $f = 1$. We recall that the meaning of f in our system is not the usual one and that the probability of the corresponding assumption being violated is even lower than in other BFT replication algorithms. The values we present are averages of around 100 executions of each experiment. The average does not include outliers, which represent less than 1% of the executions. The standard deviation is low, showing that most of the results are close to the average, and for that reason we do not include this information in any of the following graphs. Each job processed from 50 to 1000 input splits of 64 MB stored in HDFS data nodes. We use the default data-block size of 64 MB to minimize the cost of seeks [3]. To allow results to be comparable, we used a standard configuration for all tests. The times reported were obtained from the logs produced by GridMix. We choose to run the experiments with the original Hadoop in 100 cores and those with the BFT MapReduce in 200 cores, as our framework uses twice as many resources as the first with $f = 1$ and no faults, which is the case we are considering. This allows a fair apples-to-apples comparison. We presented results for setups with the same number of cores in the preliminary version of this work [1].

Overall, the original Hadoop and the BFT MapReduce in non-speculative mode had similar makespan in all experiments (see Figure 3). This may seem counterintuitive but recall that we provided the BFT MapReduce with twice the number of cores of Hadoop. This similarity of makespans shows something interesting: the additional communication contention in the network and in the nodes (caused by the comparisons of all map replica's outputs) did not impact significantly the performance of the BFT MapReduce (twice as much computation was done using twice as many resources in the same time).

The objective of the speculative mode is to improve the makespan by starting reduces earlier, when there are results from at least one replica of each map. The speculative mode improved the makespan in three of the benchmarks — webdatascan, combiner, and monsterquery. We can observe that the improvement gets larger as there are more splits to process, reaching an improvement of 30-40%, which is to be expected as more splits mean more map tasks to process them. Interestingly, the speculative mode had almost no impact in the other three benchmarks with up to 1000 splits. An analysis of the logs of these experiments has shown that the reduce tasks were launched when around three quarters of the maps finished, instead of one half as would be expectable with $f + 1 = 2$. This late start of the reduce tasks led to a very small benefit in using the speculative execution.

In summary, there is a cost associated to running BFT MapReduce in comparison to the original Hadoop, as approximately twice as many resources are used ($\alpha = 2$ in Equation 1). Given twice the number of cores, the time to run BFT MapReduce is essentially the same as Hadoop's, somewhat better if the speculative mode is used. With the same number of cores the makespan is



(a) Webdatascan.

(b) Webdatasort.

(c) Combiner.

(d) Javasort.

(e) Streaming.

(f) Monsterquery.

Fig. 3: Makespan of total execution time of the six GridMix2 benchmark applications varying the number of splits.

approximately the double [1].

5.2.2 Locality

The Hadoop MapReduce default scheduler tries to run each map task in the location where its input split is stored. This objective is called locality and a map task for which this is achieved is called a *data-local* task. When this is not possible, the task will be preferably processed in a node in the rack where the input split resides — *rack-local* tasks. If not even that is possible, the task is executed in a node in a different rack. Locality is an important property in MapReduce as moving large amounts of data (splits of 64 MB in our experiments) for a different node takes time and occupies resources, with a negative impact in the makespan and the load in the systems where MapReduce is executed.

Figure 4 shows the percentage of data local tasks in the webdatasort, javasort, and streaming benchmarks in the same experiments that were reported in the previous section. The results of the others are similar so we do not show them in the interest of space. The first conclusion is that the locality of the BFT MapReduce in both modes is around 90% and similar to the one achieved in the original Hadoop with half of the nodes. A second conclusion is that although the absolute number of non-data-local tasks increases considerably, the percentage of non-data-local tasks stays reasonably stable when the number of tasks increases.

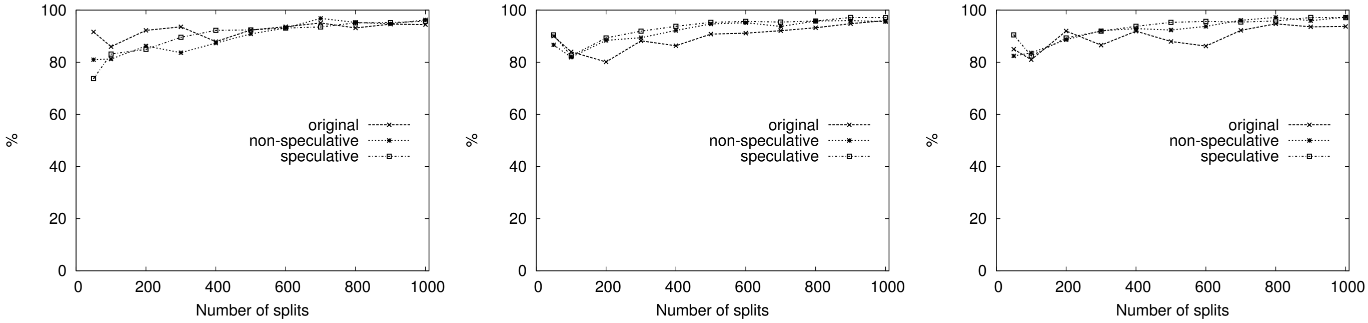
5.2.3 Data Volume

This section compares the quantity of data processed in the original and in the BFT Hadoop. Figure 5 shows the total size of the data produced by the maps and reduces in each experiment, i.e., the sum of the size of the outputs of all maps and the sum of the size of the outputs of all reduces. Recall that each input split has 64 MB. For the webdatasort application, the output of each map task had approximately the same size of its input (64 MB), whereas for streaming the output of a map had an average of 17 MB. The main conclusion is that the total output data of the BFT MapReduce with $f = 1$ and two replicas executed without faults is twice the value for Hadoop, which is the expected result.

5.2.4 Makespan with Faults

The experiments of the previous sections were executed in a scenario without faults. We created a simple *fault injector* that tampers outputs and digests of outputs of map and reduce tasks. The component injects random bits leading the job tracker to detect differences in the outputs of replicas, forcing the system to run additional tasks. The percentage of tasks affected by faults is configurable.

We set the percentage of faults to 10%. Figure 6(a) shows the makespan of webdatasort with different numbers of splits. The graph shows two lines with a slope similar to those in Figure 3. To better compare the makespan with and without faults Figure 6(b) shows the ratio between them. We can see that the makespan

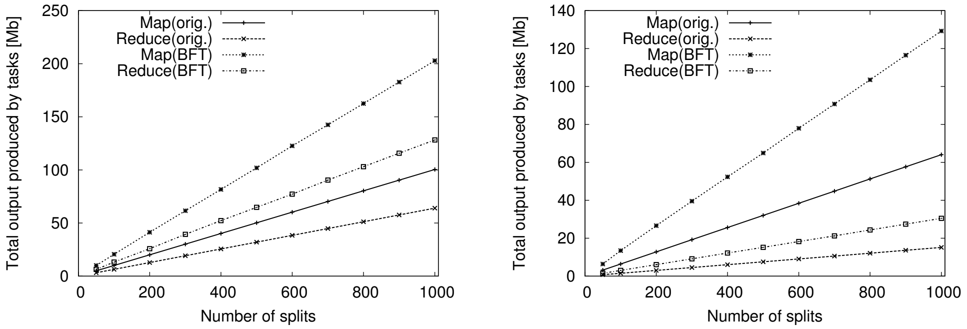


(a) Webdatasort.

(b) Javasort.

(c) Streaming.

Fig. 4: Percentage of data-local tasks in three of the GridMix2 benchmarks varying the number of splits.



(a) Webdatasort.

(b) Streaming.

Fig. 5: Total size of map and reduce outputs in two of the GridMix2 applications with different number of splits.

was roughly 10% longer, which is to be expected after executing 10% more tasks (see Equation 1 and consider that Ts and Ω are negligible in comparison to the time taken to execute the tasks).

5.2.5 Makespan vs. Parallelism

In the tests presented so far we used a fixed number of nodes. That configuration allowed running the experiments as quickly as possible, within the resource constraints imposed by Grid'5000. This section presents experiments in which we fixed the input data size and varied the number of nodes (without faults). As we increase the number of nodes, we allow more tasks to run in parallel.

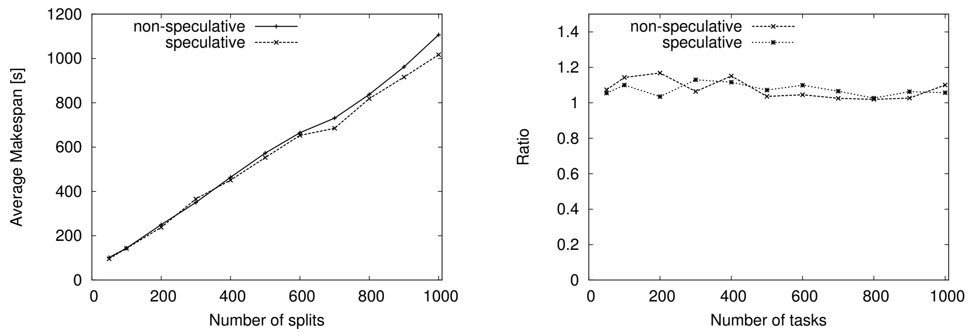
The experimental results are presented in Figure 7. We also plotted values from the original Hadoop estimated using Equation 1 (with $\Omega = 0$ and values of Ts, Tm and Tr obtained experimentally). The horizontal axis is the number of map/reduce tasks that can be executed in parallel (i.e., $Pm \cdot N = Pr \cdot N$) and the vertical axis is the makespan. Interestingly, the equation provides a good approximation of the curves (also for the BFT versions, although we do not plot those curves).

The graphs show an exponential drop as the number of map/reduce tasks executed in parallel increases. From the equation it becomes clear that the curves converge asymptotically to $Ts - \Omega$.

6 Related Work

MapReduce has been the topic of much recent research. Work has been done in adapting MapReduce to perform well in several environments and distinct applications, such as multi-core and multiprocessor systems (Phoenix) [24], heterogeneous environments as Amazon EC2 [25], dynamic peer-to-peer environments [26], high-latency eventual consistency environments as Windows Azure (Microsoft's Daytona) [27], iterative applications (Twister) [28], and memory and CPU intensive applications (LEMO-MR) [29]. Another important trend is using MapReduce for scientific computing, e.g., for running high energy physics data analysis and Kmeans clustering [30], and for the generation of digital elevation models [31]. Other systems are similar to MapReduce in the sense that they provide a programming model for processing large data sets, allowing more complex interactions and/or provide a higher level of abstraction: Dryad and DryadLINQ [32], [33], Pig Latin [34], and Nephele [35]. All these works show the importance of the MapReduce programming model, but none of them improves the original MapReduce in terms of fault tolerance.

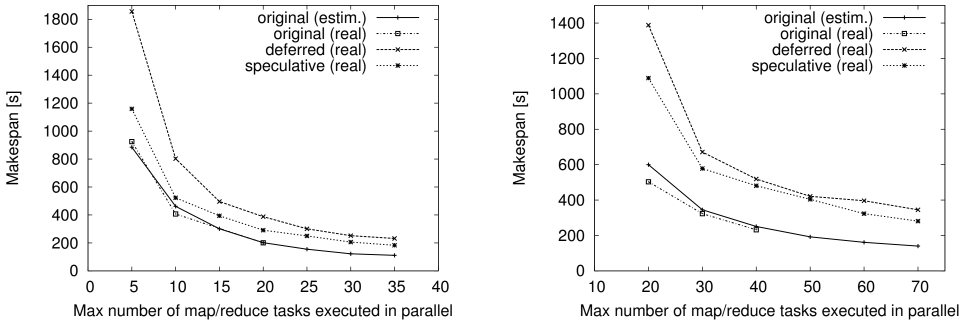
Tolerance to arbitrary faults is a long trend in fault tolerance. Voting mechanisms for masking Byzantine faults in distributed systems were introduced in the early 1980s [11]. State machine replication is a generic solution



(a) Makespan of both modes of execution.

(b) Faulty over faultless makespan ratio.

Fig. 6: Makespan of the webdatasort benchmark with fault injector enabled.



(a) Combiner.

(b) WebdataSort.

Fig. 7: Makespan varying the parallelism without faults.

to make a service crash or Byzantine fault-tolerant [14]. It has been shown to be practical to implement efficient Byzantine fault-tolerant systems [15] and a long line of work appeared, including libraries such as PBFT [15], UpRight [16], and BFT-SMaRt [36]. 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 nodes, but the cost would be high.

Byzantine quorum systems have been used to implement data stores with several concurrency semantics [19], [20], even in the cloud [21]. Although the voting techniques they use have something in common with what we do in our framework, these solutions cannot be used to implement BFT MapReduce because it is not a storage service, but a system that does computation.

For volunteer computing and bag-of-tasks applications, Sarmenta proposed a voting mechanism for sabotage-tolerance [13]. Most of that work focus on scheduling the workers in a way that no more than a number of false results are obtained. Albeit we also use a voting mechanism, we do not consider malicious behavior of workers, only accidental faults, so there is no point in complicating the scheduling except for avoiding running the same task in the same node twice. 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 Sarmenta's work. Another work studies the same problem and presents optimal scheduling algorithms [37]. Fernandez 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 [38]. Again, our problem is different and this kind of analysis is not the goal in this paper.

Very recently, a similar work on volunteer computing focusing MapReduce applications appeared [39]. 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 or improve 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 running in unreliable parallel machines was studied by Kedem et al. long ago [40]. However they proposed a solution based on auditing intermediate steps of the computation to detect faults. We assume that it is not feasible to create a detector that monitors arbitrary programs to detect arbitrary failures. On the contrary, we compare results of different executions of the same program/task with the same inputs to detect the effects of arbitrary faults.

Our BFT MapReduce replicates unmodified map and reduce functions. SWIFT is a compiler that makes programs Byzantine fault-tolerant by replicating every binary instruction [41]. PASC is a library that transparently hardens processes against Byzantine faults [42]. On the contrary to our scheme, both solutions require modifying the programs. They are also unrelated to MapReduce.

7 Conclusion and discussion

The paper presents a Byzantine fault-tolerant MapReduce algorithm and its experimental evaluation. The fault tolerance mechanisms of the original MapReduce cannot deal with Byzantine faults. These faults in general cannot be detected, so they can silently corrupt the output of any map or reduce task. Our algorithm masks these faults by executing each task more than once, comparing the outputs of these executions, and disregarding non-matching outputs. This simple but powerful idea allows our BFT MapReduce to tolerate any number of faulty task executions at the cost of one re-execution per faulty task.

The experimental evaluation confirmed what might be intuited from the algorithm: with $f = 1$ the resources used and the makespan essentially double. Although this is a considerable cost, we have shown that it is much better than alternative solutions: state machine replication and result comparison scheme. This cost may be acceptable for a large number of applications that handle critical data. We also believe that setting f to 1 is a reasonable option as this parameter is the maximum number of faulty replicas that return the same output. We have shown experimentally that the impact of faults in the total execution time is low even in a harsh fault scenario.

As future work, we plan to study the possibility of running MapReduce in several datacenters in order to tolerate faults that severely impact a subset of them. Furthermore, we aim to study to what extent similar schemes can be applied to generalizations of MapReduce like Dryad or Pig Latin.

Acknowledgements

This work was partially supported by the EC FP7 through project TCLOUDS (ICT-257243), by the FCT through project RC-Clouds (PTDC/EIA-EIA/115211/2009), the Multi-annual Program (LASIGE), and contract PEst-OE/EEI/LA0021/2011 (INESC-ID). We warmly thank Fernando Ramos for the feedback on earlier versions of this paper and Fabricio Silva, Luciana Arantes, Olivier Marin, Pierre Sens, Julien Sopena, and Vinicius Cogo for their contributions on an initial version of this work, done in the FTH-grid project (supported by FCT and EGIDE). 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] P. Costa, M. Pasin, A. Bessani, and M. Correia, "Byzantine fault-tolerant MapReduce: Faults are not just crashes," in *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science*, 2011, pp. 32–39.

[2] 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.

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

[4] "Daytona – Iterative MapReduce on Windows Azure," http://research.microsoft.com/en-us/projects/daytona/.

[5] "Amazon Elastic MapReduce," http://aws.amazon.com/elasticmapreduce/.

[6] 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.

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

[8] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, 2007.

[9] 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*, 2009, pp. 193–204.

[10] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs," in *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2011, pp. 343–356.

[11] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programing Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[12] 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*, vol. 1, no. 1, pp. 11–33, Mar. 2004.

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

[14] F. B. Schneider, "Implementing fault-tolerant service using the state machine aproach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[15] M. Castro and B. Liskov, "Practical Byzantine fault-tolerance and proactive recovery," *ACM Transactions Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[16] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Rich, "UpRight cluster services," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009.

[17] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? Byzantine fault tolerance with a spinning primary," in *Proceedings of the 28th IEEE Symposium on Reliable Distributed Systems*, Sep. 2009.

[18] "MapReduce 0.22 Documentation – GridMix," http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.

[19] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, Oct. 1998.

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

[21] 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 ACM SIGOPS/EuroSys European Conference on Computer Systems*, Apr. 2011, pp. 31–46.

[22] M. Correia, N. F. Neves, and P. Verssimo, “From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures,” *The Computer Journal*, vol. 49, no. 1, pp. 82–96, Jan. 2006.

[23] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, “Separating agreement form execution for Byzantine fault tolerant services,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Oct. 2003, pp. 253–267.

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

[25] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, “Improving MapReduce performance in heterogeneous environments,” in *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 29–42.

[26] 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.

[27] 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*, 2010, pp. 565–572.

[28] 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*, 2010, pp. 810–818.

[29] 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*, 2010, pp. 1–8.

[30] J. Ekanayake, S. Pallickara, and G. Fox, “MapReduce for data intensive scientific analyses,” in *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, 2008, pp. 277–284.

[31] 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*, 2010, pp. 33–40.

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

[33] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, “DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language,” in *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, 2008, pp. 1–14.

[34] 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*, 2008, pp. 1099–1110.

[35] 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*, 2010, pp. 119–130.

[36] “BFT-SMART - High-performance Byzantine fault-tolerant State Machine Replication,” http://code.google.com/p/bft-smart/.

[37] 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*, 2004, pp. 443–458.

[38] 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*, 2006, pp. 39–50.

[39] 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.

[40] 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*, 1991, pp. 381–390.

[41] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, “SWIFT: Software implemented fault tolerance,” in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[42] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini, “Practical hardening of crash-tolerant systems,” in *Proceedings of the USENIX Technical Conference*, 2012.

Pedro Costa Pedro Costa is a Ph.D. student at the Department of Informatics, University of Lisboa. He is a member of the Large-Scale Informatics Systems Laboratory (LASIGE) laboratory and the Navigators research group. His research interests are concerned with fault-tolerance, distributed systems, and cloud computing. He is involved in TCLOUDS (EC FP7) project. More information about him at http://pedrosacosta.wikidot.com

Marcelo Pasin Marcelo Pasin is a researcher at the University of Neuchatel, Switzerland. He has a PhD in CS from the INP Grenoble, a MS in CS from the Fed. Univ. of Rio Grande do Sul, and an EE degree from the Fed. Univ. of Santa Maria. He worked two years in the industry, was an associate professor at the Federal Univ. of Santa Maria, a researcher at INRIA in Lyon, and an assistant professor at the Univ. of Lisbon. He worked in several research projects, including CoreGRID, EGEE, EC-GIN, TClouds, SECFUNET, SRT-15 and LEADS, with research interests in parallel and distributed computing.

Alysson Neves Bessani is an Assistant Professor of the Department of Informatics of the University of Lisboa Faculty of Sciences, Portugal, and a member of LASIGE research unit and the Navigators research team. He received his B.S. degree in Computer Science from Maringá State University, Brazil in 2001, the MSE in Electrical Engineering from Santa Catarina Federal University (UFSC), Brazil in 2002 and the PhD in Electrical Engineering from the same university in 2006. His main interests are distributed algorithms, Byzantine fault tolerance, coordination, middleware and systems architecture.

Miguel Correia is Associate Professor at Instituto Superior Técnico of the Universidade Técnica de Lisboa and researcher at INESC-ID. He has been involved in several international research projects related to intrusion tolerance and security, including the TCLOUDS, MAFTIA and CRUTIAL EC-IST projects, and the ReSIST NoE. He has more than 100 publications in journals, conferences and workshops. His main research interests are: security, intrusion tolerance, distributed systems, distributed algorithms, cloud computing.