Parallel Computing 39 (2013) 233–244

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

MRSG – A MapReduce simulator over SimGrid

Wagner Kolberg [a,*], Pedro de B. Marcos [a], Julio C.S. Anjos [a], Alexandre K.S. Miyazaki [a], Claudio R. Geyer [a], Luciana B. Arantes [b]

[a] Federal University of Rio Grande do Sul (UFRGS), Institute of Informatics – GPPD, Caixa Postal 15.064 – 91.501-970, Porto Alegre, RS, Brazil
[b] Universit Pierre et Marie Curie, CNRS INRIA – REGAL, 4 Place Jussieu, 75005 Paris, France

A R T I C L E   I N F O

Article history:
Received 19 July 2011
Received in revised form 6 November 2012
Accepted 15 February 2013
Available online 24 February 2013

Keywords:
MapReduce
Data-intensive
Simulator
SimGrid
Grids

A B S T R A C T

MapReduce is a parallel programming model to process large datasets, and it was inspired by the Map and Reduce primitives from functional languages. Its first implementation was designed to run on large clusters of homogeneous machines. Though, in the last years, the model was ported to different types of environments, such as desktop grid and volunteer computing. To obtain a good performance in these environments, however, it is necessary to adapt some framework mechanisms, such as scheduling and data distribution algorithms. In this paper we present the MRSG simulator, which reproduces the MapReduce work-flow on top of the SimGrid simulation toolkit, and provides an API to implement and evaluate these new algorithms and policies for MapReduce. To evaluate the simulator, we compared its behavior against a real Hadoop MapReduce deployment. The results show an important similarity between the simulated and real executions.

1. Introduction

MapReduce is a parallel programming model, proposed by Google, to process large datasets. It was inspired by the Map and Reduce primitives from functional languages, like Lisp and Haskell. This approach was adopted because, as observed in real data-intensive applications, most of the algorithms needed to group input data fragments into an identifying key, and afterwards, process all data associated with the same key [1]. Thus, the main task of the programmer is to implement these two functions, describing how data mapping and reducing should be performed, while the framework provides mechanisms for communication between nodes, task and data distribution, among other common activities of distributed environments.

The original MapReduce implementation, proposed in 2004 by Google, was designed to run on large clusters of homogeneous COTS (Commercial off-the-shelf) machines. However, during the last years, MapReduce was ported to different types of environments, like desktop grids [2], volunteer computing grids [3], cloud computing [4,5] and others [6,7].

In order to obtain a good performance in these new environments, it is necessary to adapt several framework mechanisms and policies, such as job and task scheduling algorithms, data placement and speculative execution policies, among improvements in other MapReduce features. As an example, [8] studied the impact of speculative tasks, which are re-execution of slow tasks, when running MapReduce on heterogeneous environments. As a result, they also proposed a new task scheduler algorithm, called LATE [8], to improve MapReduce's performance in that scenario.

However, testing these new solutions may have high costs, and requires great effort. When testing new approaches, the researcher must consider issues of scalability, for example, which requires a very large and complex infrastructure. To evaluate a new task scheduler, it is necessary to observe its behavior when the tasks are long, short, or when they have varying

* Corresponding author.
E-mail addresses: wkolberg@inf.ufrgs.br (W. Kolberg), pbmarcos@inf.ufrgs.br (Pedro de B. Marcos), jcsanjos@inf.ufrgs.br (J.C.S. Anjos), aksmiyazaki@inf.ufrgs.br (A.K.S. Miyazaki), geyer@inf.ufrgs.br (C.R. Geyer), luciana.arantes@lip6.fr (L.B. Arantes).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.02.001

sizes. Many other examples could be presented, and like the ones that were exposed, require time, effort, and have high costs.

In order to make these processes of evaluation and testing more efficient, we propose the MRSG (MapReduce over SimGrid) simulator. MRSG provides an application programming interface (API) on top of a simplified and high-level model, which reproduces the behavior of a MapReduce deployment. Through the API, theoretical algorithms can be quickly translated into executable code and analyzed in various simulated environments, and with different parameters. MRSG also helps in creating new solutions, because it allows researchers to perform simulated deployments of their ideas, and to identify possible design errors earlier, before an actual implementation.

To evaluate MRSG we performed experiments running the Hadoop [9] implementation of MapReduce over the Grid'5000, a French scientific grid. After, we simulated the same environments on MRSG and compared the behaviors of each execution. The results show an important similarity between the simulations and the real deployments.

The rest of the paper is organized as follows. Section 2 shows a technology overview around MapReduce. Section 3 shows the MRSG architecture and its functionalities. Section 4 presents the evaluation of the simulator. Section 5 presents a discussion regarding related work. Finally, in Section 6 we expose our conclusions.

2. MapReduce overview

The MapReduce framework was developed to execute data-intensive applications. Therefore, the input data may have hundreds of gigabytes or even petabytes. To execute these applications a master/worker approach was proposed. In it, the *master* node controls task distribution and scheduling, and *workers* perform the computation of map and reduce functions. The MapReduce data flow is illustrated on Fig. 1.

Internally, MapReduce has three main phases: (1) The Map phase reads the data from the distributed file system and calls the user map function to emit (key, value) pairs as intermediate results. (2) In the shuffle phase, the map nodes sort their output keys in partitions, that are then pulled by the reduce nodes. Therefore, each reduce task will process the keys that belong to a specific partition. When all data transfers are done, the reduce nodes execute a sort to merge the data pulled from the map nodes. (3) Finally, the Reduce phase then calls the user's reduce function and writes the output back into the DFS.

Although the MapReduce model is straightforward and easy to comprehend, there is a more complex system behind it. It includes a distributed file system, a fault tolerance system, a task scheduler and a control mechanism. All these components affect the execution and data flow of the system and, therefore, must be considered when simulating MapReduce deployments.

We now give a brief overview of these mechanisms, in order to introduce the main aspects of MapReduce that are simulated in MRSG. In Section 3 we present the details of how they were implemented in the simulator.

2.1. Distributed file system

The distributed file system stores the input and output data of MapReduce jobs. Its architecture is very similar to the MapReduce framework itself. It has a *master* node, which handles the file system metadata and access control, and several *chunkservers*, which actually store the file blocks (called *chunks*). Google's implementation is called GFS (Google File System) [10].

An important relation between the DFS and the MapReduce deployment is that the worker nodes are also chunkservers. Therefore, to avoid network transfer, the master will consider the location of chunks when assigning tasks to workers. In other words, the master will try to schedule a task, to a worker, that will process a chunk stored locally in that worker. This property is known as *locality* [1].

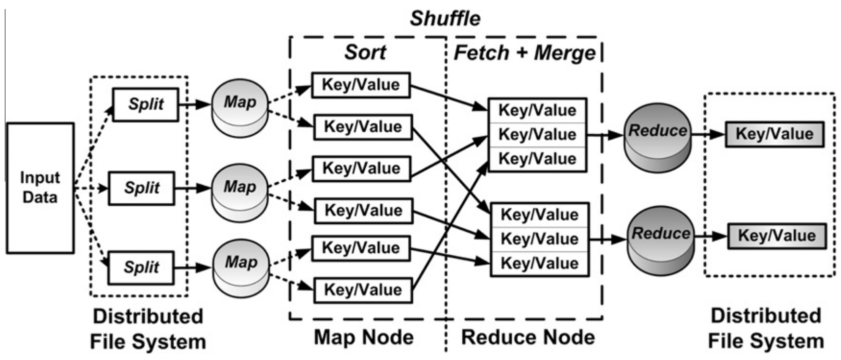


Fig. 1. MapReduce data-flow model adapted from [9].

Other aspects of the DFS are worth mentioning, such as: the replication of chunks, which performs an important role in fault tolerance and data availability; and the data balancing between chunkservers, that tries to keep the same percentage of local storage usage in each node.

2.2. Task scheduling

MapReduce's task scheduler is mainly based on data locality. This means that, for map tasks, it will try to avoid transferring input chunks over the network, as previously explained. However, the input for reduce tasks is spread among the workers that processed map tasks, therefore, there is no data locality in the reduce phase. Moreover, reduce tasks are scheduled before the map phase end. This allows the reduce tasks to gradually download the intermediary pairs during the map phase. We will refer to this property as *data prefetching*.

In addition to the map and reduce tasks, there are also the *backup tasks* [1] (also known as *speculative tasks* [9]). These are replicas of slow tasks, that could delay the whole job. It is important to notice that these backup tasks are scheduled only at the end of the map and reduce phases.

2.3. Job control and fault tolerance

In order to control the progress of jobs, and the status of workers, MapReduce uses a *heartbeat* mechanism [9]. This means that workers communicate periodically with the master node, sending information regarding availability and the progress of assigned tasks [1].

Regarding fault tolerance, MapReduce uses re-execution as its main resource [1]. When a task fails, it is re-scheduled to another worker, for example. Moreover, when a mapper node fails, the tasks processed by the worker are re-executed, since all temporary results are stored locally.

3. MRSG simulator

In this section we present the main aspects of MRSG,[1] including its goals, the technology behind it, and how we reproduce MapReduce's components in the simulator. Also, we introduce MRSG's usage through simplified API examples.

As its main goal, the simulator aims to facilitate research on the behavior of MapReduce platforms, and possible changes in the technology. To that end, MRSG seeks to provide: a complete API to translate theoretical algorithms, such as task scheduling and data distribution, into executable code; ease of change and test of different system configurations; and the possibility to evaluate solutions on a large scale, since no real infrastructure is required.

Due to the impact of Google's MapReduce version, an open-source alternative, called Hadoop [9], was created. Also, a new distributed file system, named HDFS (Hadoop Distributed File System) [11], was implemented. Both follow strictly the designs of Google's MapReduce and GFS, respectively, and are maintained by the Apache Software Foundation. Since Google's MapReduce source-code is not available, we used the Hadoop implementation as guideline to develop the MRSG simulator.

3.1. Simulation core

MRSG is developed on top of SimGrid, which is a simulation-based framework for evaluating cluster, grid and P2P (peer-to-peer) algorithms and heuristics [12]. Its goal is to provide a generic evaluation tool for large-scale distributed computing, and to make experiments on modern distributed platforms feasible and reproducible [12].

SimGrid was chosen as the basis to our simulator because it provides a compromise between execution speed and simulation accuracy [12]. Also, it scales better in comparison to other grid tools, which is one of the goals of MRSG.

As illustrated in Fig. 2, SimGrid is responsible for the simulation of all network communication and task processing in our implementation. Hence, MRSG reproduces only the behavior of the MapReduce platform, invoking SimGrid operations whenever a network transfer or task processing must be performed, without modifying SimGrid's source code.

The communication between MRSG and SimGrid is achieved through the use of MSG, one of the many application programming interfaces provided by SimGrid. The MSG interface is the most widely used API of SimGrid, and it proved to be perfectly usable for the study of scheduling algorithms and other contexts, such as desktop grids [12]. MRSG is developed in the C language, and therefore it uses the C version of the MSG interface, which provides functions very similar to a real message passing implementation, but with a simplified simulation model.

SimGrid is also responsible by the construction of the environment and the platform description. Hence, users are able to describe the distributed environment through a XML file. The description may contain several components, such as nodes, network links, cluster descriptions, among others. Moreover, each of these components have a series of characteristics that can also be defined. A node, for example, will have properties such as computation power, number of cores, and so forth.

[1] Source code available at https://github.com/MRSG/MRSG.

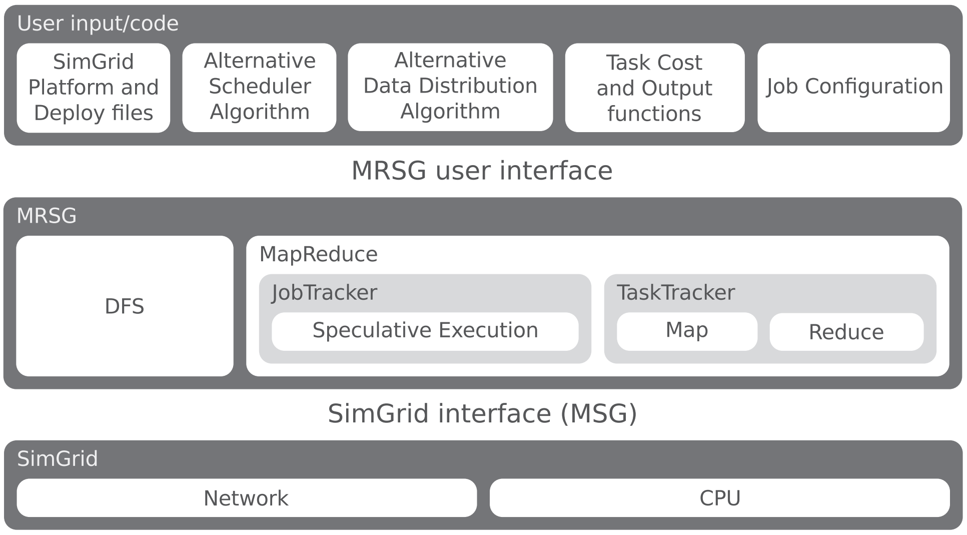


Fig. 2. MRSG architecture.

In the following we describe the MapReduce components and features developed on top of SimGrid, how they were modeled, and which abstractions were made. Also, we provide examples of how the end user may interact and modify specific components in the simulator.

3.2. Job control and fault tolerance

Following the Hadoop implementation, MRSG uses a heartbeat mechanism to control the job execution. The master node is implemented with a single SimGrid process, which listens for heartbeat signals, and schedules tasks according to workers' availability information.

The workers, however, run multiple processes. They are initialized with three fixed processes: (1) The heartbeat process, which sends periodic signals to the master node, indicating the amount of free slots for processing map and reduce tasks. (2) A process that transfer data, including DFS chunks and the intermediary pairs. (3) The process that waits for task assignments, and spawn sub-processes to perform the computation.

MRSG does not implement any fault tolerance for MapReduce, hence, in its current version, the simulator is not able to handle faults nor volatile workers. This feature is intended as future work.

3.3. Distributed file system

In MRSG, the distributed file system is abstracted to a matrix that maps chunks to nodes. This way, the master node knows where each chunk is placed, exactly as it happens in the real implementation. Moreover, each chunk can be associated to more than one node, allowing MRSG to simulate chunk replicas. The dimensions of the matrix are defined by the amount of input chunks n, and the amount of worker nodes m. Therefore, each line C_i represents a chunk with identifier i, and the each column W_j represents a worker with identifier j. The matrix is filled with the values one or zero to indicate if the chunk is stored or not in a specific node. The representation below illustrates the DFS matrix in the simulator.

$$\begin{array}{ccccc}
 & W_0 & W_1 & W_2 & W_{m-1} \\
C_0 & 1 & 0 & 0 & 1 \\
C_1 & 0 & 1 & 0 & 0 \\
C_2 & 1 & 0 & 0 & 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
C_{n-1} & 0 & 0 & 1 & 0
\end{array}$$

This information regarding chunk placement, provided by the matrix, is used by MRSG to implement the locality property of the task scheduler, and to indicate from where data should be downloaded when a worker receives a non-local map task.

Regarding configuration parameters for the distributed file system, users should indicate the amount of input chunks for a job, and the amount of chunk replicas desired. A default value of three replicas is assumed, following Hadoop's default configuration. These parameters are defined in a job configuration file required by MRSG.

Through the API, users can also define a function to implement different DFS data distribution algorithms. This function receives as parameters the DFS matrix, the number of input chunks, the number of workers, and the number of replicas in the configuration. Therefore, the user function should fill the structure using any desired policy. The following pseudo-code illustrates this situation.

```
function dfs (dfs_matrix, chunks, workers, replicas)
    for each c in chunks:
        for each r in replicas:
            w = choose_worker ()
            dfs_matrix[c][w] = 1
end function
```

The implementation of a data distribution function is optional to users, since MRSG already has a built-in default algorithm based on Hadoop's distribution. It performs a uniform distribution of chunks, and replicas, since it assumes the same amount of storage for each node. Hence, the simulation of heterogeneous storage capacities could be achieved through this API function.

It is important to verify that the chunk replication factor can be ignored by the user defined function. Hence, it is feasible to simulate more complex scenarios, such as variable chunk replication, which is a possibility in the Hadoop framework [9].

3.4. Task scheduling

The built-in task scheduler in MRSG follows the locality principle described in Section 2. Hence, when the master node receives a heartbeat from a worker, and it verifies available slots for map processing, it will try to schedule a task according to the following criteria:

1. Unassigned task that processes a chunk stored locally in the worker;
2. Unassigned task that is stored in another worker;
3. Speculative task that processes a local chunk;
4. Speculative task with non-local input.

A reduce task, however, presents no locality since its input is spread among the workers that processed the map tasks. Therefore, when assigning reduce tasks, the scheduler only distinguishes between unassigned and speculative tasks. In both map and reduce phases, a speculative task is only scheduled when all regular tasks were already processed or assigned to other workers.

Furthermore, based on the scheduled task, all necessary data transfer are handled by the simulator. In the map phase, MRSG knows which chunk is associated to a task, and where it is stored, and therefore the simulator generates the data requests automatically. The same happens in the reduce phase, however, the information regarding the input for a reduce task is defined by the user, through an API function (Section 3.5). Moreover, MRSG simulates the data prefetching for reduce tasks, as described in Section 2.2.

To develop a new scheduling algorithm, users can implement a function that receives the target worker and the MapReduce phase as parameters. The function should then return a unique task identifier, indicating which map or reduce task should be assigned to the worker. It is important to notice that, for all alternative algorithms, users can call MRSG predefined functions to obtain details about the simulated environment and its properties.

```
function scheduler (phase, worker_id)
    if phase == MAP:
        return choose_map_task (worker_id)
    if phase == REDUCE:
        return choose_reduce_task (worker_id)
end function
```

3.5. Task cost and output data

Since tasks can have variable costs in a MapReduce job, MRSG users can describe task costs through a function, rather than a single value. The user function receives the task identifier and the task phase as parameters. The function should then return the task cost in MFlops, as required by SimGrid. This allows the users to use several task cost distributions when evaluating new schedulers or policies. It is also possible to read the costs from a log file obtained from a real MapReduce job, allowing MRSG to reproduce real experiments.

In MRSG, the task cost represents all local operations performed by a MapReduce task. As explained in Section 2, map tasks sort the intermediary data into partitions and spill them to disk. Similarly, reduce tasks perform a merge operation over the fetched data. Therefore, all these steps, excluding the data transfers, are considered to be embedded in the SimGrid MFlops returned by the task-cost user function. This is because we are interested mainly in tasks' makespans, and how they

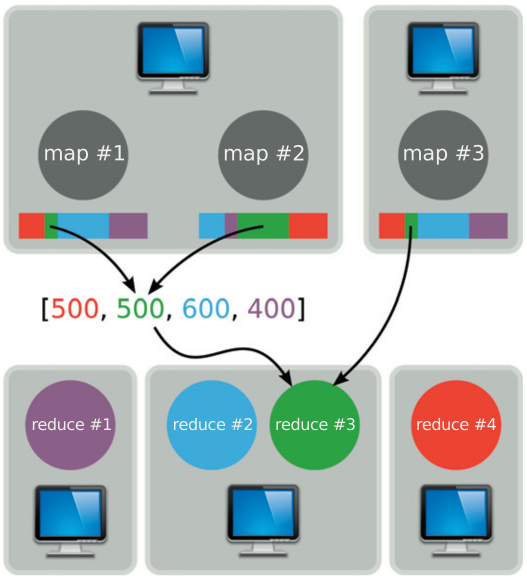


Fig. 3. Intermediary data in MRSG.

affect the distributed solutions developed with the simulator. Hence, the current version of MRSG does not simulate low-level local operations performed by the computing nodes, and all disk I/O overhead should be embedded in the task cost.

As an example of task cost setting, in the pseudo-code below, the function uses a skewed distribution for map tasks and a bimodal distribution for reduce tasks. Notice that these distribution functions are user defined, and are currently not provided by the MRSG programming interface. A real example for the task cost function is presented in Appendix A.

```
function task_cost (phase, task_id, worker_id)
    if phase == MAP:
        return skwed_cost_distribution (task_id)
    if phase == REDUCE:
        return bimodal_cost_distribution (task_id)
end function
```

Another important aspect related to the MapReduce data-flow is the generation of intermediary data. Since MRSG does not work with a real input, nor real applications to process the data, we use a simplified model based on the amount of data generated to each reduce partition. This is achieved through a user defined function. Therefore, the user can specify how much data each map task emits to each reduce task. Fig. 3 illustrates the intermediary data mechanism in MRSG.

The fact that MRSG provides separate interfaces for task cost and data generation, makes it highly flexible, since it is possible to simulate unusual MapReduce applications that do not have their complexity attached to the amount of data processed. Though, users can obviously use the amount of input data in their task cost functions, in order to simulate typical MapReduce applications.

4. MRSG evaluation

To evaluate our simulator we performed several tests over the Grid'5000, a French scientific grid. We used the Hadoop MapReduce implementation and collected its results. After, we simulated the same experiments on the MRSG simulator and compared the results to assess if the MapReduce behavior was reproduced correctly. The experiments were made with three MapReduce applications and were repeated 30 times.

4.1. Applications

Below we describe the MapReduce applications used in our tests, and the results of the comparison between their execution in the real environment and their simulation. In all tests we used two map slots, two reduce slots and three replicas for each chunk.

4.1.1. Log Filter

This application receives a file containing traces with machine availability information from a volunteer computing environment (one line for each period that a machine became available) and produces an output with one line per machine. This line contains the machine ID and its average availability time.

The map function processes each line and emits a pair (key, value), where the key is the machine ID and the value is a tuple (start, end) containing the start and the end time of the availability period. Each reduce function receives all pairs (key, value) associated with a specific machine and calculates the mean time of availability.

To analyze the execution of this application, we used two distinct configurations (Table 1). The results for both executions are presented in Fig. 4, that shows the amount of active map and reduce tasks throughout the MapReduce job, for each execution.

The first configuration presents a visible difference in the number of active tasks at the end of the map phase. This happened in the real execution because the computation power was not exactly the same for all nodes, despite the fact that they had the same configuration, generating speculative tasks in the Grid'5000. Since the SimGrid platform was configured with the exact same computation power in all nodes, all tasks were progressing at the average rate in the simulation. This same behavior can be observed in the Word Count execution (Fig. 6).

However, the results for the second configuration of the Log Filter application (Fig. 4), and both executions of Tera Sort (Fig. 5), are similar in the number of active tasks, as well as in the overall behavior. The speculative execution mechanism did not affect these tests since a speculative task, in Hadoop and MRSG, will only be assigned after the slow task has ran for at least one minute.

4.1.2. Tera Sort

Another application tested is Tera Sort. Its input is a file containing numbers to sort, and the output is a file with the sorted content. The map function receives a chunk of data and emits one key for each element. The value of the key is the value of the element. Hence, the keys are sorted in the partitioner step. After, the reduce function emits the sorted numbers.

The Tera Sort application was also compared using two different configurations, indicated in Table 2. The results are presented in Fig. 5.

4.1.3. Word Count

The third tested application is Word Count. The application receives a text file and produces as output a count of occurrences of each word. For each line, the map function emits a pair (key, value), where the key is one word and the value is constant 1. The reduce function sums all values of each key and emits a list of words and the number of times that each one appears in the text.

To evaluate the behavior of the Word Count application, we performed an experiment with the configuration presented in Table 3. In this test we used one reduce task, with only 0.01% of map output data relative to its input. To achieve this ratio in the real execution, we concatenated a 150 MB text file several times until we reached the total input size. Thus, using a combine function, we were able to obtain a high degree of reduction. The numbers of active map and reduce functions, are presented in Fig. 6.

As previously explained for the Log Filter application, the main difference occurred due to the speculative execution on the real environment, which was not triggered in the simulation due to the platform description.

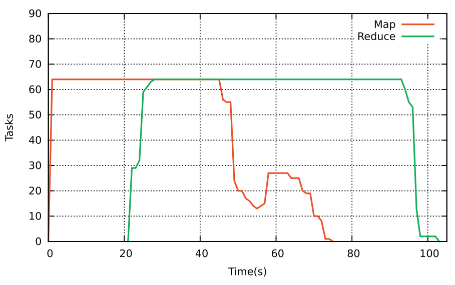
4.2. Result analysis

The results obtained in Section 4.1 indicate an important level of similarity between the real executions and the simulations. However, as expected, the simulations presented some minor variations in comparison to the Hadoop executions. Also, the speculative execution mechanism generated additional tasks at the end of the map phase in specific cases. This shows the importance of considering backup tasks when simulating MapReduce.

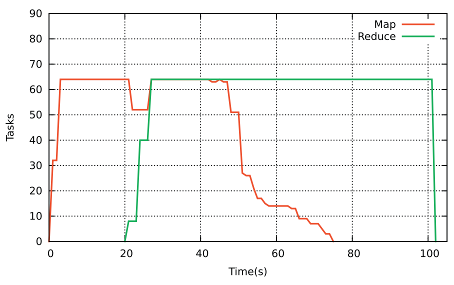
In order to demonstrate the speculative execution in MRSG, we present an additional test, using an heterogeneous platform with tasks longer than one minute. This experiment used eight workers in total, which were equally divided in two groups of node configurations. The nodes in the second group had 86% of the computing power of the nodes in the first

Table 1
Configuration used on the Grid'5000 for the Log Filter execution.

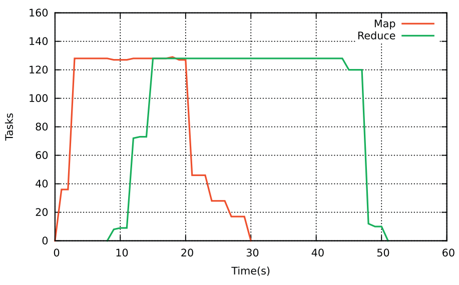
	Config. 1	Config. 2
Nodes	32	64
Cores	64	512
CPU	Intel Xeon EM64T 3.0 GHz	Intel Xeon L5420 2.5 GHz
Memory	2 GB	16 GB
Bandwidth	1 Gb/s	1 Gb/s
Input	8.8 GB	17.6 GB
Maps	141	282
Reduces	64	128
Map output	3.6 GB	7.2 GB



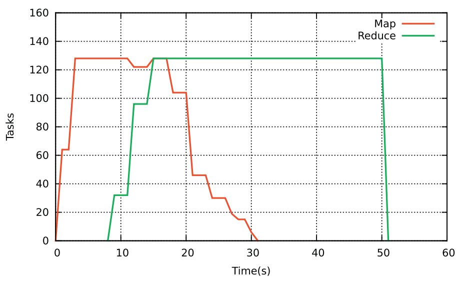
(a) Grid’5000 - Config. 1



(b) MRSG - Config. 1

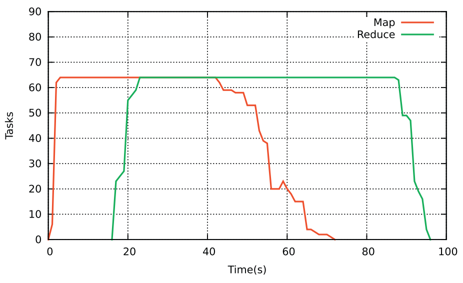


(c) Grid’5000 - Config. 2

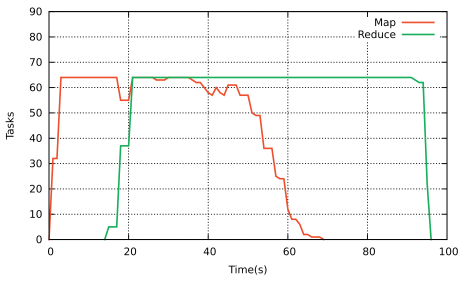


(d) MRSG - Config. 2

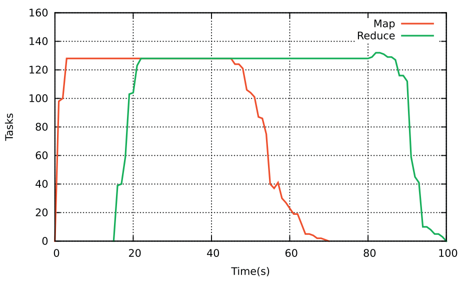
Fig. 4. Comparison between Grid’5000 and MRSG for the Log Filter application.



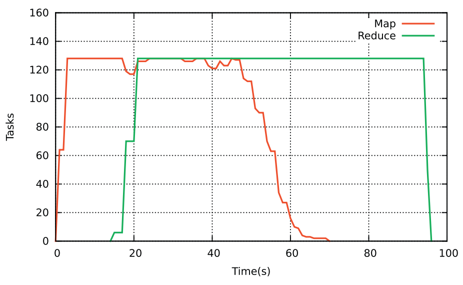
(a) Grid’5000 - Config 1



(b) MRSG - Config 1



(c) Grid’5000 - Config 2



(d) MRSG - Config 2

Fig. 5. Comparison between Grid’5000 and MRSG for the Tera Sort application.

group. The results can be observed in Fig. 7, and it is possible to identify the same behavior presented by the real executions where speculative tasks were scheduled.

Table 2
Configuration used on the Grid'5000 for the Tera Sort execution.

	Config. 1	Config. 2
Nodes	32	64
Cores	64	128
CPU	AMD Opteron 246 2.0 GHz	AMD Opteron 246 2.0 GHz
Memory	2 GB	2 GB
Bandwidth	1 Gb/s	1 Gb/s
Input	12 GB	24 GB
Maps	192	384
Reduces	64	128
Map output	12 GB	24 GB

Table 3
Configuration used on the Grid'5000 for the Word Count execution.

	Config.
Nodes	32
Cores	64
CPU	AMD Opteron 250
Memory	2 GB
Bandwidth	Gigabit ethernet
Input	8.1 GB
Maps	128
Reduces	1
Map output	83 MB

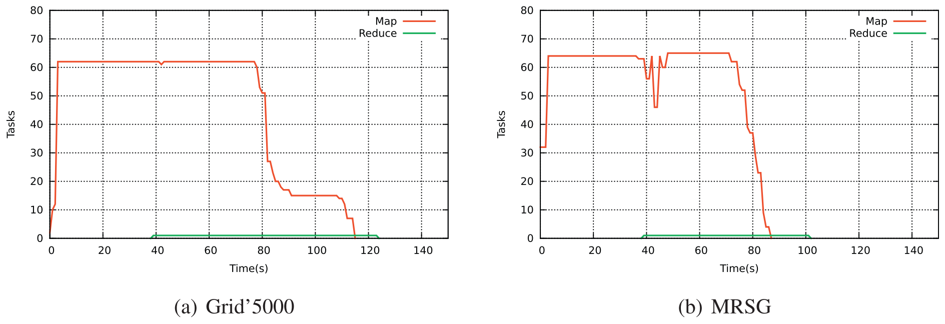


Fig. 6. Comparison between Grid'5000 and MRSG for the Word Count application.

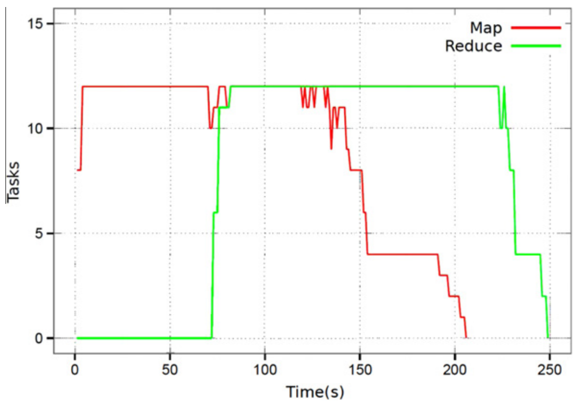


Fig. 7. Heterogeneous platform simulation in MRSG.

With the results presented in this section, it is possible to conclude that the correct modeling of task costs, input sizes and platform have an important impact in the simulation quality. This is specially relevant for validations, when the user must compare the simulated execution with a real environment. Though, one of the goals of MRSG is to allow the users to test their solutions varying these parameters, through the definition of synthetic workloads.

5. Related work

Cardona et al. [13] present a simulation, based on GridSim, to evaluate a new MapReduce scheduler. It focuses on simulating the map and reduce functions as well as the distributed file system. The goal of the work was to evaluate a new solution for data mining using MapReduce and machine learning.

Another simulator is MRPerf [14], which goal is to facilitate the study of MapReduce. It captures various aspects of the model setup and uses this information to predict the expected application performance. Although based on Hadoop, MRPerf does not simulate important aspects of the platform. Its implementation is able to simulate only a single storage device per node and allows only one replica for each chunk. It also does not implement some MapReduce optimizations, such as the speculative execution [14].

MRSim is another simulator for MapReduce, and it is based on GridSim. Its features include the data split and data replication at local rack level [15]. However, real execution environments have hundreds or thousands of nodes distributed over many racks. Also, no interface is provided to modify the framework algorithms, such as task scheduling and data distribution.

In MRSG it is possible to change the data replication, enable or disable the speculative execution, and use variable task cost and intermediary data. Also, MRSG provides an API to easily modify MapReduce mechanisms, as explained in Section 3.

Table 4
Comparison between simulations of the TeraSort application.

	Real	MRSG	MRPerf	MRSim
Map phase	72.015 s	60.155 s	107.044 s	–
Reduce phase	79.122 s	67.345 s	113.685 s	–
Job total	95.518 s	85.370 s	114.684 s	–
Simulation	–	0 m 0.401 s	3 m 37.724 s	+12 h

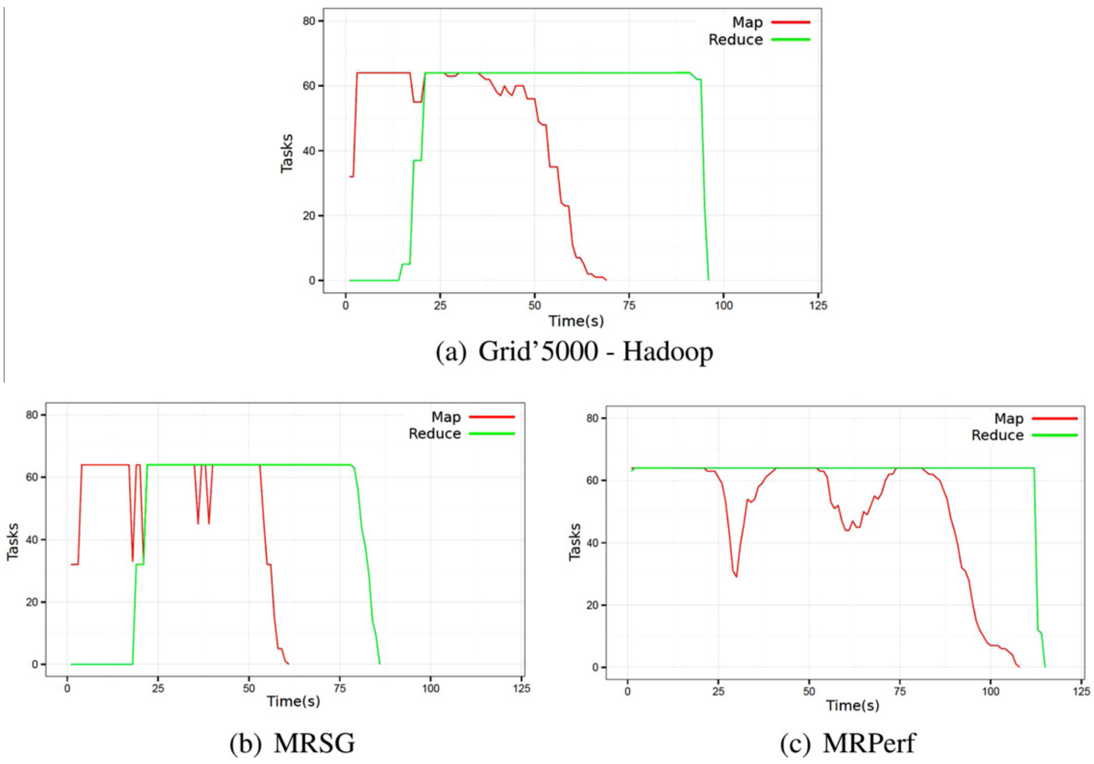


Fig. 8. Comparison between simulators.

Another important difference is the scalability of the simulation frameworks used by each tool. MRSG uses SimGrid, which was shown to be more scalable than ns2 and GridSim, which are used by MRPerf and MRSim respectively [12].

In Table 4 we present a comparison between simulations of the Tera Sort application. The experiments were executed with MRSG, MRPerf, and MRSim, based on the configuration from Table 2 (Config. 1).

In our experiments, when increasing the amount of reduce tasks, MRSim did not finish the jobs. The tests for this simulator were aborted after 12 h of execution.

MRPerf returned a job duration with approximately 19 s of difference to the real execution, and did not presented appropriate duration regarding the map and reduce phases. This can be observe in Fig. 8. It is also possible to notice that the reduce phase starts the data prefetching sooner than expected.

In this particular experiment, we used an average task size for MRSG, since the other simulators took a single value to configure task costs. Hence, the map phase in MRSG was faster than the real execution, which affects also the reduce phase, since the data prefetching is also shorter. However, as presented in Fig. 4, it is possible to make MRSG behave more closely to Hadoop when using a task cost function, for the same configuration. MRSG was also faster to simulate the experiment, due to its simplified model, and the use of SimGrid.

6. Conclusion

In this paper we presented MRSG, a high-level simulator for MapReduce. The tool was developed on top of the SimGrid framework, and provides functionalities absent in other simulators, such as the speculative execution mechanism and data replication. Also, MRSG allows users to define task costs and intermediary data through API functions, rather then a single value. These features make the simulator's behavior closer to a real MapReduce deployment.

The evaluation results showed that the workload and environment descriptions are highly important to obtain an accurate simulation, when trying to reproduce a real Hadoop execution. The experiments also indicated that MRSG is able to reproduce real executions of MapReduce accurately, as well as the viability of using MRSG to simulate MapReduce with different cluster configurations.

In a future work, we intend to incorporate new features in MRSG, including the ability to handle faults and volatile environments. Also, we intend to extend the MRSG API to allow users to develop new algorithms and solutions to deal with these faults.

Acknowledgments

This work was partially supported by FAPERGS/CNPq 008/2009 Project Research: "GREEN-GRID: Sustainable High Performance Computing" and MCT-CNPq – 70/2009 – PGAEST. 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).

Appendix A. API example

```
#include <stdio.h>
#include <string.h>
#include <msg/msg.h>
#include <mrsg.h>
/* User cost function that reads the duration of a
    real G5K task, and convert it to a cost in FLOPs. */
double my_task_cost_function (int phase, size_t tid, size_t wid)
{
    FILE *cost_file = NULL;
    char file_name [ 32 ];
    double cost;
    double duration;
    int line;

    switch (phase)
    {
    case MAP:
        strcpy (file_name, "map-cost");
        break;
```

(continued on next page)

```
    case REDUCE:
        strcpy (file_name, "reduce-cost");
        break;
    }

    cost_file = fopen (file_name, "r");

    for (line = 0; line < tid; line++)
    {
    fscanf (cost_file, "%*f");
    }
    fscanf (cost_file, "%lf", &duration);

    cost = duration * MSG_get_host_speed (worker_hosts[ wid ]);

    fclose (cost_file);

    return cost;
}

int main (int argc, char* argv [ ])
{
    MRSG_init ();

    /* Set the user defined cost function. */
    MRSG_set_task_cost_f (my_task_cost_function);

    MRSG_main ("test/platwordc.xml",
               "test/d-platwordc.xml",
               "test/mrsg.conf");

    return 0;
}
```

References

[1] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in: OSDI '04, San Francisco, CA, 2004, pp. 137–150.
[2] B. Tang, M. Moca, S. Chevalier, H. He, G. Fedak, in: 2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC '10, IEEE, Washington, DC, USA, 2010, pp. 193–200.
[3] F. Costa, L. Silva, M. Dahlin, in: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Ph.d. Forum, IEEE, 2011, pp. 1855–1862.
[4] H. Liu, D. Orban, in: Proceedings of IEEE International Symposium on Cluster Computing and the Grid 2011.
[5] T. Gunarathne, T.-L. Wu, J. Qiu, G. Fox, in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, IEEE, 2010, pp. 565–572.
[6] W. Fang, B. He, Q. Luo, N.K. Govindaraju, IEEE Transactions on Parallel and Distributed Systems 22 (2011) 608–620.
[7] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, in: 2007 IEEE 13th International Symposium on High Performance Computer Architecture, IEEE, 2007, pp. 13–24.
[8] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, I. Stoica, in: Proceedings of the Eighth USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 29–42.
[9] T. White, Hadoop: The Definitive Guide, third ed., O'Reilly Media, 2012.
[10] S. Ghemawat, H. Gobioff, S.-T. Leung, in: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles – SOSP '03, SOSP '03, vol. 37, ACM Press, New York, NY, USA, 2003, p. 29.
[11] The Hadoop Distributed File System, Yahoo!, IEEE, Sunnyvale, CA, USA, 2010.
[12] H. Casanova, A. Legrand, M. Quinson, in: Tenth International Conference on Computer Modeling and Simulation (uksim 2008), IEEE, 2008, pp. 126–131.
[13] K. Cardona, J. Secretan, M. Georgiopoulos, G. Anagnostopoulos, A grid based system for data mining using MapReduce, Technical Report, AMALTHEA, 2007.
[14] G. Wang, A.R. Butt, P. Pandey, K. Gupta, in: Proceedings of the First ACM Workshop on Large-Scale System and Application Performance – LSAP '09, ACM Press, New York, NY, USA, 2009, p. 19.
[15] S. Hammoud, M. Li, Y. Liu, N. Alham, Z. Liu, in: 2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery, IEEE, 2010, pp. 2993–2997.