Future Generation Computer Systems 42 (2015) 22–35

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

MRA++: Scheduling and data placement on MapReduce for heterogeneous environments

Julio C.S. Anjos[a,*], Iván Carrera[a], Wagner Kolberg[a], Andre Luis Tibola[a], Luciana B. Arantes[b], Claudio R. Geyer[a]

[a] *Federal University of Rio Grande do Sul (UFRGS), Institute of Informatics - PPGC, 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*

HIGHLIGHTS

- MRA++—MapReduce with Adapted Algorithms for Heterogeneous Environments.
- To address the main problems caused by the simplification of the MapReduce model.
- The algorithms will allow the use of data-intensive applications in Internet.
- This proposal suggests a potential value for use on desktop grids.
- A low delay in the setup phase of jobs justifies the use of this algorithms.

ARTICLE INFO

Article history:
Received 25 July 2013
Received in revised form
25 August 2014
Accepted 1 September 2014
Available online 16 September 2014

Keywords:
MapReduce
Distributed systems
Data placement
Data-intensive computing
Scheduling

ABSTRACT

MapReduce has emerged as a popular programming model in the field of data-intensive computing. This is due to its simplistic design, which provides ease of use for programmers, and its framework implementations such as Hadoop, which have been adopted by large business and technology companies. In this paper we make some improvements to the Hadoop MapReduce framework by introducing algorithms that are suitable for heterogeneous environments. The goal is to efficiently perform data-intensive computing in heterogeneous environments. The need for these adaptations derives from the fact that, following the framework design proposed by Google, Hadoop is optimized to run in large homogeneous clusters. Hence we propose MRA++, a new MapReduce framework design that considers the heterogeneity of nodes during data distribution, task scheduling and job control. MRA++establishes a training task to gather information prior to the data distribution. However, we show that the delay introduced in the setup phase is offset by the effectiveness of the mechanisms and algorithms, that achieve performance gains of more than 70% in 10 Mbps networks.

1. Introduction

MapReduce is a programming model proposed by Google in 2004, that was designed to simplify the inverted index construction for web searches [1]. The algorithms used in *MapReduce* allow handling of data-intensive applications, abstractions for parallelism and fault control. Large companies like Facebook, Amazon and IBM use the *MapReduce* model as a tool for Cloud Computing applications through implementing Hadoop, an open source code of *MapReduce*, produced by Apache Software Foundation. Other proposals were developed later, but with a different approach, like Dryad [2], Phoenix [3] and Pig [4].

Data-intensive applications like petroleum extraction simulations, weather forecasting, natural disaster prediction, and biomedical research have to process an increasing amount of data. In view of this, data-intensive applications lead to the need to find new solutions to the problem of how this should be carried out. In *MapReduce*, the applications have to be written in the form of two functions: a *Map* and a *Reduce* function. Tasks are sent to be executed on the machines and thus, the *Map* benefits from knowing the location of the data [5].

The result of the processing produces new intermediate data that will be fed to the *Reduce* task in the next phase. A *Reduce* task

* Corresponding author. Tel.: +55 05134803898.
E-mail addresses: julio.c.s.anjos@gmail.com, jcsanjos@inf.ufrgs.br (J.C.S. Anjos), ivan.carrera@inf.ufrgs.br (I. Carrera), wkolberg@inf.ufrgs.br (W. Kolberg), altibola@inf.ufrgs.br (A.L. Tibola), luciana.arantes@lip6.fr (L.B. Arantes), geyer@inf.ufrgs.br (C.R. Geyer).

http://dx.doi.org/10.1016/j.future.2014.09.001
0167-739X/© 2014 Elsevier B.V. All rights reserved.

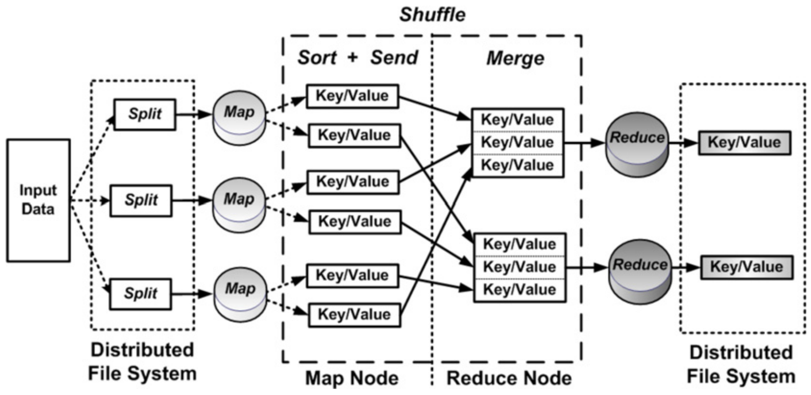


Fig. 1. MapReduce data flowchart model adapted from [6].

does not initiate an activity unless all the *Map* tasks have been completed. In this way, a computing barrier is created to synchronize the tasks that are carried out by a producer and a consumer. At the same time, the *Reduce* tasks do not take advantage of the location of the data, which results in data transfer costs during the time of execution [6].

The implementation was simplified in some ways to facilitate the construction of *MapReduce* in homogeneous clusters; this involved disregarding the cost of data transfer between the machines and adopting equal execution times for the *Map* and *Reduce* phases. However, in a heterogeneous environment, these simplifications cause a load imbalance in the computing capacity of the machines [7].

MRA++ (***M****ap****R****educe* with **A**dapted **A**lgorithms for **H**eterogeneous **E**nvironments), proposed by this paper, aims to address the main problems originating from the simplification of the *MapReduce* model while it is being implemented in clusters. Thus, the developed algorithms allow the use of data-intensive applications in large-scale environments with the use of Internet. Simulation results show that a short delay in the *jobs* setup phase justifies the use of this proposal in heterogeneous environments yielding a gain of nearly 70% in 10 Mbps networks.

This work is structured as follows: Section 2 provides an overview of the *MapReduce* architecture. Section 3 describes the proposed model and its structure. Section 4 sets out the *MapReduce* algorithms that are adjusted to heterogeneous environments. Section 5 describes the methodology, the experiments and the obtained results. Section 6 analyzes related works. The conclusion and suggestions for future work are summarized in Section 7.

2. An overview of MapReduce

The computation of large volumes of data and the efficient use of computational resources with a low execution time, are still matters under discussion in the scientific community [8–12]. *MapReduce* is a programming framework that abstracts the complexity of parallel applications. The management architecture is based on the master/worker model, while a slave-to-slave data exchange requires a P2P model.

The simplified computational model for data handling means the programmer is unaware of the complexity of data distribution and management. There is a considerable degree of complexity because of the large number of data sets, scattered across hundreds or thousands of machines, and the need for lower computing runtime [6]. The *MapReduce* architecture consists of a master machine that manages other worker machines. Fig. 1, adapted from [6], shows the data flow in *MapReduce* in three distinct phases.

The *Map* and *Reduce* phases are handled by the programmer, while the intermediate, called *Shuffle*, is created by the system during the *job* execution. The input data is broken into smaller pieces called *chunks*, that typically have a size of 64 MB. The *job* is divided into several tasks (*Map* and *Reduce*). A number of slots is set to indicate how many tasks can be processed simultaneously by each worker [6].

The master assigns tasks to machines that will run each processing step. The machine that receives a Map task, handles a Map function and emits key/value pairs as intermediate results. Intermediate data produced during the map phase is temporarily stored in the workers' disks. The execution model creates a computational barrier, which allows tasks to be synchronized between producers and consumers. A *Reduce* task does not start its processing until all the *Map* tasks have been completed.

A hash function is applied to the intermediate data produced during the *Map* phase to determine which keys will compose a *Reduce* task. All the pairs combined with these keys are transferred to one machine, during the *Shuffle*, so that they can be processed by a *Reduce* task. After a reduction function has been applied to this data, a new key/value pair is issued. The result is then stored in the distributed file system and thus can be made available to the client who submitted the *job*. The Shuffle phase consists of two steps: one performed on the machine that processes the *Map* task, which sorts the keys and serializes the data. The other is performed after the intermediate data has been sent to the reducer machine, which arranges the received data to allow the keys to be properly grouped and then runs the *Reduce* task [6].

Management mechanisms, data replication and execution control were added to the framework during its implementation. When the execution time of a task is greater than average for the cluster, the machine running the task is characterized as a straggler [5,6]. Thus, the scheduler initiates a speculative task to re-execute those that are taking too long to complete. In addition, a task can be assigned to a machine that does not contain the data in its local disk. In this last case, the machine will have to make a copy of the data before it can perform the task [5].

3. Proposed model

The strategy adopted in this work is to examine three areas of the *MapReduce* implementation: grouping, data distribution and task scheduling. In *MapReduce*, the difference in computational power between machines in a heterogeneous environment causes an unbalanced load. In the original model, a machine characterized as a straggler (slow machine), after the first task distribution, will not receive a new task in free slots.

If a machine is unable to receive tasks that process local data, remote tasks are initiated (this means that it is necessary to copy data before the execution). As a result, the original algorithms cause an increase in the number of unnecessary, speculative and remote tasks in heterogeneous environments. The data must be distributed in accordance with the heterogeneity of the machines to prevent a large increase in execution time. This problem can be overcome by grouping the machines according to their computational capabilities.

The approach adopted to balance the processing load in the *Reduce* phase is similar to the fine partitioning technique proposed by Gufler et al. [13], but adapted to the heterogeneous environment. The difference is in the way the data is available to the machines and the calculation methodology employed for data splitting. The data information for each partition is placed in a global FIFO queue of tasks.

The replication mechanism that is present in the distributed file system, influences the system's performance. In addition to providing a mechanism for fault tolerance, replication allows task launching without the need to copy data over the network at runtime. Thus, replicating the data according to their computational capabilities can also help prevent the initiation of remote tasks that require data copy. However, the number of replicas is limited to a configuration parameter that is defined by the programmer.

3.1. Structure of the MRA++ model

Before the *MapReduce* model could be adapted to a heterogeneous environment, several modifications were required for the existing algorithms. New modules were introduced to support the task scheduler and the data distribution mechanism. Fig. 2 shows the different modules that compose *MRA++*. The blank modules are those that were altered from the original model.

Data division module: this is responsible for dividing the data into *Map* and *Reduce* phases according to the computational capacity of the machines. Data division in *Map* is carried out on top of the distributed file system, and in *Reduce* phases by a process executed in the master node, inside the partitioning management module, and allows a task queue to be created. These queues will be managed by the task scheduling module. The replication module supports the data replication of *Map* phase by means of machine clustering.

Task scheduling module: this is the module that controls the task scheduling in both the *Map* and *Reduce* phases. The control of the task progress is divided into two modules, one for the *Map* phase and another for the *Reduce* phase. The *Remote Task Control* module takes control of remote task execution.

Clustering control module: this is responsible for generating the machine clustering. The machine clustering, along with task times, control the task executions.

Measuring task module: this organizes, controls and distributes data to execute the measuring task. The collected information in this module is vital for the data division and the control of the task execution.

4. Heterogeneous algorithms

Some of the related works shown in Section 6 introduce partial solutions to the problem of employing the *MapReduce* model in heterogeneous environments. Thus, the most important problems that arise from adopting data-intensive applications in heterogeneous environments like clusters having heterogeneous nodes, can be attributed to some of the simplifications in the model, such as the following: optimizing for cluster implementation; task execution time: scheduler for equal-time tasks; data copy: imperceptible cost for copying data in low latency networks; heterogeneity problems—low performance in heterogeneous environments, mostly in slow links.

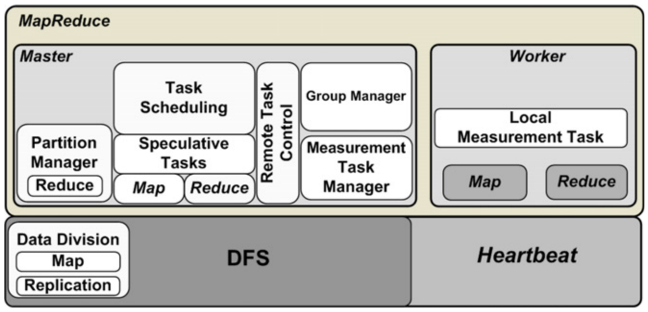


Fig. 2. MRA++ structure model.

4.1. Clustering

Clusters are formed naturally in *MapReduce* when the machines are characterized as having a status of failure, (slow and normal). A *g_dist_bruta* cluster is a group of machines with similar computing capacity that receive the same number of *chunks* to be processed. In *MRA++*, the algorithms for data distribution and clustering estimate the task execution time from measurements of the local task execution time before they are effectively partitioned. These measurements are calculated through a *measurement task*.

The measurement task is created from the *job* submitted by the user. The 1 MB *chunk* is executed in every machine when the system starts before a *job* execution. The size of the *chunk* was defined experimentally from the data distribution time, in order to achieve the lowest possible system overload. The objective is to obtain the execution time for the *Map* and *Reduce* phases in each machine. The execution time will vary according to the type of application and the data input, thus altering the task schedule automatically.

The 1 MB sample *chunk* is extracted from the data itself that is spread across the distributed file system. This data distribution is carried out according to the *k-d Tree* algorithm model, which uses the mean method [14]. When the measuring task execution is over, each slave machine reports to the master: the execution time that is needed for the *Map* phase processing, called t_eMap is measured in seconds; the execution time needed for the *Reduce* phase processing, called $t_eReduce$ is measured in seconds; the computational capacity of the machine is measured in *flops*; the network latency in seconds and the bandwidth in bits per second. The network latency can be measured in each machine through a 1 MB transfer test sent to each machine.

When the execution times for each machine in the measuring task have been obtained, the predicted execution time of the tasks is calculated and the clustering is defined, as shown in Eq. (1). In the case of existing machines where the length or the execution time can be harmful to the *job* execution, this machine will not be given data or tasks to process.

- t_e is the execution time for a *Map* task in the slowest machine;
- $\varepsilon_{h\to r}$ is the transfer time of a *chunk* from a local machine to a remote machine. If the task is executed in the local machine $\varepsilon_{h\to r} = 0$;
- t_eMap is the *Map* task execution time in the local machine;
- C_{total} is the sum of the computing capacities of all the machines in the cluster;
- *Num_chunks* is the total number of *chunks* that have to be distributed.

$$D = \left\lceil \frac{C}{C_{total}} * Num\_chunks \right\rceil \tag{1}$$

The value of a *dist_bruta* is between $\frac{t_e}{(t_eMap_j+\varepsilon_{h\to r})} \leqslant D \leqslant$ *Num_chunks*, thus the machine with the lowest computing capacity must execute at least one *chunk* for each *slot* in a t_e interval to make effective use of the computing resources. The machine with the largest capacity will execute the largest number of *chunks* in this same interval.

The solution proposed in *MRA++* characterizes a smaller number of machines as *stragglers* and executes a larger number of tasks concurrently in a heterogeneous environment. The reason for this is that the scheduler assumes that the progress of the tasks can be measured as an average of the separate executions of each cluster. In view of this, it can be expected that the machines in the same cluster will end their tasks with the same execution time, as in original *MapReduce* model, although this cannot be guaranteed. Thus it is comparable to saying that, in a heterogeneous environment, the machines in the *Map* phase that have the same *dist_bruta* end their tasks at approximately the same execution time.

Algorithm 4.1 introduces the clustering algorithm. In lines 2–6, the starting process for the values occurs, (as shown in Table 1(a)). The first round of Algorithm 4.1, (lines 7–13), redistributes and corrects the number of *chunks*. In the second round of the algorithm, (lines 14–17), the slow machines that undertake the execution time of the whole *job*, are excluded from the processing. In lines 18–21, after the values for D_j have been adjusted, the machines that have the same D_j are identified so that they can form the clusters (*g_dist_bruta*), that will be used to handle the remaining algorithms.

Algorithm 4.1 Clustering Algorithm

```
1.  for $j \leftarrow 1$ to $Num\_hosts$ do
2.      $prev\_exec_j = t_e Map_j * D_j$
3.      $temp\_corr_j = t_e Map_j + prev\_exec_j$
4.      $Maior\_prev\_exec = \mathrm{Max}(prev\_exec_j)$
5.      $menor\_temp\_corr = \mathrm{Min}(temp\_corr_j)$
6.  $Soma\_dist\_bruta = \sum_j D_j$
7.  while $Soma\_dist\_bruta \neq Num\_chunks$ do
8.      if $Soma\_dist\_bruta > Num\_chunks$ then
9.          $D_{Host\_Maior\_prev\_exec} = D_{Host\_Maior\_prev\_exec} - 1$
10.     Recalculates the Value $Maior\_prev\_exec$ and $D$
11.     if $Soma\_dist\_bruta < Num\_chunks$ then
12.         $D_{Host\_menor\_temp\_corr} = D_{Host\_menor\_temp\_corr} + 1$
13.     Recalculates the Value $Maior\_prev\_exec$ and $D$
14. while $\lfloor Maior\_prev\_exec - menor\_temp\_corr \rfloor > 0$ do
15.     $D_{Host\_Maior\_prev\_exec} = D_{Host\_Maior\_prev\_exec} - 1$
16.     $D_{Host\_menor\_temp\_corr} = D_{Host\_menor\_temp\_corr} + 1$
17.     Recalculates the Value $Maior\_prev\_exec$ and $menor\_temp\_corr$
18. for $j \leftarrow 1$ to $Num\_hosts$ do
19.     while $i < Num\_hosts$ do
20.         if $D_{Host_j} = D_{Host_i}$ then
21.             $g(D_k) = \{host_j, host_i, host_{i+1}, ..., host_n\}$ {Creates Clusters $g\_dist\_bruta$}
```

Algorithm 4.1 has two rounds comprising a group of 5 machines with processing capacities, as described in the Capac (Mflops) column in Table 1. After the information has been collected, the data from Table 1(a) are initialized. In the first round, (Table 1(b)), the total number of *chunks* in D is adjusted, together with the number of real tasks belonging to the *job*. If the sum of the calculated value for D is larger than the number of *chunks*, D will be reduced to one, from the row with the largest value of *prev_exec*. The values of *prev_exec* and *temp_corr* are then recalculated and the values of maximum *prev_exec* and minimum *temp_corr* have to be compared again. The process is repeated until the value of the sum of D is equal to the number of *chunks*.

In the second round, (Table 1(c)), the difference between the largest *prev_exec* and smallest *temp_corr* is checked. If the lowest integer of the difference is larger than one, D will be reduced to one from the row with the largest value of *prev_exec*. The *prev_exec* and *temp_corr* are recalculated in the lines that have been altered, and the maximum *prev_exec* and minimum *temp_corr* are calculated again. The process is repeated until the lowest integer is equal to zero. Thus, after Algorithm 4.1 has been finally executed, depending on the size of the data input, the machines that affect the performance of all the pipelines, will not be given any tasks or data.

4.2. Distributed file system modifications

Where necessary, alterations can be made in the distributed file system to improve the performance of the proposed algorithms. While in the case of the original model, the data is distributed equally between the machines, in *MRA++* these will be distributed in accordance with the processing capacity of the machines. A training *chunk* with 1 MB will be used again to determine the computational power. When re-balancing the file system, it will be necessary to consider both the replica positioning and the computing capacity distribution that each machine that hosts a given *chunk* belongs to. This operation should be carried out frequently to achieve a better performance.

The replicas of a given *chunk* will be positioned in a set of machines belonging to the same *g_dist_bruta*. Supposing the probability of failure in a machine is independent of its *g_dist_bruta*, the reliability of the file system will not be affected. This is because if a *chunk* is unavailable, it will still be necessary for all the replicas to fail at once. However, it could be assumed that these *g_dist_bruta* will group machines from the same period, as newer machines are expected to be faster than previous ones. Thus if it is believed that old ones are less reliable, it can be argued that some chunks are more likely to be lost than others in the new model.

When they are employed by a wide range of users with different *job* profiles, this will lead to a need to define policies that enable distinct performance measurements to be adopted when positioning different files or directories. Some cases can be easily handled; for example when text processing and image processing are executed with different data, or data is stored separately. In scenarios where distinct applications are executed with the same data, (for instance *Sort* and *Wordcount*), performance degradation can be expected as usually the data will not be ideally positioned for both *jobs*. In this work these factors have not been addressed.

4.3. Data division algorithm for Reduce phase

In Algorithm 4.2, the data division is adjusted to the computing capacity of the *Reduce* machines. The reduction of the intermediate data granularity creates a larger number of small tasks. Thus, with a larger number of partitions of intermediate data, there is a greater degree of parallelism in the executions and a reduction in the *job* processing time. The smaller granularity for intermediate data is intended to adjust the execution time required for the tasks, increase performance during the *Reduce* phase in the heterogeneous environment and reduce the effects of the unbalanced intermediate data that is produced during the *Map* phase.

Intermediate keys produced in the *Map* tasks are independent and have intermediate results $I \subseteq \mathbb{K} \times \mathbb{V}$. The I results contain all the intermediate pairs (key/value) produced in the *Map* function. The keys are formed by a data group where $C(k) = (k, v) \in I$. The intermediate results are divided into P key partitions by a *Hash* function. The partitions have one or more data groups with keys of the same index, as in Eq. (2) [13].

$$P(j) = \biguplus_{k \in \mathbb{K}:Hash(k)=j} C(k) \qquad (2)$$

The number of *Reduce* tasks is defined as a function of a granularity factor, called *Fg*, and is calculated from the collected data during the measuring task. The intermediate data is divided by Algorithm 4.2 through the sum of the granularity factors Fg_j of each machine represented in Eq. (3). Thus, for each machine, a value of the division of the execution time of each *Reduce* task by the

Table 1
Example of the execution of the clustering algorithm.

Host	$t_e Map$ (s)	Capac (*Mflops*)	% Capac.	*D*	*prev_exec* (s)	*temp_corr* (s)
(a) Start						
1	0.3450	14,780	0.7503	15	5.1750	5.5200
2	1.9510	2,614	0.1327	3	5.8530	7.8040
3	4.2827	1,190	0.0604	1	4.2827	8.5655
4	9.1269	558	0.0283	1	9.1269	18.2538
5	9.1269	558	0.0283	1	9.1269	18.2538
		19,700		21	9.1269	5.5200
(b) First round						
1	0.3450	14,780	0.7503	15	5.1750	5.5200
2	1.9510	2,614	0.1327	3	5.8530	7.8040
3	4.2827	1,190	0.0604	1	4.2827	8.5655
4	9.1269	558	0.0283	1	9.1269	18.2538
5	9.1269	558	0.0283	0	0	9.1269
		19,700		20	9.1269	5.5200
(c) Second round						
1	0.3450	14,780	0.7503	16	5.520	5.8650
2	1.9510	2,614	0.1327	3	5.8530	7.8040
3	4.2827	1,190	0.0604	1	4.2827	8.5655
4	9.1269	558	0.0283	0	0	9.1269
5	9.1269	558	0.0283	0	0	9.1269
		19,700		20	5.8530	5.8650

fastest *Reduce* task (t_{e_r}), is obtained when the measuring task is executed.

$$Fg_j = \left\lceil \frac{t_e Reduce_j}{t_{e_r}} \right\rceil \tag{3}$$

In the original algorithm, the number of *Reduce* tasks is a parameter defined by the programmer. In Algorithm 4.2 the sum of the granularity factors defines the number of *Reduce* tasks (called *Q_reduces*). The partitions $P_{int_{Host_j}}$ that are created in each machine for a *Hash* function, are FIFO-queued in *Q_reduces*, as described in lines 6 and 7 of Algorithm 4.2. When a machine is free, it receives the location of the stored data so that this queue can be executed.

When a *Hash* function is applied to the intermediate pairs, it divides the data according to the number of *Reduce* tasks. The number of *Reduce* tasks is normally equal to the number of available machines. Thus, several intermediate keys are distributed for the same *Reduce* task, which means that instead of using the number of *Reduce* tasks, the *Hash* must use other kinds of information that reflect a suitable level of granularity.

The number of partitions of the intermediate data will be greater than the number of machines of the cluster. *Q_reduces* is the sum of the *Fg* factors. The result of *Q_reduces* will replace the definition of the number of *Reduce* tasks automatically and will be a larger number of short tasks produced by the *Map* phase. The tasks are queued in master node, with the location of intermediate data in the network and, then, the machines request the processing of *Reduce* tasks.

Algorithm 4.2 *Reduce* Data Division

```
Require: $t_{e_r} = 100$; $t_e Reduce$;
1. while $t_{e_r} > t_e Reduce_i$ do
2.     $t_{e_r} = t_e Reduce_i$ {Find the smallest $t_{e_r}$}
3. for $j \leftarrow 1$ to $Num\_hosts$ do
4.     $Fg_j = \lceil \frac{t_e Reduce_j}{t_{e_r}} \rceil$ {Calculates the granularity factor for each machine.}
5. $Q\_reduces = \sum_1^{Num\_hosts} Fg_j$
6. Executes $funcHash(P_{int_{Host_j}}\{Key, Value\} \bmod Q\_reduce)$ {Data division.}
7. $Fila() \leftarrow (taskReduce, Host_j)$ {Tasks are queued with data location}
```

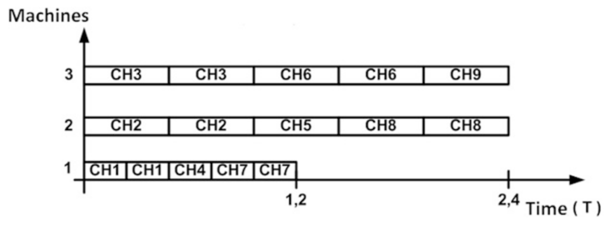
For example, there are 3 selected machines in a cluster to execute a determined *Reduce* function over 15 intermediate keys produced in the *Map* phase. A machine executes the tasks in half the time the other two would do. To simplify this example, the network and the transfer data are not taken into account. Each key/value pair is only represented by CH and the execution time is a generic value T. Fig. 3(a) shows the execution flow of the *Reduce* tasks for the original algorithm and Fig. 3(b) the flow for the *MRA*++.

In the original algorithm, data from intermediate keys is partitioned in 3 *Reduce* tasks, P1 = {CH1, CH1, CH4, CH7, CH7}, P2 = {CH2, CH2, CH5, CH8, CH8} and P3 = {CH3, CH3, CH6, CH6, CH9}, and distributed to machines 1, 2 and 3 respectively.

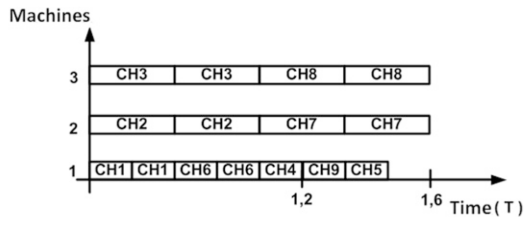
The goal is to distribute a uniform number of keys to each machine. Thus, in Fig. 3(a), for the slower machines (2 and 3), the *Reduce* task execution time is 2, 4 T and the cost of executing each key 0, 48 T. In the faster machine (1) the *Reduce* task execution time is 1, 2 T and the cost of executing each key 0, 24 T. This means that the task scheduler should assign a speculative task for Machine 1 from either one of the slower machines, as it is free and there are no pending tasks.

When *Q_reduces* is calculated in *MRA*++, it results in 5 tasks for the original model; otherwise, it results in 3 tasks. Thus, the intermediate keys can be partitioned by a *Hash* function such as, for example, in P1 = {CH1, CH1, CH6, CH6}, P2 = {CH2, CH2, CH7, CH7}, P3 = {CH3, CH3, CH8, CH8}, P4 = {CH4, CH9} an P5 = {CH5}, and after that they can be queued. Machines 1, 2 and 3 are given tasks by partitions P1, P2 and P3 respectively, as shown in Fig. 3(b). After the fastest Machine 1 has carried out its tasks, it requests more tasks from the master, then, the scheduler sends Task 4, from partition P4. When it has been completed, the machine again requests more tasks from the master, which sends Task 5 to be executed. When this strategy was adopted, the total amount of time of the *Reduce* phase went from 2, 4 T to 1, 6 T, as can be seen in the example of Fig. 3(b). The task executions in the slower and fastest machines tend to be similar.

After the data division, it is necessary to distribute tasks according to the computing capacity of the machines and monitor the progress of their executions. The management of slower machines was divided into two controls, one for the *Map* and another for the *Reduce* phases. The use of different controls is justified because in the *Map*, remote task executions are not required and the number of speculative tasks must be as low as possible. However, as determining the size of intermediate pairs is expensive, in *Reduce* the strategy was to let the faster machines execute more tasks than the slower ones.



(a) Example *Reduce* tasks on original algorithms.



(b) Example of *Reduce* tasks on MRA++ algorithms.

Fig. 3. *Reduce* tasks flowchart.

Table 2
Configuration example for 64 machines.

Configuration	Amount	Configuration	Amount
Intel Xeon 3040 (1860 MHz) 2 GB RAM	1	Pentium Dual-Core E6300 (2800 MHz) 2 GB RAM	2
Intel Celeron E1500 (2200 MHz) 2 GB RAM	2	Pentium Dual-Core E6300 (2800 MHz) 4 GB RAM	1
Dual-Core AMD Opteron 1214 (2200 MHz) 2 GB RAM	1	AMD Phenom II X2 555 (3200 MHz) 4 GB RAM	3
Intel Core i5 430UM (1200 MHz) 4 GB RAM	1	Pentium Dual-Core E5700 (3000 MHz) 2 GB RAM	1
Intel Core2 Duo T7500 (2200 MHz) 2 GB RAM	1	Intel Pentium G6950 (3010 MHz) 2 GB RAM	3
AMD Athlon 64 X2 Dual Core 4800 (2500 MHz) 2 GB RAM	1	Intel Core2 Duo P9700 (2800 MHz) 2 GB RAM	2
Intel Xeon 3050 (2130 MHz) 2 GB RAM	3	Intel Core2 Duo P9700 (2800 MHz) 4 GB RAM	2
Intel Core2 Duo E6540 (2330 MHz) 2 GB RAM	1	Intel Core i3 370M (2400 MHz) 4 GB RAM	2
Intel Core2 Duo E4600 (2400 MHz) 2 GB RAM	2	AMD Athlon X3 440 (3000 MHz) 4 GB RAM	2
Intel Core2 Duo E4600 (2400 MHz) 4 GB RAM	1	AMD Phenom II X3 720 (3200 MHz) 4 GB RAM	2
Dual Core AMD Opteron 185 (2600 MHz) 2 GB RAM	2	Intel Core2 Duo E8500 (3160 MHz) 2 GB RAM	1
Intel Pentium Dual Core E5300 (2616 MHz) 2 GB RAM	1	Intel Celeron G540 (2500 MHz) 2 GB RAM	3
Intel Pentium P6200 (2128 MHz) 2 GB RAM	1	AMD Phenom II X3 720 (3335 MHz) 4 GB RAM	1
Intel Core 2 Duo E6550 (3002 MHz) 2 GB RAM	1	Intel Core i5-2537M (1400 MHz) 4 GB RAM	1
Intel Xeon 3065 (2330 MHz) 2 GB RAM	2	AMD Phenom II X3 720 (3600 MHz) 4 GB RAM	1
Intel Xeon 3065 (2330 MHz) 4 GB RAM	2	Intel Xeon E5335 (2000 MHz) 4 GB RAM	1
Intel Core2 Duo E6750 (2660 MHz) 2 GB RAM	3	Intel Xeon X3210 (2130 MHz) 4 GB RAM	2
AMD Phenom II N640 Dual-Core (2900 MHz) 4 GB RAM	3	AMD Phenom II X3 720 (3816 MHz) 4 GB RAM	1
Intel Core i7 640M (2800 MHz) 4 GB RAM	3	Pentium Dual-Core E6300 (2660 MHz) 2 GB RAM	1

Before assigning a remote task, the scheduler in the *master* node, first checks if it is worth assigning it to another available machine or, if it is best to wait for a current task to end, (through Eq. (4)). The remaining processing time of a task in a machine $task_{Time\_Rest}(Host_h)$ added to the task execution time in the local machine, $New\_taskTime(Host_h)$ should be longer than the time for transferring the data to the remote machine $\varepsilon_{h\to r}$ added to the execution time of the new task in this remote machine $New\_taskTime(Host_r)$. Thus, if time of the first is shorter, the task is not executed and the scheduler waits for the execution to be over; otherwise, the remote task will be executed. Thus, it is natural for a lower number of remote tasks to occur.

$$task_{Time\_Rest}(Host_h) + New\_taskTime(Host_h) > \varepsilon_{h\to r} + New\_taskTime(Host_r) \qquad (4)$$

5. MRA++ evaluation

In this section we will describe the methodology employed in the validation tests for the algorithms and the configurations of the experiments, as well as providing a justification of the choice evaluation parameters. The objective of the tests is to determine if the algorithms of *MRA++* are suitable for heterogeneous environments. *MRA++* is available for downloading in https://github.com/MRSG-MRA/MRA.

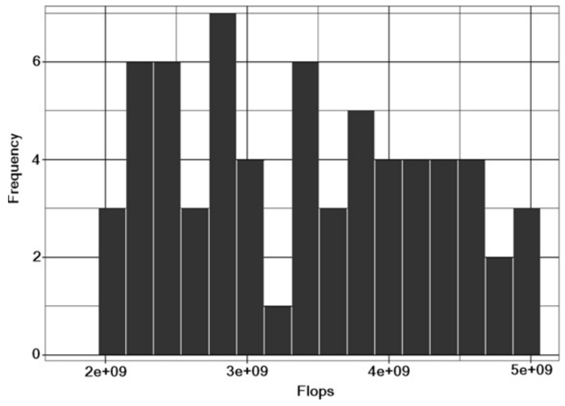
5.1. Methodology

When building real large-scale heterogeneous environments, carrying out measurements with thousands of machines is an extremely complex task owing to a lack of resources. Creating environments for measuring depends on networking devices and software where the implementing standards may vary considerably and involve dealing with different environments and interactions that are very difficult to control [15].

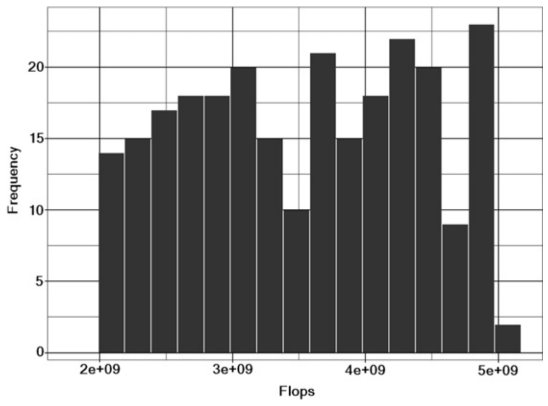
Thus, the deterministic simulator MRSG [16], based on SimGrid, was adopted to help overcome these problems and confirm the selection of the scheduling algorithms. MRSG was built by the GPPD/MR UFRGS research group, and is available for downloading in https://github.com/MRSG/MRSG. In order to validate the proposed approach we resort to discrete-event simulations because they provide means for controlled and repeatable experiments. A reader with access to the simulator's source code would be able to replicate the obtained results. We believe that deploying the proposed solution in a given cluster or grid would add too many variables and would certainly compromise the reproducibility of our work. Moreover, simulation of *MapReduce* applications is a topic of intense interest of the research community [17–20,16].

The experiments test the performance of *MapReduce* by using the new algorithms in heterogeneous environments. The computing capacity of the equipment was generated from a random function written in Python; this varied from 2E+09 to 5E+09 *flops*. These values were chosen based on the values shown in the work of [21]. Table 2 shows an example of a heterogeneous configuration of machines used in an experiment with 64 machines, (other configurations can be found in Tables A.8–A.10). Each machine has a different computing capacity within this range and different generations of processors such as Core I3, Core I7, Intel Xeon, Pentium, AMD Athlon, Celeron and so on. On MRA++ runs it is considered that the distributed file system is balanced with new algorithms, and will thus represent the maximum possible gain.

The experiments use from 32 to 2048 *multi-core* machines. The amount of data for the *Reduce* phase is the same as the data input in *Map* function. The workload is shown in Table 3. The computational cost of each *Map* and *Reduce* task is equal to 1000 *flops*/Byte. The size of the task represents applications of short and typical tasks,

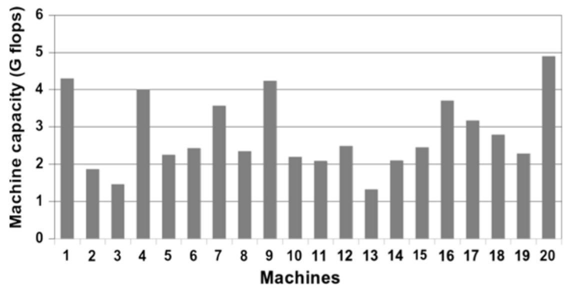


(a) Frequency of machine capacity @ 64 nodes.

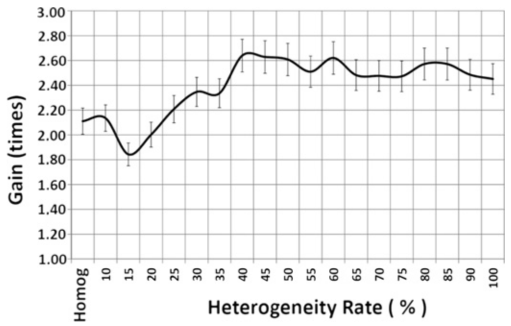


(b) Frequency of machine capacity @ 256 nodes.

Fig. 4. Frequency of machine capacity.



(a) Distribution of machine capacity.



(b) Degree of heterogeneity.

Fig. 5. Heterogeneity.

Table 3
Workload.

Machines	*Chunks*	Data
32	256	16 GB
64	512	32 GB
128	1,024	64 GB
256	2,048	128 GB
512	4,096	256 GB
1024	8,196	512 GB
2048	16,384	1 TB

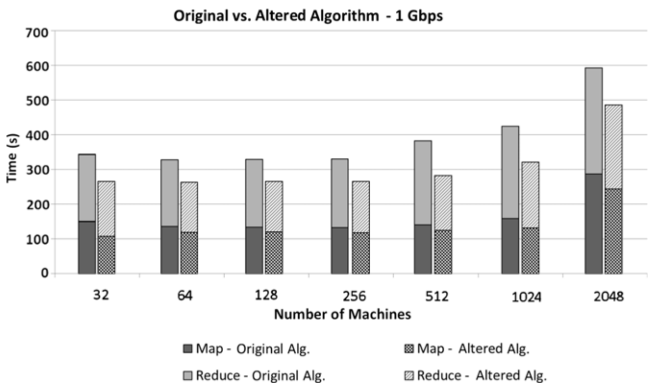


Fig. 6. MapReduce phases—original vs. altered algorithm @ 1 Gbps.

executed by *Google* [5]. The criteria for choosing a network with 10 Mbps speed is that it must reflect the worldwide standards for access via the Internet [22] and 1 Gbps is normally achieved when forming *clusters* in real *MapReduce* environments.

Fig. 4 shows an example of frequency of machine capacity used in an experiment with 64 (Fig. 4(a)) and 256 machines (Fig. 4(b)). Other distributions of the frequency of machine capacity can be found in Fig. A.10. This represent the number of machines distributed among 2E+09–5E+09 *flops*, for the two experiments.

Fig. 5 shows the theoretical degree of heterogeneity. *Heterogeneity Rate* is the rate between the number of machines with same computational power and total number of cluster machines. When all machines have the same computational power the *Heterogeneity Rate* is called homogeneous (Homog) on the left side of Fig. 5(b). It was calculated for the *MRA++* algorithms, with 20 machines to carry out *jobs* with long and short runtime. This study considers the data distribution (Fig. 5(a)) for a cluster of 55.8E+09 *flops*, with mean 2.79E+09 *flops*, a data input size between 8 and 16 GB and a 10 Mbps network. The average gain depends both on the number of heterogeneous machines, and on their computational power with a standard deviation of 5%. The results indicate that both homogeneous and heterogeneous environments justify the system implementation; When heterogeneity is above 20% the gain is better. *MRA++* outperforms original *MapReduce* algorithms in homogeneous setting in Fig. 5(b) in function of network bandwidth and the influence of granularity of *Reduce* phase.

5.2. Experiments

We will now show the results of the experiments that were conducted to evaluate the proposed algorithms. Every test in this section was simulated by means of MRSG with heterogeneous platforms. Fig. 6 shows the duration of *Map* and *Reduce* for *jobs* which involved running the original and new algorithms, in a 1 Gbps network. In this experiment, the *job* execution time with the new algorithms was 22% faster in the worst case scenario, and 35.76% faster in the best case, when compared with the original algorithms. The difference is not greater because the performance

Table 4
Speculative and remote tasks @ 1 Gbps.

Machines	32	64	128	256	512	1024	2048
Map Remote tasks[a]	24	60	91	175	449	1168	2499
Map Remote tasks[b]	0	0	0	0	0	0	0
Reduce Speculative tasks[a]	11	9	24	47	136	439	1017
Reduce Speculative tasks[b]	0	3	7	11	7	52	0

[a] Original algorithm.
[b] New algorithms.

Table 5
Remote and speculative executions @ 10 Mbps.

Machines	32	64	128	256	512	1024	2048
Map Remote tasks[a]	24	59	91	175	444	1143	2438
Map Remote tasks[b]	0	0	0	0	0	0	0
Reduce Speculative tasks[a]	11	9	24	47	136	439	1017
Reduce Speculative tasks[b]	0	0	0	0	0	0	0

[a] Original algorithm.
[b] New algorithms.

of the original algorithm is offset by the availability of a larger bandwidth, which conceals the impact of the data transfer between the machines during execution time.

With regard to the *Map* phase, its duration when the new algorithms were used, was 10.27% faster in the worst case scenario, and 27.96% in the best case. The *Reduce* phase represents between 51.47% and 63.10% of the total makespan for the *jobs*, which implies that they had a greater impact. The duration of *Reduce* is 17.76% faster in the worst case scenario and 34.89% in the best case.

Table 4 compares the number of (remote and speculative) executed tasks, between the original algorithm and the new algorithms for a 1 Gbps network. In the original algorithm, the scheduler executes remote tasks to improve *Map* performance, and copies data during execution time.

Remote task execution improves the *job* performance, but it can slow down the system. This slowness is hidden because of the high bandwidth availability between the machines, which is common in *data centers*. However, this model generates larger execution times in slower networks and the data copying ends up by having a bad influence on the performance. Before the scheduler executes a remote task, it first checks if it is worth carrying out the task on another available machine or if it is best to wait for a concurrent task to be completed. As a result, naturally fewer remote tasks are executed.

Fewer speculative tasks are executed due to the new clustering model, which allows a slow machine to be classified according to the average performance of machines from the same *g_dist_bruta*, and a larger number of short tasks in *Reduce* phase. Thus, there is a decline in the number of false positives and a as a result the system has a better performance.

Fig. 7 shows the behavior of the *job* with a 10 Mbps network. The execution of the *MapReduce* model with heterogeneous machines in slow networks has a slow performance because, with intensive applications such as *MapReduce*, there is a large amount of data to be copied to the machines during the *Reduce* phase. It can be seen that with the new algorithms the total amount of *job* time is 52.04% faster for the worst case scenario, and 72.08% for the best case. This corresponds to a gain of 2.1–3.5 times. The impact of the *Map* task execution time is smaller than the *Reduce* tasks. The proportion of *Reduce* functions in execution time to the *job* total time varies from 73.98% to 95.76%.

The *Map* tasks show an improvement in time of 75.06%–89.27% when the new algorithms are used; this represents a gain of 4–9.3 times. The improvement in the execution times of *Reduce* tasks, was 43.27% for the worst case scenario, and 69.73% for the best case, corresponding to a gain of 1.8–3.3 times, respectively.

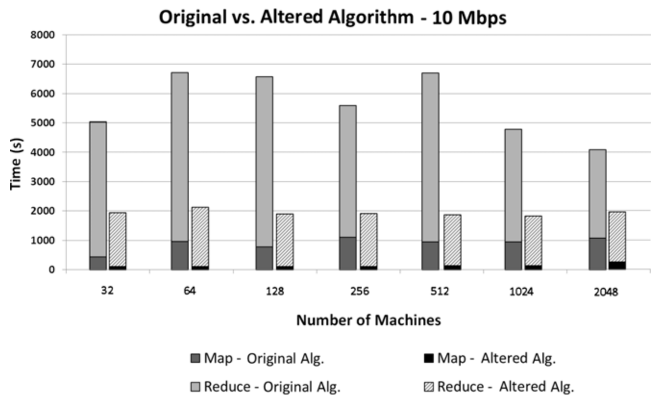


Fig. 7. MapReduce phases—original vs. new algorithms @ 10 Mbps.

Although the gain obtained in *Map* phase is better, there is less impact on the performance gain of the new algorithms.

Table 5 shows the number of remote and speculative executions for the tests with a 10 Mbps network. As can be seen, there is a large number of remote executions, which is similar to the tests shown in Table 4 with a 1 Gbps network. Therefore, it can be said that the number of these types of tasks is related to the *MapReduce* model and not to bandwidth availability.

The obtained results show a noticeable difference between the improvements in the 1 Gbps and 10 Mbps networks. This can be explained by the penalty associated with executing remote tasks in each case. With a higher bandwidth, the large amount of remote tasks does not affect the system as much as with a lower bandwidth. Hence, by reducing the amount of remote tasks, MRA++ provides greater improvements for slow bandwidth networks.

In *MapReduce* with original algorithms in homogeneous environments, we would expect there to be fewer remote and speculative executions, because of the high computational power of the machines, with balanced data and the same execution times. Thus, the results obtained in Table 5 for the new algorithms in heterogeneous environments show the unsuitable behavior of remote executions.

Fig. 8 shows the influence of the data granularity of input during the *Reduce* phase in a 1 Gbps network. The best execution time is 34.89% faster, when obtained with a 1/4 grain in the *Map* output, and the size is equivalent to 16 MB. In the worst case scenario the increase in execution time is 19.32%, with a 1/2 grain in the data output, equivalent to 32 MB.

Table 6
Measuring task costs @ 1 Gbps.

Costs	Machines						
	32	64	128	256	512	1024	2048
Map time (s)	0.53	0.69	0.91	1.26	2.13	3.20	8.52
Reduce time (s)	6.15	6.28	6.55	7.08	12.07	24.75	46.83
Job time (s)	6.68	6.98	7.46	8.34	14.20	27.94	55.35
Transfer time (s)	0.10	0.12	0.18	0.20	0.34	0.35	0.62
Total cost (s)	6.78	7.10	7.64	8.54	14.54	28.29	55.97
Transferred data (MB)	32	64	128	256	512	1024	2048

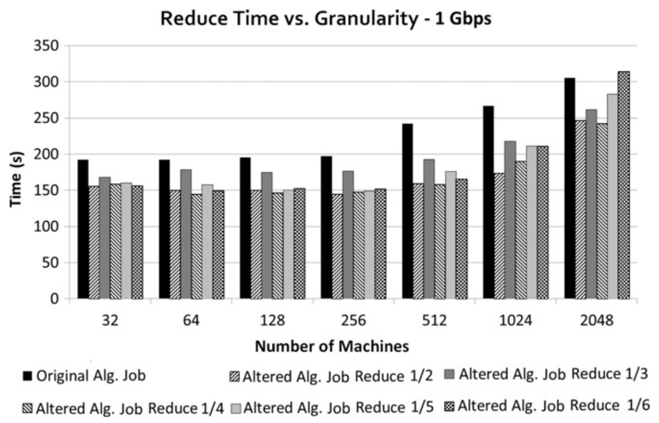


Fig. 8. Influence of granularity on *Reduce* @ 1 Gbps.

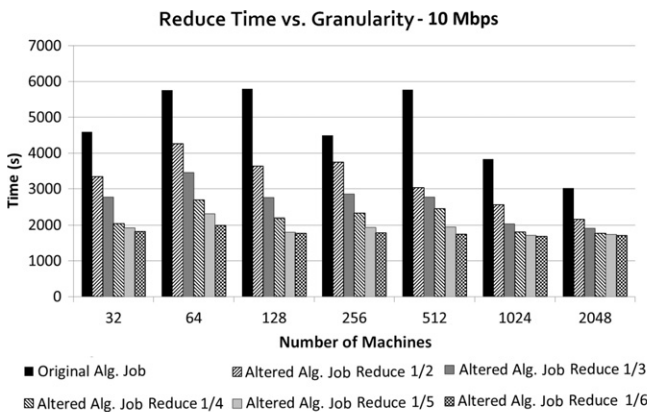


Fig. 9. Influence of granularity on *Reduce* @ 10 Mbps.

A smaller data granularity implies less time for the intermediate data copy among the machines that need to execute the *Reduce* tasks. Thus, there is a smaller connection time between each machine for transferring the intermediate data. The result is a greater degree of parallelism with regard to the copy tasks among the machines, even though the total amount of data is the same.

It can be seen that the approach adopted to reduce the amount of data results in less time for executing the *Reduce* tasks. Smaller granularity leads to a greater degree of task parallelism. Thus, a faster machine processes more tasks in the same time required for the execution of a single task with a slower machine.

However, this approach can result in unsuitable execution times if the granularity overloads the network with lots of small tasks, which is just the opposite of what was intended. In the test shown in Fig. 8, with a grain larger than 1/4 the system will generate execution times longer than desired.

Fig. 9 shows the influence of the data granularity of input during the *Reduce* phase in a 10 Mbps network. The best execution time with the new algorithms is 69.73% faster than the execution with the original algorithms. This result is accomplished with a grain that is 1/6 times the value of the output data of *Map*—the equivalent of 11 MB.

In the worst case scenario, the use of the new algorithms with a grain of 1/2 of output value causes a reduction of 16.33% in the execution time, compared with that of the original algorithms. In the cases shown in Fig. 9, the smallest granularity which does not affect the system performance is obtained with a grain of 1/6. A greater grain value will cause longer than desired execution times. In the original model, intermediate data copy in low availability networks causes poor performance. Thus, a smaller data granularity in *Reduce* provides acceptable execution times for this type of network.

5.3. *Cost estimates of the measuring task*

The total cost of the measuring task comprises the cost of copying 1 MB for every machine plus the execution time of the measuring task. The transfer cost t_t is the time in seconds that is necessary to copy 1 MB in a 1 Gbps network. The *k-d Tree* algorithm is used for calculating the transfer cost of the 1 MB sample. The time to transfer 1 MB to every machine is $t_t = \frac{1024*1024*8}{1000000000}$ s. The master node spends $2^{Md} * t_t$ s to transfer the data to its child nodes, in each level, a time of $2 * t_t$ is spent to transfer the data from the masters of each level to their child nodes.

The total transfer time for the 1 MB sample is the sum of the time spent by the master plus the time spent at the intermediate level, as can be seen in Table 6. Thus, by means of the parallelism obtained at the sub-levels, there is a low transfer time compared with the time needed to transfer data from the master to every machine, in a 1 Gbps network.

Costs are higher for a 10 Mbps bandwidth, because the available bandwidth is smaller. At the same time, the execution times of *MapReduce* in 10 Mbps are longer, since they benefit from using the new algorithms. The total cost of the measuring task comprises the cost of transferring the 1 MB sample plus the execution time of the measuring task, (both shown in Table 7).

The time required for the measuring task in 32 machines, for example, will be the sum of the transfer cost of 1 MB to 32 machines, (equal to 10.07 s), plus the execution time of the measuring *job* of 13.04 s, in a total of 23.11 s. This value is not considered because the execution time for a *job* without the new algorithms takes 84 min, whereas with the new algorithms it is only a bit longer than 32 min.

The cost of transferring 1 MB to 2048 machines is 62.08 s whereas the execution time of the measuring job is 57.10 s, which adds up to 1 min and 59 s and is the time needed to carry out the measuring task. The *job* with the original algorithms takes 68 min to execute while it takes a bit longer than 33 min when the new algorithms are used.

The initial costs can be considered to be environmental self-configuration costs, which are necessary for a better task scheduling. In view of this, as can be seen, the measuring task cost is very small, and practically insignificant when compared with the obtained gain. This gain can be even greater, when the approach offered by *MapReduce Write-Once-Read-Many* is taken into account.

Table 7
Measuring task costs @ 10 Mbps.

Costs	Machines						
	32	64	128	256	512	1024	2048
Map time (s)	0.53	0.69	0.91	1.26	2.13	4.26	8.52
Reduce time (s)	12.50	11.76	11.31	12.90	16.54	25.45	48.58
Job time (s)	13.04	12.45	12.22	14.16	18.67	29.71	57.10
Transfer time (s)	10.07	11.74	18.45	20.13	33.55	35.23	62.08
Total cost (s)	23.11	24.19	30.67	34.29	52.22	64.94	119.18
Transferred data (MB)	32	64	128	256	512	1024	2048

Hence some types of applications can benefit from using heterogeneous *clusters* for the execution of data-intensive *jobs*.

6. Related works

The *MapReduce* model was originally conceived for large homogeneous cluster environments. Some simplifications to the model were adopted with the aim of optimizing the task distribution. However, in heterogeneous environments, these simplifications may entail system degradation. Anjos et al. [23] show that data redistribution based on the processing capacity of machines is suitable for the workload distribution of tasks in these environments.

In [7], Zaharia et al. address performance issues in *MapReduce* in heterogeneous environments, which arise from the execution of different applications in large *clusters* with virtual environments, *e.g.* Amazon *EC2*. Their study points out that there ares some concerns over the simplification of the *MapReduce* model since this may generate an excessive number of speculative tasks.

To overcome this problem, the authors proposed LATE (*Longest Approximate Time to End*), a new task scheduler. Although LATE does not completely avoid speculative tasks, it considerably reduces performance degradation in heterogeneous environments. Experimental evaluation results show that, compared with the native Hadoop scheduler in speculative mode, LATE achieves a gain from 8.5% to 58%, depending on the application and number of working machines. However, the gain obtained by LATE is smaller with *WordCount*-type tasks, that are executed with a cluster of more than 900 nodes.

Xie et al. [24] propose to improve the performance of *MapReduce* applications in heterogeneous environments by allocating data according to the capacity of the machines. The authors' solution involves including some new algorithms in the data distribution module of the file system responsible for data relocation, which can move data from one machine to another in execution time, for the sake of performance.

The algorithms were included in HDFS (Hadoop Distributed File System) and have different roles: data division based on machine capacity, data redistribution according to CPU usage, and tackling new data. An execution of the application is performed in order to estimate the execution time for the tasks after the distribution. One of the drawbacks of this approach is that it entails the removal of the existing data replication of *Mapreduce*, which has an impact on the data loss if there are machine failures.

Xie et al. use large data input to characterize the different execution times, (*e.g.* 1 GB), in their performance evaluation experiments. However, experiments with such a large amount of data are not feasible in large-scale heterogeneous environments owing to the costs incurred by data transfer. In contrast, in *MRA++*, the proposed mechanism for estimating machine capacity does not require this order of data since it is based on task execution time.

Several works in the literature focus on adapting *MapReduce* algorithms tor volatile and heterogeneous environments. One of the first proposals was the MOON project (***M****apReduce* ***O****n* ***O****pportunistic* *e****N****vironments*) [25], a hybrid model for voluntary computing which considers that the system is composed of volatile and reliable non-volatile machines. Their solution involves applying the LATE algorithm [7]. Loss of data from the volatility machines is overcome by replicating data in reliable machines. However, the number of reliable machines that are needed to replicate the data restrains scalability.

In the work of Chen et al. [26] the authors outline an adaptive scheduler inspired by LATE, called SAMR, which changes the optimization parameters of each node. SAMR uses the history of *job* executions in a score called HP, with a range of 0–1. If the HP score is high (close to 1), the parameters of the current task are very dependent on the history of execution information. However, if the score is small (close to 0), the parameters do not depend on the history and the performance optimizations are not carried out. In addition, the authors use five virtual machines to determine the success of the experiments and the execution parameters. Hence it is not possible to ascertain the benefits of the implementation or the system load in large-scale environments. In contrast, MRA++ does not need a history of executions as it creates a knowledge base of execution times during the setup phase, before the *job* is executed. This allows us to adjust the scheduling before Hadoop's data split phase. MRA++ also avoids sending data to machines with low processing capability that would later be characterized by the system as stragglers.

Gufler's work [13] addresses the partitioning problem of the *Reduce* tasks in scientific applications that show characteristics for which the current *MapReduce* systems were not designed. The scientific applications studied by the authors have non-linear complexity in time for the execution of the *Reduce* tasks. In this case, the data distribution is generally heterogeneous. This means that, machines with a low workload have to wait for other machines with a high workload.

The model has two algorithms, one to calculate the cost of partitioning data and another to calculate the workload for the machines that execute the *Reduce* tasks. The algorithms are called fine-grain partitioning, as they divide the input data into a fixed number of parts, and dynamic fragmentation, as they divide the intermediate data locally in execution time. The authors warn that since the *bin packing problem* algorithm has *NP-Hard* complexity, the calculation of a moderate number of aggregates can be more expensive than the cost of the *Reduce* tasks normal system. In view of this, exact monitoring at an aggregate level may not be an easy task. Moreover, the adopted approach is only used in homogeneous *clusters*.

7. Conclusion

MapReduce is a framework used for handling data-intensive applications. The programming of applications does not require prior knowledge of the system's architecture, network topology or even how tasks should be parallelized. A prior knowledge of the amount of data or the execution time required for each task is not necessary either. However, the model created for large clusters precludes its efficient use in heterogeneous environments, especially those with low bandwidth links.

MRA++ adapts the amount of data processed during the *Map* phase to the distributed computing capability of the machines. The *Reduce* data is partitioned into smaller sizes according to the sum of the granularity factors. This means that the fastest machines process more data than the slowest ones.

The reduction in size of the intermediate data partitions increases the data granularity and number of executed tasks. Thus, a global *Reduce* task queue controls the execution and provides the available machines with a data location so that they can process it. The result is a higher degree of parallelism in the tasks and a shorter execution time for the *job*.

The speculative and remote task launches control, and provides an effective control of the distribution of tasks. Thus, it avoids initiating remote tasks that are not worth being launched. If there was no management, the run times would be longer and a larger number of speculative tasks would be launched in the last wave of task assignments in each phase.

MRA++ not only proposes to change a single part of *MapReduce*, but also a set of algorithms that are suited to the computational capabilities of the machines. Thus, *MapReduce* can be used in heterogeneous environments with slow links, in particular by means of the Hadoop implementation. *MRA++* also introduces changes to the way data is divided, task assignments, clustering and the way of controlling task progress.

Most studies have only been concerned with the *Map* phase and the launch speculative tasks. Speculative tasks only occur in the last wave of tasks and have little impact on the performance of *jobs*. So far, not may simultaneous changes have been proposed for most of the *MapReduce* algorithms, as proposed in this work.

The implementations proposed in *MRA++* are concerned about whether or not the workload can be adapted to the computing capability of each machine and also the reduction of remote tasks, which occurs because the machines are free to perform tasks and do not have to process local data. Thus, copies of data are performed in other machines at runtime. These data copies, affect the execution time of the entire *job* (especially over slow links). Hence, they are one of the causes of low performance in heterogeneous environments with low network bandwidth.

In heterogeneous environments, the new algorithm for data distribution, in the *Map* phase, increased the performance of task executions by up to 9.3 times. In the *Reduce* phase, the policy of reducing the data granularity of intermediate data, resulted in an improvement in performance of up to 69.73%. This led to an increased speed of 72.08% for the *job* with the new algorithms, as compared with the original model. Therefore, in a 10 Mbps network, a *job* with 16 GB of input data and 32 machines, which runs in 84 min in the original model, can be executed in just 32 min.

The experiments also show that a 1 TB *job*, with 2048 machines connected by 10 Mbps links, has a runtime of 33 min with the new algorithms, in contrast with 68 min required by the original model. Thus, if the volatility is ignored, the new algorithms have a satisfactory performance for task execution, and enable the deployment of data-intensive applications with low-bandwidth links; typical of Internet connections. This means that the developed algorithms enable data-intensive applications in large-scale environments on the Internet. This demonstrates the potential of using desktop grids and heterogeneous grids, although some problems are need to be addressed with regard to volatility.

The policy of performing a measurement task before the data distribution incurred a low cost. For example, with 2048 machines, the cost was only 56 s in 1 Gbps networks, and 57.1 s in 10 Mbps networks. Costs of this magnitude can be considered negligible when set beside the benefits achieved. If there are multiple *jobs* with the same *Map* and *Reduce* functions and using the same data, multiple executions of the measurement tasks are not necessary.

The proposed solutions in *MRA++* allow a greater balance to be achieved in task processing for *MapReduce*. This larger balancing occurs as a result of a heterogeneous load, but is suited to the computational capability of each machine in the cluster. The solution not only applies to low-bandwidth networks, but can also be deployed in heterogeneous environments with high bandwidth, (as demonstrated in Section 5). The performance improvements of the *job* are in the order of 22% in the worst case scenario and 35.7% in the best case, for fast links. If the results are compared with other studies they still show a satisfactory performance, such as the improvements of 27%–31% obtained in the work of Zaharia [7] and 29% obtained in the work of Lin [25]. When the slow links are taken into account, the performance improvements range from 52.04% in the worst case scenario to 72.08% in the best case.

On the basis of the results obtained in this work, combined with the low cost of implementing the new model, we believe that the objectives of the original proposal – adapting the *MapReduce* framework to heterogeneous environments – have been achieved.

Acknowledgments

This work was partly supported by CAPES, CNPq and FAPERGS/CNPq 008/2009 Project Research: "GREEN-GRID: Sustainable High Performance Computing". The experiments discussed in this paper were conducted with the aid of the Grid'5000 experimental testbed, under the INRIA ALADDIN development plan with support from CNRS, RENATER and several universities as well as other funding bodies (see https://www.grid5000.fr).

Appendix. Configuration example and frequency of machines capacity

See Tables A.8–A.10 and Fig. A.10.

Table A.8
Configuration example for 32 machines.

Configuration	Amount	Configuration	Amount
Intel Xeon 3050 (2130 MHz) 2 GB RAM	1	Intel Pentium Dual Core E6500 (3520 MHz) 4 GB RAM	1
Intel Core2 Duo E6540 (2330 MHz) 2 GB RAM	1	AMD Athlon II X2 250 (3750 MHz) 4 GB RAM	1
Intel Core2 Duo E6540 (2330 MHz) 4 GB RAM	1	Intel Core 2 Duo E6600 (3700 MHz) 2 GB RAM	2
Intel Pentium Dual Core E5300 (2616 MHz) 2 GB RAM	1	Intel Core 2 Duo E6600 (3700 MHz) 4 GB RAM	3
Intel Core 2 Duo E6550 (3002 MHz) 3 GB RAM	1	AMD Phenom II X3 720 (2809 MHz) 4 GB RAM	2
AMD Phenom II X2 550 (3100 MHz) 4 GB RAM	2	AMD Phenom II X3 720 (3200 MHz) 4 GB RAM	1
Intel Core 2 Duo E8200 (3200 MHz) 4 GB RAM	2	AMD Phenom II X3 720 (3335 MHz) 4 GB RAM	1
AMD Athlon II X2 255 (3410 MHz) 2 GB RAM	1	AMD Phenom II X3 720 (3600 MHz) 2 GB RAM	2
AMD Phenom II 550 (3567 MHz) 2 GB RAM	1	Intel Xeon L5320 (1800 MHz) 4 GB RAM	1
AMD Phenom II 550 (3567 MHz) 4 GB RAM	1	Intel Xeon E5335 (2000 MHz) 4 GB RAM	2
Pentium Dual-Core E5400 (2700 MHz) 4 GB RAM	1	Intel Xeon X3210 (2130 MHz) 4 GB RAM	1
Intel Core2 Duo E8135 (2660 MHz) 4 GB RAM	1	AMD Phenom II X3 720 (3816 MHz) 4 GB RAM	1

Table A.9
Configuration example for 128 machines.

Configuration	Amount	Configuration	Amount
Intel Core2 Duo T7300 (2000 MHz) 2 GB RAM	1	Intel Core2 Duo E8200 (2660 MHz) 4 GB RAM	1
Intel Pentium Dual T3400 (2160 MHz) 4 GB RAM	1	Intel Pentium B950 (2100 MHz) 2 GB RAM	2
AMD Sempron Dual Core 2300 MHz 2 GB RAM	1	Pentium Dual-Core E5700 (3000 MHz) 2 GB RAM	2
AMD Turion II Neo K685 Dual-Core (1800 MHz) 2 GB RAM	1	Intel Core i3 330M (2130 MHz) 2 GB RAM	2
Intel Celeron E1500 (2200 MHz) 2 GB RAM	2	AMD Phenom II X2 560 (3300 MHz) 2 GB RAM	3
Dual-Core AMD Opteron 1214 (2200 MHz) 2 GB RAM	1	Intel Core2 Duo T9600 (2800 MHz) 4 GB RAM	2
Pentium Dual-Core E5800 (3200 MHz) 4 GB RAM	1	Intel Core i7 640LM (2130 MHz) 4 GB RAM	2
Intel Core2 Duo T7500 (2200 MHz) 2 GB RAM	1	Intel Core i3 350M (2270 MHz) 2 GB RAM	2
AMD Athlon 64 X2 Dual Core 4800 (2500M Hz) 2 GB RAM	1	Intel Core2 Duo P9700 (2800 MHz) 4 GB RAM	6
Intel Xeon 3050 (2130 MHz) 2 GB RAM	2	Intel Core2 Duo T9800 (2930 MHz) 4 GB RAM	2
Intel Core2 Duo E6540 (2330 MHz) 2 GB RAM	2	Pentium Dual-Core E6700 (3200 MHz) 4 GB RAM	3
Intel Core2 Duo E4600 (2400 MHz) 4 GB RAM	2	Intel Core 2 Duo E8400 (3000 MHz) 4 GB RAM	3
Dual Core AMD Opteron 185 (2600 MHz) 2 GB RAM	2	Intel Core2 T9550 (266 0 MHz) 4 GB RAM	1
Intel Pentium P6200 (2128 MHz) 2 GB RAM	3	Intel Pentium G630 T (3300 MHz) 2 GB RAM	2
AMD Phenom II X2 550 (3100 MHz) 4 GB RAM	5	Intel Pentium B960 (2200 MHz) 4 GB RAM	2
Intel Core 2 Duo E6550 (3002 MHz) 2 GB RAM	3	Intel Core i3 380M (2530 MHz) 4 GB RAM	2
Intel Xeon 3065 (2330 MHz) 2 GB RAM	5	Intel Xeon L5240 (3000 MHz) 2 GB RAM	3
Intel Core2 Duo E6750 (2660 MHz) 2 GB RAM	3	Intel Celeron G540 (2.500 MHz) 2 GB RAM	3
AMD Phenom II N640 Dual-Core (2900 MHz) 4 GB RAM	7	Intel Core i5 560M (2670 MHz) 4 GB RAM	1
Intel Xeon 3060 (2400 MHz) 2 GB RAM	2	Intel Core i5-2537M (1400 MHz) 2 GB RAM	2
Pentium Dual-Core E5200 (2500 MHz) 2 GB RAM	3	Intel Xeon E5335 (2000 MHz) 4 GB RAM	4
Pentium Dual-Core E6300 (2660 MHz) 2 GB RAM	1	AMD Phenom II X3 B75 (3000 MHz) 4 GB RAM	3
Pentium Dual-Core E5400 (2700 MHz) 2 GB RAM	2	Intel Core i3-2310M (2100 MHz) 2 GB RAM	2
Intel Core2 X6800 (2930 MHz) 2 GB RAM	3	Intel Xeon X5260 (3330 MHz) 2 GB RAM	3
AMD Phenom II N660 Dual-Core (3000 MHz) 2 GB RAM	3	Intel Core i5-2557M (1700 MHz) 2 GB RAM	3
AMD Phenom II X2 B59 (3400 MHz) 4 GB RAM	1	Intel Core i3-2330M (2200 MHz) 4 GB RAM	4
Intel Core i7 640M (2800 MHz) 4 GB RAM	4		

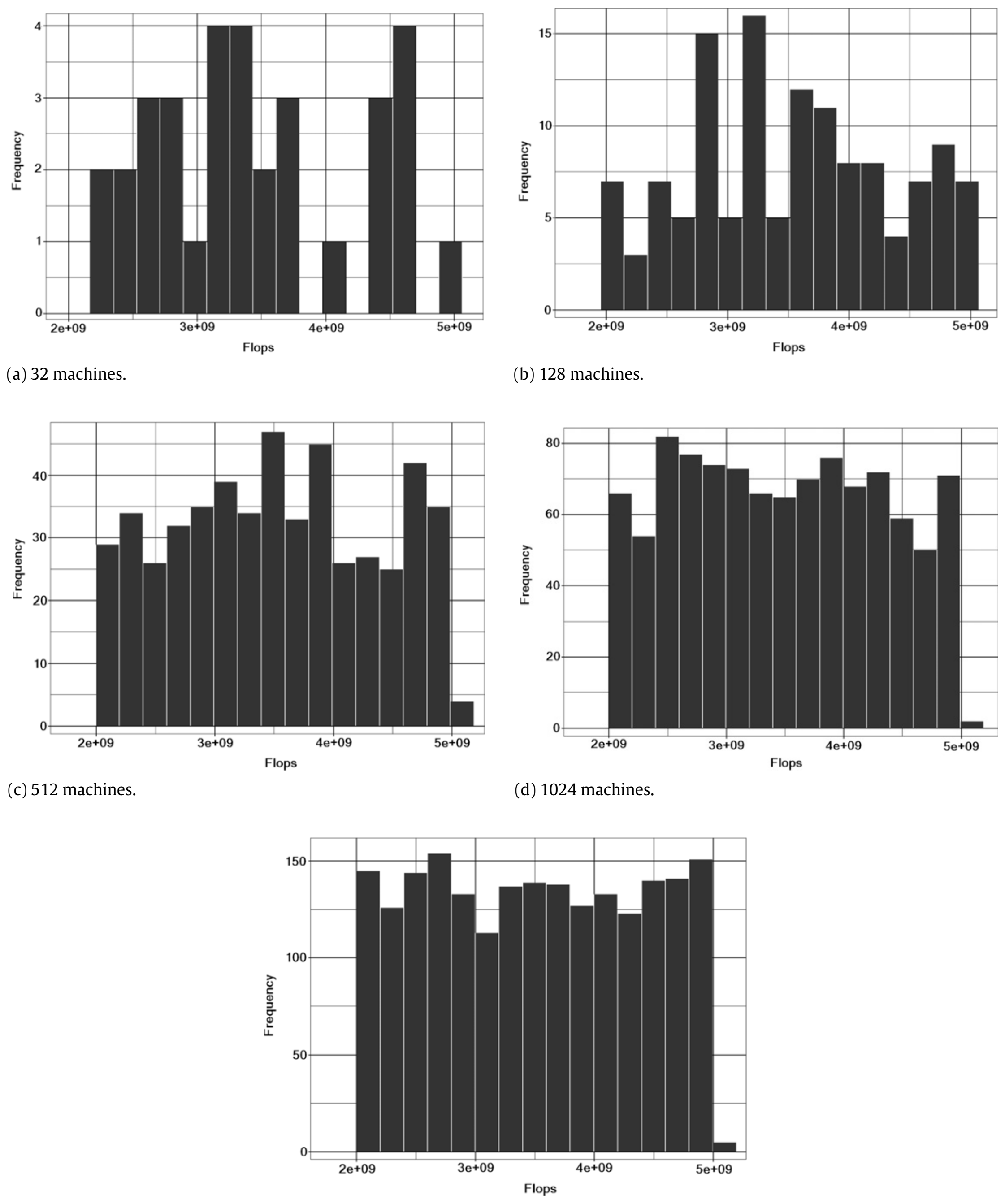
Table A.10
Configuration example for 256 machines.

Configuration	Amount	Configuration	Amount
Intel Core2 Duo T7300 (2000 MHz) 2 GB RAM	4	AMD Turion II P560 Dual-Core (2500 MHz) 2 GB RAM	3
AMD Sempron Dual Core 2300 MHz 2 GB RAM	2	Intel Core i7 620LM (2000 MHz) 4 GB RAM	3
AMD Turion II Neo K685 Dual-Core (1800 MHz) 2 GB RAM	1	Intel Celeron P4600 (2000 MHz) 2 GB RAM	4
Intel Celeron E1500 (2200 MHz) 2 GB RAM	3	Intel Core2 Extreme X7800 (2600 MHz) 2 GB RAM	2
Dual-Core AMD Opteron 1214 (2200 MHz) 2 GB RAM	2	Intel Core2 Duo E6700 (2660 MHz) 2 GB RAM	4
Dual-Core AMD Opteron 1214 (2200 MHz) 4 GB RAM	2	AMD Athlon 64 X2 Dual Core 5800+ (3000 MHz) 2 GB RAM	1
Intel Core2 Duo E4500 (2200 MHz) 2 GB RAM	3	AMD Athlon II X2 B24 (3000 MHz) 2 GB RAM	2
AMD Sempron X2 180 (2400 MHz) 2 GB RAM	2	Intel Pentium P6200 (2130 MHz) 4 GB RAM	2
AMD Athlon Dual Core 5000B (2600 MHz) 2 GB RAM	2	Intel Xeon 5148 (2330 MHz) 4 GB RAM	2
AMD Athlon 64 X2 Dual Core 4800 (2500 MHz) 2 GB RAM	3	AMD Phenom II N620 Dual-Core (3000 MHz) 4 GB RAM	2
Intel Xeon 3050 (2130 MHz) 2 GB RAM	1	Intel Core2 Duo T9300 (2500 MHz) 4 GB RAM	2
Intel Core2 Duo E6420 (2130 MHz) 2 GB RAM	2	Intel Core2 Duo E8135 (2400 MHz) 2 GB RAM	2
Intel Core2 Duo E6540 (2330 MHz) 2 GB RAM	2	Intel Core2 Duo E7200 (2530 MHz) 2 GB RAM	2
Intel Core2 Duo T6400 (2000 MHz) 4 GB RAM	2	AMD Phenom II N640 Dual-Core (2900 MHz) 4 GB RAM	5
Dual-Core AMD Opteron 2214 (2200 MHz) 4 GB RAM	3	Intel Xeon 3060 (2400 MHz) 4 GB RAM	3
Intel Core2 Duo E4600 (2400 MHz) 2 GB RAM	3	Intel Core2 Duo P9600 (2530 MHz) 2 GB RAM	3
Intel Pentium Dual E2220 (2400 MHz) 2 GB RAM	3	Intel Core2 Duo P8700 (2530 MHz) 4 GB RAM	2
Intel Core2 Duo P7350 (2000 MHz) 2 GB RAM	1	AMD Phenom II P860 Triple-Core (2000 MHz) 4 GB RAM	1
Dual Core AMD Opteron 185 (2600 MHz) 2 GB RAM	3	Intel Core2 Duo T9500 (2600 MHz) 2 GB RAM	4
Intel Pentium P6000 (1870 MHz) 2 GB RAM	2	Intel Core i3-2367M (1400 MHz) 2 GB RAM	4
Intel Core2 Duo E6550 (2330 MHz) 2 GB RAM	3	AMD Athlon II X2 260 (3200 MHz) 2 GB RAM	3
Intel Core2 Duo T7800 (2600 MHz) 4 GB RAM	4	Pentium Dual-Core E6300 (2800 MHz) 4 GB RAM	2
Intel Core2 Duo E6600 (2400 MHz) 2 GB RAM	4	Pentium Dual-Core E5500 (2800 MHz) 2 GB RAM	2
AMD Athlon II X2 B22 (2800 MHz) 4 GB RAM	2	AMD Athlon II X2 4450e (2300 MHz) 2 GB RAM	2
AMD Phenom II N830 Triple-Core (2100 MHz) 2 GB RAM	2	Intel Core i3 390M (2670 MHz) 2 GB RAM	2
Intel Xeon 5160 (3000 MHz) 4 GB RAM	2	Pentium Dual-Core E6800 (3330 MHz) 2 GB RAM	4
Pentium Dual-Core E5700 (3000 MHz) 4 GB RAM	2	Intel Celeron G540 (2.500 MHz) 2 GB RAM	6
AMD Phenom II N870 Triple-Core (2300 MHz) 2 GB RAM	2	Intel Core i5 540M (2530 MHz) 2 GB RAM	4
AMD Phenom 8750B Triple-Core (2400 MHz) 2 GB RAM	3	Intel Core i5 560M (2670 MHz) 2 GB RAM	3
Intel Pentium G6950 (2800 MHz) 4 GB RAM	6	Intel Core i5 560M (2670 MHz) 4 GB RAM	6
Intel Core i7 640LM (2130 MHz) 2 GB RAM	3	AMD Phenom II X3 720 (3600 MHz) 4 GB RAM	4
Intel Core2 Duo E8235 (2800 MHz) 2 GB RAM	2	Intel Core i5-2537M (1400 MHz) 4 GB RAM	4
Intel Core i3 350M (2270 MHz) 2 GB RAM	2	AMD Athlon II X3 440 (3000 MHz) 2 GB RAM	2
Intel Core2 Duo E7600 (3060 MHz) 4 GB RAM	3	AMD Athlon II X3 440 (3000 MHz) 4 GB RAM	3
Intel Core 2 Duo E8600 (4200 MHz) 4 GB RAM	3	Intel Core i5 580M (2670 MHz) 4 GB RAM	2
AMD Athlon II X3 415e (2500 MHz) 2 GB RAM	1	Intel Xeon E5335 (2000 MHz) 2 GB RAM	3
Intel Core2 Duo T9800 (2930 MHz) 4 GB RAM	2	AMD Athlon II X3 435 (2900 MHz) 2 GB RAM	2
Pentium Dual-Core E5800 (3200 MHz) 4 GB RAM	5	Intel Xeon X3210 (2130 MHz) 4 GB RAM	3
AMD Phenom 8850B Triple-Core (2500 MHz) 2 GB RAM	2	Intel Xeon X5260 (3330 MHz) 2 GB RAM	2
Intel Xeon E5603 (1600 MHz) 4 GB RAM	3	Intel Core i3 530 (2930 MHz) 2 GB RAM	4

(*continued on next page*)

Table A.10 (*continued*)

Configuration	Amount	Configuration	Amount
Intel Core i3 370M (2400 MHz) 2 GB RAM	2	Intel Core i3 530 (2930 MHz) 4 GB RAM	6
Intel Pentium G630T (2300 MHz) 4 GB RAM	1	Intel Pentium G850 (2900 MHz) 2 GB RAM	2
Intel Pentium B960 (2200 MHz) 4 GB RAM	2	Intel Core i3-2330M (2200 MHz) 2 GB RAM	2
Intel Core i5 520M (2400 MHz) 4 GB RAM	3	Intel Core i7 640M (2800 MHz) 4 GB RAM	3
AMD A8-3520M APU (1600 MHz) 2 GB RAM	2	AMD Phenom II X3 740 (3000 MHz) 2 GB RAM	5
Intel Core i5 480M (2670 MHz) 2 GB RAM	5	Intel Xeon X5272 @ 3.40 GHz (3400 MHz) 2 GB RAM	2
Intel Core i3 380M (2530 MHz) 3 GB RAM	3		



(a) 32 machines.

(b) 128 machines.

(c) 512 machines.

(d) 1024 machines.

(e) 2048 machines.

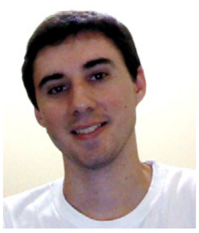
Fig. A.10. Frequency of machine capacity.

References

[1] J. Dean, S. Ghemawat, MapReduce—simplified data processing on large clusters, in: OSDI, 2004, pp. 137–150.
[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad—distributed data-parallel programs from sequential building blocks, in: EuroSys'07—Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, ACM, New York, NY, USA, 2007, pp. 59–72. http://dx.doi.org/10.1145/1272996.1273005.
[3] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, Evaluating MapReduce for multi-core and multiprocessor systems, in: HPCA'07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, IEEE Computer Society, Washington, DC, USA, 2007, pp. 13–24. http://dx.doi.org/10.1109/HPCA.2007.346181.
[4] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig Latin—a not-so-foreign language for data processing, in: SIGMOD'08—Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2008, pp. 1099–1110. http://dx.doi.org/10.1145/1376616.1376726.
[5] J. Dean, S. Ghemawat, MapReduce—a flexible data processing tool, Commun. ACM 53 (1) (2010) 72–77. http://dx.doi.org/10.1145/1629175.1629198.
[6] T. White, Hadoop—The Definitive Guide, Vol. 1, thirs ed., OReilly Media, Inc., 2012.
[7] M. Zaharia, A. Konwinski, A.D. Joseph, Y. Katz, I. Stoica, Improving MapReduce performance in heterogeneous environments, in: OSDI, 2008, pp. 29–42.
[8] G. Mackey, S. Sehrish, J. Lopez, J. Bent, S. Habib, J. Wang, Introducing MapReduce to high end computing, in: Petascale Data Storage Workshop at SC08, Austin, Texas, 2008.
[9] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, D. Thain, All-Pairs—an abstraction for data-intensive computing on campus grids, IEEE Trans. Parallel Distrib. Syst. 21 (1) (2010) 33–46. http://dx.doi.org/10.1109/TPDS.2009.49.
[10] P.K. Mantha, A. Luckow, S. Jha, Pilot-MapReduce: an extensible and flexible MapReduce implementation for distributed data, in: Proceedings of Third International Workshop on MapReduce and its Applications Date, MapReduce'12, ACM, New York, NY, USA, 2012, pp. 17–24. http://dx.doi.org/10.1145/2287016.2287020.
[11] S. Narayan, S. Bailey, A. Daga, Hadoop acceleration in an OpenFlow-based cluster, in: 2012 SC Companion: High Performance Computing, Networking, Storage and Analysis, SCC, 2012, pp. 535–538. http://dx.doi.org/10.1109/SC.Companion.2012.76.
[12] V.S. Martha, W. Zhao, X. Xu, h-MapReduce: a framework for workload balancing in MapReduce, Advanced Information Networking and Applications, in: 2013 IEEE 27th International Conference on, AINA, 2013, pp. 637–644. http://dx.doi.org/10.1109/AINA.2013.48.
[13] B. Gufler, N. Augsten, A. Reiser, A. Kemper, Handling data skew in MapReduce, in: F. Leymann, I. Ivanov, M. van Sinderen, B. Shishkov (Eds.), CLOSER, 2011, pp. 574–583.
[14] R.W. Floyd, R.L. Rivest, Expected time bounds for selection, Commun. ACM 18 (1975) 165–172. http://dx.doi.org/10.1145/360680.360691.
[15] P. Velho, A. Legrand, Accuracy study and improvement of network simulation in the SimGrid framework, in: Proceedings of the 2nd International Conference on Simulation Tools and Techniques, Simutools'09, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, 2009, pp. 13:1–13:10. http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5592.
[16] W. Kolberg, P.D.B. Marcos, J.C.S. Anjos, A.K.S. Miyazaki, C.R. Geyer, L.B. Arantes, MRSG—a MapReduce simulator over SimGrid, Parallel Comput. 39 (4–5) (2013) 233–244. http://dx.doi.org/10.1016/j.parco.2013.02.001.
[17] G. Wang, A.R. Butt, P. Pandey, K. Gupta, Using realistic simulation for performance analysis of MapReduce setups, in: LSAP'09: Proceedings of the 1st ACM Workshop on Large-Scale System and Application Performance, ACM, New York, NY, USA, 2009, pp. 19–26. http://dx.doi.org/10.1145/1552272.1552278.
[18] S. Hammoud, M. Li, Y. Liu, N.K. Alham, Z. Liu, MRSim: a discrete event based MapReduce simulator, in: M. Li, Q. Liang, L. Wang, Y. Song (Eds.), FSKD, IEEE, 2010, pp. 2993–2997.
[19] F. Bai, X. Hu, Cloud MapReduce for particle filter-based data assimilation for wildfire spread simulation, in: Proceedings of the High Performance Computing Symposium, HPC'13, Society for Computer Simulation International, San Diego, CA, USA, 2013, pp. 11:1–11:6. URL: http://dl.acm.org/citation.cfm?id=2499968.2499979.
[20] A. Chakraborty, M. Pathirage, I. Suriarachchi, K. Chandrasekar, C. Mattocks, B. Plale, Storm surge simulation and load balancing in Azure cloud, in: Proceedings of the High Performance Computing Symposium, HPC'13, Society for Computer Simulation International, San Diego, CA, USA, 2013, pp. 14:1–14:9. URL: http://dl.acm.org/citation.cfm?id=2499968.2499982.
[21] E. Heien, D. Kondo, D. Anderson, A correlated resource model of Internet end hosts, IEEE Trans. Parallel Distrib. Syst. 23 (6) (2012) 977–984.
[22] A.F. Barbosa, Pesquisa sobre o uso das tecnologias de informação e comunicação no Brasil: TIC Domicílios e TIC Empresas 2010, Vol. 1, first ed., Comitê Gestor da Internet no Brasil, 2011, pp. 1–584.
[23] J.C.S. Anjos, W. Kolberg, L. Arantes, C.F.R. Geyer, Estratégias para Uso de MapReduce em Ambientes Heterogêneos, in: L.A.C.o.H.P. Computing (Ed.), CLCAR Conferência Latinoamericana de Computación de Alto Rendimiento, Vol. 1, first ed., Evangraf, 2010, pp. 322–325.
[24] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, X. Qin, Improving MapReduce performance through data placement in heterogeneous Hadoop clusters, in: IEEE International Symposium on Parallel and Distributed Processing, Workshops and Ph.D. Forum, IPDPSW, 2010, pp. 1–9. http://dx.doi.org/10.1109/IPDPSW.2010.5470880.
[25] H. Lin, J. Archuleta, X. Ma, W.-c. Feng, Z. Zhang, M. Gardner, MOON—MapReduce On Opportunistic eNvironments, Tech. Rep., Virginia Polytechnic Institute and State University, 2009, URL: http://eprints.cs.vt.edu/archive/00001089.
[26] Q. Chen, D. Zhang, M. Guo, Q. Deng, S. Guo, SAMR: a self-adaptive MapReduce scheduling algorithm in heterogeneous environment, in: 2010 IEEE 10th International Conference on Computer and Information Technology CIT, 2010, pp. 2736–2743. http://dx.doi.org/10.1109/CIT.2010.458.

Julio C.S. Anjos is a Ph.D. student in Computer Science at the Federal University of Rio Grande do Sul UFRGS/RS. He received the Master in Computer Science degree from the Federal University of Rio Grande do Sul (Concept 6—Capes) in 04/2012 on the theme of Adaptation in Data Intensive Computing Environments Desktop Grid using MapReduce. He is a graduate in Electrical Engineering with emphasis on Electronics from PUC/RS – Pontifical Catholic University of Rio Grande do Sul – in 1991. His research interests include grid computing, distributed systems, data intensive computing, MapReduce and virtual systems.

Iván Carrera is an M.Sc. student in Computer Science at the Federal University of Rio Grande do Sul—UFRGS/RS. He has completed Electronics and Networking Engineering at Escuela Politécnica Nacional in Quito/Ecuador, in 2011. He is a Professor in the National Polytechnic School and University of the Americas in Quito/Ecuador. He is a researcher in Computer Science focused on the areas of largely distributed systems, Cloud computing and MapReduce.


Wagner Kolberg is an M.Sc. student in Computer Science at Universidade Federal do Rio Grande do Sul (UFRGS)—Brazil. He received the B.Sc. degree in Computer Science from Universidade Federal do Rio Grande do Sul—Brazil, in 2010. He is a Computer Science researcher interested in the areas of parallel programming, distributed systems, data-intensive computing.

Andre Luis Tibola is an M.Sc. student in Computer Science at the University Federal of Rio Grande (FURG)—Brazil. He received the B.Sc. degree in Computer Engineering from the University Federal of Rio Grande—Brazil, in 2011. He is a Computer Science researcher interested in the areas of parallel programming, distributed systems, data-intensive computing.

Luciana B. Arantes is an Assistant Professor in University Pierre et Marie Currie (Paris 6), member of Laboratoire d'Informatique de Paris 6 (LIP6) and INRIA/LIP6 Regal Project. Her research interests include: adapting distributed algorithms for large scale, dynamic or heterogeneous environments; scalability, distributed algorithms fault-tolerance, and also with distributed support for multi-agent systems in large scale environments.

Claudio R. Geyer received his Ph.D. in Informatics from Université de Grenoble I, in 1991. He received the M.Sc. degree in Computer Science from UFRGS/RS, in 1986. He completed his Mechanics Engineering at UFRGS/RS, in 1978. He is a Tenured Professor in UFRGS/RS, in Computer Science. His research interests include: pervasive and ubiquitous computing, grid and volunteer computing, scheduling and data-intensive computing.