

# Towards MapReduce for Desktop Grid Computing

Bing Tang<sup>\*</sup>, Mircea Moca<sup>†</sup>, Stéphane Chevalier<sup>‡</sup>, Haiwu He<sup>§</sup>, Gilles Fedak<sup>§</sup>,

<sup>\*</sup>Hunan University of Science and Technology, China

Email: btang@hnust.edu.cn

<sup>†</sup>University of Babeş-Bolyai of Cluj-Napoca, Romania

Email: mircea.moca@econ.ubbcluj.ro

<sup>‡</sup>Ecole Normale Supérieure de Lyon, France

Email: stephane.chevalier@ens-lyon.fr

<sup>§</sup>INRIA, Université de Lyon, France

Email: {haiwu.he, gilles.fedak}@inria.fr

**Abstract**—MapReduce is an emerging programming model for data-intensive application proposed by Google, which has attracted a lot of attention recently. MapReduce borrows ideas from functional programming, where programmer defines Map and Reduce tasks to process large set of distributed data. In this paper we propose an implementation of the MapReduce programming model. We present the architecture of the prototype based on BitDew, a middleware for large scale data management on Desktop Grid. We describe the set of features which makes our approach suitable for large scale and loosely connected Internet Desktop Grid: massive fault tolerance, replica management, barriers-free execution, latency-hiding optimisation as well as distributed result checking. We also present performance evaluation of the prototype both against micro-benchmarks and real MapReduce application. The scalability test shows that we achieve linear speedup on the classic WordCount benchmark. Several scenarios involving lagger hosts and host crashes demonstrate that the prototype is able to cope with an experimental context similar to real-world Internet.

**Keywords**—Desktop Grid computing, MapReduce, data-intensive application.

## I. INTRODUCTION

Although Desktop Grids have been very successful in the area of High Throughput Computing[1], data-intensive computing is still a promising area. Up to now, the application scope of Desktop Grids have mostly been restricted to Bag-of-Tasks applications with low requirements in terms of disk storage or communication bandwidth and without dependencies between tasks. We believe that applications requiring an important volume of data input storage with frequent data reuse and limited volume of data output could take advantage not only of the vast processing power but also of the huge storage potential offered by Desktop Grid systems[2]. There exists a broad range of scientific applications, such as bioinformatics and simulation in general as well as non scientific application such as web ranking or data mining which meet these criteria. The MapReduce programming model, initially proposed by Google[3], adapts well to this class of applications. MapReduce borrows ideas from functional programming, where programmer defines Map and Reduce tasks executed on large set of distributed data. Its key strengths are the high degree of parallelism combined with the simplicity of the programming model and its applicability to a large variety of application domains.

However, enabling MapReduce on Desktop Grids raises many research issues with respect to the state of the art in existing Desktop Grid middleware. In contrast with traditional Desktop Grids[4], [5], [6] which have been built around Bag-of-Tasks applications

with few I/O, MapReduce computations are characterized by the handling of large volume of input and intermediate data. The first challenge is the support to collective file operation which exists in MapReduce. In particular the redistribution of intermediate results between the execution of Map and Reduce tasks, called the *Shuffle* phase, presents some similarities with the MPI\_AlltoAll collective[7]. Collective communications are complex operations in Grids with heterogeneous network, and they are even more difficult to achieve on Desktop Grids because of the hosts volatility, churn and extremely varying network conditions. The second challenge is that some components of a Desktop Grid have to be decentralized. For instance, a key security component is the results certification[8] which is needed to check that malicious volunteers do not tamper with the results of a computation. Because intermediate results might be too large to be sent back to the server, results certification mechanism cannot be centralized as it is currently implemented in existing Desktop Grid systems. The third challenge is that dependencies between the Reduce tasks and the Map tasks, combined with hosts volatility and lagers can slowdown dramatically the execution of MapReduce applications. Thus, there is a need to develop aggressive performance optimisation solution, which combines latency hiding, data and tasks replication and barriers-free reduction.

In this paper we present a complete runtime environment to execute MapReduce applications on Desktop Grid. At our knowledge there exists no such environment which has been specifically designed for Desktop Grid. Although existing MapReduce middleware such as Hadoop[9] have fault-tolerant capabilities, it has been demonstrated that it is not suitable to the Desktop Grid context[10]. Our prototype is based on an open source middleware BitDew[11], developed by INRIA, which is a programmable environment for automatic and transparent data management on computational Desktop Grids. BitDew relies on a specific set of metadata to drive key data management operations, namely life cycle, distribution, placement, replication and fault-tolerance with a high level of abstraction. The BitDew runtime environment is a flexible distributed service architecture that integrates modular P2P components such as DHTs for a distributed data catalog, and collaborative transport protocols for data distribution, asynchronous and reliable multi-protocols transfers.

In this paper, we present the design choice and the system architecture of our prototype. We describe the set of features which makes our approach suitable for large scale and loosely connected Internet Desktop Grid: massive fault tolerance, replica management, barriers-free execution, latency-hiding optimisation as well as distributed

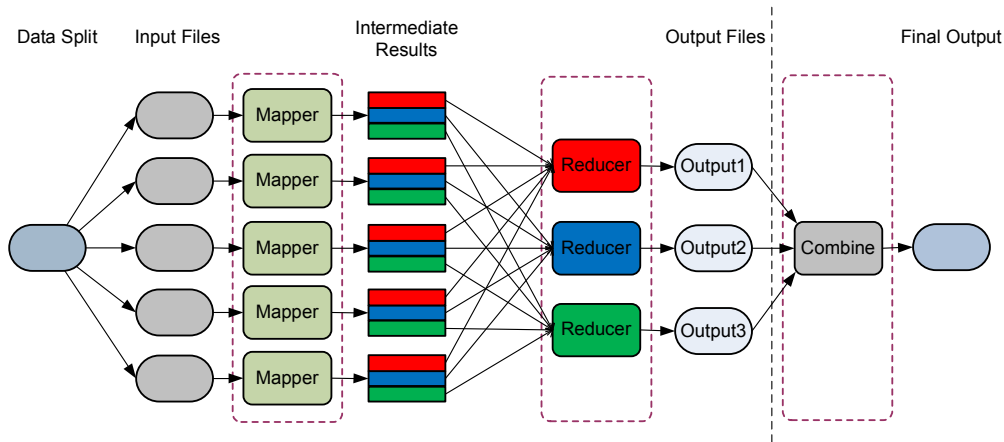


Figure 1. Implementation of MapReduce, overview of the dataflow.

result checking. We evaluate our implementation of the collective file operations on micro-benchmarks and the efficiency of the prototype against real MapReduce applications. The scalability test shows that we achieve linear speedup on the WordCount benchmark. Several scenarios involving laggings host and numerous host crashes demonstrate that the prototype is able to cope with experimental context similar to real-world Internet.

The rest of the paper is organized as follows, in Section II we give the background of our research, in Section III, we detail the system architecture, in Section IV we report on performance evaluation, we give related work in Section V and finally we conclude in Section VI.

## II. BACKGROUND

### A. MapReduce

The MapReduce programming model allows the user to write a Map function which processes input data and a Reduce function which processes the output of Map tasks and produce a final result. Figure 1 illustrates the dataflow of the system and now we detail each steps of a MapReduce computation. In the first step, input data are divided into chunks and distributed over the computing elements using a distributed file system such as GFS[12]. Computing elements can be classified into Mapper nodes, which execute Map tasks and Reducer nodes which execute Reduce tasks. In the second step, Mapper nodes apply the Map function on each file chunk. The result of the execution of a Map task is  $list(k, v)$ , a list of key and value pairs. Then, the Partition phase achieves splitting the keys space on Mapper node, so that each Reducer node get a part of the key space. This is typically done by applying a hash function to the keys although programmers can define their own partition function. The new data produced are called the *intermediate results*. During the Shuffle phase, intermediate results are sent to their corresponding Reducer. In the Reduce phase, Reducer nodes apply the Reduce function to all of the values  $(k, list(v))$  for a specific key. At the end, all the results can be assembled and sent back to the master node, and this is the Combine phase.

### B. BitDew

BitDew<sup>1</sup> provides simple APIs to programmers for creating, accessing, storing and moving data easily even in highly dynamic and volatile environments. BitDew relies on a specific set of

metadata to drive key data management operations, namely life cycle, distribution, placement, replication and fault-tolerance with a high level of abstraction. The Bitdew runtime environment is a flexible distributed service architecture that integrates modular P2P components such as DHTs for a distributed data catalog, and collaborative transport protocols for data distribution, asynchronous and reliable multi-protocols transfers.

In consequence, implementing the MapReduce model using BitDew allows to leverage on many of the needed features already provided by BitDew.

## III. SYSTEM ARCHITECTURE

In this section we describe the architecture of our prototype. First we present an overview of the system, then we focus our discussion on the algorithms and implementation of the main software components and we highlight the main features.

### A. General Overview

The system contains three main software components: the MR Master and MR worker programs, the MapReduce library, and several functions written by the user for his particular MapReduce application.

Our prototype heavily relies on the BitDew middleware that it extends with additional modules. Figure 2 illustrates the new components and extensions (white boxes) that has been specifically developed versus the existing components (grey boxes) provided by BitDew.

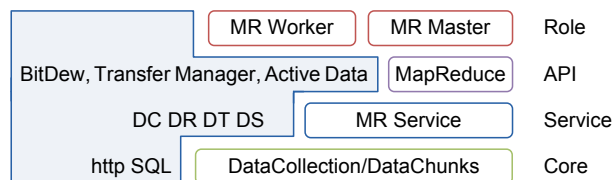


Figure 2. General overview of the system.

The architecture follows an approach commonly found in Desktop Grids which separates the nodes in two groups: stable nodes (Master and service node) and volatile nodes (Worker).

The *service node* run various independent services which compose the runtime environment. BitDew provides 4 basic services which cover the most common data management operations, namely the Data Repository (DR) service, Data Catalog (DC) service, Data

<sup>1</sup><http://www.bitdew.net/>

Transfer (DT) service and Data Scheduler (DS) service. We have enriched this set by adding a new service, called MapReduce service (MR), launched by the Master, which monitors MapReduce execution on Worker nodes. Workers periodically reports to the MR service the Map and Reduce tasks processing status.

Normally, programmers will not use the various services directly, instead they call the APIs which encapsulate the complexity of internal protocols. We have added a new API called as MapReduce, which is similar to Hadoop API. When programmers develop MapReduce applications, they define Map and Reduce functions by implementing the Map and Reduce interfaces provided by the API.

Volatile nodes, called *Workers* provide the storage and computing resources to run the Map and Reduce tasks. Classically, the execution is controlled by a *Master* node which distributes and schedules the data chunks, launches MapReduce computations, monitors the progress of tasks execution and collects final results.

When the user aims to achieve a MapReduce application, he provides the master node with a set of  $F_n$  initial input files. Then, the master splits all files into chunks of size  $S_c$ , registers and uploads all chunks to the BitDew services, which will be scheduled and distributed to a set of workers as input data for Map task, as explained in Section II-A. File splitting methods are varied with different applications. For the word counting example, the new-line character can be simply used as split-point.

### B. Map-Reduce Mechanism

When a worker receives data from the BitDew service node, a `data_scheduled` event is raised. At this time it determines whether the received data is treated as Map or Reduce input, and then the appropriate Map or Reduce function is called. The Map function parses the file (according to particular semantic), and, for each key  $k$ , *Emits* a value  $v$ . Thus, the Map function produces a set of *intermediate key/value* pairs,  $list(k, v)$ . The list is then sorted by  $k$  and splitted into  $R$  subsets of *key/value* pairs. The key interval depends on the number of reducers  $R$ , which is user defined. Finally, each subset is written to a separate file  $ir_{m,r}$  and sent back to BitDew to be scheduled.

A worker which has finished a Map task reports to the BitDew services all intermediate results  $ir$ . The respective files will be sent to the appropriate nodes as input data for Reduce tasks. A worker received input data for a Reduce task calls the Reduce function, which is specified by user. We notice that the worker behaves as Reducer based on a particular data it received. The Reduce function accepts an input like  $(k, list(v))$  and outputs a  $list(v)$ . In our system, only one node is responsible for reducing keys on  $ir_r$  partition. Thus, the BitDew server sends all intermediate files according to the  $ir_r$  partition to the correct node to be Reduced. When reducing, a worker merges 2 intermediate files into one, and this file is merged again with the next intermediate file received from BitDew services. The Reduce function is called repeatedly, and this process continues until no new intermediate file is received from BitDew. So that, the  $ir_r$  partitions converges to a final result.

### C. Algorithm and Implementation

Before explaining detailed algorithms and our implementation, we shortly present a set of key-notions which are specific to BitDew. As mentioned previously, BitDew allows the programmer to develop data-driven applications. That is, components of the system react when *Data* are scheduled or deleted. *Data* are created by nodes

Table I

MAIN ATTRIBUTE KEYS AND MEANING OF THE CORRESPONDING VALUES

<code>replica</code>	stands for replication and indicates the number of copies in the system for a particular <i>Data</i> item
<code>resilient</code>	is a flag which indicates if the data should be scheduled to another host in case of machine crash
<code>lifetime</code>	indicates the synchronization with existence of other <i>Data</i> in the system,
<code>affinity</code>	indicates that data with an affinity link should be placed on the same node
<code>protocol</code>	indicates the file transfer protocol to be employed when transferring files between nodes,
<code>distrib</code>	which indicates the maximum number of pieces of <i>Data</i> with the same <i>Attribute</i> should be sent to particular node.

either as simple communication messages (without “payload”) or associated with a file. In the later case, the respective file can be sent to the BitDew data repository service and scheduled as input data for computational tasks. The way the *Data* is scheduled depends on the *Attribute* associated to it. An attribute contains several (key, value) pairs which could influence how the BitDew scheduler will schedule the *Data*. Table I provides the list of the attribute keys and their descriptions.

The BitDew API provides a  $schedule(Data, Attribute)$  function by which, a client requires a particular behavior for *Data*, according to the associated *Attribute*.

The initial MapReduce input files are handled by the Master (see Algorithm 1). Input files can be provided either as the content of a directory or as a single large file with a file chunk size  $S_c$ . In the second case, the Master splits the large file into a set of file chunks, which can be treated as regular input files. We denote  $F = f_1, \dots, f_n$  the set of input files. From the input files, the Master node creates a *DataCollection*  $DC = d_1, \dots, d_n$ , and the input files are uploaded to BitDew. Then, the master creates *Attribute MapInputAttr* with `replica=1` and `distrib=-1` flag values, this means that each input file is one input data for the Map task. Each  $d_i$  is scheduled according to the *MapInputAttr* attribute. The node where  $d_i$  is scheduled will get the content associated to the respective *Data*, that is  $f_i$  as input data for a Map task.

---

#### Algorithm 1 DATA CREATION

---

**Require:** Let  $F = f_1, \dots, f_n$  be the set of input files

1. {on the Master node}
  2. Create a *DC*, a *DataCollection* based on  $F$
  3. Schedule data *DC* with attribute *MapInputAttr*
- 

The algorithm 2 presents the *Map* phase. To initiate a MapReduce distributed computation, the Master node first creates a data *MapToken*, with an attribute whose affinity is set to the *DataCollection DC*. This way, *MapToken* will be scheduled to all the workers. Once the token is received by a worker, the Map function is called repeatedly to process each chunks received by the worker, creating a local  $list(k, v)$ .

After finishing all the Map tasks, the worker splits its local  $list_m(k, v)$  into  $r$  intermediate output result files  $if_{m,r}$  according to the partition function, and  $R$ , the number of reducers (see Algorithm 3). For each intermediate files  $if_{m,r}$ , a reduce input data  $ir_{m,r}$  is created. How to transmit  $ir_{m,r}$  to their corresponding reducer? This is partly implemented by the master, which creates  $R$  specific data called *ReduceToken<sub>r</sub>*, and schedules this data with the attribute *ReduceTokenAttr*. If one worker receives the token, it is selected to be a reducer. As we assume that there are less reducers than

---

**Algorithm 2** EXECUTION OF MAP TASKS

---

**Require:** Let  $Map$  be the Map function to execute  
**Require:** Let  $M$  be the number of Mappers and  $m$  a single mapper  
**Require:** Let  $d_m = d_{1,m}, \dots, d_{k,m}$  be the set of map input data received by worker  $m$

1. {on the Master node}
  2. Create a single data  $MapToken$  with affinity set to  $DC$
  - 3.
  4. {on the Worker node}
  5. **if**  $MapToken$  is scheduled **then**
  6.     **for all** data  $d_{j,m} \in d_m$  **do**
  7.         execute  $Map(d_{j,m})$
  8.         create  $list_{j,m}(k, v)$
  9.     **end for**
  10. **end if**
- 

---

**Algorithm 3** SHUFFLING INTERMEDIATE RESULTS

---

**Require:** Let  $M$  be the number of Mappers and  $m$  a single worker  
**Require:** Let  $R$  be the number of Reducers  
**Require:** Let  $list_m(k, v)$  be the set of key, values pairs of intermediate results on worker  $m$

1. {on the Master node}
  2. **for all**  $r \in [1, \dots, R]$  **do**
  3.     Create Attribute  $ReduceAttr_r$  with  $distrib = 1$
  4.     Create data  $ReduceToken_r$
  5.     schedule data  $ReduceToken_r$  with attribute  $ReduceTokenAttr$
  6. **end for**
  - 7.
  8. {on the Worker node}
  9. split  $list_m(k, v)$  in  $if_{m,1}, \dots, if_{m,r}$  intermediate files
  10. **for all** file  $if_{m,r}$  **do**
  11.     create reduce input data  $ir_{m,r}$  and upload  $if_{m,r}$
  12.     schedule data  $ir_{m,r}$  with  $affinity = ReduceToken_r$
  13. **end for**
- 

workers, the  $ReduceTokenAttr$  has  $distrib=1$  flag value which ensures a fair distribution of the workload between reducers. Finally, the *Shuffle* phase is simply implemented by scheduling the portioned intermediate data with an attribute whose affinity tag is equal to the corresponding  $ReduceToken$ .

Algorithm 4 presents the *Reduce* phase. When a reducer, that is a worker which has the  $ReduceToken$ , starts to receive intermediate results, it calls the Reduce function on the  $(k, list(v))$ . When all the intermediate files have been received, all the values have been processed for a specific key. If the user wishes, he can get all the results back to the master and eventually combine them. To proceed to this last step, the worker creates an  $MasterToken$  data, which is *pinnedAndScheduled*. This operation means that the  $MasterToken$  is known from the BitDew scheduler but will not be sent to any nodes in the system. Instead,  $MasterToken$  is pinned on the Master node, which allows the result of the Reduce tasks, scheduled with a tag affinity set to  $MasterToken$ , to be sent to the Master.

#### D. Desktop Grid Features

We detail now some of the high level features that have been developed to address the characteristics of Desktop Grid systems.

*Latency hiding:* Because computing resources are spread over the Internet and because we cannot assume that direct communication between hosts is always possible due to firewall settings, the communication latency can be orders of magnitude higher than latency provided by interconnection networks found in clusters. One of the mechanisms to hide high latency in parallel systems is

---

**Algorithm 4** EXECUTION OF REDUCE TASKS

---

**Require:** Let  $Reduce$  be the Map function to execute  
**Require:** Let  $R$  be the number of Mappers and  $r$  a single worker  
**Require:** Let  $ir_r = ir_{1,r}, \dots, ir_{m,r}$  be the set of intermediate results received by reducer  $r$

1. {on the Master node}
  2. Create a single data  $MasterToken$
  3.  $pinAndSchedule(MasterToken)$
  - 4.
  5. {on the Worker node}
  6. **if**  $ir_{m,r}$  is scheduled **then**
  7.     execute  $Reduce(ir_{m,r})$
  8.     **if all**  $ir_r$  have been processed **then**
  9.         create  $o_r$  with  $affinity = MasterToken$
  10.     **end if**
  11. **end if**
  - 12.
  13. {on the Master node}
  14. **if all**  $o_r$  have been received **then**
  15.     Combine  $o_r$  into a single result
  16. **end if**
- 

overlapping communication with computation and prefetch of data. We have designed a multi-threaded worker (see Figure 3) which can process several concurrent files transfers, even if the protocol used is synchronous such as HTTP for instance. As soon as a file transfer is finished, the corresponding Map or Reduce tasks are enqueued and can be processed concurrently. The number of maximum concurrent Map and Reduce tasks can be configured, as well as the minimum number of tasks in the queue before computations can start. In MapReduce, prefetch of data is natural as data are staged on the distributed file system before computations are launched.

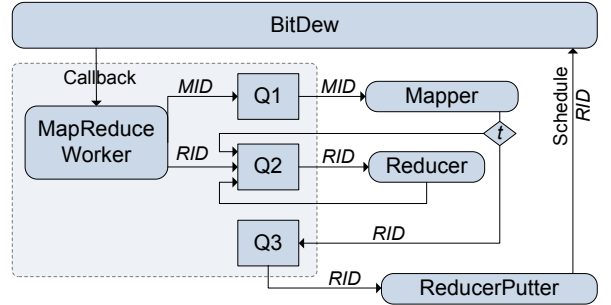


Figure 3. Worker handles the following threads: Mapper, Reducer and ReducerPutter. Q1, Q2 and Q3 represent *queues* and are used for communication between threads.

*Collective file operation:* MapReduce processing is composed of several collective file operations which in some aspects, look similar with collective communications in parallel programming such as MPI. Namely, the collectives are the *Distribution*, the initial distribution of file chunks (similar to `MPI_Scatter`), the *Shuffle*, that is the redistribution of intermediate results between the Mapper and the Reducer nodes (similar to `MPI_AlltoAll`) and the *Combine* which is the assemblage of the final result on the master node (similar to `MPI_Gather`). We have augmented the BitDew middleware with the notion of *DataCollection* to manipulate set of data as a whole, and *DataChunk*, to manipulate a part of data individually to implement the collective efficiently.

*Fault-tolerance:* In Desktop Grid, computing resources have high failure rates, therefore the execution runtime must be resilient to a massive number of crash failures. In the context of MapReduce,

failures can happen during the computation, either execution of Map or Reduce tasks, or during the communication, that is file upload and download. If a Map task fail, the Map input chunk must be distributed to another node in order to restart the Map task. To obtain this behavior, we simply toggle the flag `resilient` to the attribute `MapInputAttr`. Situation slightly differs if a Reducer node crashes because all the intermediate results should be sent to the replacing Reducer node. To do so, we enable the `resilient` flag to the `ReduceToken` data. Because intermediate results  $ir_{j,r}, j \in [1, m]$  have the `affinity` set to `ReduceToken_r`, they will be automatically downloaded by the node on which the `ReduceToken` data is rescheduled. To tolerate file transfer failure, we rely on BitDew which features file transfer reliability.

*Barrier-free computation:* Desktop Grids are not only prone to node crashes, but also to host churn, i.e. can join and leave the system at anytime. Because of fault tolerance, several replica of the same data can exist in the system, in particular, intermediate results can be duplicated. To tolerate the replication of intermediate results, we have adopted a naming scheme which allows Reducer nodes to detect that several versions of the same intermediate file exist in the queue. Thus, only the first version of the intermediate file is reduced. Traditional cluster implementation of MapReduce such as Hadoop introduces several barriers in the computation and in particular between the execution of Map and Reduce tasks. In Desktop Grid systems, because there can be a long period of time before nodes reconnect, it is necessary to remove any barriers so that the Reduce task can start as soon as intermediate files are produced. To do so, we have slightly changed the API of the Reduce task to allow the programmer to write the reduce function on segment of keys interval. Because the number of file chunks and the number of reducers are known in advance, we are able to detect when a reduction finishes. The early reduction combined with the replication of intermediate results allowed us to remove the barrier between Map and Reduce tasks.

*Scheduling:* Traditional scheduling associate computation to processing nodes. In contrast, our implementation relies on a two-level scheduler. First, the placement of data on host is ensured by the BitDew scheduler, which is mainly guided by the attribute properties given to data. However this would not be enough to efficiently steer the complex execution of MapReduce application. For instance, this would not allow to detect *lagers*, that is, nodes which spend an unusual long time to process the data and slow down the whole computation. To cope with this situation, workers periodically report to the master node the state of their ongoing computation using the MR service. The master node can then determine if there are more nodes available than tasks to execute. In this case, increasing the replication factor of the remaining tasks to compute can avoid the lagger effect. While this strategy is simple, it has been proved efficient[13].

*Distributed result checking:* A key security component of Desktop Grid systems is the result certification which is needed to check that malicious volunteers do not tamper with the results of a computation. Because intermediate results might be too large to be sent back to the server, result certification mechanism cannot be centralized as it is currently implemented in existing Desktop Grid systems. In consequence, we have adapted the majority voting heuristics, as it exists in BOINC to the decentralized network of storage Reducers. Once a reducer has obtained  $n$  out of  $p$  intermediate results, the result that appears most often is assumed to be

correct. This scheme is implemented by increasing the replica of each input map file, which in turn replicates intermediate results and by using the Reducer’s queue to buffer intermediate results. Although majority voting involves larger redundant computations, the heuristic is efficient at detecting erroneous results[14], which are likely to be significantly higher for disk bound jobs.

#### IV. PERFORMANCE EVALUATION

In this section, we present the performance evaluation of the MapReduce runtime environment. To measure precisely performances of the execution runtime, we conduct experiments within an environment where experimental conditions are reproducible. We used the *Grid Explorer (GdX)* cluster which is part of the Grid5000 infrastructure[15]. GdX is composed of 356 IBM eServer nodes featuring AMD Opteron CPU running at 2.4GHz with 2GB RAM interconnected by Gigabit Ethernet network. To emulate a Desktop Grid on GdX, we inject faults by killing worker processes, and simulate heterogeneity by launching concurrent processes.

##### A. Collective Communication

DataCollection and DataChunk are two new features specifically added to BitDew, which have not been benchmarked before. The first experiment aims at determining the optimal chunk size when sending/receiving large file. We run a PingPong benchmark that is “Ping” on one node and “Pong” on another node, using a 2.7GB large file and make the chunk size to vary between 5MB to 2.7GB. In consequence, the number of chunks varies from 540 to 1. The file transfer protocol used to send and receive data is the FTP protocol.

Figure 4 presents the execution time versus the chunk size. The execution time consists of splitting time, i.e. reading the input file and create file chunks, creation time, which involves computing the MD5 checksums of file chunks as well as registering data, time for uploading and downloading file, and the time for combining data chunks into a single file. As can be seen, the time to slice the file and create the corresponding chunks remains constant and does not depend on the chunk size. The operation of downloading and uploading the file reaches a minimum when chunks size is between 32MB and 80MB. We found that the optimal chunks size for our Desktop Grid system is similar to those observed in a cluster system [3]. In future work, we might reduce the splitting time by implementing striped file transfer when the file transfer protocol supports this operation.

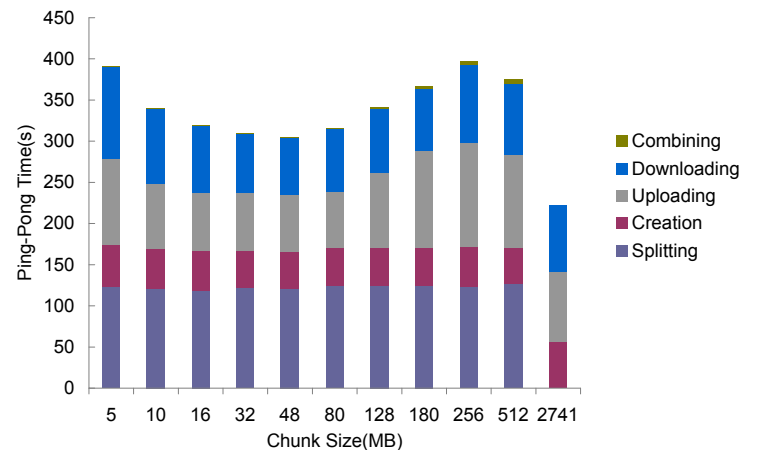


Figure 4. Breakthrough of the execution of the PingPong benchmark with varied chunk size.

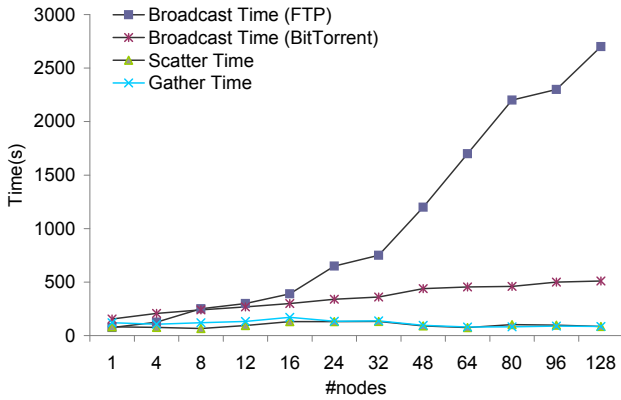


Figure 5. Execution time of the Broadcast, Scatter, Gather collective for a 2.7GB file to a varying number of nodes.

The next experiment evaluates the performance of the collective file operation. Considering that we need enough chunks to measure the Scatter/Gather collective, we selected 16MB as the chunks size for Scatter/Gather measurement and 48MB for the broadcast. The benchmark consists of the following: we prepare a 2.7GB large file into chunks and measure the time to transfer the chunks to all of the nodes according to the collective scheme. Broadcast measurement is conducted using both FTP and BitTorrent, while FTP protocol is used for Scatter/Gather measurement. The broadcast, scatter, gather results are presented respectively in the Figure 5. The time to broadcast the data linearly increases with the number of nodes because nodes compete for network bandwidth, and using BitTorrent obtains a better efficiency than using FTP in the same number of nodes. With respect to the gather/scatter, each node only downloads/uploads part of all chunks, and the time remain constant because the amount of data transferred does not increase with the number of nodes.

### B. MapReduce Evaluation

We evaluate now the performance of our implementation of MapReduce. The benchmark used is the WordCount application, which is a representative example of MapReduce application, adapted from the Hadoop distribution. WordCount counts the number of occurrences of each word in a large collection of documents. The file transfer protocol used is HTTP protocol.

The first experiment evaluates the scalability of our implementation when the number of nodes increases. Each node has a different 5GB file to process, splitted into 50 local chunks. Thus, when the number of nodes doubles, the size of the whole document counted doubles too. For 512 nodes<sup>2</sup>, the benchmark processes 2.5TB of data and executes 50000 Map and Reduce tasks. Figure 6 presents the throughput of the WordCount benchmark in MB/s versus the number of worker nodes. This result shows the scalability of our approach and illustrates the potential of using Desktop Grid resources to process a vast amount of data.

The next experiment aims at evaluating the impact of a varying number of mappers and reducers. Table II presents the time spent in Map function, the time spent in Reduce function and the total makespan of WordCount for a number of mappers varying from 4 to 32 and a number of reducers varying from 1 to 16. As expected, the Map and Reduce time decreases when the number of mappers and reducers increases. The difference between the makespan and

<sup>2</sup>GdX has 356 double core nodes, so to measure the performance on 512 nodes we run two workers per node on 256 nodes.

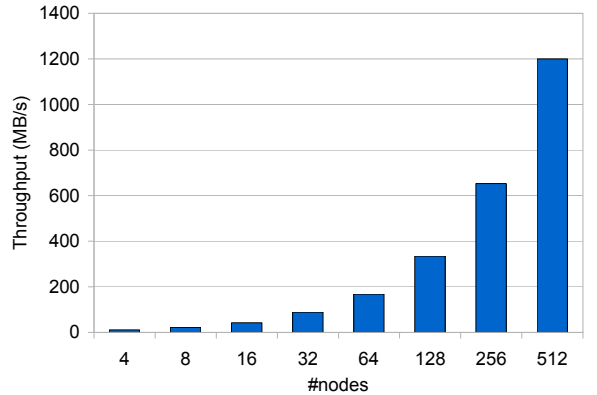


Figure 6. Scalability evaluation on the WordCount application: the y axis presents the throughput in MB/s and the x axis the number of nodes varying from 1 to 512.

Table II  
EVALUATION OF THE PERFORMANCE ACCORDING TO THE NUMBER OF MAPPERS AND REDUCERS.

#Mappers	4	8	16	32	32	32	
#Reducers	1	1	1	1	4	8	16
Map (sec.)	892	450	221	121	123	121	125
Reduce (sec.)	4.5	4.5	4.3	4.4	1	0.5	
Makespan (sec.)		473	246	142	146	144	150

the Map plus Reduce time is explained by the communication and time elapsed in waiting loops. Although the Reduce time seems very low compared to the Map time, this is typical of MapReduce application. A survey [16] of scientific MapReduce applications at the research Yahoo cluster showed that more than 93% of the codes where Map-only or Map-mostly applications.

### C. Desktop Grid Scenario

In this section, we emulate a Desktop Grid on the GdX cluster by confronting our prototype to scenarios involving host crashes, laggier hosts and slow network connection.

The first scenario aims at testing if our system is fault tolerant, that is, if a fraction of our system fails, the remaining participants are able to terminate the MapReduce application. In order to demonstrate this capacity, we propose a scenario where workers crash at different times: the first fault ( $F_1$ ) is a worker node crash while downloading a map input file, the second fault ( $F_2$ ) occurs during the map phase and the third crash ( $F_3$ ) happens after the worker has performed the map and reduce tasks. We execute the scenario and emulate worker crash by killing the worker process. In Figure 7 we report the events as they were measured during the execution of the scenario in a Gantt chart. We denote  $w_1 - w_5$  the workers and  $m$  the master node. The execution of our experiment begins with the master node uploading and scheduling two reduce token files:  $U_{t_1}$  and  $U_{t_2}$ . Worker  $w_1$  receives  $t_1$  and worker  $w_2$  receives  $t_2$ . Then, the master node uploads and schedules the map input files (chunks) ( $U_{C_1-5}$ ). Each worker downloads one such chunk, denoted with  $D_{C_1-5}$ . Node  $w_4$  fails ( $F_1$ ) while downloading map input chunk  $D_{C_4}$ . As the BitDew scheduler periodically checks whether participants are still present, after a short moment following the failure, node  $w_4$  is considered to be failed. This conveys to the rescheduling of  $C_4$  to node  $w_2$ . Node  $w_3$  fails ( $F_2$ ) while performing the map task  $M(C_3)$ . Then, chunk  $C_3$  is rescheduled to node  $w_5$ . At  $F_3$ , node  $w_1$  fails after having already performed the map task  $M(C_1)$  and several reduce tasks:  $R_{F_1,1}$ ,  $R_{F_1,2}$  and  $R_{F_1,5}$ . The notation  $R_{F_p,k}$  refers to the reduce task which takes as input the intermediate

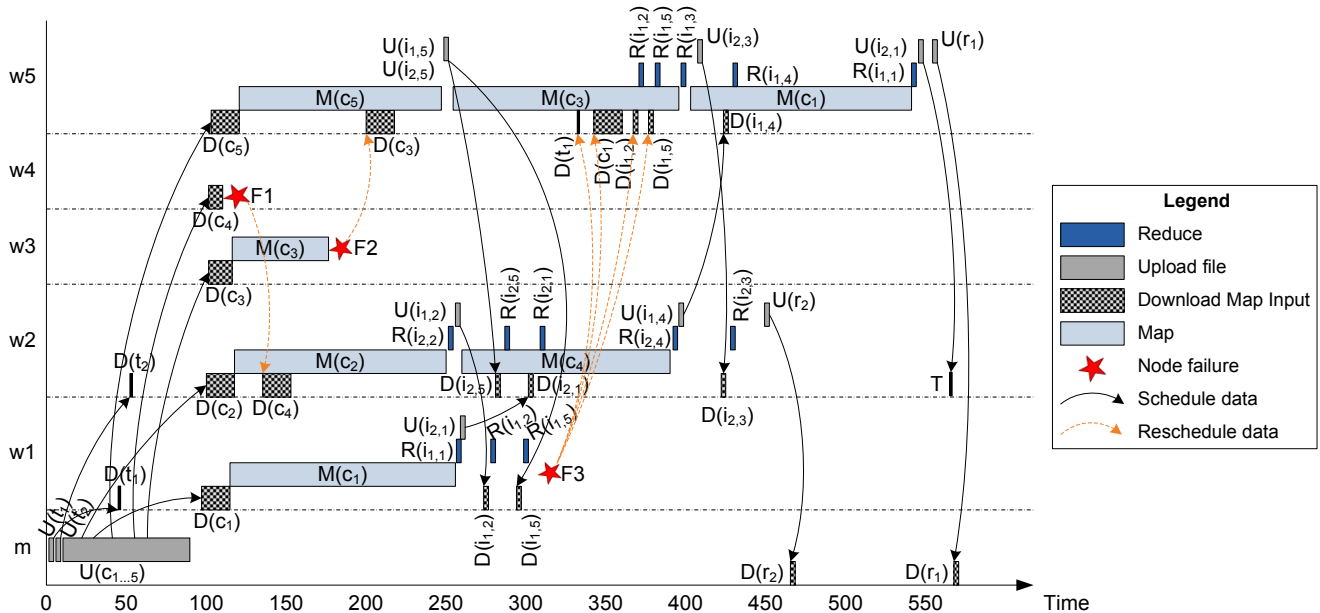


Figure 7. Fault-tolerance scenario.

result  $F_{i,k}$ , obtained from chunk  $C_k$ .  $F_3$  causes the BitDew scheduler to reschedule the token file  $t_1$ , the chunk  $C_1$  and the intermediates  $F_{1,2}$  and  $F_{1,5}$  to node  $w_5$ .

After finishing the map task  $M(C_1)$ , worker  $w_5$  performs the reduce task  $R(F_{1,1})$ . Then, it uploads and schedules the intermediate result  $F_{1,2}$  to node  $w_2$ . Node  $w_2$  downloads the respective intermediate result  $D(F_{2,3})$ , but without performing any further reduce task for this input (symbolized with  $G$ ). This is because every reducer keeps a list with already performed reduce tasks to prevent our system from processing multiple copies of the same intermediate result.

In the next scenario, we confront our system against the lagggers effect. To emulate a lagger, we launch several concurrent processes simultaneously to the worker process. Figure 8 shows the execution time averaged over 100 runs according to two parameters, the replication factor and the lagger rate. The lagger rate is the number of lagger hosts on the total number of hosts. The replication factor is the number of replica for the mappers input file (here, replicating the reducers has no time impact on the execution time). We launch the wordcount benchmark on  $16 * (1 + \#replica)$  nodes, so that there is enough nodes to run all the Map tasks. When there is no lagger, we obtain the reference execution time, that is around 25s in average. When there are lagggers and no replica, we obtain the lagger time, which is close to 6 times the reference time. We observe that increasing the number of replica significantly decreases the average execution time. Even a low number of replica can soften the lagger effect, as long as the the number of lagggers remain low.

## V. RELATED WORK

In the last years, several advances have been made in supporting data-intensive Bag-of-Tasks (DI-BoT) applications on Desktop Grids. In [17] and [18], authors investigate the use of the BitTorrent protocol with the XtremWeb and BOINC Desktop Grid. If the peer-to-peer approach seems efficient, it assumes that volunteers would agree that their desktop PC connects directly to another participant's machine to exchange data. Unfortunately, this could be seen as a potential security threat and is unlikely to be widely accepted by users. The second approach is to use a content delivery

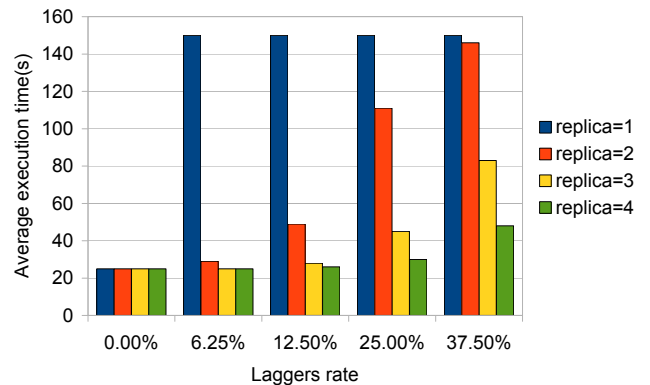


Figure 8. Lagger scenario. Average execution time for different lagggers rate and varying number of replica.

approach where files are distributed by a secure network of well-known and authenticated volunteers[19]. This approach is followed by the ADICS project (Peer-to-Peer Architecture for Data-Intensive Cycle Sharing)[20]. However data distribution in DI-BoT differs than MapReduce in the sense that the input file for the Map task is splitted in chunks, and thus the shareness between distributed data is less than of DI-BoT.

Only few initiatives associate data-intensive computing with large scale distributed storage on volatile resources. In [21], the authors present an architecture following the super-peer approach where the super-peers serve as cache data server, handle jobs submissions and coordinate execution of parallel computations.

There exists many MapReduce implementations for various architectures[22], [23] and systems[24], [3], but few specific implementations of MapReduce for Desktop Grid have been proposed. The closest work to ours is Moon[10] which stands for MapReduce On Opportunistic eNvironments. The starting point is adapting Hadoop to Desktop Grid context. Firstly, it has showed that without modification Hadoop and its HDFS file system were not adapted to high volatility context, partly because HDFS creates too many Map input replicas, while Hadoop does not create replica of intermediate results. Next, several enhancements are proposed to the scheduler in particular to take advantage of hybrid architecture where some of the

computing nodes are stable and dedicated to MapReduce execution. However, Moon does not demonstrate that Hadoop would work in multi-domain or Internet Desktop Grid. Hadoop assumes that Reducers can fetch intermediate results directly from the Mappers. In contrast, our solution does not assume any direct communication between the nodes. Instead, file transfer and scheduling are realized through the BitDew services. Readers can find in [11], experiments where hosts on the broadband Internet can exchange files using BitDew despite firewall, NAT and limited connectivity.

## VI. CONCLUSION

We have introduced an implementation of MapReduce for Desktop Grid that leverages on the BitDew middleware. Our prototype features massive fault tolerance, replica management, barrier-less MapReduce, latency-hiding mechanism, dedicated 2-level scheduler as well as distributed result checking. The scalability test shows that we achieve linear speedup on the classical WordCount benchmark. Several scenarios involving lagging hosts and numerous host crashes demonstrate that our approach is suitable for large scale and loosely connected Internet Desktop Grid. However to be really efficient, input data have to be highly shared between hosts so that P2P protocol can be used to distribute the data or frequently used by tasks in order to increase the computation/communication ratio. Our future work will focus on improving the scheduling heuristics, providing QoS and improving distributed result checking.

In conclusion, we showed that although MapReduce is significantly more complex than traditional Bag-of-Tasks application, it is possible to build a runtime efficient and secure to enable data-intensive application on Desktop Grid.

## ACKNOWLEDGMENT

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>).

This work is partly supported by the Romanian Authority for Scientific Research under doctoral scholarship no. 399/2008 and project IDEI 2452 and by the French Agence National de la Recherche through the MapReduce ARPEGE grant (22265) as well as INRIA ARC BitDew.

## REFERENCES

- [1] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - A Hunter of Idle Workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*. Washington, DC: IEEE Computer Society, 1988, pp. 104–111.
- [2] D. Anderson and G. Fedak, "The Computational and Storage Potential of Volunteer Computing," in *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, Singapore, May 2006, pp. 73–80.
- [3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. USA: USENIX Association, 2004, pp. 137–149.
- [4] G. Fedak, C. Germain, V. Néri, and F. Cappello, "XtremWeb: A Generic Global Computing Platform," in *Proceedings of 1st IEEE International Symposium on Cluster Computing and the Grid (CCGRID'2001), Special Session Global Computing on Personal Devices*, Brisbane, Australia, May, pp. 582–587.
- [5] D. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th IEEE/ACM International GRID Workshop*, Pittsburgh, USA, 2004, pp. 4–10.
- [6] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray, "Labs of the world, unite!!!" *Journal of Grid Computing*, vol. 4, no. 3, pp. 225–246, September 2006.
- [7] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards Efficient MapReduce Using MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting*. Springer, Sep. 2009, pp. 240–249.
- [8] L. F. G. Sarmenta, "Sabotage-Tolerance Mechanisms for Volunteer Computing Systems," *Future Generation Computer Systems*, vol. 18, no. 4, pp. 561–572, 2002.
- [9] "Hadoop. <http://hadoop.apache.org/>."
- [10] H. Lin, J. Archuleta, X. Ma, and W. Feng, "Moon: Mapreduce on opportunistic environments," in *ACM International Symposium on High Performance Distributed Computing (HPDC)*, Chicago, 2010.
- [11] G. Fedak, H. He, and F. Cappello, "BitDew: A Data Management and Distribution Service with Multi-Protocol and Reliable File Transfer," *Journal of Network and Computer Applications*, vol. 32, no. 5, pp. 961–975, Sep. 2009.
- [12] S. Ghemawat, H. Gobioff, and S. T. Leung, "The google file system," in *SOSP'03: Proceeding of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 29–43.
- [13] D. Kondo, A. Chien, and H. Casanova, "Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids," in *ACM Conference on High Performance Computing and Networking (SC'04)*, Pittsburgh, 2004.
- [14] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L. M. Silva, G. Fedak, and F. Cappello, "Characterizing Result Errors in Internet Desktop Grids," in *European Conference on Parallel and Distributed Computing EuroPar'07*, Rennes, France, August 2007, pp. 361–371.
- [15] R. Bolze and all, "Grid5000: A Large Scale Highly Reconfigurable Experimental Grid Testbed," *International Journal on High Performance Computing and Applications*, pp. 481–494, 2006.
- [16] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, Melbourne, Australia, 2010, pp. 94–103.
- [17] B. Wei, G. Fedak, and F. Cappello, "Towards Efficient Data Distribution on Computational Desktop Grids with BitTorrent," *Future Generation Computer Systems*, vol. 23, no. 7, pp. 983–989, Nov. 2007.
- [18] F. Costa, L. Silva, G. Fedak, and I. Kelley, "Optimizing Data Distribution in Desktop Grid Platforms," *Parallel Processing Letters*, vol. 18, no. 3, pp. 391–410, September 2008.
- [19] C. Mastroianni, P. Cozza, D. Talia, I. Kelley, and I. Taylor, "A scalable super-peer approach for public scientific computation," *Future Generation Computer Systems*, vol. 25, no. 3, pp. 213 – 223, 2009.
- [20] I. Kelley and I. Taylor, "Bridging the data management gap between service and desktop grids," in *Distributed and Parallel Systems*, Springer, Ed., Hungary, 2008, pp. 13–26.
- [21] E. Cesario, N. Caria, C. Mastroianni, and D. Talia, "Distributed data mining using a public resource computing framework," in *Proc. of CoreGrid Workshop*, Delft, Netherlands, 2009, pp. 34–44.
- [22] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 13th Intl. Symp. on High Performance Computer Architecture (HPCA)*, Phoenix, AZ, 2007, pp. 13–24.
- [23] B. He, W. Fang, N. K. Govindaraju, Q. Luo, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Parallel Architectures and Compilation Techniques (PACT)*, Toronto, Canada, 2008.
- [24] B. Nicolae, G. Antoniu, and L. Boug, "Enabling high data throughput in desktop grids through decentralized data and metadata management: The blobseer approach," in *Proceedings of EuroPar*, Delft, Netherlands, 2009, pp. 404–416.