freeCycles - Efficient Data Distribution for Volunteer Computing

Rodrigo Bruno
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa
Lisboa, Portugal
rodrigo.bruno@tecnico.ulisboa.pt

ABSTRACT

Volunteer Computing (VC) is a very interesting solution to harness large amounts of computational power, network bandwidth, and storage which, otherwise, would be left with no use. In addition, recent developments of new programming paradigms, namely MapReduce, have raised the interest of using VC solutions to run MapReduce applications on the large scale Internet. However, current data distribution techniques, used in VC to distribute the high volumes of information which are needed to run MapReduce jobs, are naive, and therefore need to be re-thought.

We present a VC solution called freeCycles, that supports MapReduce jobs and provides two new main contributions: i) improves data distribution (among the data server, mappers, and reducers) by using the BitTorrent protocol to distribute (input, intermediate, and output) data, and ii) improves intermediate data availability by replicating it through volunteers in order to avoid losing intermediate data and consequently preventing big delays on the MapReduce overall execution time.

We present the design and implementation of freeCycles along with an extensive set of performance results which confirm the usefulness of the above mentioned contributions, improved data distribution and availability, thus making VC a feasible approach to run MapReduce jobs.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Data communications*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

General Terms

Algorithms, Performance, Realiability

Keywords

Volunteer Computing, BitTorrent, BOINC, MapReduce

1. INTRODUCTION

With the ever growing demand for computational power, scientists and companies all over the world strive to harvest more computational resources in order to solve more and bigger problems in less time, while spending the minimum money possible. With these two objectives in mind, we think of volunteer computing (VC) as a viable solution to access huge amounts of computational resources, namely CPU cycles, network bandwidth, and storage, while reducing the cost of the computation (although some commercial solutions give incentives, pure VC projects can produce computation at zero cost).

With time, more computing devices (PCs, gaming consoles, tablets, mobile phones, and any other kind of device capable of sharing its resources) join the network. By aggregating all these resources in a global volunteer pool, it is possible to obtain huge amounts of computational resources that would be impossible, or impractical, for most grids, supercomputers, and clusters. For example, recent data from BOINC [3, 4] projects shows that, currently, there are 50 supported projects sustained by an average computational power of over 7 PetaFLOPS[1].

The creation and development of large scale VC systems enables large projects, that could not be run on grids, supercomputers, or clusters due to its size and/or costs associated, to run with minimized costs. VC also provides a platform to explore and create new projects without losing time building and for setting up execution environments like clusters.

In addition, recent developments of new programming paradigms, namely MapReduce[2] [9], have raised the interest of using VC solutions to run MapReduce applications on large scale networks, such as the Internet. Although increasing the attractiveness of VC, it also brings the need to rethink and evolve current VC platforms' architectures and protocols (in particular, the data distribution techniques and protocols) to adapt to these new programming paradigms.

We present a system called freeCycles, a VC solution which

[1]Statistics from boincstats.com

[2]MapReduce is a popular programming paradigm composed of two operations: Map (applied for a range of values) and Reduce (operation that will aggregate values generated by the Map operation).

has as its ultimate goal to aggregate as many volunteer resources as possible, in order to run MapReduce jobs in a scalable and efficient way. Therefore, the output of this project is a middleware platform that enables upper software abstraction layers (programs developed by scientists, for example) to use volunteer resources all over the Internet as they would use a supercomputer or a cluster. Therefore, using freeCycles, applications can be developed to solve problems that require more computational resources than what it is available in private clusters, grids or even in supercomputers.

To be successful, freeCycles must fulfill several key objectives. It must be capable of scaling up with the number of volunteers; it must be capable of collecting volunteers' resources such as CPU cycles, network bandwidth, and storage in an efficient way; it must be able to tolerate byzantine and fail-stop failures (which can be motivated by unreliable network connections, that are very common in large scale networks such as the Internet). Finally, we require our solution to be capable of supporting new parallel programming paradigms, in particular MapReduce, given its relevance for a large number of applications.

MapReduce applications commonly have two key features: i) they depend on large amounts of information to run (for both input and output) and ii) may run for several cycles (as the original algorithm for page ranking). Therefore, in order to take full advantage of the MapReduce paradigm, freeCycles must be capable of distributing large amounts of data (namely input, intermediate and output data) very efficiently while allowing applications to perform several MapReduce cycles without compromising its scalability.

In order to understand the challenges inherent to building a solution like freeCycles, it is important to note the difference between the main three computing environments: clusters, grids, and volunteer pools. Clusters are composed by dedicated computers so that every node has a fast connection with other nodes, node failure is low, nodes are very similar in hardware and software and all their resources are focused on cluster jobs. The second computing environment is a grid. Grids may be created by aggregating desktop computers from universities, research labs or even companies. Computers have moderate to fast connection with each other and grids may be inter-connected to create larger grids; node failure is moderate, nodes are also similar in hardware and software but their resources are shared between user tasks and grid jobs. At the end of the spectrum there are volunteer pools. Volunteer pools are made of regular desktop computers, owned by volunteers around the world. Volunteer nodes have variable Internet connection bandwidth, node churn is very high (since computers are not managed by any central entity), nodes are very asymmetrical in hardware and software and their computing resources are mainly focused on user tasks. Finally, as opposed to grids and clusters, volunteer computers cannot be trusted since they may be managed by malicious users.

Besides running on volunteer pools, freeCycles is also required to support MapReduce, a popular data-intensive programming paradigm, that was initially created for clusters and that depends on large amounts of data to run. Therefore, feeding large inputs to all volunteer nodes through unstable Internet connections, and coping with problems such as node churn, variable node performance and connection, and being able to do it in a scalable way is the challenge that address with freeCycles.

By considering all the available solutions, it is important to note that solutions based on clusters and/or grids do not fit our needs. Such solutions are designed for controlled environments where node churn is expected to be low, where nodes are typically well connected with each other, nodes can be trusted and are very similar in terms of software and hardware. Therefore, solutions like HTCondor [24], Hadoop [25], XtremWeb [11] and other similar computing platforms are of no use to attain our goals.

When we turn to VC platforms, we observe that most existing solutions are built and optimized to run Bag-of-Tasks applications. Therefore, solutions such as BOINC, GridBot [22], Bayanihan [19] and many others [2, 20, 26, 5, 6, 17, 16, 21, 15] do not support the execution of MapReduce jobs, which is one of our main requirements.

With respect to the few solutions [14, 23, 10, 8] that support MapReduce, we are able to point out some frequent problems. First, data distribution (input, intermediate, and final output) is done using naive techniques. Current solutions do not take advantage of task replication (that is essential to avoid stragglers and to tolerate byzantine faults). Thus, even thought multiple mappers have the same output, reducers still download intermediate data from one mapper only. The second problem is related to intermediate data availability. It was shown [13] that if some mappers fail and intermediate data becomes unavailable, a MapReduce job can be delayed by 30% of the overall computing time. Current solutions fail to prevent intermediate data from becoming unavailable. The third problem comes from the lack of support for applications with more than one MapReduce cycle. Current solutions typically use a central data server to store all input and output data. Therefore, this central server is a bottleneck between every cycle (since the previous cycle needs to upload all output data and the next cycle needs to download all the input data before starting).

In order to support the execution of programming paradigms like MapReduce, that need large amounts of data to process, new distribution techniques need to be employed. In conclusion, current VC solutions raise several issues that need to be improved in order to be able to run efficiently new parallel programming paradigms, MapReduce in particular, on large volunteer pools.

freeCycles is a BOINC-compatible VC platform that enables the deployment of MapReduce applications. Therefore, freeCycles provides the same features as BOINC regarding scheduling, tasks replication and verification. Besides supporting MapReduce jobs, freeCycles goes one step further by allowing volunteers (mappers or reducers) to help distributing both the input, intermediate output and final output data (through the BitTorrent[3] protocol and not point-to-point). Since multiple clients will need the same input and will produce the same output (for task replication purposes), multiple clients can send in parallel multiple parts of the same file to other clients or to the data server. freeCycles

[3]Official BitTorrent specification can be found at www.bittorrent.org

will therefore benefit from clients' network bandwidth to distribute data to other clients or even to the central server. As a consequence of distributing all data through the BitTorrent protocol, freeCycles allows sequences of MapReduce cycles to run without having to wait for the data server, i.e., data can flow directly from reducers to mappers (of the next cycle). Regarding the availability of intermediate data, freeCycles proposes an enhanced work scheduler that adapts the replication factor for map tasks to minimize the risk of loosing intermediate data.

By providing these functionalities, freeCycles achieves higher scalability (reducing the burden on the data server), reduced transfer time (improving the overall turn-around time), and augmented fault tolerance (since nodes can fail during the transfer without compromising the data transfer).

2. RELATED WORK

In this section we analyse and discuss the systems that we found to be closer to freeCycles. In addition, we also analyse and discuss some data distribution protocols that we considered when designing freeCycles.

2.1 Computing Platforms

From all the existent VC solutions, most of them are focused and optimized to run Bag-of-Tasks (embarrassingly parallel) applications, and thus, cannot directly support MapReduce applications. Nevertheless, as MapReduce's popularity increased, some platforms decided to use volunteer resources to run MapReduce tasks. Therefore, solutions such as SCOLARS [8], MOON [14], Tang [23] and Marozzo [10] already support MapReduce applications.

MOON (MapReduce On Opportunistic Environments) is an extension of Hadoop (an open source implementation of MapReduce). MOON ports MapReduce to opportunistic environments mainly by: i) modifying both data and task scheduling (to support two types of nodes, stable and volatile nodes), and ii) performing intermediate data replication. However, although MOON was designed to run MapReduce tasks on volunteer nodes, it still relies on a large set of dedicated nodes (mainly for hosting dedicated data servers). This assumption does not hold in a pure volunteer computing setting.

The solution presented by Tang [23] is a MapReduce implementation focused on desktop grids. It was built on top of a data management framework, Bitdew [12]. Even though BitDew supports BitTorrent, the authors do not state if it was used to distribute all the data (input, intermediate, and output). Moreover, there is no performance evaluation with other applications than word count and no performance evaluation with simulated Internet connections (with limited upload bandwidth). Notwithstanding, we believe that BitDew would produce high overhead in a VC setting and it also assumes high availability of a particular set of management nodes (which is prohibitive in a volunteer pool).

Marozzo [10] presents a solution to exploit the MapReduce model in dynamic environments. The major drawbacks of this solution are: i) data is distributed point-to-point (which fails to fully utilize the volunteer's bandwidth), and ii) there is no intermediate output replication.

SCOLARS (Scalable Complex Large Scale Volunteer Computing) is a modified version of BOINC that supports MapReduce applications and presents two additional contributions: i) inter-client transfers (for intermediate data only), and ii) hash based task validation (where only a hash of the intermediate output is validated on the central server). However, it presents the same issues as the previous solution: only point-to-point transfers, and no intermediate data replication.

In conclusion, all the analysed solutions present problems that invalidate them as candidate solutions for the problem we are addressing. Additionally, no solution showed that it was able to fully utilize the volunteer's upload bandwidth, and no solution showed that it was possible to run multiple MapReduce cycles, while avoiding a bottleneck on the data server.

2.2 Data Distribution Protocols

Regarding data distribution systems, we analysed several ones but we only discuss those we think that are relevant to our objective (distribute efficiently large amounts of data within a large set of nodes): FastReplica [7] and BitTorrent [18].

FastReplica is a replication algorithm focused on replicating files in Content Distribution Networks (CDNs). It was designed to be used in large-scale distributed networks of servers and it uses a push model (where the sender triggers the data transfer). FastReplica works in two steps: i) distribute equally sized pieces of the original data among the destination servers; ii) all destination servers send to all other destination servers their piece (at the end of this step, all destination servers can merge all pieces to obtain the original file).

Despite being very efficient, we point out two main issues: i) FastReplica relies on a push model, which can be very difficult to use when there is a variable set of destination nodes; ii) it does not cope with node failure (since it was designed for servers, which are supposed to be up most of the time).

BitTorrent is a peer-to-peer data distribution protocol widely used to distribute large amounts of data over the Internet. It was designed to avoid the bottleneck of file servers (like FTP servers). BitTorrent can be used to reduce the server and network impact of distributing large files by allowing ordinary users to spare their upload bandwidth to spread the file. So, instead of downloading the whole file from a server, a user can join a swarm of users that share a particular file and thereafter, download and upload small file chunks from and to multiple users until the user has all the file chunks. Using this protocol, it is possible to use computers with low bandwidth connections and even reduce the download time (compared to a centralized approach).

In order to find other users sharing the target file, one has to contact a BitTorrent Tracker. This tracker has to maintain some information about the nodes such as its IP address, port to contact and available files. This information is kept so that when a node asks for a file, the tracker is able to return a list of nodes to whom the node should contact.

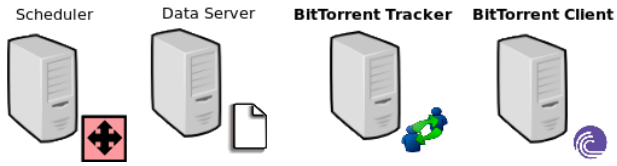

Figure 1: Server-Side Architecture

This protocol has been proving to be capable of scaling up to millions of users and providing high efficiency in distributing large amounts of data [18]. Thus, we decided to integrate this protocol in our solution given that i) it enables a pull model where data can flow as volunteers need it (as opposed to FastReplica), ii) it is more reliable to faults (if one client fails, other clients can still download the file), and iii) it scales up to millions of users.

3. FREECYCLES

freeCycles is a MapReduce-enabled and BOINC-compatible VC platform. It provides two new contributions: i) efficient data distribution (for input, intermediate output, and final output) by using the BitTorrent protocol and ii) enhanced task scheduling that minimizes the loss of intermediate data. Throughout this section, we describe freeCycles' architecture and data distribution algorithms (for both server and client sides). As some of its basic components are already present in BOINC, we focus only on freeCycles' extensions.

The basic server-side architecture is composed by several components: i) a database where information about jobs is held; ii) a data server where input and output data are stored; and iii) a scheduler that is responsible for several tasks (such as creating tasks, scheduling tasks, validating tasks). freeCycles adds two additional components (highlighted in bold in figure 1): a BitTorrent tracker (to enable volunteers to u se the BitTorrent protocol to download and upload data to and from clients and the central server), and a BitTorrent client (that is used to share the initial input and to receive the final output through the BitTorrent protocol).

As for the client-side architecture, freeCycles reuses the BOINC client runtime (that manages all the issues related to server communication, process management and so on). To support the BitTorrent protocol, freeCycles augments the client software with a BitTorrent client. Therefore, all volunteers can download and upload data to and from other volunteers or the data server.

By including such BitTorrent clients and a tracker, freeCycles is able to move data between clients (mappers and reducers) and server taking advantage of the download and upload bandwidth available at the volunteer nodes. This brings a significant decrease in the used bandwidth and load of the central server.

3.1 Data Distribution Algorithm

Having described the components on (client-side) volunteers and on the server, we now detail how we use the BitTorrent file sharing protocol to coordinate input, intermediate and final output data transfers. Still on our data distribution algorithm, we show how freeCycles is able to run multiple MapReduce cycles without compromising its scalability (i.e. avoiding high burden on the data server).

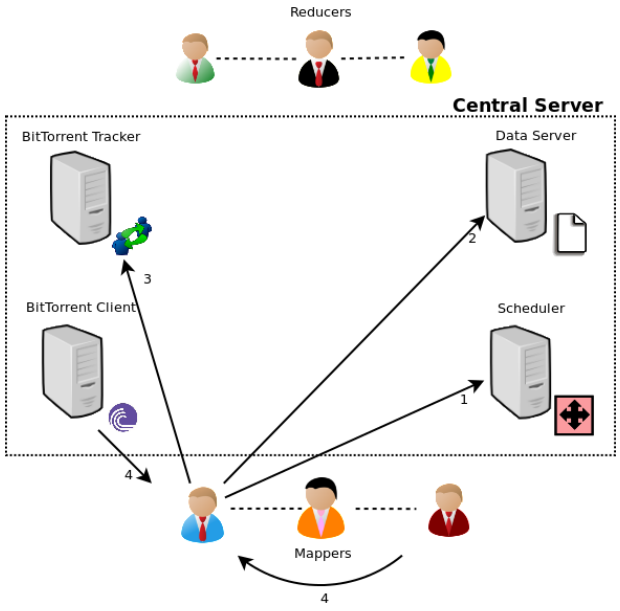

Figure 2: Input Data Distribution

3.1.1 Input Distribution

Input distribution is the very first step in every MapReduce application. Input must be split over multiple mappers. To do so, each mapper downloads a `.torrent` file [4] pointing to the corresponding input file in the central data server.

For each input file, the server plays as initial seed (the origin). If we take into consideration that each map task is replicated over at least three volunteers (for replication purposes), then, when a new map task begins, the volunteer will have at least one seed (the data server) and, possibly up to the task replication factor minus one, additional volunteers sharing the file (each volunteer shares all the input file chunks, that it owns, using the BitTorrent protocol).

Therefore, we can leverage the task replication mechanisms to share the burden of the data server. Even if the server is unable to respond, a new mapper may continue to download its input data from other mappers. The transfer bandwidth will also be bigger since a mapper may download input data from multiple sources.

Input data distribution is done as follows (see Figure 2): 1) a new volunteer requests work from the central server scheduler and receives a workunit;[5] 2) the new mapper downloads the `.torrent` file from the data server (a reference of the `.torrent` file was inside the workunit description file); 3) the new mapper contacts the BitTorrent tracker to know about other volunteers sharing the same data; 4) the volunteer starts downloading input data from multiple mappers and/or from the server.

[4] A .torrent file is a special metadata file used in the BitTorrent protocol. It contains several fields describing the files that are exchanged using BitTorrent. A `.torrent` file is unique for a set of files to be transfered (since it contains a hash of the files).

[5] A workunit is a concept used in BOINC to refer a computational task that is shipped to volunteer resources. A workunit contains all the information needed to run a task.

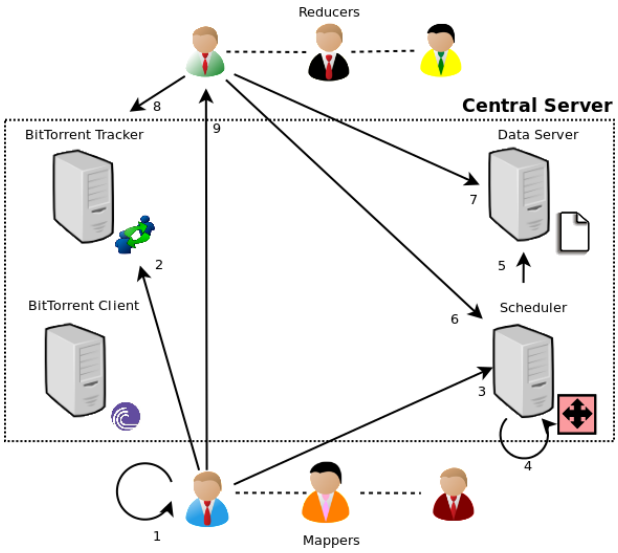

Figure 3: Intermediate Data Distribution

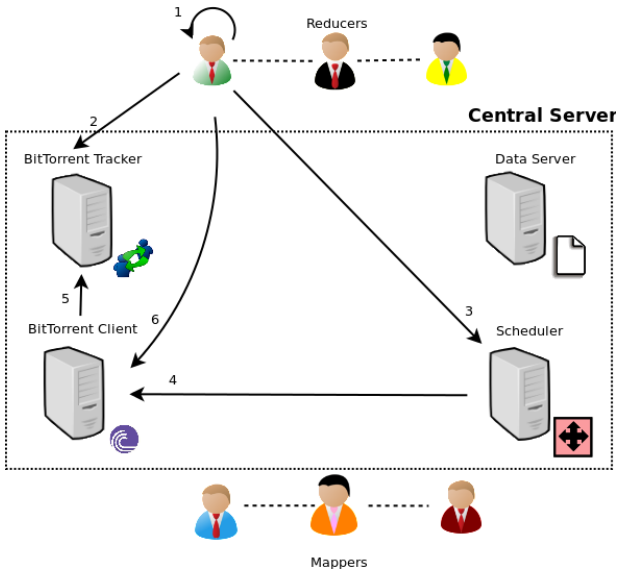

Figure 4: Output Data Distribution

3.1.2 *Intermediate Output Distribution*

Once a map task is finished, the mapper has an intermediate output ready to be used. The first step is to create a `.torrent` file. From this point on, the mapper is able to share its intermediate data using the BitTorrent protocol: the BitTorrent client running at the volunteer node automatically informs the BitTorrent tracker, running at the central server, that some intermediate files can be accessed through this volunteer. The second step is to make the server aware of the map task finish. To this end, the mapper contacts the server and sends the `.torrent` file just created for this intermediate output.

As more `.torrent` files arrive at the server, the server is able to decide (using a quorum of results) which volunteers (mappers) have the correct intermediate files comparing the hashes (that came inside the `.torrent` files). When all the intermediate outputs are available, the server shuffles all `.torrent` files and prepares sets of inputs, one for each reduce task. When new volunteers request work, the scheduler starts issuing reducer tasks. These reducer tasks contain references to the `.torrent` files that were successfully validated and that need to be downloaded. Once a reducer has access to these `.torrent` files, it starts transferring the intermediate files (using the BitTorrent protocol) from all the mappers that completed the map task with success. Reduce tasks start as soon as all the needed intermediate values are successfully transfered.

Figure 3 illustrates the steps for the intermediate data distribution: 1) the mapper computes a `.torrent` file for each of its intermediate outputs; 2) the mapper informs the central BitTorrent tracker that it has some intermediate data ready to share; 3) a message acknowledging the map task finish and containing the computed intermediate output hashes (`.torrent` files) is sent to the central scheduler; 4) the server validates and shuffles all intermediate output hashes; 5) then, all `.torrent` files are stored in the data server; 6) when a new volunteer (reducer) asks for work, a workunit is delivered with references to several `.torrent` files (one for each intermediate output); 7) the reducer downloads the `.torrent` files from the data server; 8) it then contacts the BitTorrent tracker to know the location of mapper nodes that hold intermediate data; 9) the reducer uses its BitTorrent client to download the intermediate data from multiple mappers.

3.1.3 *Output Distribution*

Given that map and reduce tasks are replicated over at least three volunteers, it is possible to accelerate the upload of the final output files from the reducers to the data server.

The procedure is similar to the one used for intermediate outputs. Once a reduce task finishes, the reducer computes a `.torrent` file for its fraction of the final output. Then, it informs the BitTorrent tracker that some output data is available at the reducer node. The next step is to send a message to the central scheduler containing the `.torrent` file and acknowledging the task finish. Once the scheduler has received enough results from reducers, it can proceed with validation and decide which `.torrent` files will be used to download the final output. All the trustworthy `.torrent` files are then used by the BitTorrent client at the central server to download the final output.

Using BitTorrent to transmit the final outputs results in a faster transfer from volunteers to the data server, a lower and shared bandwidth consumption from the volunteer's perspective, and an increased fault tolerance (since a volunteer node failure will not abort the file transfer).

Output data distribution works as follows (see Figure 4): 1) `.torrent` file is generated for the fraction of the final output; 2) the reducer informs the BitTorrent tracker that it has some data to share; 3) a message is sent to the central scheduler acknowledging the task finish and reporting the `.torrent` file; 4) the central server is able to validate results and gives some `.torrent` files to the BitTorrent client at the server; 5) the BitTorrent client contacts the BitTorrent tracker to know the location of the reducers; 6) final output files are downloaded from multiple vounteer nodes.

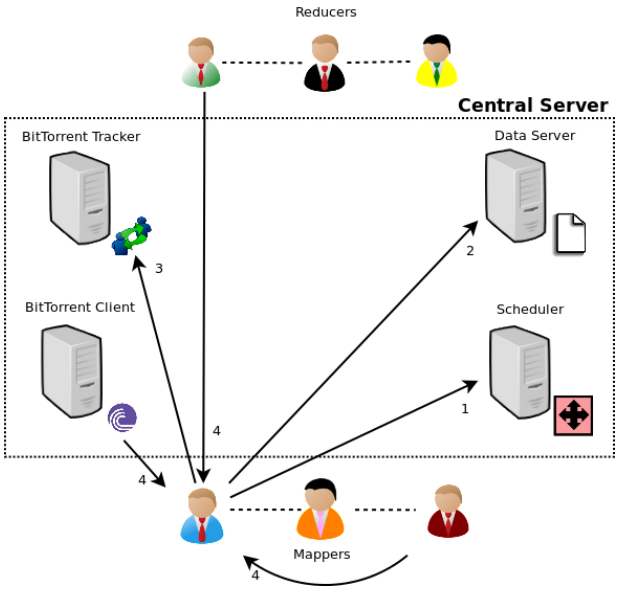

Figure 5: Data Distribution Between MapReduce Cycles

3.1.4 Multiple MapReduce Cycles

Using the data distribution techniques just described, where the central server and all volunteers have a BitTorrent client and use the BitTorrent Tracker to find peers with data, it is very easy to use freeCycles for running applications that depend on multiple MapReduce cycles. The difference between our solution and previous ones (namely SCOLARS) is that output data need not to go to the central server before it is delivered to new mappers (i.e., data can flow directly from reducers to mappers from one cycle to the other).

From mappers's perspective, when a workunit is received, volunteers ask the BitTorrent tracker for nodes with the required data. It does not differentiate between the single cycle scenario (where the volunteer downloads from the central server) or the multiple cycle scenario (where the volunteer downloads from reducers of the previous cycle). Regarding the central server's perspective, the scheduler only needs to know that some map tasks depend on the output of some reduce tasks (more details in the next section).

Figure 5 shows how volunteers and server interact to feed the new cycle with the output from the previous one (assuming that all steps in figure 4 are finished): 1) when new volunteers ask for work, the scheduler delivers new map tasks with references to the `.torrent` files sent by the reducers (of the previous cycle); 2) the new mappers download the `.torrent` files from the data server; 3) after retrieving the `.torrent` files, the mappers ask the BitTorrent tracker for nodes that are sharing the output data; 4) mappers from the next cycle can now download output data from multiple reducers (from the previous cycle).

3.2 Availability of Intermediate Data

Previous studies [13] show that the availability of intermediate data is a very sensitive issue in programming paradigms like MapReduce. The problem is that, for performance reasons, typical implementations do not replicate intermediate results. However, when applied to (MapReduce) volunteer computing, where node churn is very high, such lack of replication leads to a loss of intermediate data. It was shown that loosing a fraction of intermediate data incurs a 30% delay of the overall MapReduce execution time. To cope with this problem, freeCycles presents two methods.

First, replicate map tasks aggressively when volunteers designated to execute a particular map task take too long to report results. By imposing a shorter interval time to respond with results, we make sure that we keep at least a few replicas of the intermediate output.

Second, when there are intermediate outputs that have already been validated (by the central server) and other map tasks are still running, new tasks designed to replicate intermediate output might be delivered to new volunteers. Therefore, volunteers might be used to replicate intermediate data to compensate other mappers that might die while waiting for the reduce phase to start. These tasks would simply download `.torrent` files and use them to start downloading intermediate data.

4. IMPLEMENTATION

In this section we explain how freeCycles is implemented and how to use our VC solution to create and distribute MapReduce applications over a volunteer pool. We further present a detailed description of all its components and how they cooperate with each other.

freeCycles is implemented directly on top of BOINC. It does not, however, change BOINC's core implementation since: i) it would create a dependency between our project and a specific version of BOINC; ii) it would be impossible to have a single BOINC server hosting MapReduce and non-MapReduce projects at the same time.

Assuming a working BOINC server, to start running MapReduce jobs on volunteer nodes and using BitTorrent to distribute data, one has to:

1. create a BOINC project;
2. replace and/or extend the provided implementations for some of the project specific daemons (work generator and assimilator);
3. use and/or adapt the provided script to prepare input data and the project configuration file;
4. reuse and/or extend our sample MapReduce application and provide a Map and a Reduce function implementations;
5. start the project.

4.1 Server

As a normal BOINC server, freeCycles' server has two global components (a database and a data server) and several project specific daemons (mainly the work scheduler, work generator, validator, and assimilator).

In order to support the BitTorrent protocol, freeCycles adds two new global components: a BitTorrent client (to share

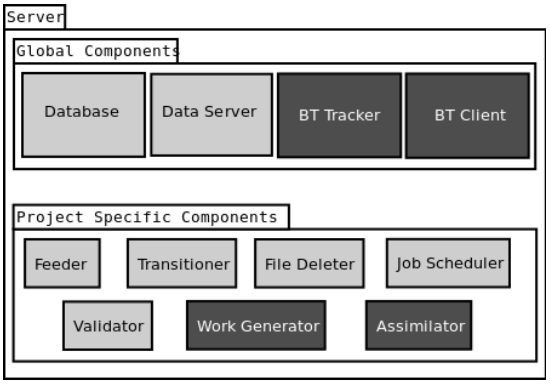

Figure 6: freeCycles Server-side Implementation

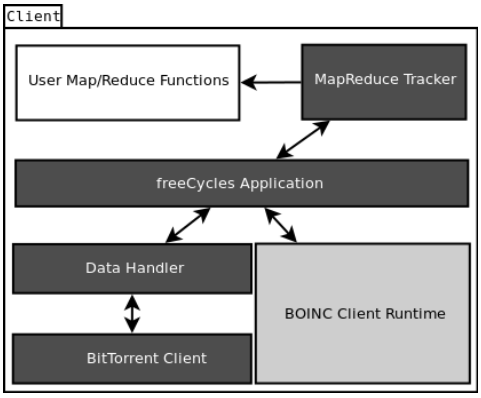

Figure 7: freeCycles Client-side Implementation

the initial input and to retrieve the final output), and a BitTorrent tracker that is used (by all BitTorrent clients, both on the server and client sides) to know the locations of other BitTorrent clients sharing a desired file.

Regarding the handling of MapReduce applications, freeCycles provides modified implementations for two of the project specific daemons, namely: the work generator (daemon that creates tasks) and the work assimilator (daemon that closes tasks when finished).

These two special implementations are needed to introduce the dependency between reduce tasks and map tasks (since the reduce phase can only start when all map tasks are finished). Therefore, when a task is finished, the assimilator moves the result to some expected location that is periodically verified by the work generator. The job configuration (input and output paths, number of mappers and number of reducers) is held in a well known configuration file (that is automatically generated).

When all intermediate results (`.torrent` files) are present and validated, the work generator triggers the shuffle. This operation is responsible for assembling and assigning sets of intermediate results to reduce tasks (since every map task typically produces data for every reduce task).

Figure 6 presents a graphical representation of a freeCycles server. Light gray boxes are the components inherited from BOINC (and therefore, remain unchanged). Dark gray boxes represent the components that were included (in the case of the BitTorrent client and BitTorrent tracker) or modified (work generator and assimilator) by our solution.

4.2 Client

Using freeCycles, all volunteer nodes run the client software which is responsible for: i) performing the computation (map or reduce task), and ii) sharing its input and output data (which might be input, intermediate, or final output data) with all other volunteers and possibly, the central server.

To be and remain compatible with current BOINC clients, our project is implemented as a regular BOINC application. Therefore, all volunteers will be able to join a MapReduce computation without upgrading their VC software. If it were not a regular application, volunteers would have to upgrade their client software in order to fully explore freeCycles' capabilities namely, use BitTorrent to share files. Previous solutions (e.g. SCOLARS) do not use this approach and modify the BOINC client. Therefore, it cannot be used without forcing users to upgrade their client software.

freeCycles's client side application is meant to be used as a framework, i.e., developers would simply call freeCycles's code to register the map and reduce functions. All other issues related to managing map and reduce task execution, downloading and uploading data is handled by our implementation.

Notwithstanding, VC application developers might analyse and adapt the application code to specific application needs (e.g. if one needs to implement a special way to read/write input/output data). Other optimizations like intermediate data partitioning or combining intermediate results might also be easily implemented.

Figure 7 presents a graphical representation of the freeCycles client implementation. The light gray box, BOINC Client Runtime, is the core component in the client software. All volunteers that are already contributing using BOINC will have this component and it is, therefore, our only requirement. Dark grey boxes are the components offered by our project:

- freeCycles Application is the central component and the entry point of our application. It coordinates the overall execution by: i) asking for input from the Data Handler, ii) preparing and issuing the MapReduce computation and iii) sending output data (via the Data Handler); Additionally, this component is responsible for interacting with BOINC Client Runtime (initialize BOINC runtime, obtain task information and acknowledging the task finish);
- Data Handler is data management API. It is the component responsible for downloading and uploading all the necessary data and is the only one that needs to interact with the communication protocol (BitTorrent). This API, however, does not depend on the communication protocol and therefore, can be used for multiple protocols;
- BitTorrent Client is the low level and protocol specific component. It was implemented using an open source

BitTorrent library (libtorrent);

- the MapReduce Tracker introduces the logic related to MapReduce applications. It uses a previously registered function to run map or reduce tasks and manages all key and value pairs needed for the computation.

To create and use one application, one would only need to provide a map and a reduce function implementation (white box from figure 7). These functions are then registered in our MapReduce Tracker module and thereafter, are called for all keys and value pairs.

The client side application code is divided into two modules: the Data Handler (handles all data downloading and uploading), and the MapReduce Tracker (controls the map or reduce task execution). By using such division of responsibilities, we provide project developers with modules that can be replaced (e.g. one could decide to use our Data Handler implementation with BitTorrent to distribute data in other types of computations, not just MapReduce).

5. EVALUATION

We now proceed with an extensive evaluation of our system. We compare freeCycles with SCOLARS (a BOINC compatible MapReduce VC system), and BOINC, one of the most successful VC platforms, and therefore, a good reference for performance and scalability comparison. We use several representative benchmark applications and different environment setups that verify the performance and scalability of freeCycles.

5.1 Evaluation Environment

To conduct our experiments, we use a set of university laboratory computers. These computers, although very similar in software and hardware, are, most of the time, running tasks from local or remote users. It is also important to note that all tasks run by remote users have a higher niceness.[6] With this setting of priorities, our computations will compete with local user tasks (as in a real volunteer computing scenario). In fact, all experiments run several times to avoid scenarios where some machines were overloaded with other tasks.

In order to be more realistic, most of our evaluation was performed with throttled upload bandwidth. This is an important restriction since Internet Service Providers tend to limit users' upload bandwidth (the ratio between download and upload bandwidth usually goes from 5 to 10 or even more).

During this experimental evaluation, all MapReduce workflows used 16 map tasks and 4 reduce tasks both with a replication factor of 3. All map and reduce tasks run on different nodes (to simulate what would probably happen in a regular VC project). Thus, we used a total of 60 physical nodes (48 map nodes and 12 reduce nodes).

5.2 Application Benchmarking

For this paper, we use a set of representative benchmark applications to compare our solution with SCOLARS and BOINC.

[6]The nice value is process attribute that increases or decreases the process priority for accessing the CPU. The higher this value is, the lower the priority is when deciding which process should run next.

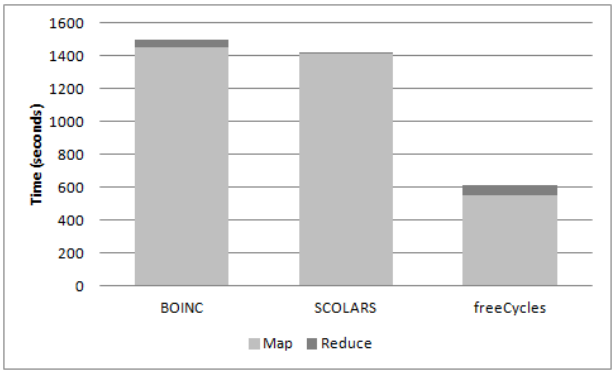

Figure 8: grep Benchmark Application

From the data handling perspective (and according to some previous work [1]), each one of the three selected benchmarks belongs to a different MapReduce application class: small intermediate output (grep), medium intermediate output (word count) and large intermediate output (terasort).

While running these benchmarks, several variables are set: i) the MapReduce application was setup to 16 map tasks and 4 reduce tasks; ii) the replication factor is 3 (then we have 48 map nodes plus 12 reduce nodes); iii) all nodes have their bandwidth throttled to 10Mbps; iv) a 512MB data file was used as input. These parameters hold for every benchmark application and for all the three evaluated systems.

5.2.1 grep

Our first benchmark application is grep. As from the original Unix system, grep is a program that searches plain-text data for lines matching regular expressions. For this evaluation, we built a simple implementation of grep that was used to match a single word. The word was selected so that it was possible to have very small intermediate data.

From figure 8, it is possible to see that freeCycles is able to run the benchmark application in less than half the time took by BOINC and SCOLARS. The application turnaround time is clearly dominated by the input distribution time. Since freeCycles uses BitTorrent, it uses available upload bandwidth from volunteers to help the server distributing the input.

A slight overhead can be noticed in the reduce phase for our solution. This comes from the fact that the intermediate output is so small that the time to distribute all the intermediate and final output was dominated by the BitTorrent protocol overhead (contact the central tracker, contact several nodes, wait in queues, etc).

5.2.2 Word Count

Our second benchmark application is the famous word count. This program simply counts the number of occurrences for each work in a given input text. In order to maintain a reasonable size of intermediate data, we do combine (operation that merges intermediate data still on the mappers) output data from mappers.

Figure 9 shows the results of running word count. As intermediate data is larger than in the previous application

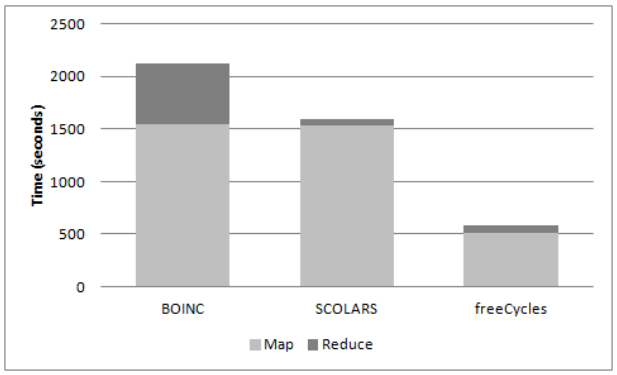

Figure 9: Word Count Benchmark Application

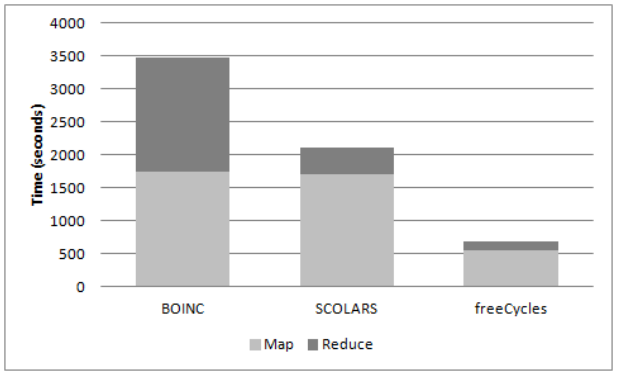

Figure 10: Terasort Benchmark Application

(186MB of generated intermediate data versus 2MB, in grep), BOINC has a worse performance compared to SCOLARS and freeCycles. SCOLARS is able to be much more efficient than BOINC in the reduce phase (since it allows intermediate results to travel from mappers to reducers, avoiding the central server). Nevertheless, freeCycles continues to be the best system mainly by having a very small input data distribution time.

5.2.3 Terasort

The last benchmark application is Terasort. Terasort is yet another famous benchmark application for MapReduce platforms. At a very high level, it is a distributed sorting algorithm that: i) divides numbers in smaller chunks (with certain ranges), and ii) sorts all chunks. We produced a simple implementation of this algorithm to be able to compare the performance of freeCycles with other systems.

Despite being a very popular benchmark, this application is also important because it generates large volumes of intermediate and output data.

Looking at figure 10, it is possible to see a big reduce phase in BOINC. This has to do with the large intermediate data generated by terasort. As SCOLARS implements inter-client transfers, it cuts much of the time needed to perform the intermediate data distribution (which is the dominating factor). As intermediate data is larger than in the previous application, our system suffered a slight increase in the reduce phase.

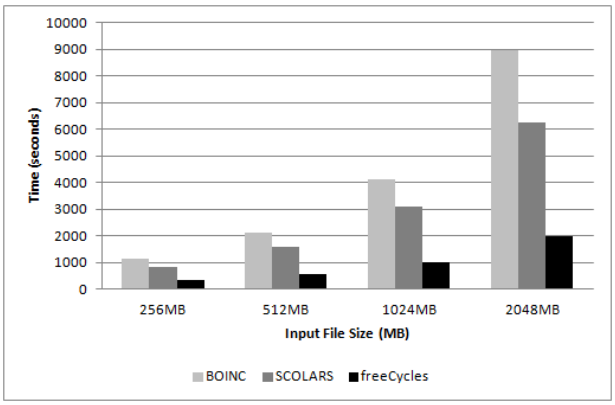

Figure 11: Performance varying the Input File Size

Despite the size of intermediate data (512MB), freeCycles is able to distribute intermediate data very efficiently. It is almost 5 times faster than BOINC and 3 times faster than SCOLARS.

5.3 Varying Input File Size

For the next experiment, we change the input file size. In the previous scenarios 512MB files were used. For this experiment, we start with a smaller input file (256MB) and increase the file size until 2048MB.

The reason for this experiment is to show how VC systems behave when large input files are used. Typical VC projects use small input files that require lots of computation. However, as already said, MapReduce applications need large input files that, most of the time, will be consumed quickly.

The considered input file is the one that will be divided by all mappers. In our implementation, we equally divide the file using the number of lines as parameter. Therefore, all mappers will have a very similar input file size to process (and consequently, will have similar execution times).

For this experiment we use the word count application (as it has medium intermediate file sizes). We keep all nodes with their upload bandwidth limited to 10Mbps and we maintain the MapReduce configuration of 16 mappers and 4 reduces (all replicated 3 times).

Figure 11 shows the results for this experiment. BOINC is the system with worse execution times. SCOLARS is able to lower the time by using inter-client transfers, which effectively reduces the reduce operation time. Our system, freeCycles, has the best execution times, beating the other two systems with great advantage (4,5 times faster than BOINC and 3 times faster than SCOLARS).

It is also interesting to note that, as the input file size increases, all three solutions have different execution time increases. This has to do with the fact that, as the file size goes up, the total number of bytes uploaded goes up as well. However, as freeCycles uses volunteers to distribute data, each volunteer will have a little more data to share. On the other hand, BOINC data server will have to upload all data resulting in an almost doubled execution time when the input file size is doubled. SCOLARS has an intermediate behaviour

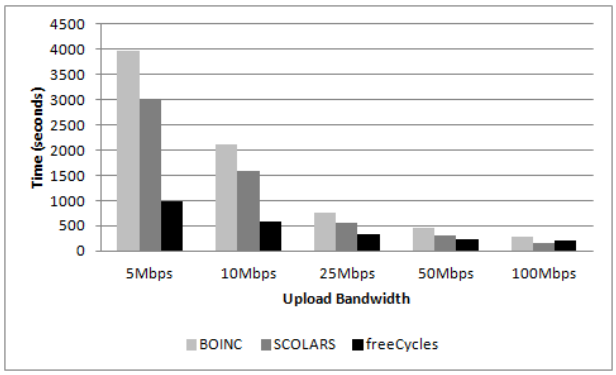

Figure 12: Performance varying the Upload Bandwidth

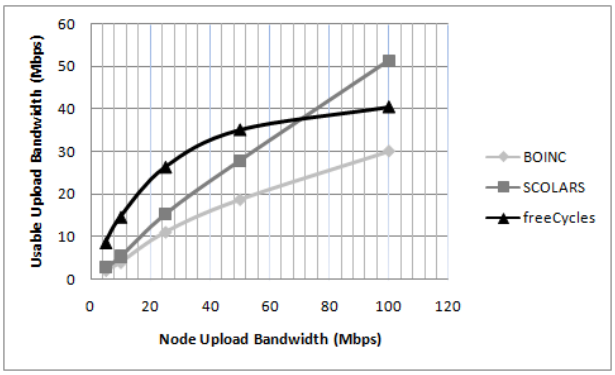

Figure 13: Aggregate Upload Bandwidth

since it spares some upload bandwidth from volunteers to spread intermediate data.

5.4 Varying Upload Bandwidth

Another important restriction on real-life volunteer computing projects is the speed at which a data server can export computation. In other words, upload bandwidth is a very limited resource that determines the number of volunteer tasks that can be delivered at any moment.

In order to evaluate the behaviour of different systems with different upload bandwidth, we present this evaluation scenario where we repeated the execution of the word count application with a 512MB input file. We keep the 16 map and 4 reduce configuration for all runs.

We use a wide range of upload bandwidth. We start with 5Mbps and 10Mbps, reasonable values for a home Internet connections. Then we move to higher bandwidths that come closer to what is expected for a grid or cluster.

With figure 12, we can see that as the upload bandwidth goes up, the difference between different solutions decreases. The reason for this is that the input file size was kept constant (at 512MB). Hence, freeCycles, for example, always have a small overhead that comes from the fact of using BitTorrent (delay to contact the central tracker, contact several volunteers, delay in priority queues, etc). This overhead is particularly noticeable in the last scenario (with 100Mbps) where SCOLARS performs better (since it has far less overhead).

Yet, when we move to more realistic scenarios, with upload bandwidths such as 5Mbps or even 10Mbps, freeCycles is able to perform much better than the other solutions. This is possible since freeCycles uses all the available upload bandwidth at the volunteer nodes to distribute all the data (input, intermediate and final output). On the other end of the spectrum there is BOINC, that uses a central data server to distribute all data (and thus, becomes as critical bottleneck).

With the same data from this experiment, we divided the total number of uploaded mega bytes (which is the same for the three platforms) by the time it took to distribute that amount of data. The result is an approximate average of the total upload bandwidth effectively used. We performed this division for all the bandwidths in figure 12 (5, 10, 25, 50, and 100 Mbps) and the results are shown in figure 13.

Figure 13 shows that, for some server upload bandwidths (the actual link bandwidth), freeCycles can, harvesting volunteer's upload bandwidth, obtain up to three times the total bandwidth that BOINC is able to use. It is also possible to see that, when the server upload bandwidth goes up, our solution is not able to increase the usable upload bandwidth. This has to do with the fact that we do not increase the input file size (which is always 512MB) as we increased the server upload bandwidth. Therefore, with higher bandwidths, the overhead of using BitTorrent becomes noticeable. Hence, it is possible to conclude that BitTorrent should be used when the ratio of the number of bytes to upload by the server upload bandwidth is high.

5.5 Iterative MapReduce Applications

For this next experiment, our goal is to measure the performance of the three solutions (BOINC, SCOLARS, and freeCycles) when a MapReduce application needs to run for several cycles (as the original Page Ranking algorithm).

For that purpose, we implemented a simple version of the Page Ranking algorithm which is composed by two steps: i) each page gives a share of its rank to every outgoing page link (map phase); ii) every page sums all the received shares (reduce phase). These two steps can be run iteratively until some criteria is verified (for example, if the value has converged).

We maintained the setup used in previous experiments: upload bandwidth throttled to 10Mbps, input file size of 512MB, 16 map tasks, 4 reduce tasks and 3 replicas for each task. To limit the size of the experiment (and since no extra information would be added) we use only two iterations.

Results are shown in figure 14. BOINC has the highest execution times since all the data has to go back to the server after each map and reduce task. This creates a high burden on the data server which contributes to a higher overall time. It is also interesting to note that, in this experiment, intermediate data is almost 50% bigger than input and output data. This is why the reduce phase takes longer than the input phase.

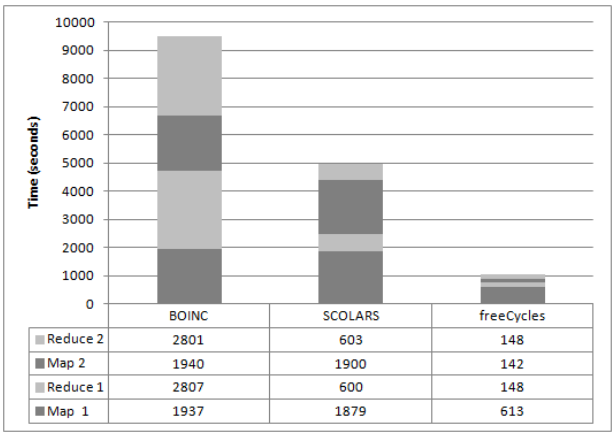

Figure 14: Two Page Ranking Cycles

Benchmark	Hadoop Cluster	Volunteer Pool
grep	102 sec	610 sec
Word Count	106 sec	578 sec
Terasort	92 sec	686 sec

Table 1: Benchmark Execution Times

SCOLARS performs much better than BOINC since intermediate data flows from mappers to reducers. However, between cycles, all data must go to the central data server and therefore, N iterations will result in N times the time is takes to run one iteration (as in BOINC).

freeCycles presents the most interesting results. The first map phase (from the first iteration) still takes a significant time to finish. Subsequent phases (first reduce, second map and second reduce) execute much faster. However, the big difference between our solution and previous ones is that output data need not to go to the central data server. Thus, input data distribution for the next iterations is much faster since multiple reducers, can feed data to the next map tasks avoiding a big bottleneck on the data server.

5.6 Comparison with Hadoop Cluster

Another interesting experiment to perform is to compare the execution times for the three benchmark applications on: i) volunteer pool (freeCycles) and Hadoop cluster (Hadoop is a MapReduce implementation by Apache).

To that end, we use a cluster of 60 machines (the same number of machines as in our volunteer pool) in which 10 machines also play as datanodes (datanodes are nodes that are hosting the distributed file system, HDFS). All nodes have 4 cores, a total 8GB of RAM and are interconnected with Gigabit Ethernet.

For this experiment, we run the three application benchmarks: grep, word count and terasort. We did not build these applications for Hadoop, but instead, we use the implementation provided with the platform. Regarding the freeCycles platform (Volunteer Pool), we use the same implementation as in previous experiments. All applications run with 16 mappers and 4 reduces (replicated 3 times).

Results are shown in table 1. By these results, it is possible to conclude that a MapReduce application deployed on a cluster runs approximately 6 times faster than the same application deployed on a volunteer pool. This is the performance price that it takes to port an application to volunteer computing.

Session Time	1 vol/min	10 vol/min	100 vol/min
1 hour	INF	2710 sec	2282 sec
2 hour	3922 sec	2527 sec	2114 sec
3 hour	3880 sec	2398 sec	2104 sec
4 hour	3754 sec	2354 sec	2104 sec
5 hour	3754 sec	2354 sec	2104 sec

Table 2: Real World Simulations

Nevertheless, it is important to note two factors that motivate this big performance discrepancy (between the used cluster and the volunteer pool): i) we limited the bandwidth to 10Mbps on the volunteer pool while the cluster nodes where connected via 1000Mbps; ii) input data was previously distributed and replicated among 10 datanodes (in the Hadoop cluster) while our freeCycles deployment only had one data server.

5.7 Real World Environment

In this last experiment we test our solution on a simulated realistic environment, i.e., with nodes coming and going in the middle of a MapReduce job. To that end, we used different churn rates by manipulating the average node session time and the new volunteer rate.

Results from Table 2 shows that the time needed to complete a job is reduced by: i) increasing the volunteer session time and ii) increasing the new volunteer rate. The interesting result is found when the session time is 1 hour and there is 1 new volunteer per minute. Our results show that there are not enough volunteers and volunteers fail to ofter to complete the job. For those reasons, most of our runs never ended.

From this experiment, we conclude that, as intermediate data is kept on volatile nodes, one should make sure that the job completion time is, at most, equal to the average session time. Therefore, when running MapReduce on top of volunteer pools, one should use many small jobs (possibly with multiple cycles) instead of few large jobs.

6. CONCLUSIONS

freeCycles is a new, MapReduce-enabled, VC platform. It presents a new data distribution technique for input, intermediate and final output using the BitTorrent protocol. Using BitTorrent, freeCycles is able to use volunteer's upload bandwidth to help distributing data. Moreover, it proposes an adaptive map task replication and is able to efficiently run MapReduce applications with multiple cycles.

From the experiments, we conclude that freeCycles is to able perform much better (performance and scalability wise) than current VC platforms. Hence, with our work, it is possible to improve MapReduce applications' execution time on large volunteer pools such as the Internet.

7. REFERENCES

[1] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on

heterogeneous clusters. *SIGARCH Comput. Archit. News*, 40(1):61–74, Mar. 2012.

[2] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Superweb: towards a global web-based parallel computing infrastructure. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 100–106, 1997.

[3] D. Anderson. Boinc: a system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10, 2004.

[4] D. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 73–80, 2006.

[5] A. Baratloo, M. Karaul, Z. Kedem, and P. Wijckoff. Charlotte: Metacomputing on the web. *Future Generation Computer Systems*, 15(5âĂŞ6):559 – 570, 1999.

[6] A. Chakravarti, G. Baumgartner, and M. Lauria. The organic grid: self-organizing computation on a peer-to-peer network. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 35(3):373–384, 2005.

[7] L. Cherkasova and J. Lee. Fastreplica: Efficient large file distribution within content delivery networks. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[8] F. Costa, L. Veiga, and P. Ferreira. Internet-scale support for map-reduce processing. *Journal of Internet Services and Applications*, 4(1):1–17, 2013.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[10] D. T. Fabrizio Marozzo and P. Trunfio. Adapting mapreduce for dynamic environments using a peer-to-peer model, 2008.

[11] G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: a generic global computing system. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 582–587, 2001.

[12] G. Fedak, H. He, and F. Cappello. Bitdew: A data management and distribution service with multi-protocol file transfer and metadata abstraction. *Journal of Network and Computer Applications*, 32(5):961 – 975, 2009. Next Generation Content Networks.

[13] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 181–192. ACM, 2010.

[14] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 95–106, New York, NY, USA, 2010. ACM.

[15] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao. Cluster computing on the fly: P2p scheduling of idle cycles in the internet. In *Peer-to-Peer Systems III*, pages 227–236. Springer, 2005.

[16] N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over the internet-the popcorn project. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 592–601, 1998.

[17] H. Pedroso, L. M. Silva, and J. G. Silva. Web-based metacomputing with jet. *Concurrency: Practice and Experience*, 9(11):1169–1173, 1997.

[18] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV*, pages 205–216. Springer, 2005.

[19] L. F. Sarmenta and S. Hirano. Bayanihan: building and studying web-based volunteer computing systems using java. *Future Generation Computer Systems*, 15(5âĂŞ6):675 – 686, 1999.

[20] P. C. B. C. M. F. I. M. O. N. K. E. Schauser and D. Wu. Javelin: Internet-based parallel computing using java, 1996.

[21] K. Shudo, Y. Tanaka, and S. Sekiguchi. P3: P2p-based middleware enabling transfer and aggregation of computational resources. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 1, pages 259–266 Vol. 1, 2005.

[22] M. Silberstein, A. Sharov, D. Geiger, and A. Schuster. Gridbot: execution of bags of tasks in multiple grids. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 11:1–11:12, New York, NY, USA, 2009. ACM.

[23] B. Tang, M. Moca, S. Chevalier, H. He, and G. Fedak. Towards mapreduce for desktop grid computing. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2010 International Conference on*, pages 193–200, 2010.

[24] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[25] T. White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[26] L. Zhong, D. Wen, Z. W. Ming, and Z. Peng. Paradropper: a general-purpose global computing environment built on peer-to-peer overlay network. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 954–957, 2003.