J Internet Serv Appl (2012) 3:329–346
DOI 10.1007/s13174-012-0072-0

SI: DATA INTENSIVE COMPUTING

Large-scale volunteer computing over the Internet

Fernando Costa · João Nuno Silva · Luís Veiga · Paulo Ferreira

Received: 10 July 2012 / Accepted: 4 October 2012 / Published online: 25 October 2012

Abstract Cycle sharing over the Internet has increased in popularity during the last decade, with increasingly powerful machines being made available to existing projects. In this paper, we present GiGi-MR, a framework that allows non-expert users to run CPU-intensive jobs on top of volunteer resources over the Internet. GiGi-MR has several distinctive features: it allows non-expert users to easily partition their jobs in several parallel tasks; such Bag-of-Tasks (BoT) are executed in parallel as a set of MapReduce applications; the volunteer resources that provide the best match for the tasks being executed are chosen (using attenuated bloom filters); it provides a portable checkpointing fault-tolerance mechanism based on virtualization; it does not rely exclusively on a central server (or servers) at all times (thus minimizing the bottleneck effect); it deals with malicious participants (possibly byzantine) using an efficient partial replication mechanism to validate the results obtained; and it is compatible with BOINC (one of the most popular open-source software platforms for computing using volunteered resources). We describe GiGi-MR's architecture and evaluate its performance by executing several MapReduce applications on a wide area testbed. Furthermore, we use micro-benchmarks to assess each one of GiGi-MR's components independently. The system's overhead is minimal. When compared to an unmodified volunteer computing system, GiGi-MR obtains a performance increase of over 60 % in application turnaround time, while reducing the bandwidth used by an order of magnitude.

Keywords Volunteer computing · Distributed systems · MapReduce · Adaptive middleware

This work was partially supported by national funds through FCT-Fundação para a Ciência e Tecnologia, under projects PTDC/EIA-EIA/102250/2008, PTDC/EIA-EIA/108963/2008, PTDC/EIA-EIA/113993/2009 and PEst-OE/EEI/LA0021/2011.

F. Costa · J. N. Silva · L. Veiga · P. Ferreira (✉)
Distributed Systems Group, INESC-ID , Technical University of Lisbon, R. Alves Redol, 9, 1000-029 Lisboa, Portugal
e-mail: paulo.ferreira@inesc-id.pt

1 Introduction

The use of volunteer PCs across the Internet to execute distributed applications has been increasing in popularity since its inception in the early 1990s, with the creation of projects such as Distributed.net,[1] Seti@home [3] or Folding@home [19]. These Volunteer Computing (VC) systems harness computing resources from machines running commodity hardware and software, and perform highly parallel computations, also called Bag-of-Tasks (BoT), that do not require any interaction between network participants.

Existing VC systems support over 60 scientific projects,[2] and have over a million participants, rivaling supercomputers in computing power. The most popular middleware, BOINC [2], is currently being used by over 40 projects, from scientific fields ranging from climate prediction to protein folding.

Projects must have a large visibility to attract enough cycle donors and be composed of hundreds of individual tasks or workunits. Furthermore, project creators must have a large knowledge on C++ or Fortran programming. To achieve fault tolerance during task execution, developers must modify their application code and insert explicit checkpoints. Users not satisfying these requirements cannot take advantage of available remote cycles. Even if the user has enough

[1] Distributed.net website. http://www.distributed.net.

[2] List of active VC projects. http://www.distributedcomputing.info/projects.

programming knowledge to create a project, if the project is short lengthened or not capable of attracting enough donors, the gains will be low. This kind of operation greatly limits the scope of users capable of creating projects to be remotely executed.

1.1 Goal

Our goal is to create a framework (called GiGi-MR) that allows non-expert users to create jobs and submit the corresponding Bag-of-Tasks to a VC system, supporting the MapReduce paradigm and making an efficient usage of the resources available, while being fault-tolerant and resilient to byzantine clients and compatible with BOINC.

There are several challenges and requirements to consider, in order to achieve our goal. First and foremost, GiGi-MR must be able to take advantage of the huge amount of VC resources that we previously mentioned. We must consider both the hardware capabilities of individual machines and the network bandwidth that is at our disposal, at the last mile of the Internet. The platform needs to be portable, in order to handle the heterogeneity of machines, and adaptable to environmental changes (i.e., resource availability). To that end, it must able to organize clients into a virtual network, and have them exchange information that is then used by the server.

Our system must also be compatible with existing VC solutions (e.g., BOINC [2]). Developing a whole new platform from scratch would be of no practical use. Therefore, we must take into account existing systems and use their existing infrastructure to come up with a final prototype that can actually be used, in a real-world scenario. In fact, our solution would undoubtedly bring significant disadvantages if it required that only our system's clients were attached to a project.[3] To avoid this situation, we must guarantee compatibility with existing projects. Any client must be able to run any project application. On the other hand, our solution must support existing applications, and successfully schedule tasks on existing clients.

To include non-expert users as job creators, two key requirements are to be met: (1) the users should be allowed to use the applications or programming languages they are literate on, and (2) there should be enough cycle donors to speed even small jobs. The system must also be able to take sequential applications representative of BoT problems (with iterations that process different data sets) as input, and modify them into parallel tasks without user intervention. Some applications, due to being more complex and not easily transformed into a set of map and reduce tasks, do require some manual intervention. This is provided by means of a simple interface that non-experts users can use (e.g., to define which executable should run with which set of data).

The execution of our system on unreliable, non-dedicated resources requires fault tolerance mechanisms. This means it must account for unreachable clients, which have disconnected from the server, or are simply offline. Our solution must be able to withstand transient server failures. This is particularly important in our case because we will be dealing with long running applications, with a potentially high level of server interactions. We need to prevent the execution on the clients to come to a halt, as they wait for the server to come back up. Finally, we must also consider byzantine behaviour. Clients may maliciously return incorrect results, or inadvertently produce an incorrect output by encountering errors during the computation or data transfers. Therefore, we must provide result validation that accounts for this environment and provides reliability.

1.2 Shortcomings of current solutions

Existing solutions do not fulfill our goal while ensuring the requirements mentioned above. We highlight some of those shortcomings in this section (more details in Sect. 4).

Although creation, distribution and execution of tasks over the Internet are handled by existing middleware, there is still a steep entry barrier for anyone trying to start a VC project. This makes cycle sharing over the Internet a one-way deal. Computer owners only have one role in the process: to donate their computers' idle time.

The development of Bag-of-Tasks applications for execution on multiprocessors or clusters requires the use of APIs not designed for this kind of problem. For instance, MPI [29] allows the parallel execution of tasks, but was developed for much more complex parallel applications, with high data communication between tasks. The use of such APIs requires the programmers to learn them, and add complexity to the final parallel solution. Existing VC systems typically do not provide any tool to convert simpler, sequential applications to parallel BoT.

A considerable limitation of existing VC systems is their focus on BoT applications, with little communication and without dependencies between the tasks. As parallel and distributed computing becomes the answer for increased scalability for varied computational problems, several paradigms and solutions have been created during the last decade. In particular, MapReduce [11] has taken its place as one of the most widely used paradigms in cloud computing environments, such as Amazon's EC2.[4] Its wide use, simplicity, and scalability make it a prime candidate for execution on

[3] A VC Project runs on top of existing middleware (e.g., BOINC) by developing an application and defining all parameters concerning its execution.Project developers only have to make sure their tasks are properly configured and provide a publicly accessible machine to act as the VC server.

[4] Amazon EC2. http://aws.amazon.com/ec2.

VC systems. None of the current VC platforms support MapReduce, a programming model that adapts well to a data-intensive class of applications. Supporting MapReduce requires fundamental changes on existing algorithms, and the introduction of on-the-fly task creation. This is currently not available on any present system.

To deal with BoT applications, scheduling and resource discovery algorithms are designed with the least complexity possible. Despite reducing the probability of introducing errors in computation or validation, this approach underestimates the benefits of taking advantage of user resources. Current systems are limited to specifying the minimum hardware requirements for each computation, and typically do not consider adaptive algorithms to deal with ever-changing machine availability and resources.

A server in existing VC systems is only capable of using host information periodically reported by each client when requesting work. After assigning a work unit, the scheduler can make an educated guess on when the client will finish execution and request further work, based on past behaviour and task deadlines. However, there is no further update of this schedule until there is another request. This greatly reduces the system's capacity to predict future work requests and schedule tasks accordingly.

Most VC systems have a centralized architecture, with all communication going through a single server (or cluster). There are few exceptions and they were created with a smaller scope or environment in mind [8]. In BOINC [2], XtremWeb [5] and Folding@home[19], the server or coordinator must fulfill the role of job scheduler, by handling all the task distribution aspects and result validation. This approach inevitably creates a bottleneck, as projects expand and storage and network requirements become more demanding. Existing projects such as Climateprediction.net and MilkyWay@home have encountered scalability problems when dealing with large files or having the same data shared by many clients [9]. Although some potential solutions have been proposed [10, 13], they have not been deployed in the most widely used systems.

Fault tolerance is mostly confined to the client-side in current VC systems. Although some projects do have a set of mirrors that act as data repositories, all client requests and task scheduling goes through the central server. Therefore, any server fault that prevents it from communicating with clients has a very high probability of disrupting clients and stopping further task execution.

Finally, there is a considerable limitation with respect to result validation mechanisms. Most existing systems are content with providing integral replication of data, without considering communication overhead or potentially more attractive alternatives. There is also little or no use of redundant task execution (we call this sampling technique - more details in Sect. 2.3) which can constitute definite proof in cases of malicious behaviour (user returning an incorrect result).

In summary, existing VC solutions allow the execution of BoT in a master/worker model, with simple replication and fault tolerance mechanisms. They guarantee valid results but do not take advantage of the ample client resources, and create a high entry barrier for anyone wishing to take advantage of their platform.

1.3 Our solution: GiGi-MR

In this paper we present GiGi-MR, a framework that allows ordinary users to execute MapReduce tasks over the large scale Internet, on top of volunteer resources. MapReduce is a fitting choice for running data-intensive applications on top of volunteer resources, since it is a popular paradigm, representative of different tasks.

MapReduce leverages the concept of Map and Reduce commonly used in functional languages: a map task runs through each element of a list and produces a new list; reduce applies a new function to a list, reducing it to a single final value or output. In MapReduce, the user specifies a map function that processes tuples of key/values given as input, and generates a new intermediate list of key/value pairs. This map output is then used as input by a reduce function, also predefined by the user, that merges all intermediate values that belong to the same key. Therefore, all reduce inputs are outputs from the previous map task. Throughout the rest of the paper, we will refer to them as map outputs.

Our system is compatible with existing solutions (in particular BOINC), and provides users with the ability to submit jobs through a web interface. GiGi-MR supports client to client transfers, thus minimizing the volume of data sent through the server. This also allows GiGi-MR to tolerate transient server failures, as the clients depend merely on other peers for data. It is also capable of tolerating VC clients' failure using replication (i.e., running the same task on several VC machines). By increasing the replication factor, the probability of a failure of all clients running a certain task is lowered.

Byzantine client behaviour is controlled through the use of task validation in the server. Different data partitioning flavours among the tasks are supported, and the use of sampling on the server further increases security. By replicating each task, it is possible to compare the outcome and accept only the results in which a quorum has been reached.

Our framework follows a layered approach, ranging from top-level user interaction tools to lower-level modifications that arrange clients into a connected topology. We decentralize some of the mechanisms of existing systems that place an excessive burden on the central server, by taking advantage of user resources. Additionally, we introduce new algorithms for scheduling and validation that increase our system's adaptability and usefulness.

Task scheduling is improved through the use of information provided by running clients, which are organized in an overlay network [27]. Several criteria, such as bandwidth or resource availability, are subject to analysis for the choice of neighbours. Bloom filters [4] are used to identify different types of resources, from applications to libraries or services. The system's resource discovery mechanism is coupled with a resource evaluation algorithm that uses fuzzy logic and combined utility functions to prioritize hosts [28].

This paper is organized as follows: GiGi-MR is presented in more detail in Sect. 2; Sect. 3 describes some implementation details, and presents micro-benchmarks and experimental results, conducted with several MapReduce applications, on a large scale testbed [7]; related work is discussed in Sect. 4; and Sect. 5 concludes.

2 GiGi-MR architecture

GiGi-MR's high-level architecture is presented in Fig. 1. A server is responsible for scheduling and validating tasks, while taking advantage of information provided by host clients. Clients are organized into a network overlay, which allows them to exchange information independently from the server.

GiGi-MR is compatible with BOINC (Berkeley Open Infrastructure for Network Computing), the most successful and popular volunteer computing middleware to date. Consequently, our client can participate in GiGi-MR as well as in BOINC projects, and borrows many primitives and algorithms available to BOINC clients.

The GiGi-MR client software in shown in Fig. 1. The top layer, *User Interface*, is responsible for user interface on the client. Users can use it to transform sequential applications into parallel tasks, thus making them runnable on GiGi-MR. In addition, this layer also lets ordinary users submit their jobs from their machine, by registering the application's executable file. On the server, the *Web Interface* provides a web page for users to submit jobs, and define their parameters and input files (which are then uploaded to the *Data Server*). The *RPC Interface* is responsible for interacting with the client when registering new applications.

The *MapReduce VC* layer enables the execution of MapReduce tasks on the system. The server stores information on each job's parameters (e.g., number of map and reduce tasks) in a configuration file, which is accessed when creating tasks. Map tasks are distributed to clients, and once all mappers have returned their result, the reduce tasks are created and scheduled for execution on reducers. As previously mentioned, the transfer of map outputs to reducers is done through inter-client transfers, without server interference.

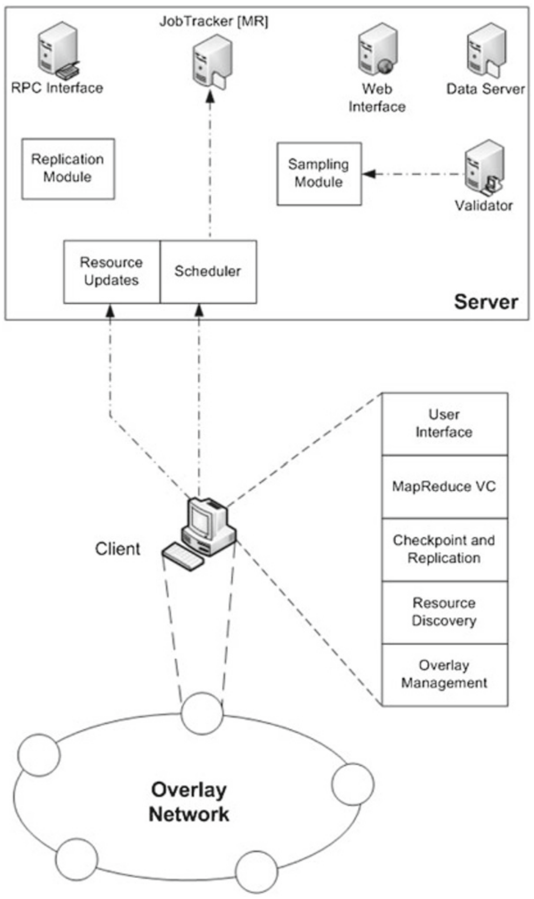


Fig. 1 GiGi-MR model

The following layer, *Checkpoint and Replication*, provides a checkpointing mechanism, through the use of virtual machines (VMs), and provides several options for partitioning and replicating input data. Using VMs removes the need for changes to the application source code to achieve task fault tolerance.

Resource Discovery is used for enhancing the server's scheduling performance. Clients exchange messages within their overlay network, concerning their current availability and volunteered resources. This information is then sent to the *Resource Updates* module whenever there is an interaction with the server (e.g., work request).

The bottom layer, *Overlay Management*, is responsible for routing and addressing in the overlay network. When changes in volunteered resources occur, they are announced to the nodes of the local node neighbour set throughout update messages. The neighbour set is established and managed at this level. Moreover, this layer maintains all the information about the availability of resources that each node of its neighbour set has. This layer separates the system from the overlay network used, thus providing the freedom of choosing the most appropriate solution (e.g., CAN [25], Chord [30], Pastry [27], etc).

Each layer is described in more depth in the following subsections.

2.1 User Interface

The top layer provides two features: (1) transformation of sequential applications into parallel tasks, and (2) their submission to GiGi-MR by ordinary users.

To perform a transformation, the user must define which methods and classes should be parallelized. This information is saved in a configuration file, which is read by the GiGi-MR client. Afterwards, it loads the application, and transforms it in run-time so that the specified methods are executed concurrently. The transformation itself is performed without user intervention. The resulting tasks are submitted to the system and executed remotely. This layer is responsible for spawning the necessary threads, and synchronizing the invocation of the methods.

The proposed solution is implemented in Python and uses metaclasses, allowing the modification of the code to be done in run-time, without any need to transform and recompile the source code. The developed metaclass intercepts all class creations and modifies the implementation of those that are to be parallel, without any user intervention: the user must only state what classes have methods that can be executed concurrently with the rest of the code.

The distribution of work among several computers or processors by existing systems can be done using libraries such as MapReduce, but requires the programmer to know their API. Our system removes this burden from the application developer through run-time code adaptation, and allows the submission of sequential applications. It is worth noting that users may skip the transformation step, as our system supports the deployment of parallel tasks and MapReduce applications.

To submit tasks, and make them available for execution, a developer would typically have to run scripts and console commands from the server. However, an ordinary user can take advantage of the *User Interface* layer, which provides a client GUI and a web interface on the server to facilitate the submission process.

GiGi-MR supports efficient execution of user submitted jobs, while allowing any user to have two complementary roles: owner of the jobs that are executed on remote computers and owner of the computers where jobs will be executed. To accomplish this, we modified both the client and server software, and developed a custom application. The data processing code used by these jobs comprises commodity applications that are installed in the remote computers, only after their owners allow their use.

The job submission process is shown in Fig. 2. To submit and create new jobs, users must: (1) select the commodity application that should be used to process the data and register it through the *Application Registrar* GUI; (2) provide the input files (data or code) to the *Data Server*, and (3) use the server's *Web Interface* to define the number of tasks to create, the name of the output files and the arguments that should be used to invoke the commodity application. For a MapReduce job, the user must provide both the map and reduce application to be used.

The web interface is shown in Fig. 3. In this page, the user uploads the input files and selects the application that should be used to process them. In the example, the user wants to process a file (anim.pov) with the POVray ray tracer and generate a movie with 200 frames. In order to submit a MapReduce job, the user must provide additional information such as the number of map and reduce tasks.

After creating and storing the information for each job, the server waits for client work requests to distribute tasks. Once it receives a work request from a client with the required commodity application, it replies with task information (input files and arguments). Once all required files have been down-

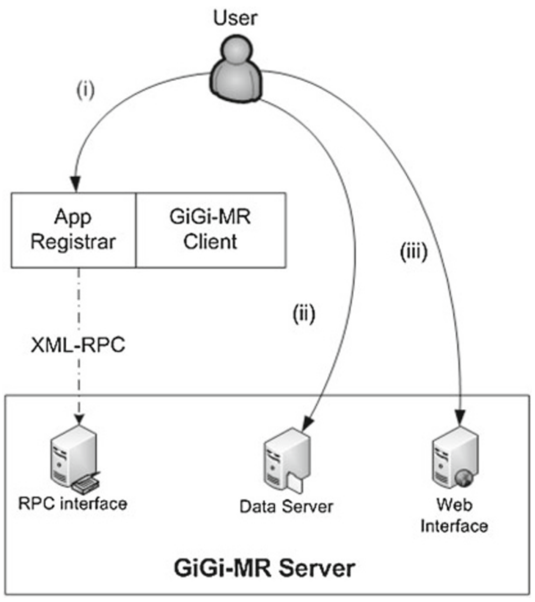
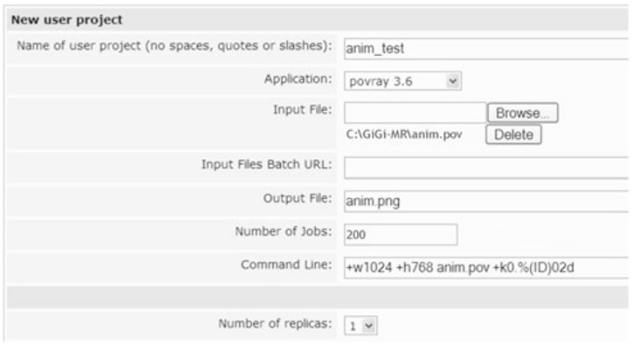


Fig. 2 User job submission to GiGi-MR



New user project

Name of user project (no spaces, quotes or slashes):	anim_test
Application:	povray 3.6
Input File:	Browse... C:\GiGi-MR\anim.pov Delete
Input Files Batch URL:	
Output File:	anim.png
Number of Jobs:	200
Command Line:	+w1024 +h768 anim.pov +k0.%(ID)02d
Number of replicas:	1

Fig. 3 User job submission interface

loaded from the server, the client invokes the correct commodity application to process the input files. After each job completion, the client submits the output to the server, as a normal application.

In general, the applications that our system handles best are those which can be easily decomposed in a set of map and reduce tasks; thus, as an example, Monte-Carlo based applications are good candidates.

2.2 MapReduce VC

This layer is responsible for handling all aspects of execution and management of MapReduce jobs on the system. As previously mentioned, a user must define the parameters of the MapReduce job through the *User Interface* layer. This information is stored in the GiGi-MR server. Once all the MapReduce job characteristics have been defined, the server creates the map tasks, and stores this information in the GiGi-MR server's database—the GiGi-MR database is responsible for holding all persistent information on tasks, clients, and applications being executed.

The overall GiGi-MR execution model for a MapReduce job is presented in Fig. 4. We consider two types of clients in GiGi-MR: mappers, which are responsible for bag-of-tasks in the map stage; and reducers, which perform the aggregation of all map output in the reduce step. A group of mappers first requests work from the server (1). The server follows a scheduling procedure which takes into account host resources and availability (see Sect. 2.4 for further details) when selecting which available task is assigned. Whenever it receives a work request, it matches each task's predefined hardware or software requirements to the client's machine characteristics. If the client is the most suitable for the task, the server assigns it the task and saves this information in its database. After selecting an appropriate map task for the requesting mapper, the server sends back information on the task that the mapper must execute. This information includes the location of input and executable files, the deadline for task completion and the previously mentioned task requirements. The machines holding input and executable files are called data servers. Although some VC projects do use a set of mirrors to act as data servers, most store the data in the central server, as represented in Fig. 4.

The mapper must then download the required data from the data server (2) before starting the computation (3). After the task execution is completed, the mapper creates an MD5 hash for each of the map output files. Therefore, at the end of the computation, each mapper is left with both the map output files and the same number of corresponding hashes. These hash sums are sent back to the server in place of the output files (4) (so it is compatible with current VC solutions, e.g., BOINC). It's worthy to note that this greatly reduces the upload volume from mappers to the server.

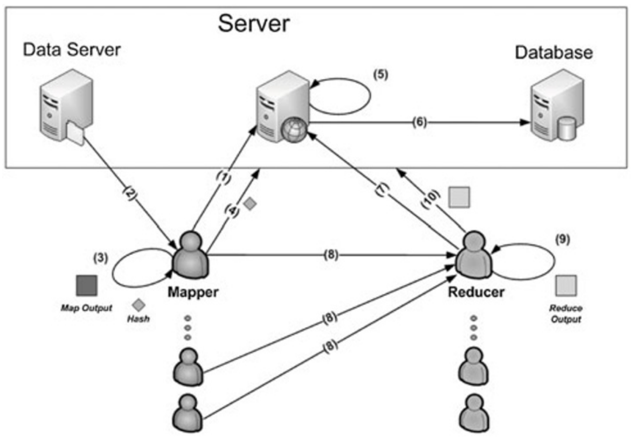


Fig. 4 GiGi-MR MapReduce job execution

The hashes are compared at the server to validate each corresponding task (5). If the result is valid, the mapper's address is stored in GiGi-MR's database (6). Each time a map result is validated, the GiGi-MR server checks if all map tasks have been executed and validated. Once this condition is met, the server creates the predefined number of reduce tasks. Upon receiving a work request from a reducer (7), the server follows task scheduling procedure mentioned earlier in this section and looks through the database to find a task that can be assigned. Once it has ascertained that the reducer meets all the hardware and availability requirements, the server replies with a reduce task that fits the request.

MapReduce jobs require communication between map and reduce stages since map outputs are used as input for reduce tasks. In the reduce step, each task performs join operations on the map outputs. Therefore, each reduce task must obtain all the map outputs that correspond to the key range it is responsible for. To achieve good performance in MapReduce jobs, we leverage clients' resources by moving as much of the communication as possible to the client-side. This helps reduce the load on the central server, and creates a more suitable decentralized model for data-intensive scenarios, typical of MapReduce.

Note that, as previously stated, in current VC systems all data would have to be uploaded and downloaded from the server. However, the GiGi-MR server stores the address of all mappers that returned valid map results. This information is included in the work request reply, and allows reducers to download the map output directly from the mappers, without having to go through the server (8). Once the input files have been downloaded, the reduce task is executed (9) and the final result is returned to the server (10) for validation.

2.3 Checkpoint and replication

This layer is responsible for: (1) checkpointing tasks to account for task failure and allow restarts in remote nodes, without any source code modification, through the use of VMs; (2) providing different options for partitioning input data, chosen by the user; (3) local sampling at the server (more details afterwards in this section), for validation purposes. In Fig. 1, the Replication and Sampling modules represent this layer in the server.

An application can be checkpointed if we run it on top of a virtual machine (VM) with checkpoint/restart capabilities (e.g., qemu[5]), as the application's state is saved within the virtual machine's state. This also provides some extra security to the clients, since they will be executing untrusted code with a high level of confinement.

Furthermore, using virtual machines allows us to reduce the impact of byzantine behaviour, caused by the different software and hardware configurations found at each machine. By running tasks on top of VMs, the same software drivers and programs are used during execution. This guarantees that each task produces the same result regardless of the underlying system. VMs also help developers by removing the need for building multiple application versions for different architectures.

Currently, there are many VMs available that can be used in desktop computers. The overhead of such a VM, when compared to a case in which there is no such software layer, is negligible as is mostly proved by the large amount of installations used both in academic and non-academic settings.

The major drawback of this approach is the size of the checkpoint data, incurring considerable transmission overhead. To attenuate this: (1) we assume that one base-generic running checkpoint image is accessible to all the clients; (2) the applications start their execution on top of this image once it is locally resumed; and (3) at checkpoint time, we only transmit the differences between the current image and the base image.

GiGi-MR provides redundant computing in which each computation is performed on multiple clients through the replication of input files. When a sufficient number of successful results have been returned, the GiGi-MR server compares them and sees if there is a consensus. In that case, the corresponding outputs are considered valid.

Each replication method provided by GiGi-MR is based on a different data partitioning technique, which consists of dividing a task into multiple subtasks that execute separately. This is achieved by splitting the initial input file into several smaller chunks, and requires the application to be completely parallel. For an application to be amenable to distributed computation, it must be possible to have its work partitioned in multiple tasks that run separately.

Through the use of data partitioning and task replication, GiGi-MR is able to detect collusion and validate results by

[5] Qemu is a generic and open source machine emulator and virtualizer. http://wiki.qemu.org/

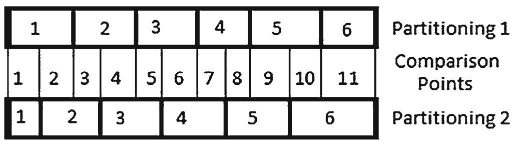


Fig. 5 The same work divided differently, creating an overlapped partitioning

comparing the outputs of redundant computations. However, the techniques used to identify incorrect results incur considerable overhead. None of the existing result verification techniques is able to ensure with 100 % certainty that a result is correct, though in some cases they can identify an incorrect one. The degree of certainty that a result is correct usually grows along with the overhead the technique incurs. Therefore, a compromise between the overhead and the reliability of the results can be found, and must be dynamically adaptable to the variable conditions/resources of the system.

This layer proposes a number of data partitioning approaches and a complementary sampling technique, which give the user ample choice on how to reach the desired compromise. The supported partitioning techniques are presented in the following sections.

2.3.1 Overlapped partitioning

Using overlapped partitioning, the tasks are never exactly equal, even though each individual piece of data is still replicated with the predetermined factor. Colluders must always execute part of the task, even when they are trying to return forged results. Figure 5 depicts the same work (input file) divided into two different overlapped partitionings, with two different sets of partitions. The file is divided into six chunks, but following different division offsets (where to split the initial file). There are 11 different comparison points (common chunks between two partitions) between each set of partitions, instead of the typical 6 of an integral replication (assuming a division of the file in six different partitions). These overlapped partitions can use a random offset and require strong communication among the colluders to identify the common part of the job. Although it is more probable for colluders to have common parts of the tasks, these common parts are smaller.

2.3.2 Relaxed partitioning

Overlapped partitioning can be implemented in a relaxed flavour, where only some parts of the job are executed redundantly. This lowers the overhead, but also lowers the reliability of the results. However, it can be useful if the system has low computational power available. Malicious participants are able to detect the common part of the job, however they can never be sure that the non-common part is not being

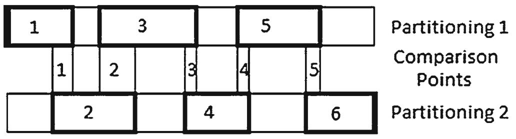


Fig. 6 Overlapped tasks for relaxed replication

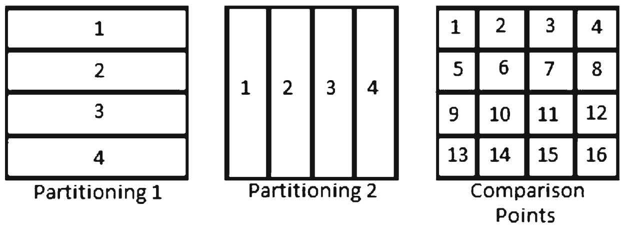


Fig. 7 Meshed partitioning using replication factor 2

executed redundantly. Figure 6 depicts a relaxed overlapped partitioning. We can see that in each partitioning scheme, the file is divided into three partitions. However, those partitions do not encompass the whole file (i.e., there are parts of the file that are not replicated). Therefore, the comparison points in which we can validate the output are much smaller than the whole file.

2.3.3 Meshed partitioning

Some applications can have their work divided in more than one dimension. Figure 7 depicts the partitioning of the work for a ray-tracer. The initial input file is split horizontally to create the first four partitions, and then vertically to create the remaining 4. When validating the results, there are 16 (4×4) comparison points between partitions. Like the overlapped partitioning, this influences the way colluders are able to introduce bad results: more points where they can collude, with a smaller size too. This partitioning provides a number of points of comparison, which are used to establish the "reputation" of a result. Each task's output is compared to 4 other tasks' outputs, according to the existing comparison points, and is evaluated according to the number of consensual results. For example, in Fig. 7 the 1st partition of "Partitioning 1" will have comparison points 1, 2, 3 and 4 (each one for a different partition of "Partitioning 2").

The algorithm for calculating the "reputation" of a result must take into account the outcome of comparison points (i.e., equal or different output). Since the majority of the participants is expected to be honest, finding the same result (equal) adds positive reputation while a different outcome adds negative reputation. For the acceptance of each point, equal outputs from both tasks are accepted on the fly, while disparate outputs are disambiguated according to the combined reputation of the two tasks that produced it. For example, if task 1 produces 4 correct outcomes, while task 2 produces 2 incorrect and 2 correct, then task 1 would have a better reputation (considering other tasks also produced correct results). Thus, in the discrepant comparison point between task 1 and 2, task 1's output would be accepted. If the reputation of both tasks is the same, the common portion of both results must be re-executed to achieve a voting quorum.

2.3.4 Samplification

Sampling consists on the local (in the server, in our case) execution of a fragment, as small as possible, of each task to be compared with the returned result. In essence, sampling points act as hidden embedded quizzes. Replication bases all its result verification decisions in results/info provided by third parties, i.e., the participant workers. In an unreliable environment, this may not be enough. Therefore, local sampling by the server can have an important place in the verification of results.

Sampling ensures that the malicious participants execute part of the task for this to have any chance of being accepted. Although random sampling can only ensure that a result is correct with a given probability (based on the size of the work, the number of samples and the percentage of the work that is corrupted), it can identify wrong results with certainty and deliver very useful information to a reputation mechanism.

We define Samplification as the combination of replication and random sampling, used sequentially to achieve higher reliability of the results: the winning result of the voting quorums is considered correct if it matches a random sample that was executed by the server. This technique is applicable to MapReduce jobs by having the server run a small part of a map input and then check against the returned outputs. For example, if running a *word count* application, the server would count the words present in a small part of an input file, and check if they were present in at least the same number inside the returned output files. Samplification allows the system to take advantage of the best of both mechanisms, while adding only marginal overhead (defined by the application owner).

Finally, samplification is also used to make sure that the parallelization of sequential tasks (provided in the User Interface layer, Sect. 2.1) does not alter the expected result. To that end, the GiGi-MR periodically executes the original application in the background, offline, sequentially and compares its results with the distributed version.

2.4 Resource discovery

This layer is responsible for implementing the Resource Discovery mechanisms on the GiGi-MR clients (that execute either map or reduce tasks). It is extremely important for the scheduling algorithm used by the server since the information obtained by the clients, through the exchange of resource and availability data, is sent back to the server, to the *Resource Updates* module. This module updates the server database with hosts' updated data, and is accessed by the *Scheduler* whenever replying to a client work request. This way, the server is more frequently updated with current knowledge on hosts, and is able to perform more reliable scheduling decisions.

Our Resource Discovery layer is also capable of searching not only for physical resources (e.g., CPU, Memory, etc.), but also services (e.g., facial recognition, high-resolution rendering, etc.) and applications (e.g., ffmpeg video encoder, programming language compilers, etc.)

In the GiGi-MR client, each type of resource is assigned a value from 0 to 1, where 0 means that the resource is unavailable and 1 that the resource is powerful and has good availability. The global (among all types of resources) availability value of a remote node may be obtained through a simple additive model [15]. In this way, we define the relative importance of each type of resource by defining weights (using methods like the swing weights). With them, it is then possible to make a weighted sum and obtain the global availability value, which would be the node rate.

As already mentioned in Sect. 2.2, GiGi-MR supports inter-client transfers, which reduce the burden on the server, and improve performance on more data-intensive scenarios, such as MapReduce jobs. Therefore, determining the available bandwidth between nodes can be of the utmost importance (e.g., when scheduling reduce tasks). However, measuring bandwidth of a single node in these environments can yield disparaging results. Our approach is to check the time for a message to travel from one node to another and back again (i.e., the round-trip time, RTT). To avoid flooding the network, we only have each client contact a small subset of remote nodes, called its neighbour set. Within a short period of time, the minimum RTT value obtained is kept and the bandwidth is calculated. The results obtained from this process are then passed on the server.

Without proper neighbour selection, this information would not be very helpful. Slower nodes could be coupled with far away nodes, or machines with faster connections that would not be taken advantage of. Therefore, this layer provides GiGi-MR clients with a neighbour selection mechanism that maximizes the system performance metrics.

Our algorithm considers two parameters as significant: proximity and resource availability. Proximity is measured through RTT, and includes bandwidth. Each peer contacts other nodes upon bootstrap and, periodically, once it has entered the network, records the RTT. The available bandwidth is inferred from these contacts, as well as from any inter-client transfers that occur when executing a MapReduce job. The resource availability parameter is defined through the previously mentioned node rate (additive model of a remote node's resources), and is included in these contact messages. The selection of neighbours is then based on a weighted measure of both proximity and node rate. The weight of each parameter is defined by the application developer (defaults to 0.5 each).

Once reported to the server, the neighbour set information is extremely useful for the server when scheduling tasks. As an example, when submitting a reduce task, the server is able to check if any of the neighbours of the node requesting work is executing a map task. If this is true, and the available bandwidth between both is large enough, the server can make this node a reducer. If, on the other hand, the requesting node has very low bandwidth to all its neighbours, the server is able to deduce that this node has low upload bandwidth. It is marked as unfit for a data-intensive reduce task, and a more compute intensive application is selected instead.

2.4.1 Using bloom filters

Attenuated bloom filters (ABF) were proposed in [26] to optimize location performance. It uses an array of Bloom Filters with depth d, where each row i, for $1 \leq i \leq d$, corresponds to the information stored at nodes i hops away. As the depth increases, more information will be stored in that Bloom Filter row, making the respective filter more attenuated and resulting in a higher probability of false positives. Therefore, information closest to the node is more accurate, and becomes less so as the distance between nodes increases. Using it in our system, each node in the network keeps a cached version of the ABF of its neighbours. This information is then combined into one single ABF by calculating the union of each Bloom Filter at the same depth from all neighbours. For instance, say node A receives the following ABF from its neighbours with depth d = 2: (00011, 10000) and (11001, 00001). To combine the information, the OR operation is performed for each depth. So, for $d = 1$, the resulting information is 11011, and for $d = 2$ it is 10001.

These aggregated ABF are sent to the server, once in every n work requests (if there have been no changes since the last requests, they are not included), and saved by the *Resource Updates* module. This module orders them according to the node's expected availability (how soon it is expected to be available for execution), and the Filters' depth (lower to higher). Saving all the received ABF would be impossible, and create incredible overhead. Therefore, the server uses timestamps to mark the validity of each one. Whenever an ABF has been in the system for more than the time-out interval specified, it is discarded. This keeps the number of ABF to a reasonable number, while still being useful for scheduling.

Whenever the server receives a work request, it checks the available tasks and, if there are any good matches with the requesting host, they are sent in reply. However, in the case of a mismatch, the *Scheduler* contacts *Resource Updates* and checks if there is any node which is a better match, and that is expected to become available within a short time frame. This search is conducted by starting with an ABF with a depth of 1 (neighbours to the node that submitted them). If there is a hit, the server looks in the Database (DB) for other tasks more suitable for this host. However, if the query does not return any matches, the tasks whose minimum requirements are fulfilled by the requesting client are submitted. In this way, the typical scheduling algorithm serves as a fail-safe method, ensuring that tasks are executed even if there are no optimal hosts to run them.

Information about resources, applications, and services offered by each node are represented inside a Bloom Filter. However, because a Bloom Filter is only capable of performing membership tests given a key, we need to store information about those resources in the actual key. For example, say a node has a CPU of 3 GHz, we cannot simply store the name "CPU" in the Bloom Filter, as the only information we can extract from that is that a node has a CPU. We need to add information about the actual resource (e.g., its value: 3,000 MHz) to the key that is inserted in the Bloom Filter for it to be useful. Bloom Filter keys store resource information by following a naming convention, and are used to differentiate between resources and their values. Our naming convention uses a 3-level namespace, each separated using the colon (":") as a delimiter, with the following rules: Level 1—Name of the Resource, Service, or Application (e.g., CPU, ffmpeg, etc); Level 2—Type of the Resource, Service, or Application (e.g., MHz, version, etc.); Level 3—Actual value of the Resource, Service, or Application. For instance, if we wanted to store the fact that a node has a CPU of 3 GHz, the key we would insert into the Bloom Filter would be: "CPU:GHz:3". The namespace definition in stored in a configuration file in the server, which is provided to the clients.

Some resources are mostly static and do not change often, like the Operating System, or CPU and Disk speed. However, there are other resources whose values can change quite often, such as amount of RAM occupied, or the amount of CPU in use. For those cases, if we used a classic Bloom Filter then it would need to be rebuilt periodically since it does not support the removal of elements. Moreover, this rebuilding procedure would require resending information about resources that are not expected to change, thus wasting bandwidth. Therefore, instead of using a classic ABF to store the information about the dynamic resources, a separate Counting ABF [12] is used.

2.5 Overlay management

As we mentioned previously, this layer acts as an interface between the system and an underlying network that connects GiGi-MR clients. This requires the use of a robust P2P overlay. In our example, we use Pastry [27], a generic, scalable and efficient Distributed Hash Table (DHT), but any other could be used. Node identifiers are randomly generated and assigned to a precise location on the circular addressing space of Pastry. By doing so, the machines holding adjacent nodes could be completely geographically dispersed.

As a bootstrap mechanism, the GiGi-MR server provides to new clients a list of entrypoints (boot nodes' IP address and port), corresponding to some hosts with high uptime (possibly servers). Each node inside the overlay receives information on the resources of a small number of remote peers, part of their neighbour set. The neighbour set is extremely important for our system since, as we mentioned before, it identifies which remote nodes' information is sent back to the server on each work request. Nodes advertise themselves by sending update messages to their neighbours whenever there is a significant change in resource availability. These messages are also sent periodically to keep them updated. Therefore, any changes in resource availability are announced to the node's neighbours.

These messages contain the sender node's related information: its identifier, its supported application identifiers, the time required for this information to expire, and its resource availability (e.g., CPU, bandwidth). Upon receiving this information, a neighbour node calculates, with its own judgement, the global rate of the announcer node. This judgement, as described before, consists of associating weights with the measured availability of every single resource. The proximity level between the announcer node and its neighbour is also taken into account.

In summary, this layer handles all communication between GiGi-MR clients and the overlay network. All messages received from the upper layers are sent to the network. The overlay contacts this layer whenever there is a message meant for the node related to resource updates. Finally, all changes to the node's neighbour set (e.g., remote node leaving) are reported. The *Resource Discovery* layer deals with those changes appropriately.

3 Implementation and evaluation

This section reveals some of the implementation details, presents the results of our experiments and describes the applications we use.

3.1 Implementation

GiGi-MR is designed on top of a BOINC client version 6.11.1 and server version 6.11.0.

For the network management, the Overlay Management layer uses the FreePastry[6] tool which is a Java implementation of the Pastry overlay.

[6] FreePastry. http://freepastry.rice.edu.

To measure resources so that they could be compared against each other in a simple additive model (used in the Resource Discovery layer, described in Sect. 2.4), we have to convert direct indicators of availability into a common scale, rated from 0 to 1. Therefore, we rely on the following expression to do that conversion: $f_r(x) = \min(1, x/\mathrm{MAX}_r)$. MAX_r is the value that we consider as very good for the resource r, and x is the direct measured value. For example, if we consider $\mathrm{MAX}_{\mathrm{CPU}} = 500$ and $x = 250$, we obtain $f(250) = 0.5$.

In addition, FreePastry provides a proximity metric (based on the RTT value) that is also converted to the common scale and used in the additive model. Therefore, the global availability value (i.e., the global node rate) is calculated through the following expression: $NR(a) = \sum k_r(a) \cdot v_r(a)$; $k_r(a)$ is the weight of the resource r in the node a, and $v_r(a)$ the value of the resource r in the node a (i.e., $f_r(x)$). Furthermore, the user is free to define the weights and the very good reference value associated with each resource.

To differentiate map tasks from "normal"ones (i.e., non-MapReduce tasks), the *MapReduce VC* layer modifies their templates by adding "$< mapreduce >$"tags with additional information such as job id and stage. The GiGi-MR server uses an additional general configuration file (in XML) to coordinate between stages and handle task creation. GiGi-MR clients use TCP for inter-client transfers (between mappers and reducers), due to its reliability and simplicity. A mapper opens a TCP socket to listen for incoming connections whenever it has finished a map task and its output is available. Incoming requests from reducers are accepted only for specified map files, and the socket is closed when there are no more files available for upload.

In the *User Interface* layer, the interaction between the Application Registrar and the GiGi-MR server is made by XML-RPC calls. Job information organization within the GiGi-MR server implies one modification: all user submitted jobs are processed within the same GiGi-MR project but may belong to different user projects. To accommodate this new information, a new table (User Project) had to be added to the server database. Furthermore, a Commodity Application table was added to accommodate the names and versions of the commodity applications available on remote hosts.

3.2 Evaluation

We evaluate GiGi-MR by running several tests over the Internet, in a scenario that resembles a typical VC environment. We run experiments with three different MapReduce applications (word count, inverted index, and N-Gram) to gauge our system's performance under different conditions. In addition, in order to evaluate each component independently we run micro-benchmarks, tailored to measure the impact and overhead of the different layers in our system. This section presents the results of our experiments.

Table 1 Evaluation of application transformation

	Outside GiGi-MR		Inside GiGi-MR		
	Original	Modified	1 CPU	2 CPU	3 CPU
Time (s)	60.01	61.93	64.08	32.59	16.97

3.3 User Interface

In this section, we present the experiments for the two features supported by the User Interface layer: transformation of sequential applications into parallel BoTs (Sect. 3.3.1); and submission of jobs by non-expert users (Sect. 3.3.2).

3.3.1 Application transformation

Our evaluation is twofold: (1) functional, developing sample applications and executing on different environments, and (2) quantitative, where we show the overhead incurred using our solution.

We parallelize a Monte-Carlo [23] computation to integrate one function. Instead of treating each random value in a sequential way, each task is responsible for obtaining part of the solution. In order to use this feature, the definition of a class is necessary, while a more simple solution would only require a loop with the computation code inside. The overhead incurred using GiGi-MR is minimal and easily outdone by the parallelization gains. Table 1 shows the overhead when running it on a single machine.

This evaluation was performed on an Intel(R) Core 2 Quad CPU with 4 cores running at 2.40 GHz. The tested application integrates one complex function using the Monte-Carlo method while generating 50 million random points. As seen in Table 1, there is an increase of execution time when running the modified version and using GiGi-MR. One of the reasons for the execution time increase is from the rewriting of the application: the inclusion of objects, and the increase of cycle interaction and method calls. More overhead is added by our system. In the version with 1 CPU, different threads for each object were created but serialized with the help of a lock, guaranteeing that they all executed on the same processor. It is observable an increase of about 2 s on the execution time leading to an overhead of about 1/8 of a second for each parallel object. If tasks are longer, these overheads will have a lower impact. Furthermore, with concurrent working processors all overhead is subdued by the gains of concurrent processing.

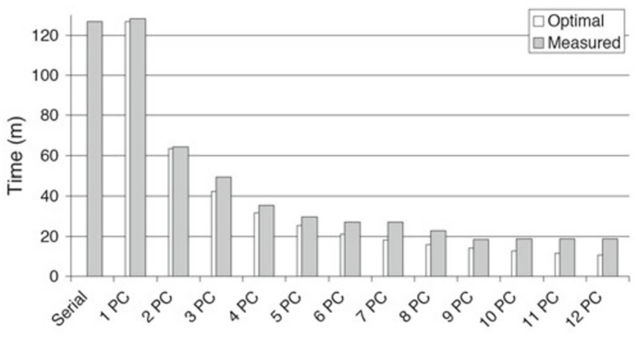


Fig. 8 Animation movie rendering times

3.3.2 Job submission by ordinary users

To evaluate the usability and performance gains, we deployed a GiGi-MR server and allowed some clients to use it. The experiments were done on our local network, to pinpoint the overhead brought on by our system more precisely. The experiment consists on using a ray tracer to generate an animation with 100 frames. On a Pentium 4 running at 3.2 GHz with Linux, each frame took between 3 and 100 s, giving a total rendering time of about 127 min. The times for the execution of these jobs on several computers, shown in Figure 8, are measured with identical computers connected by a 100 Mbit/s local network. We present the time to execute the jobs sequentially on one computer (both locally and by means of the GiGi-MR infrastructure) and on several computers.

As expected, the speedups are in line with the number of cycle donor hosts. The overhead incurred using our job distribution platform is minimal, only 2 min. This is caused by the job submission and client startup. With the participation of another host, even during a small period, this overhead is not noticeable. On an wide area network, or with larger input files, this overhead is larger but is easily surpassed with the contribution of another user.

3.4 MapReduce VC

We evaluate the performance of GiGi-MR in terms of application turnaround and network use by running several tests over the Internet, in a scenario that resembles a typical VC environment. We compare our results with an existing VC system (BOINC), referred to as VCS throughout this section. BOINC clients have the limitations mentioned previously (in Sects. 1 and 2), and do not support inter-client transfers.

The GiGi-MR server is able to support MapReduce jobs even in an environment composed solely of unmodified BOINC clients. This means that, even though the server is not able to leverage clients' resources (for scheduling and inter-client transfers), it is still able to distribute map and reduce tasks and obtain a valid final output. However, all communication must go through the central server, and as such there is no tolerance to server transient failures. To evaluate this hypothesis, in the VCS scenario we deploy a GiGi-MR server and unmodified BOINC clients (version 6.13.0).

We run experiments with three different applications (word count, inverted index, and N-Gram), in order to gauge our system's performance under different conditions. Due to space constraints, we only present the results from the N-Gram application. Note, however, that the other two applications show a similar performance.

We measure application turnaround, while differentiating between map and reduce stages to pinpoint potential bottlenecks and areas that would benefit most from improvement. Additionally, we monitor network traffic on the server. This allows us to identify the benefits of reducing the dependence on the central server. We run our experiments on PlanetLab, a wide-area testbed that supports the development of distributed systems and networks services. In these experiments, we use 50 nodes that work as the clients, and one node to act as server.

The Resource Discovery (Sect. 2.4) layer employs a neighbour selection mechanism that couples nodes with the best available bandwidth, promoting an homogenous network bandwidth; thus, the evaluation results were obtained for a network download bandwidth of approximately 700 KB/s. If other conditions (e.g., heavy churn or failure rate) are met, the network bandwidth may change accordingly thus requiring specific techniques to avoid such slow nodes (as shown in [17]).

3.4.1 Application turnaround

We begin by measuring application turnaround. We measure the time it took each MapReduce job to finish, starting from the initial download of map input files, and ending with the upload of the last reduce output. We separate the map and reduce steps to identify their respective weight in regards to the overall application turnaround time. The map stage is considered to be finished once all its output has been validated in the server.

The results obtained with N-Gram are shown in Fig. 9. The first conclusion is that GiGi-MR is able to finish the MapReduce job in half the time of VCS. We can also observe that the reduce stage on GiGi-MR is only slightly faster than VCS. This can be explained by the fast network connection of the server. Despite its large bandwidth, inter-client transfers still perform better than the centralized system. On the other hand, the differences in the map step are, as expected, much more significant. GiGi-MR is four times faster in executing the map stage, which translates to just a quarter of time needed by VCS to validate all its map tasks. This result shows that GiGi-MR performs better with applications that create large intermediate files.

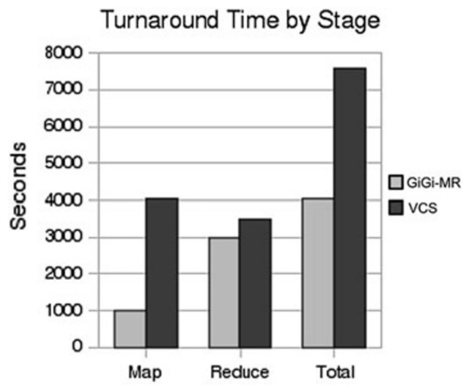


Fig. 9 Turnaround of N-Gram application by stage

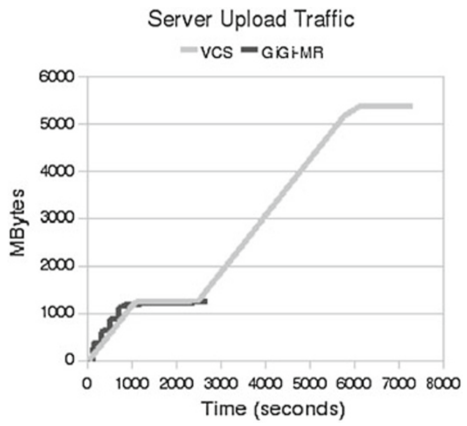


Fig. 10 Upload traffic for VCS and GiGi-MR server with N-Gram application

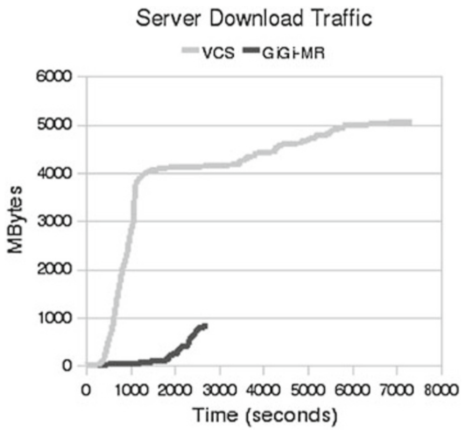


Fig. 11 Download traffic for VCS and GiGi-MR server with N-Gram application

3.4.2 Network traffic

We measure upload and download traffic in the server, for GiGi-MR and VCS while running the applications. Monitoring the network traffic on the server provides a more accurate measure of its overhead. It also allows us to quantify the impact of our solution concerning the decentralization of the VC model. We present the amount of data downloaded from the clients by the server, as well as the amount uploaded by the server to the clients.

The upload traffic for a server running N-Gram is shown in Fig. 10. Note that, as mentioned in the previous section, GiGi-MR has a much lower application turnaround than VCS. This is why the GiGi-MR line in Fig. 10 stops around second 3,000 (the same happens in Fig. 11), while VCS only finishes its execution much later. It is clear that there is a significant difference in the amount of data uploaded by GiGi-MR and VCS. This is due to the large size of intermediate files, which causes the VCS server to send almost five times more data to the clients than the GiGi-MR server in the reduce step.

The server's download traffic is exhibited in Fig. 11. Here, we can see the benefits of using hashes for map task validation (described in Sect. 2.2). Up until second 2000, the GiGi-MR server has received almost no data from the clients. At around that time in the experiment, reducers that finished their task began sending the output back to the server. The GiGi-MR server downloads a total of 820 MB from the clients. On the other hand, the VCS server is responsible for downloading all map outputs from mappers, which corresponds to the steep increase up until second 4000. The VCS server is required to download six times more data than GiGi-MR.

The inverted index and word count experiments yield very similar results, so they are not shown here. In inverted index, GiGi-MR is able to reduce the amount of data sent from the server from 6.5 to 2.3 GB and cut data received by the server by 96 %. In the word count application, the GiGi-MR server receives a mere 250 MB, a value ten times smaller than VCS's 3 GB, and is required to send 2.5 GB, whereas the VCS server sends more than double that amount to clients. Therefore, we can conclude that GiGi-MR not only performs better than VCS when running jobs with large intermediate files, but is also able to alleviate the server's network connection.

3.5 Checkpoint/restart and partitioning

In this section, we focus on the overhead of distributing tasks inside Virtual Machines, and the performance of different data partitioning techniques provided by our system when validating results.

3.5.1 Virtual machine checkpointing

The most relevant issue of the checkpoint/restart technique is the size of the checkpoint data. The potentially prohibitive VM image size is mitigated by the use of differential disk images (efficient representation of the modifications made to the virtual disk supported by the VM implementations with

Table 2 Checkpoint/Restart through a VM image: checkpoint data size using VirtualBox and Ubuntu Desktop 9.10

			Data size (KB)	
Base	Powered off	Disk image	2,651,169	2,768,998
Image	After Boot	Disk image (differential)	33	
		Volatile state	117,796	
Current	Running	Disk image (differential)	16,417	154,403
Image A	Application A	Volatile state	137,986	
Current	Running	Disk image (differential)	23,585	209,597
Image B	Application B	Volatile state	186,012	

specific disk image format files, such as QCOW2[7]). Table 2 depicts the size of the checkpoint data from the execution of a ray-tracer (POV-Ray) on two different inputs, attenuated with the use of differential disk images.

Checkpoint A reduces the size about 17 times (154,403 KB instead of 2,768,998 KB), checkpoint B about 14 times (209,597 KB instead of 2,768,998 KB), bringing transmission and/or storage costs to reasonable values. The differential disk is an efficient representation of the different amounts of modifications made to the virtual disk, which explains the difference we observe.

3.5.2 Result verification through replication

When using replication to validate returned results, GiGi-MR is able to take advantage of different data partitioning techniques (as described in Sect. 2.3). We present results from experiments using *Overlapped* and *Meshed* partitioning. Furthermore, our samplification technique is also evaluated.

We analyse the performance of GiGi-MR's result verification algorithm by identifying the percentage of wrong results that are not detected. We use a simulator to test result verification approaches with large populations. The simulator is a Java application that simulates a scenario where an n-dimensional job is broken into work units that are randomly assigned. Among the participants, there is a group of colluders that attempt to return the same bad result (based on complete or imperfect knowledge, depending on the partition overlapping), in order to fool the replication based verification mechanisms. The simulator returns the percentage of wrong results that were not detected by the server.

Figure 12 shows the results for overlapped partitioning. We can see that overlapped partitioning performs as well as standard partitioning (i.e., exact replicas of the whole file), in a scenario where the colluders are fully able to identify the common part and collude it, while still executing the rest of the task. This is possible in theory, but harder to achieve in practice as this may require global knowledge and impose heavier coordination and matching of information among the colluders. This is the worst case scenario, therefore overlapped partitioning may improve the reliability of the results, depending on how smart the colluders are.

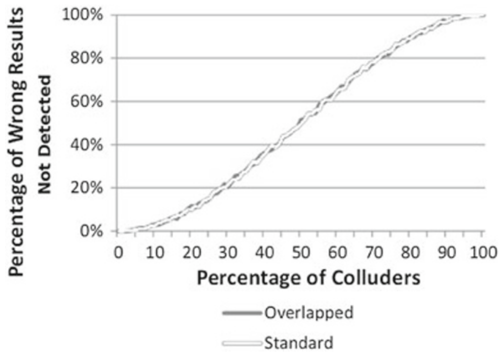


Fig. 12 Replication w/ standard partitioning vs. replication w/ overlapped partitioning, using replication factor 3

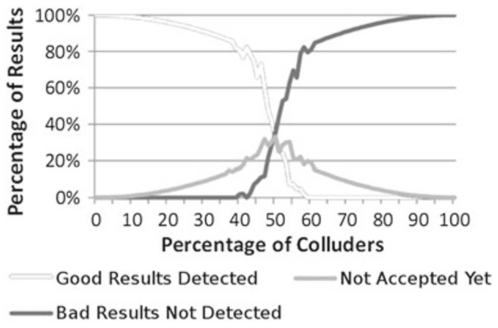


Fig. 13 Percentage of results using bi-dimensional meshed partitioning before rescheduling, in a scenario where colluders return results 100 % forged

Meshed Partitioning splits the task in more than one dimension and provides many points of comparison, which are then used to decide on the correctness of a result. Figure 13 shows that the percentage of undetected wrong results is very low, and almost null when dealing with under 40 % of colluders. However, a small number of results must be rescheduled to reach a verdict. The work that has to be

[7] The QCOW2 Image Format. http://people.gnome.org/~markmc/qcow-image-format.html.

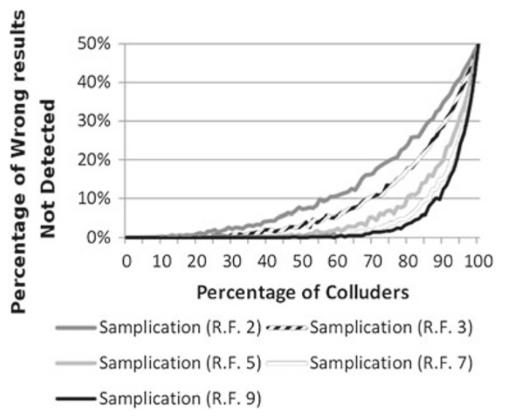


Fig. 14 Samplication: percentage of wrong results not detected in a scenario where colluders return results 50 % corrupted

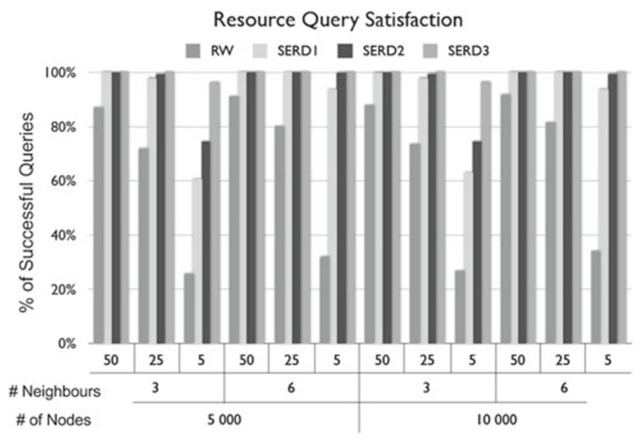


Fig. 15 Query satisfaction for static scenarios

rescheduled is mostly composed by the portions where wrong results overlap. Therefore, those results cannot be accepted, and rescheduling is the only solution. This technique proves to be very efficient as we are only using twice the base amount of work.

Samplication is a technique that combines sampling and replication without using voting quorums. Plus, this technique works with even replication factors. It uses information from replication to decide where to choose samples, rather than selecting samples randomly. It selects the samples within a replication mismatch area and discards the results that mismatch the chosen sample. If there is no mismatch in replication it resorts to random sampling. As seen in Figure 14, which shows scenarios with different replication factors (R.F.), this technique is quite effective, as it keeps the percentage of undetected wrong results very low, even for environments with up to 60 % of colluders.

3.6 Resource discovery

To evaluate GiGi-MR's discovery mechanism based on ABF, we compare it to a simpler mechanism, random walk (RW) [24], which acted as our baseline. We chose RW for being a simple, widely used discovery algorithm. We want to assess GiGi-MR's efficiency and effectiveness in finding available resources, since this can have a direct effect in the server's scheduling performance. The tests were ran using the PeerSim[8] simulator with its Event Driven capabilities, approximating the simulation more to real-life as opposed to a Cycle Driven simulation. We use an open source Bloom Filter implementation from the well known Hadoop project.[9]

The tests are executed with the random walk protocol and three variations of our algorithm for scalable and efficient resource discovery (SERD): SERD1, SERD2, and SERD3 which correspond to the ABF depths of 1, 2, and 3, respectively. In our experiments, we set the network size to 5,000 or 10,000 nodes, with either three or six neighbours per node. We define three resource distribution categories: 50 % (very abundant resource), 25 % (abundant resource), and 5 % (scarce resource). In addition, we consider the resources to be of two types: static (e.g., Operating System, or CPU) or dynamic (e.g., memory used). Every five simulation cycles, 10 % of the nodes in the network sent resource queries that could be satisfied by at least one node in the network, and we measure the percentage of successful queries.

Regarding the static scenario (see Fig. 15) SERD1 and SERD2 consistently show a percentage rate above 90 % except for the scarce scenarios with a maximum of three neighbours. This can be explained by the fact that the depth of the ABF did not allow the forwarding of queries with much hindsight, especially in a scenario where very few nodes actually contain the resource and where each node only has a maximum of three neighbours, thus further limiting a node's knowledge about the network. SERD3 has a satisfaction rate of 100 % in almost all scenarios, and 99 % in the rest. As the algorithm had a greater depth, it was able to direct queries in the right direction for them to be satisfied. The RW algorithm's lack of intelligence in the forwarding of queries is a great contrast, with almost all satisfaction rates below or around 80 %.

Figure 16 shows the query satisfaction for the dynamic resource scenarios, which are expected to not be as high as the static scenarios due to the varying values of the resources. Once again, SERD outperforms the RW protocol, which display a success rate of 80 % and lower. In almost all tests, the SERD protocols were above 80 %, except for the scarce scenario tests. In those, SERD1 struggled the most because it has little information about the neighbourhood. SERD2 and SERD3 only display a satisfaction rate lower than 80 %

[8] PeerSim. http://peersim.sourceforge.net/.

[9] Apache Hadoop. http://hadoop.apache.org/.

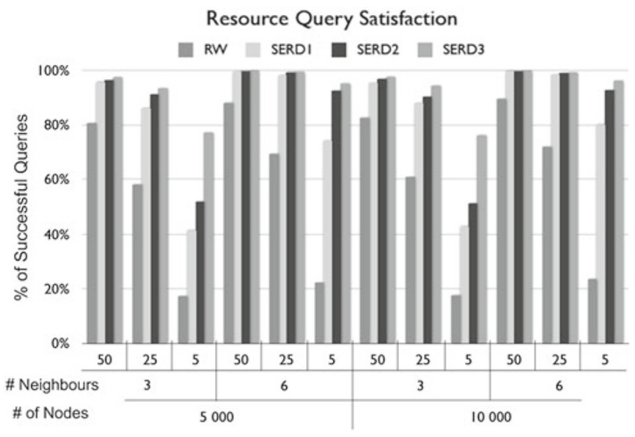


Fig. 16 Query satisfaction for dynamic scenarios

when the scarce scenario was combined with a maximum of 3 neighbours, which limited the available options when forwarding query messages. RW in those cases was hardly able to reach 20 % query satisfaction, making its lack of intelligence ever so apparent.

Analysis: In this section, we presented the evaluation of GiGi-MR. We summarize briefly its key aspects. First, we experienced performance improvements (in the execution and turnaround times) on a set of applications that are representative of those currently used, both in academic and commercial environments, such as e-science (ray tracing, imaging) and big data analytics (namely MapReduce as used by Google and others in production settings). Second, we obtained significant reductions in network traffic directed to servers, during execution, which improve server scalability and allow each server to handle larger computations with more participant nodes, i.e., scale to larger numbers of slaves to execute more tasks concurrently. Third, we improve the reliability of voluntary computing with a set of novel replication and sampling techniques, that require colluders always to execute part of their tasks, and by imposing more coordination and overhead to successfully forge results, all this combined with efficient and low overhead checkpointing. Finally, we showed how our system can scale to large populations of volunteers, while achieving efficient resource discovery and high resource utilization, thus taking advantage of idle resources scattered on the Internet, by means of the SERD resource discovery protocol.

4 Related work

XtremWeb [5] and Leiden Classical[10] are distributed computing projects that allow registered users to submit their jobs, as opposed to plain BOINC installations where only the system administrator creates jobs. In Leiden Classical, there is only one data processing application and users only submit input files to be processed by that application. XtremWeb is more versatile as it hosts several installed applications. In XtremWeb, users provide the input files and define the command line arguments used to invoke the application. XtremWeb allows the use of a broader set of applications, but still requires the system administrator to install them. A user is not allowed to install a new data processing application to solve his problems.

Supercomputing and data centers typically employ MPI task farmers when running BoT applications [16]. Task farming follows a master/worker model in which the master coordinates task creation and scheduling, distributes tasks among workers, and receives the results. In [16], MPI is extended to support dynamic process management and task creation in client/server applications. Despite having several common goals to our solution such as adaptive execution, or maximizing resource utilization, these systems operate in tightly coupled environments such as clusters. GiGi-MR's deployment over the Internet creates an entirely new set of requirements and challenges, which prevents us from adapting existing MPI Task Farming solutions.

Nimrod [1] is targeted at parameter sweep applications, and follows a model similar to task farming. In Nimrod, the user defines the input files, the type of parameters and how they vary. Nimrod then generates all parameter combinations and assigns each parameter combination to a task. Even though Nimrod helps on the combination of all parameters, the user must still have some programming knowledge, because the processing application must be coded and the data type of each parameter must be defined.

Combining the concepts of Cloud and Volunteer Computing has been proposed in [18], in which the authors studied the cost and benefits of using clouds as a substitute for volunteers or servers.

In [22], the authors define a P2P model under the MapReduce framework. Their system is tailored to a dynamic cloud environment, creating a cloud of clouds. It has a similar organization to existing Grid infrastructures, but, much like OurGrid [8], is meant to create a federation or cluster of data centers through a P2P overlay network.

MOON (MapReduce On opportunistic eNvironments) [20] proposes an extension to Hadoop that implements adaptive task scheduling to account for node failure. However, MOON is tailored for a cluster environment, such as a research lab, in which nodes are trusted or even dedicated.

MapReduce was also adapted to desktop grids in [31]. The system was designed on top of BitDew [13], a middleware the handles data management through the use of various transfer protocols. The authors claim that it is able to run MapReduce jobs on XtremWeb [5], over the Internet. However, their experiments were conducted in a cluster interconnected by

10 University of Leiden. Leiden classical. http://boinc.gorlaeus.net/.

Gigabit Ethernet. This environment more closely resembles the common scenario of XtremWeb, which consists of a federation of research labs.

BOINC, on the other hand, has million of users, and is actually tailored for a truly volunteer environment over the Internet. By moving from benchmarks and proof-of-concepts to actual applications in a realistic testbed, we can state with more certainty what are the advantages and shortcomings of this paradigm on a volunteer computing environment.

Bloom Filters have been applied in a variety of systems [6], such as dictionaries, databases, and network applications. They are implemented as bit arrays, therefore, the union of two sets can be computed by performing the OR operation between the two, while their approximate intersections can be computed using the AND operation. To test whether an element is in the set or not, it has to be passed through all hash functions and if all the resulting positions in the array are set to one, then the element hash a high probability of being in the set. If any position has the value zero, then we know that it is definitely not in the set. The small false positive rate arises from the fact that when querying for an element that is not in the set, some hash functions may result in positions that were already used (have the value one) for a previously inserted item. Therefore, the more elements are inserted into the Bloom Filter, the higher the chance of a query resulting in a false positive. Another shortcoming is the inability to remove an element from the Bloom Filter, as simply setting the positions given by the k hash functions to zero have the side effect of removing other elements as well.

Our solution is different to the existing systems because it combines all types of different resources into one discovery mechanism. It is especially different to the works [14,21] that also make use of ABF due to to the usage of one aggregated ABF (explained in Sect. 2.4.1), and the fact that all the different types of basic resources, services, and applications are encoded in the Bloom Filter.

5 Conclusion

We have presented GiGi-MR, a Volunteer Computing platform that allows ordinary users to create and submit jobs for execution on volunteer machines over the Internet. Our system is able to execute MapReduce applications over the Internet and tolerate volunteer faults, and transient server failures. Furthermore, it is compatible with existing VC systems (in particular BOINC). It significantly reduces the dependence on the central server, which is typically overburdened in current VC platforms, thus allowing it to obtain better performance.

GiGi-MR enhances task scheduling using information exchanged by clients within an overlay network. Neighbour selection is based on resources and availability information is provided through a novel resource discovery mechanism. It is capable of locating physical resources, services, and applications from many computers connected to the same overlay. This is done in a novel way by storing all resource, application, and service information in ABF. GiGi-MR is able to distribute tasks inside Virtual Machines, and supports several partitioning mechanisms, thus increasing the system's adaptability and usefulness.

We evaluated GiGi-MR by measuring the application turnaround and server network traffic while running three different MapReduce applications. We also ran micro-benchmarks to assess the impact of each of our system's components.

The experiments show that the overhead of the *User Interface* layer is minimal, and that it is possible to take advantage of parallel processing environments without the use of complex APIs. We can also conclude that it allows the definition and execution of a myriad of jobs that can take advantage of remote idle cycles. We managed to execute a batch of image rendering, necessary to create an animation video, as well as process several MapReduce jobs. In general, the applications that our systems handles best are those which can be described as Bag-of-Tasks problems, or easily decomposed in a set of map and reduce tasks; thus, Monte-Carlo based applications are good candidates.

GiGi-MR's discovery mechanism performed well in the various test scenarios that included static and dynamic resoures, and outperformed the RW protocol which was our baseline. Our system proved to be effective in locating various types of resources, and scalable as the number of nodes in the network did not affect the mechanism's resource query satisfaction.

Our result verification schemes vary in their complexity and overhead. Replication with overlapped partitionings makes collusion harder to achieve, while ensuring that the reliability of the results is the same as using standard partitionings. Replication with meshed partitionings enables the use of even replication factors and improves the reliability of the results using its stateless result reputation algorithm. Samplication combines replication and sampling in an elegant manner, ensuring it takes the best advantage of redundant execution through the comparison with local samples rather than using voting quorums.

Our checkpoint/restart through a virtual machine overcame its biggest obstacle, checkpoint data size, through differential disk images and compression. We were able to minimize the checkpoint size about 17 times, to a transmittable amount of data.

Our solution was able to improve the performance of all the MapReduce jobs we tested. The map stage was up to 4 times faster than in an existing VC system. The reduce step also showed an improvement, thus reducing each MapReduce job's execution time down to less than half. Our experiments regarding the server's network traffic also gave us some interesting results. We were able to reduce server download traffic by an order of magnitude on the word count and inverted index applications. Therefore, we were able to witness a decrease in uploaded data to 20 % of the existing VC system server's value.

Acknowledgments The authors would like to thank students Filipe Paredes, João Paulino and Raoul Felix, for their work and enthusiasm during the project.

References

1. Abramson D, Sosic R, Giddy J, Hall B (1995) Nimrod: a tool for performing parametrised simulations using distributed workstations. In: Proceedings of the 4th IEEE international symposium on high performance distributed computing, HPDC '95. IEEE Computer Society, Washington, DC, USA, pp 112–121.
2. Anderson DP (2004) Boinc: A system for public-resource computing and storage. In: Proceedings of the 5th IEEE/ACM international workshop on grid computing, GRID '04. IEEE Computer Society, Washington, DC, USA, pp 4–10
3. Anderson DP, Cobb J, Korpela E, Lebofsky M, Werthimer D (2002) Seti@home: an experiment in public-resource computing. Commun ACM 45:56–61
4. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. Commun ACM 13(7):422–426
5. Cappello F, Djilali S, Fedak G, Herault T, Magniette F, Néri V, Lodygensky O (2005) Computing on large-scale distributed systems: Xtremweb architecture, programming models, security, tests and convergence with grid. Future Gener Comput Syst 21:417–437
6. Chazelle B, Kilian J, Rubinfeld R, Tal A (2004) The bloomier filter: an efficient data structure for static support lookup tables. In: Proceedings of the fifteenth annual ACM-SIAM symposium on discrete algorithms, Society for Industrial and Applied Mathematics, SODA '04, Philadelphia, PA, USA, pp 30–39
7. Chun B, Culler D, Roscoe T, Bavier A, Peterson L, Wawrzoniak M, Bowman M (2003) Planetlab: an overlay testbed for broad-coverage services. SIGCOMM Comput Commun Rev 33:3–12
8. Cirne W, Brasileiro F, Andrade N, Costa L, Andrade A, Novaes R, Mowbray M (2006) Labs of the world, unite!!!. J Grid Comput 4:225–246
9. Costa F, Kelley I, Silva L, Fedak G (2008a) Optimizing data distribution in desktop grid platforms. Parallel Process Lett (PPL) 18(3):391–410
10. Costa F, Silva L, Fedak G, Kelley I (2008b) Optimizing the data distribution layer of boinc with bittorrent. In: International symposium on parallel and distributed processing symposium, pp 1–8
11. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51:107–113
12. Fan L, Cao P, Almeida J, Broder AZ (2000) Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans Netw 8(3):281–293
13. Fedak G, He H, Cappello F (2008) Bitdew: a programmable environment for large-scale data management and distribution. In: Proceedings of the 2008 ACM/IEEE conference on supercomputing, SC '08. IEEE Press, Piscataway, pp 45:1–45:12
14. Goering P, Heijenk G (2006) Service discovery using bloom filters. In: Proceedings of twelfth annual conference of the advanced school for computing and imaging, pp 14–16
15. Goodwin P, Wright G (2004) Decision Analysis for Management Judgment. Wiley, New York
16. Gropp W, Lusk E (1995) Dynamic process management in an mpi setting. In: Proceedings of the seventh IEEE symposium on parallel and distributed processing, 1995, pp 530–533
17. Guo Z, Fox G, Zhou M (2012) Investigation of data locality in mapreduce. In: Proceedings of the 2012 12th IEEE/ACM international symposium on cluster, cloud and grid computing (ccgrid 2012), CCGRID '12. IEEE Computer Society, Washington, DC, pp 419–426
18. Kondo D, Javadi B, Malecot P, Cappello F, Anderson DP (2009) Cost-benefit analysis of cloud computing versus desktop grids. In: Proceedings of the 2009 IEEE international symposium on parallel and distributed processing, IPDPS '09. IEEE Computer Society, Washington, DC, USA, pp 1–12
19. Larson SM, Snow CD, Shirts M, Pande VS (2002) Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. Comput. Genomics.
20. Lin H, Ma X, Archuleta J, Feng Wc, Gardner M, Zhang Z (2010) Moon: mapreduce on opportunistic environments. In: Proceedings of the 19th ACM international symposium on high performance distributed computing, HPDC '10. ACM, New York, pp 95–106
21. Lv Q, Cao Q (2007) Service discovery using hybrid bloom filters in ad-hoc networks. In: International Conference on wireless communications, networking and mobile computing, 2007. WiCom 2007, pp 1542–1545
22. Marozzo F, Talia D, Trunfio P (2008) Adapting mapreduce for dynamic environments using a peer-to-peer model. In: Proceedings of the first workshop on cloud computing and its applications (CCA 2008), Chicago, USA
23. Metropolis N, Ulam S (1949) The Monte Carlo method. J Am Stat Assoc 44(247):335–341
24. Pearson K (1905) The problem of the random walk. Nature 72
25. Ratnasamy S, Francis P, Handley M, Karp R, Shenker S (2001) A scalable content-addressable network. In: Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications, SIGCOMM '01. ACM, New York, pp 161–172
26. Rhea S, Kubiatowicz J (2002) Probabilistic location and routing. In: Proceedings of INFOCOM 2002. Twenty-first annual joint conference of the IEEE computer and communications societies, vol 3. IEEE, New York, pp 1248–1257
27. Rowstron A, Druschel P (2001) Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui R (ed) Middleware 2001. Lecture Notes in Computer Science, vol 2218, Springer, Berlin, pp 329–350
28. Silva J, Ferreira P, Veiga L (2010) Service and resource discovery in cycle-sharing environments with a utility algebra. In: 2010 IEEE international symposium on parallel distributed processing (IPDPS), pp 1–11
29. Snir M, Otto SW, Walker DW, Dongarra J, Huss-Lederman S (1995) MPI: the complete reference. MIT Press, Cambridge
30. Stoica I, Morris R, Karger D, Kaashoek MF, Balakrishnan H (2001) Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications, SIGCOMM '01. ACM, New York, pp 149–160
31. Tang B, Moca M, Chevalier S, He H, Fedak G (2010) Towards mapreduce for desktop grid computing. In: Proceedings of the 2010 international conference on P2P, parallel, grid, cloud and internet computing, 3PGCIC '10. IEEE Computer Society, Washington, DC, pp 193–200