Exploring Fault-tolerance and Reliability in a Peer-to-peer Cycle-sharing Infrastructure

João Paulino *, Paulo Ferreira, and Luís Veiga

INESC-ID/IST
Rua Alves Redol N°9, 1000-029, Lisboa, Portugal
joaopaulino@ist.utl.pt,{paulo.ferreira, luis.veiga}@inesc-id.pt

Abstract. The partitioning of a long running task into smaller tasks that are executed separately in several machines can speed up the execution of a computationally expensive task. This has been explored in Clusters, in Grids and lately in Peer-to-peer systems. However, transposing these ideas from controlled environments (e.g., Clusters and Grids) to public environments (e.g., Peer-to-peer) raises some reliability challenges: will a peer ever return the result of the task that was submitted to it or will it crash? and even if a result is returned, will it be the accurate result of the task or just some random bytes? These challenges demand the introduction of result verification and checkpoint/restart mechanisms to improve the reliability of high performance computing systems in public environments. In this paper we propose and analyse a twofold approach: i) two checkpoint/restart mechanisms to mitigate the volatile nature of the participants; and ii) various flavours of replication schemes for reliable result verification.

Keywords: fault-tolerance, result verification, checkpoint/restart, cycle-sharing, public computing

1 Introduction

The execution of long running applications has always been a challenge. Even with the latest developments of faster hardware, the execution of these is still infeasible by common computers, for it would take months or even years. Even though super-computers could speed up these executions to days or weeks, almost no one can afford them. The idea of executing these in several common machines parallely was firstly explored in controlled environments [19, 3, 9, 4] and was later transposed to public environments [2, 12]. Although they are based on the same principles, new challenges arise from the characteristics of public environments.

Clusters [19, 3] and Grids [9, 4, 13] have been very successful in accelerating computationally intensive tasks. The major difference between these is that

* This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds. João Paulino and this work were supported by FCT research project GINGER - PTDC/EIA/73240/2006

while clusters use dedicated machines in a local network, grids consider the opportunistic use of workstations owned by institutions around the Globe. Both systems are composed by well managed hardware, trusted software and a near 24 hour per day uptime. Public computing [2, 12, 1, 5, 10, 15] stems from the fact that the World's computing power and disk space is no longer exclusively owned by institutions. Instead, it is distributed in the hundreds of millions of personal computers and game consoles belonging to the general public. These systems face new challenges inherent to their characteristics: less reliable hardware, untrusted software and unpredictable uptime.

One of the several public computing projects is GINGER (Grid Infrastructure for Non Grid EnviRonments), in the context of which the work of this paper has been developed. GINGER [20] proposes an approach based on a network of favours where every peer is able to submit his work-units to be executed on other peers and execute work-units submitted by other peers as well. A specific goal of GINGER is that in order to be able to run an interesting variety of applications without modifying them, GINGER proposes the concept of Gridlet, a semantics-aware unit of workload division and computation off-load (basically the data, an estimate of the cost, and the code or a reference to it). Therefore, GINGER is expected to run applications such as audio and video compression, signal processing related to multimedia content (e.g., photo, video and audio enhancement, motion tracking), content adaptation (e.g., transcoding), and intensive calculus for content generation (e.g., ray-tracing, fractal generation).

The highly transient nature of the participants in the system may origin a constant loss of already performed work when a peer fails/leaves or even the never ending of a task, if no peer is ever enough time available to accomplish it. To mitigate this, checkpointing/restart mechanisms shall be able to save the state of a running application to safe storage during the execution. Allowing it to be resumed in another peer from the point when it was saved if necessary.

The participants of the system are not trusted, so are the results they return. Results may be invalid (e.g., either corrupted data or format non-compliance), or otherwise valid but in disagreement with input data (e.g., repeated results from previous executions with different input, especially one computationally lighter). Therefore, result verification mechanisms shall be able to check the correctness of the results.

In the next Section, we address the relevant related work to ours. In Section 3, we propose result verification techniques and checkpoint/restart mechanisms. In Section 4 we provide a description of our implementation. In Section 5, we evaluate the proposed techniques. Section 6 concludes.

2 Related Work

In Section 2.1 we review the main approaches to provide an application with checkpoint/restart capabilities; in Section 2.2 we analyse the techniques that are mainly used to verify the correctness of the results.

2.1 Checkpoint/Restart

Checkpoint/restart is a primordial fault-tolerance technique. Long running applications usually implement checkpoint/restart mechanisms to minimize the loss of already performed work when a fault occurs [16, 8]. Checkpoint consists in saving a program's state to stable storage during fault-free execution. Restart is the ability to resume a program that was previously checkpointed. To provide an application with these capabilities, various approaches have been proposed: Application-level [2, 12] , Library-level [18, 14] and System-level [21].

Application-level Checkpoint/Restart Systems. These systems do not use any operating system support. These are usually more efficient and produce smaller checkpoints. They also have the advantage of being portable[1]. These checkpointing mechanisms are implemented within the application code, requiring a big programming effort. Checkpointing support built in the application is the most efficient, because the programmer knows exactly what must be saved to enable the application to restart. Though, this approach has some drawbacks: it requires major modifications to application's source code (its implementation is not transparent [2] to the application); the application will take checkpoints by itself and there is no way to order the application to checkpoint if needed; it may be hard, if not impossible, to restart an application that was not initially designed to support checkpointing; and it is a very exhaustive task to the programmer. This programming effort can be minimized using pre-processors that add checkpointing code to the application's code, though they usually require the programmer to state what needs to be saved (e.g., through flagged/annotated code). Public computing systems like Seti@home [2] and Folding@home [12] use this checkpointing solution.

Library-level Checkpoint/Restart Systems. This approach consists in linking a library with the application, that creates a layer between the application and the operating system. This layer has no semantic knowledge of the application and cannot access kernel's data structures (e.g., file descriptors), so this layer has to emulate operating system calls. The major advantage is that a portable generic checkpointing mechanism could be created, though it is very hard to implement a generic model to checkpoint any application. This checkpointing method requires none or very few modifications to the application's code. This approach has been explored by libckpt [18] and Condor [14].

System-level Checkpoint/Restart Systems. These systems are built as an extension of the operating system's kernel [21], therefore they can access kernel's data structures. Checkpointing can consist in flushing all the process's data and control structures to stable storage. Since these mechanisms are external to the application they do not require specific knowledge of it, and they require none

[1] Portability is the ability of moving the checkpoint system from one platform to another.

[2] Transparency is the ability of checkpointing an application without modifying it.

or minimal changes to the application. They have the obvious disadvantage of not being portable and usually more inefficient than application-level.

2.2 Result Verification

The results returned by the participants may be wrong due either to failures or malicious behaviour. Failures occasionally produce erroneous results that must be identified. Malicious participants create bad results that are intentionally harder to detect. Their motivation is to discredit the system, or to grow a reputation for work they have not executed (i.e., to cheat the public computing system and exploit other participants' resources).

Replication. One of the most effective methods to identify bad results is through redundant execution. In these schemes the same job is performed by N different participants (N being the replication factor). The results are compared using voting quorums, and if there is a majority the corresponding result is accepted. Since, it is virtually impossible for a fault or independent byzantine behaviour to produce the same bad result more than once, this technique easily identifies and discards the bad ones. However, if a group of participants colludes it may be hard to detect a bad result [17, 6]. Another disadvantage is the massive overhead it generates. Most of the public computing projects [2, 12] use replication to verify their results, it is a high price they are willing to pay to ensure their results are reliable. However, when there is no collusion, it is virtually capable of identifying all the bad results with 100% certainty.

Hash-trees. Cheating participants can be defeated if they are forced to calculate a binary hash-tree from their results, and return it with them [7]. The submitting peer only has to execute a small portion of a job and calculate its hash. Then, when receiving results, the submitting peer compares the hashes and verifies the integrity of the hash-tree. This dissuades cheating participants because finding the correct hash-tree requires more computation than actually performing the required computation and producing the correct results. The leafs of the hash tree are the results we want to check. The hash is calculated using two consecutive parts of the result concatenated, starting by the leafs. Once the tree is complete, the submitting peer executes a random sample of the whole work that corresponds to a leaf. Then this result is compared to the returned result and the hashes of the whole tree are checked. Hash-trees make cheating not worthwhile. They have a relative low overhead: a small portion of the work has to be executed locally and the hash tree must be checked.

Quizzes. Quizzes are jobs whose result is known by the submitter a priori. Therefore, they can test the honesty of a participant. Cluster Computing On the Fly [15] proposed two types of quizzes: stand-alone and embedded quizzes.

Stand-alone quizzes are quizzes disguised as normal jobs. They can test if the executing peer executed the job. These quizzes are only useful when associated with a reputation mechanism that manages the trust levels of the executing peers

[11]. Though, the use of the same quiz more than once can enable malicious peers to identify the quizzes and to fool the reputation mechanisms. The generation of infinite quizzes with known results incurs considerable overhead.

Embedded quizzes are smaller quizzes that are placed hidden into a job, the job result is accepted if the results of the embedded-quizzes match the previously known ones. These can be used without a reputation system. Though, the implementation tends to be complex in most cases. Developing a generic quiz embedder is a software engineering problem that has not been solved so far.

3 Architecture

In section 3.1 we propose two checkpointing mechanisms that enable GINGER to checkpoint and restart any application; in section 3.2 we discuss result verification techniques that we implemented in GINGER, we consider various flavours of replication and a straightforward sampling technique.

3.1 Checkpoint/Restart Mechanisms

In GINGER we want to provide a wide range of applications with checkpoint/restart capabilities, while keeping them portable to be executed on cycle-sharing participant nodes, and without having to modify them. Library-level is the only approach in the related work that would fit. However, an approach simply stating these goals is still far from being able to checkpoint any application. Therefore, we propose two mechanisms that will enable us to checkpoint any application.

Generic Checkpoint/Restart. An application can be checkpointed if we run it on top of virtual machine with checkpoint/restart capabilities (e.g., qemu), being the application state saved within the virtual machine state. This also provides some extra security to the clients, since they can be executing untrusted code. 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 peers; 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. The checkpoint data size can be further reduced using various techniques: optimized operating systems (e.g., just enough operating system or JeOS); differencing not only the disk but also the volatile state; and applying compression to the data. This approach does not have semantic knowledge of the applications, it cannot preview results. However we may be able to show some statistical data related to the execution and highlight where changes have occurred.

Result Checkpoint/Restart. This technique will only be fit for some applications and demands the implementation of specific enabling mechanisms for each application. The idea behind this technique is that the applications produce final results incrementally during their execution. Therefore, if we are able

to capture the partial results during execution and resume execution from them later, such result files can actually serve as checkpoint data. This creates a very efficient checkpointing mechanism. This technique can be implemented using two different approaches: by monitoring the result file that is being produced by the application; or by dividing the gridlet work into subtasks in the executing peer. Since this approach has semantic knowledge of the application result it can checkpoint whenever it is more convenient (e.g., every 10 lines in an image written by a ray-tracer); rather than on a predefined time interval. This awareness of the semantics of the application also enables the monitoring of the execution and the previewing of the results in the submitter.

3.2 Result Verification Mechanisms

In order to accept the results returned by the participants we propose replication with some extra considerations and a complementary sampling technique.

Incremental Replication. The insight of assigning the work iteratively according to some rules, instead of putting the whole job to execution at once can provide some benefits with only minor drawbacks.

The major benefit stems from the fact that lots of redundant execution is not even taken into consideration when the correct result is being chosen by the voting quorums. For example, for replication factor 5, if 3 out of the 5 results are equal the system will not even mind looking at the other 2 results. Then, those could and should have never been executed. And if so, the overall execution power of the system would have been optimized by avoiding useless repeated work. This replication scheme has additional benefits in colluding scenarios. In these, the same bad result is only returned once the colluders have been able to identify that they have the same job to execute. If a task is never being redundantly executed at the same time, colluders can only be successful if they submit a bad result and wait for the replica of that task to be assigned to one of them, enabling them to return the same bad result. If that does not happen, the bad result submitted by them will be detected and they might be punished by an associated reputation mechanism (e.g., blacklisted).

This technique can have a negative impact in terms of time to complete the whole work: in one hand, the incremental assignment and wait for the retrieval of results will lower the performance when the system is not overloaded; on the other hand, if the number of available participants is low it can actually perform faster than putting the whole work to execution at once. Therefore, the correct definition of an overloaded environment having into consideration various factors (e.g., the number of available participants, the maximum number of gridlets, etc.) makes possible for the system to decide whether to use this technique or not, enabling it to take the best advantage of the present resources.

Replication with 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. Therefore, it becomes more complex for

the colluders to identify the common part of the task, plus they have to execute part of the task. Figure 1 depicts the same work divided in in two different overlapped partitionings. Overlapped partitioning could be implemented in a

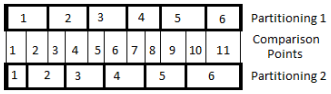

Fig. 1. The same work divided differently creating an overlapped partitioning.

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. Figure 2 depicts a relaxed overlapped partitioning.

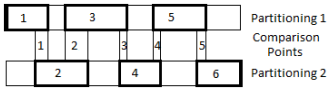

Fig. 2. Overlapped tasks for relaxed replication.

Replication with Meshed Partitioning. Some applications can have their work divided in more than one dimension. Figure 3 depicts the partitioning of the work for a ray-tracer. Like the overlapped partitioning it influences the way colluders are able to introduce bad results: more points where they can collude, with a smaller size too. This partitioning provides lots of points of comparison. This information might feed an algorithm that is able to choose correct results according to the reputation of a result, instead of using voting quorums.

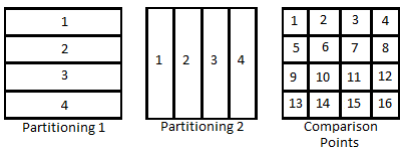

Fig. 3. Meshed partitioning using replication factor 2.

Sampling. 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 can have an important place in the verification of results. Sampling considers the local 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. This sample is the only trusted result, so even if a result that was accepted by the voting

quorums does not match the local sample it is discarded. Figure 4 depicts the sampling of an image where a sample is a pixel. We have proposed two check-

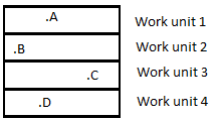

Fig. 4. Sampling for an image.

point/restart enabling techniques. Our generic checkpointing technique enables us to checkpoint any application and resume it later, the overhead it incurs derives from the size of the checkpoint data. Nevertheless, for some applications our result oriented technique will enable the system to checkpoint and resume an application with no noticeable overhead, the results are transmitted incrementally, rather than at the end of the execution. For a reliable result verification we have proposed various flavours of replication that make colluding increasingly more difficult to achieve and easier to detect. Replication is combined with a sampling technique that tests the received untrusted results against one result sample that is known to be correct.

4 Implementation

Our implementation is developed in two different deployments: i) a simulation environment, that enables us to test result verification approaches with large populations; and ii) a real environment, that proves that our result verification and the checkpointing approaches are feasible.

4.1 Simulation Environment

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 receives several parameters: number of participants; number of colluders; work-size as an array of integers (n-dimensional representation), the size as defined in terms of atoms of execution (i.e., an indivisible portion of execution); number of gridlets; replication factor; and partitioning mode (standard or overlapped). The simulator returns several results, being the most important one the percentage of bad results that were accepted by the system.

4.2 Real Environment

For being able to support a new application, semantic knowledge of it is mandatory. Therefore, 3 classes must be programmed: an application manager, a gridlet

and a result. The Application Manager is responsible for translating a command that invokes an application into several executable gridlets and reunite their partial results once it has verified their correctness. For some applications it also enable the user to preview the results (e.g., an image being incrementally produced by a ray tracer). It must extend the abstract class ApplicationManager and implement a constructor and two methods. The following is an excerpt of the Pov Ray's application manager class.

```
class PovRayManager extends ApplicationManager {

  PovRayManager(String command) throws ApplicationManagerException { ... }

  ArrayList<Gridlet> createGridlets(int nGridlets) { ... }

  void submitResult(Result result) { ... }
}
```

The Gridlet class must implement the Serializable and Runnable interfaces, this enables transportation by the Java RMI and allows it to perform a threaded execution in its destination. The following is an excerpt of the Pov Ray's gridlet class.

```
class PovRayGridlet extends Gridlet implements Serializable, Runnable {

    void run() { ... }
}
```

The Result class is just a container of the result data of a gridlet, it must be defined for each application and implement the Serializable interface, for the Java RMI mechanisms being able to transmit it.

5 Evaluation

At this stage of our work, we only have few performance measures of the techniques we have described. We present what we have been able to measure so far in this section.

5.1 Checkpoint/Restart Mechanisms

The major issue of the generic checkpoint/restart technique is transmitting the state of a virtual machine. We are mitigating this by transmitting only the differences between the current image and the base-image. The table in Figure 5 depicts the size of the checkpoint data to be transmitted.

		size (KB)	total size (KB)
Base Image	disk image - powered off (.vdi)	2.650.145,00	2.764.961,00
	disk image - after OS boot (differencial .vdi)	33,00	
	volatile state - after OS boot (.sav)	114.783,00	
Current Image	disk image - running application (differencial .vdi)	21.537,00	161.455,00
	volatile state - running application (.sav)	139.918,00	

Fig. 5. Checkpoint data size using VirtualBox and Ubuntu Desktop 9.10

5.2 Result Verification Mechanisms

Replication can be fooled if a group of colluders determines they are executing redundant work and agree to return the same bad result, forcing the system to accept it. The graphic in Figure 6 depicts that when the percentage of colluders is under 50%, the greater the replication factor the lower the percentage of bad results accepted; when the percentage of colluders is above 50%, albeit a less probable scenario, replication actually works against us. Groups of colluders are usually expected to be minorities. However we must take into account that if they are able to influence the scheduler by announcing themselves as attractive executers the percentage of bad results could even be above what this graphic shows, for the scheduler it uses is random. Overlapped partitioning influences the

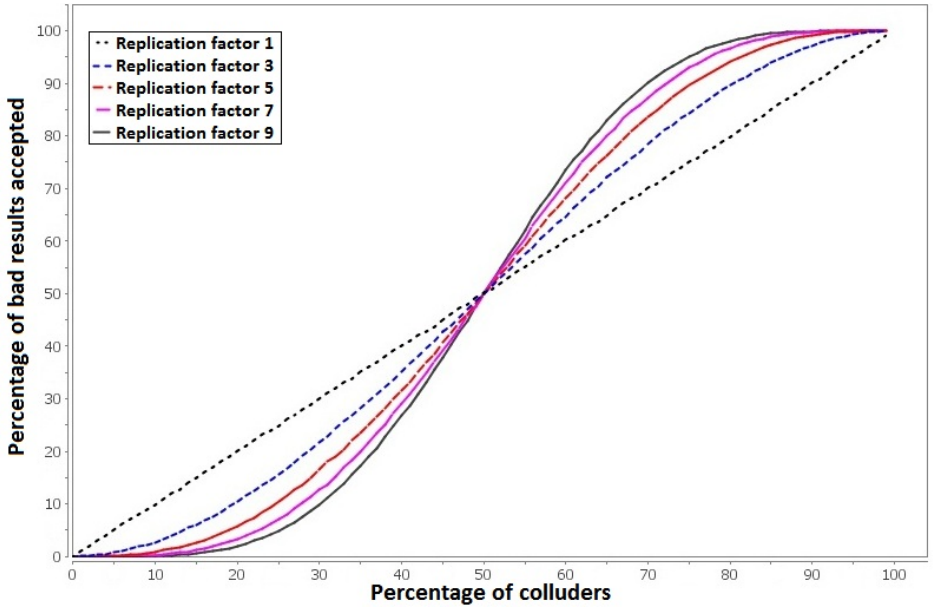

Fig. 6. Correlation between the percentage of bad results accepted and the percentage of colluders in the system for various replication factors.

way that the colluders introduce their bad results: it produces more points where collusion may happen and also may be detected; the size of each bad result is smaller, though. This happens because one task is replicated into more tasks than using standard partitioning; therefore there is a higher probability of redundant work being assigned to colluders; however they can only collude part of the task instead of the whole task as using standard partitioning. The graphic in Figure 7 depicts that overlapped partitioning is as good as standard partitioning, in a scenario where the colluders are fully able to identify the common part (in theory possible, but in practice harder to achieve as this may require global knowledge and impose heavier coordination and matching of information among the colluders) and collude it, while executing the non common part. This is the worst case scenario, therefore overlapped partitionings can improve the reliability of the results depending on how smart the colluders are.

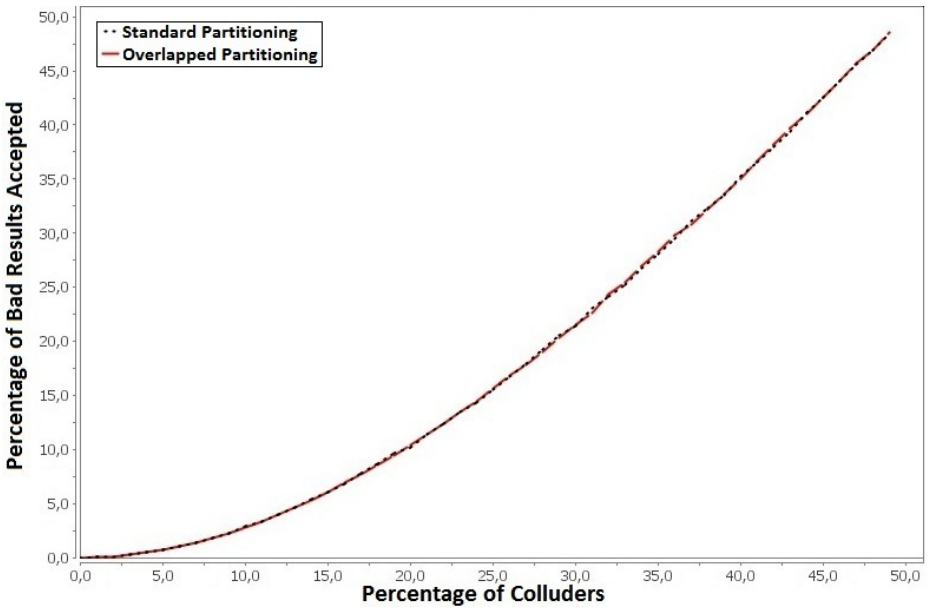

Fig. 7. Replication w/ Standard Partitioning Vs. Replication w/ Overlapped Partitioning, using replication factor 3.

6 Conclusions

In this paper, we proposed and analysed a number of checkpoint/restart mechanisms and replication schemes to improve fault-tolerance and reliability in cycle-sharing infrastructures such as those in Peer-to-peer networks.

Our generic checkpoint/restart mechanism based in virtual machine images that is able to checkpoint any application. It is yet at an early stage of development, we see no obstacles to it other than the checkpoint data size, therefore we are focused in reducing it. Our result oriented approach is only fit for some applications. We have successfully enabled POV-Ray to checkpoint and resume execution from its results automatically.

Our result verification schemes are very promising, we are trying to figure out how can we adapt them to the variable conditions of the system in order to produce a good compromise between performance and reliability.

References

1. D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.
2. D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.
3. T. E. Anderson, D. E. Culler, D. A. Patterson, , and the NOW team. A case for now (networks of workstations). *IEEE Micro*, 15:54–64, 1995.

4. L. B. Costa, L. Feitosa, E. Araujo, G. Mendes, R. Coelho, W. Cirne, and D. Fireman. Mygrid: A complete solution for running bag-of-tasks applications. In *In Proc. of the SBRC 2004, Salao de Ferramentas, 22nd Brazilian Symposium on Computer Networks, III Special Tools Session*, 2004.
5. distributed.net. Distributed.net: Node zero. In *http://distributed.net/*, 2010.
6. J. R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.
7. W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable grid computing. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 4–11, 2004.
8. J. C. e Alexandre Sztajnberg. Introdução de um mecanismo de checkpointing e migração em uma infra-estrutura para aplicações distribuídas. In *V Workshop de Sistemas Operacionais (WSO'2008)*, July 2008.
9. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.
10. GIMPS. Great internet mersenne prime search. In *http://mersenne.org*, 2010.
11. S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 640–651, New York, NY, USA, 2003. ACM.
12. S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. Technical Report arXiv:0901.0866, Jan 2009.
13. M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.
14. M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.
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 *In Proceedings of the IEEE Fourth International Conference on Peer-to-Peer Systems*, pages 227–236, 2004.
16. A. Maloney and A. Goscinski. A survey and review of the current state of rollback-recovery for cluster systems. *Concurr. Comput. : Pract. Exper.*, 21(12):1632–1666, 2009.
17. D. Molnar. The seti@home problem. In *ACM Crossroads: The ACM Student Magazine*, 2000.
18. J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.
19. T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.
20. L. Veiga, R. Rodrigues, and P. Ferreira. Gigi: An ocean of gridlets on a 'grid-for-the-masses´. In *IEEE International Symposium on Cluster Computing and the Grid - CCGrid 2007 (PMGC-Workshop on Programming Models for the Grid)*. IEEE Press, May 2007.
21. H. Zhong and J. Nieh. Crak: Linux checkpoint / restart as a kernel module. Technical Report CUCS-014-01, Department of Computer Science. Columbia University., November 2002.