Mechanisms for Reliable Execution of Tasks in Cycle-sharing Environments

Mecanismos para a Fiabilidade (Replicação, Tolerância a Faltas, Checkpointing) de Execução de Tarefas em Ambientes Cycle-sharing

João Paulino
INESC-ID/IST
Distributed Systems Group
Rua Alves Redol, 9, 1000-029, Lisboa, Portugal
joaopaulino@ist.utl.pt

ABSTRACT

The partitioning of a long running task into smaller tasks that are executed parallely 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 [16, 3, 8, 4] and was later transposed to public environments [2, 10]. Although they are based on the same principles, new challenges arise from the characteristics of public environments.

Clusters [16, 3] and Grids [8, 4, 11] have been very successful in accelerating computationally intensive tasks. The major difference between these is that 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, 10, 1, 5, 9, 13] 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 [17] 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 analyse the techniques that are mainly used to verify the correctness of the results; in Section 2.2 we review the main approaches to provide an application with checkpoint/restart capabilities;

2.1 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).

2.1.1 Replication

One of the most effective methods to identify bad results is through redundant execution and comparison between results. 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 a 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 impossible to detect a bad result. Another disadvantage of redundant execution is the overhead it generates, since every job is executed at the very least three times.

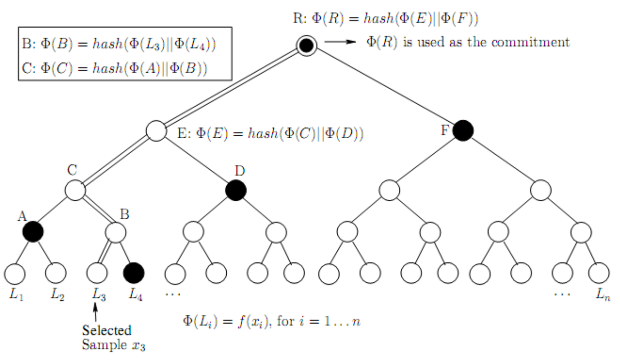
Most of the public computing projects use replication to verify their results, it is a high price they are willing to pay to ensure their results are reliable. Seti@Home [2] and Folding@Home [10] use redundant execution and voting quorums to verify their results.

Replication consumes at the very least three times more resources than the ones that are actually needed to perform the execution in order to produce more believable results. When there is no collusion, it is virtually capable of identifying all the bad results with 100% certainty.

2.1.2 Hash-trees

This technique is able to defeat cheating participants by forcing them to calculate a binary hash-tree from their results, and return it with them [6]. 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.

Figure 3 shows a hash-tree where the leafs are partitioned sequential results or the data 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 at random a small portion of the whole work (the selected sample) that corresponds to a leaf. Then this result is compared to the returned result and the hashes of the whole tree are checked.



R: $\Phi(R) = hash(\Phi(E)||\Phi(F))$
$\Phi(R)$ is used as the commitment
B: $\Phi(B) = hash(\Phi(L_3)||\Phi(L_4))$
C: $\Phi(C) = hash(\Phi(A)||\Phi(B))$
E: $\Phi(E) = hash(\Phi(C)||\Phi(D))$
F
C
D
A
B
L_1
L_2
L_3
L_4
$\cdots$
$\cdots$
L_n
Selected Sample x_3
$\Phi(L_i) = f(x_i)$, for $i = 1 \dots n$

Figure 1: Example of an 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.

Hash-trees make cheating unworthy. They have a relative low overhead: a small portion of the work has to be executed locally and the hash tree must be checked. However, they do not dissuade malicious participants that are willing to forge their results at any cost.

2.1.3 Quizzes

This technique consists in assigning jobs whose result is known by the submitter a priori. Therefore, these jobs can test the honesty of a participant. Cluster Computing On the Fly [13] 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 node executed the job. These quizzes are only useful when associated with a reputation system that manages the trust levels of the executing peers. 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. Embedded quizzes can be used with or without a reputation system. Though, their 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.

2.2 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 [14, 7].

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, 10] , Library-level [15, 12] and System-level [18].

2.2.1 Application-level Checkpoint/Restart Systems

These systems are built within the application code requiring a big programming effort. If the applications are not developed from scratch to support checkpointing mechanisms, it may be impossible to provide them with checkpoint/restart capabilities later. Since the programmer knows exactly what needs be safely stored to enable the application to restart in case of failure, application-level checkpoint/restart are usually more efficient. They achieve a better performance, and lower checkpoint data size. Applications may either checkpoint in time intervals, or constantly persist the important data. Since these systems do not use any operating system support, they are portable[1].

Thus, this approach has some drawbacks: it requires major modifications to application's source code (its implementation is not transparent 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 required the programmer to state what needs to be saved (e.g., through flagged/annotated code).

2.2.2 *Library-level Checkpoint/Restart Systems*

These consist in linking a library with the application, creating a layer between the application and the operating system that provides checkpoint/restart capabilities. The major advantage is that it is possible to create generic checkpointing mechanism that is able to checkpoint a vast range of applications without having to modify them, while maintaining portability.

Thus, the so far developed libraries require a few modifications to the applications source code, access to the source code may be a problem. Plus, the existing implementations are not able to checkpoint a vast range of applications, the major challenge is that these systems cannot access kernel's data structures (e.g., file descriptors), so this layer has to emulate operating system calls. This layer has no semantic knowledge of the application and checkpoints may be taken in the least appropriate moments generating considerable sized checkpoints. This layer may also be responsible for a slow down in the performance of the application.

2.2.3 *System-level Checkpoint/Restart Systems*

These systems are built as an extension of the operating system's kernel. They are more powerful, since they can access kernel's data structures (e.g., file descriptors). Checkpointing can consist in flushing all the process's data and control structures to stable storage (i.e., to a file on the local disk). Since these mechanisms are external to the application they do not require specific knowledge of the application, and they require none or minimal changes to the application, so they are transparent to the application.

These approach has the disadvantage of not being portable. The non-knowledge of the application may lead to least efficient checkpoint data when compared with checkpoint data generated by applications that checkpoint themselves (i.e., application-level). Plus, developing a kernel module that enables the checkpointing of any application is complex and the implementations so far are only able to checkpoint some applications.

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

3. ARCHITECTURE

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

3.1 Result Verification Mechanisms

In order to verify the results returned by the participants we propose replication with a number of flavours that enable us to take better advantage of redundant execution.

3.1.1 *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.

As an example of a rule, we can assign only the the required redundant work for the voting quorums to be able to accept it. 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 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 current resources.

3.1.2 *Replication using Overlapped Partitionings*

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 2 depicts the same work divided in in two different overlapped partitionings.

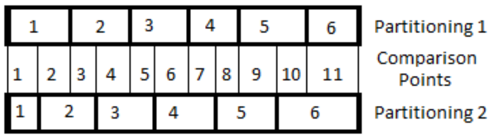


Figure 2: The same work divided differently creating an overlapped partitioning.

Overlapped partitioning could 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. If the overlapped parts are considered correct, the non redundantly executed portions of the same task are accepted as well. Figure 3 shows a relaxed overlapped partitioning.

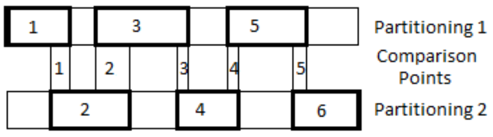


Figure 3: Overlapped tasks for relaxed replication.

3.1.3 Replication using Meshed Partitionings

Some applications can have their work divided in more than one dimension. Figure 4 depicts the partitioning of the work for a ray-tracer. This partitionings provide lots of points of comparison. This information feeds an algorithm that is able to choose correct results according to the reputation of a result, instead of using voting quorums.

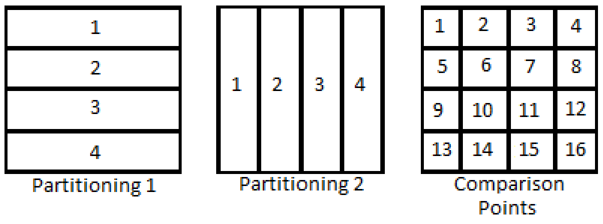


Figure 4: Meshed partitioning using replication factor 2.

The algorithm for calculating the reputation of a result is based on the comparison points result (i.e., equal or not equal). Since the majority of the participants is expected to be honest equal adds positive reputation and not equal adds negative reputation. Equal task results are accepted on the fly (if at least one of the tasks has a positive reputation), different task results are disambiguated according to the reputation of those results. We choose the result with higher reputation if its reputation is positive, otherwise the corresponding portion of execution is rescheduled. If the reputation is drawn, the portion of execution of the corresponding result must be re-executed for voting quorum like disambiguation.

Figure 5 depicts the comparison point results of a two-dimensional work partitioned in four independent tasks twice (replication factor 2) creating eight tasks (H1, H2, H3, H4, V1, V2, V3 and V4). Using the comparison point indexes of Figure 4: results 1, 2, 3, 5, 6, 7, 13,14 and 15 would be accepted on the fly for being equal; in position 9 the chosen result is the one returned in task V1 since it has reputation 2 against reputation -4 of task H3 (positions 10, 11, 4, 8 and 16 would be disambiguated in the same way); position 12 would require re-execution, since both tasks H3 and V4 have equal negative reputation -4.

This algorithm can enable the use of low, possibly even, replication factors. Requiring minimal portions of extra execution for disambiguation. Nonetheless, in a system where

	V1	V2	V3	V4
H1	1	1	1	0
H2	1	1	1	0
H3	0	0	0	0
H4	1	1	1	0

Figure 5: Meshed partitioning: results of the comparison points (1 means equal, 0 means not equal).

the majority of the participants is honest, the extra work is minimal and rare.

3.1.4 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, without generation nor identification issues. Figure 6 depicts the sampling of an image where a sample is a pixel.

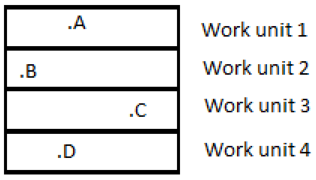


Figure 6: Sampling for an image.

3.1.5 Samplication

Replication and random sampling can be 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 in the submitter.

Nonetheless, these techniques can be combined in a more elegant manner that provides additional benefits without adding extra overhead.

The algorithm we propose is very simple, the next piece of pseudo code describes how it works:

```
 1.  Schedule redundant work, put the results in a bag;
 2.  IF(the bag is empty)
 3.    GOTO 1;
 4.  IF(all results in the bag are equal)
 5.    IF(random sample matches)
 6.      ACCEPT RESULT;
 7.    ELSE
 8.      remove all results from the bag;
 9.      GOTO 1;
10.  ELSE
11.    choose a sample within the mismatch area;
12.    compare with all results;
13.    remove results that mismatch the sample from the bag;
14.    GOTO 2;
```

This algorithm provides a number of desirable properties:

- It only discards wrong results: if at least one of the results of the redundant work is correct this approach ensures that the mentioned result is the chosen one, and that the honest participant will always receive credit for it. Whereas, using voting quorums, if the correct result is not within a majority it is discarded, the honest participant does not receive credit and might even be punished by the reputation mechanisms.

- It enables the identification of fault-prone participants and colluders: results that mismatch samples are wrong, if more than one results are equal and wrong it is very likely that they were returned by colluders;

- It enables the use of even replication factors (since it does not use voting quorums);

- The number of samples per task is low: if all results are correct, it only requires one sample per task; the maximum number of samples used when there is no rescheduling is R (R being the replication factor);

- Rescheduling only occurs if all the received results were wrong, which makes rescheduling the desirable option.

3.2 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/restart any application. Therefore, we propose two alternative approaches that will enable us to checkpoint/restart any application.

3.2.1 Checkpoint/Restart through a Virtual Machine

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.

3.2.2 Checkpoint/Restart through the Result Files

This technique will only fit some applications and demands the implementation of specific enabling mechanisms for those, the application remains unmodified though. 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 it 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 raytracer); rather than on a predefined time interval. This awareness of the application's semantics also enables the monitoring of the execution and the previewing of the results in the submitter.

. We have proposed two checkpoint/restart enabling techniques. Our checkpoint/restart through a virtual machine's running image technique enables us to checkpoint any application and resume it later, the overhead it incurs derives mainly from the size of the checkpoint data. Nevertheless, for some applications our checkpoint/restart through the result files 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. Sampling is able to test the received untrusted results against one result sample that is known to be correct. Samplication combines replication and sampling in an elegant manner without using voting quorums.

4. IMPLEMENTATION

Our implementation is developed in two different deployments: i) a simulator that enables us to test result verification approaches with large populations; and ii) a real deployment that proves that our result verification and the checkpoint/restart approaches are feasible.

4.1 Simulator

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; worksize 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 wrong results that were accepted by the system.

4.2 Real Deployment

The major concern in implementing the checkpoint/restart and result verification techniques that have been described in Section 3 was to keep the checkpoint/restart and result verification policies separated from the application specific adaptors.

For being able to support a new application, we have to develop application specific adaptors, which consists in specializing three classes: an Application Manager, a Gridlet and an Atomic Result.

4.2.1 Application Manager

The Application Manager is responsible for dividing a long running execution into several executable gridlets and reunite their results. For some applications it may also enable the user to preview the results as it receives them (e.g., an image being incrementally produced by a ray-tracer). Using our checkpoint/restart through the result files approach it is possible not only to preview the tasks that already completed their execution, but also the ones that are currently being executed.

It must extend the abstract class *ApplicationManager* and implement a constructor and three other methods. The next excerpt of code is the POV-Ray's application manager specialized class.

```
class PovRayManager
    extends ApplicationManager {

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

  int calculateWorksize() { (...) }

  Gridlet createGridlet(int offset, int worksize) { (...) }

  void submitResults(int offset, AtomicResult[] res) { (...) }

}
```

The constructor receives a string as argument, this string is the command that invokes the application (this is a simplification of the GINGER application invocation used for this work only).

For the generic application management to be able to partition any task it must have access to the total size of the long running task, this can only be calculated by the specific application adaptors. Therefore, it must implement the method *calculateWorksize* that retrieves the total size of the long running task, in terms of atoms of execution.

The *createGridlet* method receives the offset and the size of the task and returns a gridlet that matches the corresponding portion of execution. Gridlets are created with a size defined by generic application adaptors rather than the specific ones. This creation of gridlets on demand enables:

- the implementation of application independent partitioning policies, manipulating the offset and worksize parameters;
- the creation of samples to be locally executed, a sample is a gridlet whose work consists in an atom of execution;
- the creation of recovery gridlets, for both checkpoint/restart and result verification purposes;

The *submitResults* receives an offset and a variable number of ordered results in an array. This method reassembles the results, and since it receives the updated results during the execution it can display a preview of the already received results.

4.2.2 Gridlet

The gridlets are created on demand by invoking the *createGridlet* method on a specific application manager, gridlets are self-aware of their execution, and they perform it upon the invocation of the method *execute*. The following is an excerpt of the Pov Ray's gridlet class.

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

  AtomicResult[] execute(int offset, int worksize) { (...) }

}
```

The specialized 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. Implementing the Runnable interface requires the implementation of the *run* method, this method is implemented in the super class. The *run* method invokes the *execute* method manipulating its arguments, which enables the capture of partial results (making sequential invocations of the application). Those are used as checkpoint data in our checkpoint/restart through the result files based approach.

The *execute* method receives as argument the portion of execution to be executed (this is defined through the offset and worksize parameters), this is contained within the boundaries with which the gridlet was firstly created.

4.2.3 Atomic Result

The atomic result is just a container of result data (e.g., a pixel for image generation, a frame for video enhancing). The atomic result class must specialize a method that enables the comparison with another result. The following piece of code is an excerpt of the PovRay's Atomic Result.

```
class PovRayAtomicResult
    extends AtomicResult
    implements Serializable {

  boolean isEqual(Object obj) { (...) }

}
```

The *isEqual* must be able to compare atomic results. For result verification purposes, the comparison of results in replication schemes using standard partitionings can be done byte wise over raw data. However, this does not work for the other types of partitionings that we described in the architecture, nor for the comparison of samples. Therefore, the implementation of this simple method enables all the result verification techniques studied in our work, while keeping their policies transparent to the application specific adaptors.

Specializations of the *Result* class must implement the Serializable interface, for the Java RMI mechanisms being able to transmit these atomic results back to the submitter.

5. EVALUATION

This Section presents a highlight of our evaluation. Section 5.1 presents the evaluation of result verification strategies proposed earlier. Section 5.2 evaluates the proposed checkpoint/restart enabling techniques.

5.1 Result Verification Mechanisms

Result verification mechanisms introduces overhead, this overhead is the price these systems are willing to pay to ensure their results are reliable. In this section we analyse and evaluate the result verification techniques proposer earlier in this paper.

5.1.1 Replication using Overlapped Partitionings

Overlapped partitioning influences the 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.

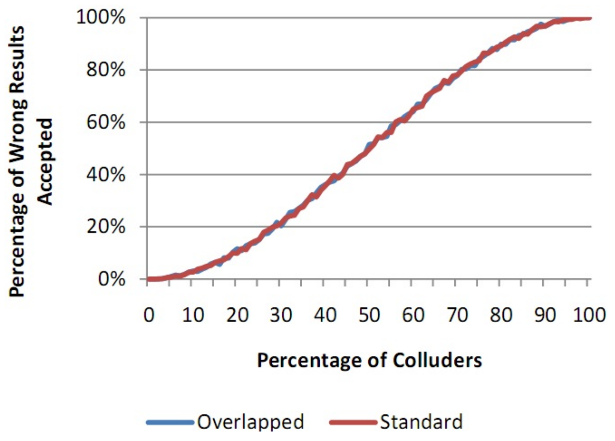


Figure 7: Replication w/ Standard Partitioning Vs. Replication w/ Overlapped Partitioning, using replication factor 3.

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.

5.1.2 Replication using Meshed Partitionings

The insight of partitioning the task in more than one dimensions provides lots of points of comparison, information that is used to calculate the reputation of a result. This reputation disambiguates mismatching results of the same task.

The graphic in Figure 8 depicts that the percentage of wrong results accepted is very low, a small number of results must be rescheduled for disambiguation, though. The work that has to be rescheduled is mostly composed by the portions where wrong results overlap. Therefore, those results cannot be accepted, rescheduling is the only solution. This technique proves to be very efficient as we are only using twice the base amount of work.

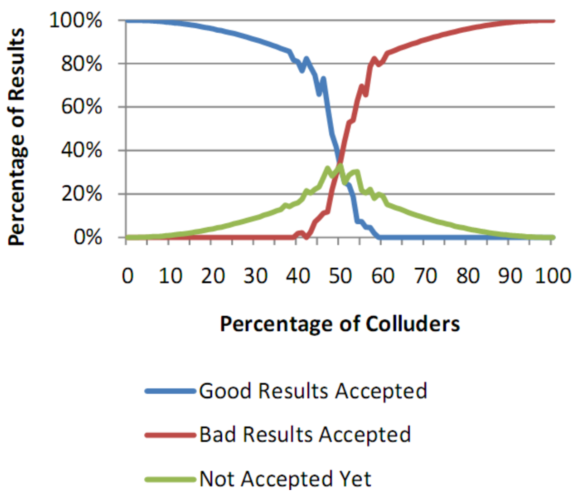


Figure 8: Replication using meshed partitionings: percentages of results using bi-dimensional before rescheduling, in a scenario where colluders return results 100% forged.

5.1.3 Samplication

Samplication is a technique that combines sampling and replication without using voting quorums. Plus, this technique works with even replication factors as well. It uses information from replication to decide where to choose samples, rather than selecting samples randomly. It chooses the samples within a replication mismatch area and discards the results that mismatch the chosen sample. If there is no mismatch in replication it uses random sampling.

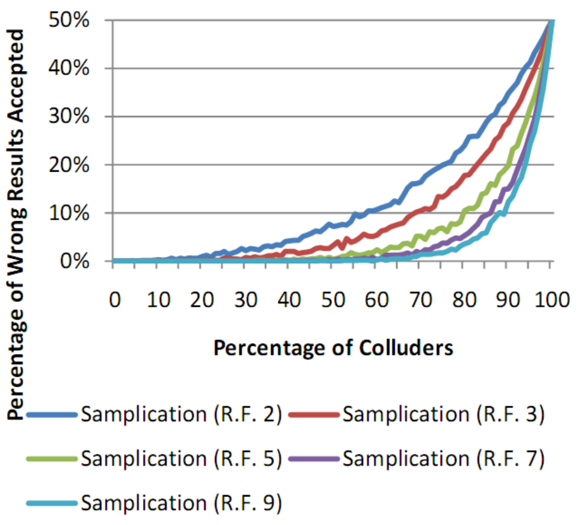


Figure 9: Result Verification - Samplication: percentage of wrong results accepted in a scenario where return results 50% corrupted.

As seen in Figure 9, this technique is quite effective, it keeps the percentage of wrong results accepted very low, even for medium groups of colluders. Improving the standalone replication effectiveness through comparison with local samples. It incurs local overhead (the execution of the samples).

5.2 Checkpoint/Restart Mechanisms

In this Section we evaluate our checkpoint/restart approaches having into consideration the overhead incurred by those techniques during fault-free execution.

5.2.1 *Checkpoint/Restart through a Virtual Machine*

The major issue of our checkpoint/restart through a virtual machine's running image is the size of the checkpoint data, because this has to be transmitted over the network. In order to reduce the size of the checkpoint data, we use differential disk images and compression.

			Data Size (KB)	
Base Image	Powered Off	disk image (.vdi)	2.651.169	2.768.998
	After Boot	disk image (differencial .vdi)	33	
		volatile state (.sav)	117.796	
Current Image A	Running Application A	disk image (differencial .vdi)	16.417	154.403
		volatile state (.sav)	137.986	
Current Image B	Running Application B	disk image (differencial .vdi)	23.585	209.597
		volatile state (.sav)	186.012	

Figure 10: Checkpoint data size using VirtualBox and Ubuntu Desktop 9.10.

The table in Figure 10 depicts the reduction of the sizes of the checkpoints using differential disk images. Checkpoint A reduces the size about 17 times, checkpoint B about 14 times. This difference is mainly justified by the different amounts of data written by the applications. Therefore, the checkpoint size will always depend on the application being checkpointed.

	Data Size (KB)	Compressed Data Size (KB)	Relation (%)
Current Image A	154.403	74.466	48,23
Current Image B	209.597	105.408	50,29

Figure 11: Checkpoint data size reduced using 7zip compression.

The table in Figure 11 shows the effect of using compression over the checkpoint data. Reductions are up to 50% of the original size using light compression techniques, the use of heavier compression techniques would be able to reduce the checkpoint size more effectively. However, it would increase the amount of time necessary to take a checkpoint.

Other techniques like differencing the volatile state can improve the checkpoint data size, their implementation is very complex, though. We believe 50MB to 100MB checkpoints are transmittable within an affordable number of checkpoints taken during execution.

5.2.2 *Checkpoint/Restart through the Result Files*

The major overhead of this approach is the sequential launch of the application. The checkpoint data is the result files, whose would have to be transmitted anyway.

The table in Figure 12 reflects the overhead incurred during fault-free execution of 4 different examples. We conclude that the number of checkpoints taken must be controlled having into account the duration/complexity of the task: taking 10 checkpoints in task 4 is almost negligible whereas taking 100 checkpoints in task 1 is an overkill.

	No Checkpoints	10 Checkpoints		100 Checkpoints	
	time (hh:mm:ss)	time (hh:mm:ss)	overhead (%)	time (hh:mm:ss)	overhead (%)
Task 1	00:00:17	00:00:22	29,41%	00:01:15	341,18%
Task 2	00:01:16	00:01:20	5,26%	00:02:08	68,42%
Task 3	00:05:21	00:05:29	2,49%	00:06:38	23,99%
Task 4	00:12:02	00:12:10	1,11%	00:13:01	8,17%

Figure 12: Checkpoint/restart through the result files: the overhead of sequential launch.

		No Checkpoints	10 Checkpoints		100 Checkpoints	
		time (hh:mm:ss)	time (hh:mm:ss)	overhead (%)	time (hh:mm:ss)	overhead (%)
Fault at 25%	Task 1	00:00:21	00:00:24	13,88%	00:01:16	256,47%
	Task 2	00:01:35	00:01:28	-7,37%	00:02:09	36,08%
	Task 3	00:06:41	00:06:02	-9,81%	00:06:42	0,18%
	Task 4	00:15:02	00:13:23	-11,02%	00:13:09	-12,60%
Fault at 50%	Task 1	00:00:26	00:00:24	-5,10%	00:01:16	197,06%
	Task 2	00:01:54	00:01:28	-22,81%	00:02:09	13,40%
	Task 3	00:08:01	00:06:02	-24,84%	00:06:42	-16,52%
	Task 4	00:18:03	00:13:23	-25,85%	00:13:09	-27,16%
Fault at 75%	Task 1	00:00:30	00:00:24	-18,66%	00:01:16	154,62%
	Task 2	00:02:13	00:01:28	-33,83%	00:02:09	-2,80%
	Task 3	00:09:22	00:06:02	-35,58%	00:06:42	-28,44%
	Task 4	00:21:03	00:13:23	-36,45%	00:13:09	-37,57%

Figure 13: Checkpoint/restart through the result files: overhead pay-off in faulty scenarios.

The table in Figure 13 shows the pay-off of checkpoint overhead in scenarios with faults occurring at different moments of a task's execution. As it show 10 checkpoints pays-off in all faulty scenarios, except for task 1 with a fault occurring at 25%. In some scenarios 100 checkpoints pays-off more than 10

The number of checkpoints incurs a constant overhead during fault free execution, though in faulty scenarios it can make it up. A compromise between the number of checkpoints and the probability of a fault occurring must be established to take the best advantage of checkpoint/restart mechanisms.

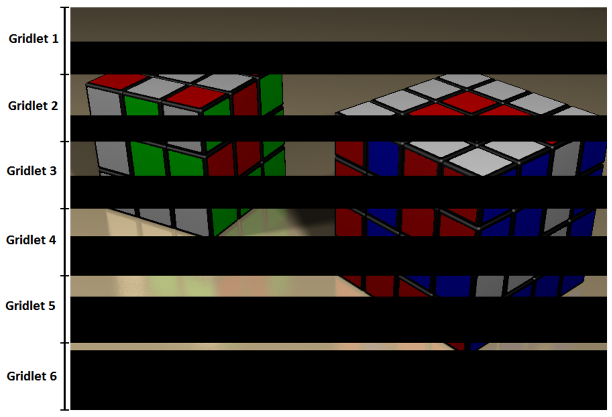


Figure 14: Checkpoint/restart through the result files: the previewing of a ray-tracer.

This technique enables the previewing of the results. This is a very pleasant functionality to users, as they can see that the work they submitted is being executed. despite it seems like an extra functionality, it can have a very positive impact on the acceptance of these systems by the public.

Figure 14 depicts the previewing of 6 independent POV-Ray tasks executing.

6. CONCLUSIONS

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

Result verification techniques ensure their results are correct with a given probability, this probability grows along with the overhead the technique incurs. Techniques that maximize the probability of results being correct or lower the overhead, are steps towards a more effective and efficient result verification.

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 (because it avoids the use voting quorums) and improves the reliability of the results using its stateless result reputation algorithm, it only fits applications that can have their work partitioned in more that one dimension, though. 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 (which also enables the use of even replication factors).

Checkpoint/restart enabling mechanism incur overhead during fault-free execution. This overhead is compensated when faults occur. The overhead incurred is the extra-time necessary to take checkpoints and the size of the checkpoints.

Our checkpoint/restart through a virtual machine has as major obstacle the checkpoint data size, using differential disk images and compression we were able to minimize the checkpoint size about 30 times, which is a transmittable amount of data. Our checkpoint through the result files approach is very efficient because checkpoint data is composed only by results that would have to be transmitted anyway, the time required to take checkpoints can be tightly controlled by the number of checkpoints per task, plus, this technique enables the previewing of the results.

7. 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] 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.

[7] 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.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[9] GIMPS. Great internet mersenne prime search. In *http://mersenne.org*, 2010.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] L. Veiga, R. Rodrigues, and P. Ferreira. Gigi: An ocean of gridlets on a 'grid-for-the-massest'. In *IEEE International Symposium on Cluster Computing and the Grid - CCGrid 2007 (PMGC-Workshop on Programming Models for the Grid)*. IEEE Press, May 2007.

[18] 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.