Strategies for Storage of Checkpointing Data using Non-dedicated Repositories on Grid Systems*

Raphael Y. de Camargo
Dept. of Computer Science
University of São Paulo, Brazil
rcamargo@ime.usp.br

Renato Cerqueira
Dept. of Computer Science
PUC-Rio, Brazil
rcerq@inf.puc-rio.br

Fabio Kon
Dept. of Computer Science
University of São Paulo, Brazil
kon@ime.usp.br

ABSTRACT

Dealing with the large amounts of data generated by long-running parallel applications is one of the most challenging aspects of Grid Computing. Periodic checkpoints might be taken to guarantee application progression, producing even more data. The classical approach is to employ high-throughput checkpoint servers connected to the computational nodes by high speed networks. In the case of Opportunistic Grid Computing, we do not want to be forced to rely on such dedicated hardware. Instead, we want to use the shared Grid nodes to store application data in a distributed fashion.

In this work, we evaluate several strategies to store checkpoints on distributed non-dedicated repositories. We consider the tradeoff among computational overhead, storage overhead, and degree of fault-tolerance of these strategies. We compare the use of replication, parity information, and information dispersal (IDA). We used InteGrade, an object-oriented Grid middleware, to implement the storage strategies and perform evaluation experiments.

Categories and Subject Descriptors

C.2.4 [**Computer-communication Networks**]: Distributed Systems—*distributed applications*; C.4 [**Performance of Systems**]: [fault tolerance]; E.4 [**Coding and Information Theory**]: [error control codes]

General Terms

Performance, Reliability

Keywords

Fault-tolerance, Distributed storage, Data coding, Checkpointing, Grid Computing

*This work is supported by a grant from CNPq, Brazil, process #55.2028/02-9.

MGC'05, November 28-December 2, 2005 Grenoble, France

1. INTRODUCTION

Executing computationally intensive parallel applications on dynamic heterogeneous environments, such as Computational Grids [3, 8, 4], is a daunting task. This is particularly true when using non-dedicated resources, as in the case of opportunistic computing [11]. Machines may fail, become unavailable, or change from idle to occupied unexpectedly, compromising the execution of applications.

Different from dedicated resources, whose MTBF (mean-time between failures) is typically in the order of weeks or even months [12], non-dedicated resources can become unavailable several times during a single day. Moreover, some machines can remain unavailable for more time than available.

A fault-tolerance mechanism, such as checkpoint-based rollback recovery [7], can be used to guarantee application execution progression in the presence of frequent failures. Moreover, the checkpointing mechanism can be used for process migration, allowing the implementation of efficient preemptive scheduling algorithms for parallel applications on the Grid.

The generated checkpoints need to be saved on a stable storage medium. The machine where the application is running cannot be considered a stable storage medium because it can become unavailable. The usual solution is to install checkpoint servers connected to the nodes by a high speed network. But since our focus is on an opportunistic computing environment, we do not want to be forced to rely on such dedicated hardware. The natural choice would be to use the Grid nodes as the storage medium for checkpoints.

We use InteGrade[1] [10], a multi-university effort to build a Grid middleware to leverage the computing power of idle shared workstations, as the platform for the implementation of the distributed storage system and experiments. The current InteGrade version has support for portable checkpointing of sequential, parameter sweeping, and BSP parallel applications [5].

A distributed storage system must ensure scalability and fault-tolerance for the storage, management, and recovery of application data. We expect to fulfill the scalability requirement by developing algorithms to distribute the data on non-dedicated repositories. We explore several techniques to provide fault-tolerance, such as data replication, Information Dispersal Algorithms (IDA) [17], and addition of parity information.

In this paper, we describe the implementation of a dis-

[1]http://integrade.incubadora.fapesp.br/

tributed repository system for InteGrade. The repository uses non-dedicated Grid nodes to store checkpoint data and supports several storage strategies. We used this repository implementation to perform experiments comparing several distributed storage strategies.

This article is organized as follows. In Section 2, we show related work in the area of distributed storage. In Section 3, we introduce several storage strategies that can be used, while, in Section 4, we present how the distributed infrastructure was implemented on InteGrade. In Section 5, we show experimental results and, finally, in Section 6, we present conclusions and future work.

2. RELATED WORK

Our research has some shared objectives with peer-to-peer content distribution technologies [2]. In peer-to-peer networks, the storage nodes are usually responsible for indexing, searching, and management of the network. This results in a design with good fault-tolerant and scalability properties, useful for systems containing large numbers of highly transient nodes, and where performance is not a major factor.

There are several works in the area of distributed storage, some which use coding techniques to improve fault-tolerance and security. Diferently from our work, they deal with dedicated storage servers.

Malluhi and Johnston [13] use an optimized version of the Information Dispersal Algorithm (IDA) [17] and 2D parity coding schemes, comparing their efficiency analytically. In our work we perform experimental evaluations and focus on non-dedicated repostories.

Alon *et al.* [1] developed a strategy for storage when up to half of the system is faulty. They use IDA and append hash information to provide both information retrieval and integrity. Garay *et al.* [9] alse address the problem of secure storage and retrieval of information using IDA and cryptographic techniques. Ellard *et al.* [6] also allow the secure distributed storage of data, but works only with immutable data objects. Our work differs from by focusing on the comparison of several storage techniques instead of implementing a single storage method. Besides, we use non-dedicated repositories.

There is also some research in the area of distributed storage of checkpoints from parallel applications. Sobe [18] analyzes the use of two different parity techniques to store checkpoints in distributed systems. The authors present a analytical study comparing the two models, but differently from our work, they do not perform experiments.

Pruyne and Livny [16] performed studies about the usage of multiple checkpoints servers to store checkpoints from parallel applications. But they only compared the usage of single and dual checkpoint servers and the servers were dedicated.

Plank *et al.* [15] propose the usage of diskless checkpointing. It consists of storing checkpointing data on system volatile memory, removing the overhead of stable storage. Similarly to our work, they evaluate the scenario where checkpoint data is stored on the processing nodes and one or more backup nodes. But the focus of their work is on comparing diskless with disk-based checkpointing. Also, the experiments were performed using parity information for fault-tolerance.

Distributed storage systems, which use dedicated repositories, target performance. Peer-to-peer storage systems, which use non-dedicated resources, focus on flexibility. Our work is positioned between these two areas, seeking to provide performance while maintaining the flexibility of using non-dedicated repositories. We expect to achieve these objectives by focusing on the particular requirements of Grid environments.

3. STORAGE STRATEGIES

An storage strategy needs to deal with scalability, computational cost, and fault-tolerance issues. We analyzed several data replication, error correction, and coding techniques with regard to the above criteria.

Another important issue to be taken into consideration relates to deciding into which nodes checkpoints should be stored. It is possible to distribute the data over the nodes executing the application, other Grid nodes, or both.

Finally, we are not dealing with data integrity and confidentiality. These features are being developed in parallel within the InteGrade project by another members of the group. But for typical scientific applications we want to execute on an opportunistic Grid, data integrity and confidentiality are not primary issues.

3.1 Data replication

Using data replication, we store full replicas of the generated checkpoints. If one of the replicas becomes unaccessible, we can use another. The advantage is that no extra coding is necessary, but the disadvantage is that it is necessary to transfer and store large amounts of data. For instance, to guarantee safety against a single failure it is necessary to save two copies of the checkpoint.

In our Grid scenario, transferring two times the checkpoint data would generate too much network traffic, so we decided to store a copy of the checkpoint locally and another remotely. Even though a failure in a machine running the application will make one of the checkpoints unaccessible, it will be possible to retrieve the other copy. Moreover, the other application processes will be able to use their local checkpoint copies. Consequently, this storage mode provides recovery as long as one of the two nodes containing a checkpoint replica is available.

3.2 Parity

In order to avoid the large storage requirements of data replication, an alternative would be to calculate the parity of the checkpoint data. Instead of storing two full replicas of the checkpoint, one stores only the checkpoint with some additional parity information. The amount of parity insformation determines the level of fault-tolerance achieved. We consider two of several approaches for evaluating checkpoint parity:

- **Parity over local checkpoints:** in this scheme, each node calculates the parity of its checkpoint locally. It first divides the generated checkpoint into m slices and calculates the parity over these slices. A checkpoint C of size n is divided into m slices U_k of size n/m, given by:

$$U_k = (u_0^k, u_0^k, ..., u_{n/m}^k), 0 \leq k < m$$

  The elements p_i, $0 \leq i < n/m$ of the parity informa-

tion vector P are calculated by:

$$p_i = (u_i^0 \oplus u_i^1 \oplus ... \oplus u_i^m), 0 \leq i < n/m,$$

where $\oplus$ represents the exclusive-or operation. The slices U_i and parity vector P from each process are then distributed for storage on other nodes.

- **Parity over global checkpoint:** in this case, the parity calculation is performed over the global checkpoint. An application composed of m processes generates m checkpoints C_k of size n, given by:

$$C_k = (u_0^k, u_0^k, ..., u_n^k), 0 \leq k < m$$

The elements p_i, $0 \leq i < n$ of the parity information vector P are calculated by:

$$p_i = (u_i^0 \oplus u_i^1 \oplus ... \oplus u_i^m), 0 \leq i < n$$

Each node keeps a copy of its local checkpoint and sends a copy for parity calculation. The parity information is then stored in an additional node.

Sobe [18] performed an analytical comparison of the two parity strategies for storing checkpointing data from parallel application. They show that parity over local checkpoints is normally faster, since the parity is calculated in parallel on each local node. The disadvantage is that it needs to create more network connections, so it is not scalable when distributing data over a large number of nodes.

Recovery when a machine executing a single application process fails is also faster using parity over local checkpoints. In this case, to reconstruct the missing checkpoint data and restart the application process, it is only necessary to fetch the slices relative the failed process. When using global parity, it is necessary to fetch data from all checkpoints to reconstruct the missing checkpoint.

The advantage of using parity, compared to other storage strategies, is that its evaluation is very efficient, requiring only simple exclusive-or operations. But the drawback is that failure of two nodes containing the checkpoint data will make the state unrecoverable.

3.3 Information dispersal algorithms

The classic information dispersal algorithm (IDA) [17] was developed by Rabin and generates a space optimal coding of data. Using IDA, one can code a vector U of size n, into $m + k$ encoded vectors of size n/m, with the property that one can regenerate U using only m encoded vectors. By using this encoding, one can achieve different levels of fault-tolerance by tuning the values of m and k. In practice, it is possible to tolerate k failures with an overhead of only $k/m * n$ elements.

This algorithm requires the computation of mathematical operations over a Galois field $GF(q)$, a finite field of q elements, where q is either prime or a power p^x of a prime number p. When using $q = p^x$, arithmetic operations over the field are carried by representing the numbers as polynomials of degree x and coefficients in $[0, p-1]$. Sums are calculated with XOR operations, while multiplications are carried out by multiplying the polynomials modulo an irreducible polynomial of degree x. In our case, we will use $p = 2$ and $x = 8$, representing a byte. To speedup calculations, we perform simple table look up for the multiplications.

The algorithm also requires the generation of $m + k$ linearly independent vectors α_i of size m. These vectors can be easily generated by choosing n distinct values a_i, $0 \leq i < n$ and setting $\alpha_i = (i, a_i, ..., a_i^{n-1})$, $0 \leq i < n$. These vectors are then organized as a matrix G defined as:

$$G = [\alpha_0^T, \alpha_1^T, \ldots, \alpha_{m+k}^T]$$

We now break a file F into n/m information words U_i of size m and generate n/m code words V of size $m+k$ where:

$$V_i = U_i \times G$$

The $m + k$ encoded vectors E_i, $0 \leq i < m + k$ are given by:

$$E_i = (V_0[i], V_1[i], \ldots, V_{n/m}[i])$$

To recover the original information words U_i, we need to recover k of the encoded $m+k$ slices. We then construct code words V_j', which are equivalent to the original code words V_i, but containing only the components of the k recovered slices. Similarly, we construct a matrix G', containing only elements relative to the recovered slices. We now recover U_i multiplying the encoded words V_j' with the inverse of G':

$$U_i = V_i' \times G'^{-1},$$

The main drawback of this approach is that coding requires $O((m + k) * n * m)$ steps and decoding $O(n * m * m)$ steps, in addition to the inversion of an m x m matrix.

Malluhi and Johnston [13] proposed an algorithm that improves coding computation complexity to $O(n * m * k)$ and also improve decoding. They showed that we can diagonalize the first m columns G and still have a valid algorithm. Consequently, the first m fields of code words V_i involves simple data copying. Coding is only necessary for the last k fields. This approach reduces encoding complexity considerably.

The biggest advantage of the IDA algorithm is that it provides the desired degree of fault-tolerance without much space overhead. For an application composed of 10 nodes, if we set m as 10, it is possible to tolerate failure of one node with a 10% space overhead, two failures with 20% overhead and so on. The disadvantage of this approach is the computational complexity of coding the data.

3.4 Error-correcting codes

Error correcting codes allow the location and correction of errors in a stream of data. One of the most common codes, Reed-Solomon [14], allows the correction of $k/2$ errors, where k is the number of extra bits used. Information dispersal algorithms can correct k errors using the same additional k bits. The difference occurs because when using Information Dispersal Algorithms we have to know in advance the presence of errors and their location. Reed-Solomon codes can detect and correct errors in arbitrary places. Since in our environment we are always able to determine where data loss occur, IDA provides a better trade-off than error-correction codes.

4. CHECKPOINTING-BASED ROLLBACK RECOVERY ON INTEGRADE

Figure 1 shows the main modules present on a InteGrade cluster. InteGrade clusters are typically composed of a few dozen machines located in a single local network. A collection of hierarchically organized InteGrade clusters form an InteGrade Grid. The current stable version of InteGrade supports only single clusters. A single cluster is adequate for

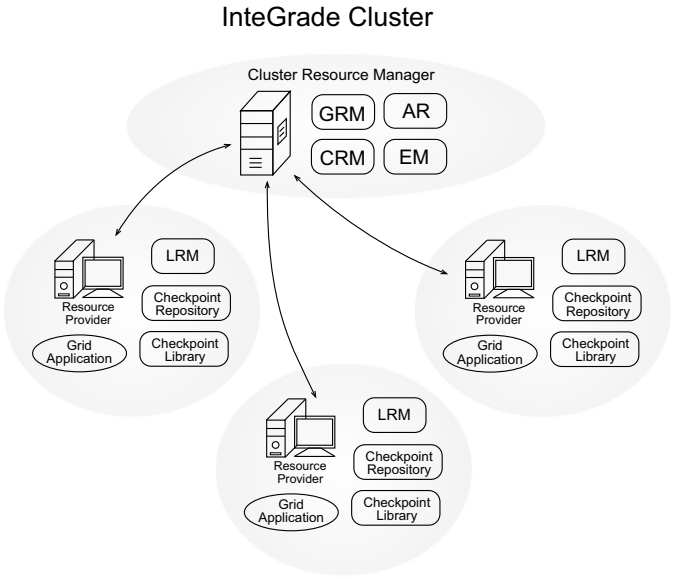

Figure 1: InteGrade's Intra-Cluster Architecture.

performing experiments comparing the overhead of different checkpointing strategies.

The Global Resource Manager (GRM) is responsible for its cluster resource management, including the scheduling of application execution requests. Local Resource Managers (LRM) manage the resources of a single machine, monitoring and controlling the execution of applications on that machine. The Application Repository (AR) stores application meta-data and binaries. Finally, the Application Submission and Control Tool (ASCT) permits users to submit applications for execution, monitor executions, and view execution results through a graphical interface.

4.1 Checkpointing modules

The checkpointing-based rollback recovery architecture on InteGrade is composed of the checkpointing library (ckpLib), the Execution Manager (EM), the distributed checkpoint repositories (ckpRep), and the Checkpoint Repository Manager (CRM).

To store a checkpoint, a checkpointing library queries the CRM about available checkpoint repositories and transfers the checkpoint data to the repositories. The number of requested repositories and the data transfered to each repository depend on the chosen storage strategy. Checkpoint repositories are instantiated on the same machines that host Grid applications.

The Execution Manager (EM) maintains a list of applications executing on the cluster. The LRM and GRM notify the Execution Manager when applications start and finish their execution. The EM coordinates the reinitialization process when an application fails, which can happen either because one of its nodes is unreachable or because one of its processes died. The reinitialization process also allows recovery from failures during checkpoint generation and application reinitialization.

The restarted processes query the Checkpoint Repository Manager (CRM) for the checkpoint repositories containing the last checkpoint. The CRM mantains meta-data containing information about stored checkpoints, such as the encoding used and the location of checkpoint slices.

The use of a centralized EM and CRM is justified by the ease of implementation and development of simpler algorithms that require less message exchanges. Although the current versions of these components require a dedicated machine, future versions will have fault-tolerance support through replication and/or logging, and will run on non-dedicated machines.

A centralized CRM provides other important benefit. The CRM has a central view of the available repositories, the amount of storage space available in each of them, and the number of processes simultaneously using the repositories. This allows the CRM to employ scheduling and data distribution strategies by determining which repositories should be assigned to each application and when checkpoints should be stored.

4.2 Checkpointing Library

To provide portable checkpointing, we developed an application-level checkpointing mechanism for regular, parameter sweeping, and BSP parallel applications [5]. The portability allows a stored state to be recovered on a machine with a different architecture. We developed a pre-compiler that instruments C/C++ application code in order to save its state periodically. The current version of the pre-compiler works with a subset of C++.

The storage strategy to be used is passed through a configuration file to the checkpointing library. The checkpointing library is implemented in C++, with a separate class for each kind of storage strategy. The strategies include storing the file locally or remotely, using the techniques described in Section 3.

Coding and transfer of checkpoints is performed by a separate thread of the application. During checkpoint generation, the data to be coded and stored is copied to a separate buffer, allowing the application to continue its execution while data is coded and stored. This greatly reduces the checkpointing overhead, since the application does not need remain stopped while the checkpoint is stored.

5. EXPERIMENTS

We performed the experiments on a laboratory containing 11 AthlonXP 1700+ with 1GB of RAM, connected by a switched 100Mbps Fast Ethernet network. During the day the machines are used by students, so the experiments were performed during the night period. The objective is to measure the overhead of checkpoint storage during normal operation, that is, without machines becoming unavaliable.

For the experiments, we used a matrix multiplication application with matrices of different sizes and composed of `long double` elements. Matrix multiplication calculations are used in several engineering and scientific applications.

We evaluated the time necessary to encode and decode data and the overhead of using different storage strategies.

5.1 Data encoding and decoding

We first measured the time spent to encode and decode a checkpoint using IDA and local parity. We used different data sizes and number of slices for the comparison. In the graphs, $IDA(m, k)$, represents the IDA algorithm using m and k as described in Section 3.3. Figure 2 shows the results we obtained.

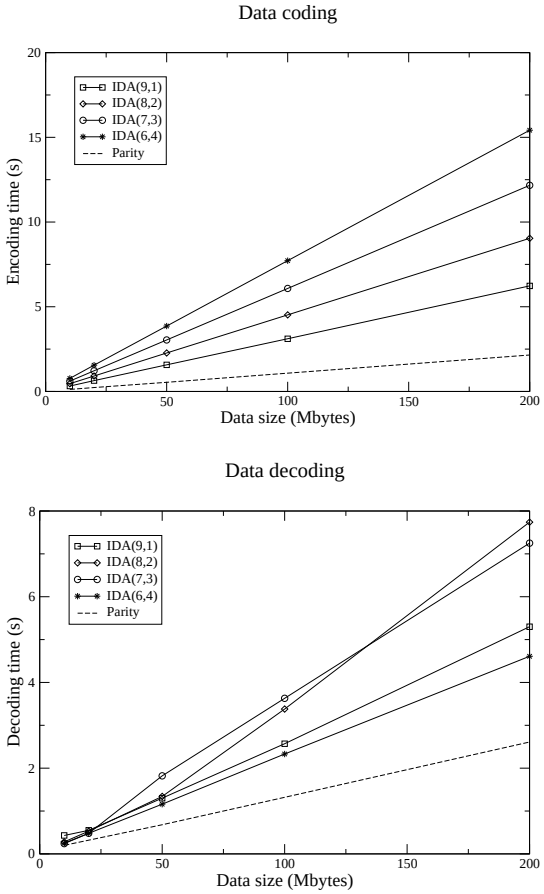

Figure 2: Time to code and decode a file.

As we can see, local parity calculation is faster than IDA on all scenarios, as expected. The most interesting result, however, is that coding with IDA is not as expensive as expected. Encoding 100Mbytes of data requires only a few seconds, with the time spent on encoding increasing linearly with the number of extra slices (k), and data size. The same is true for decoding. Recovering the data should not take more than a few seconds.

The results of this experiments are very satisfactory. With further optimizations in vector multiplications we can achieve even better results. In the future, our fault-tolerance mechanism switch between different storage strategies, such as local parity, IDA and replication, depending on fault-tolerance requirements, storage space and network bandwidth.

5.2 Execution overhead

We also evaluated the overhead incurred by checkpointing, coding, and distributed storage over parallel application executions. The objective was to compare the overhead for several of the storage strategies described in Section 3. We evaluated the following scenarios:

1. *No storage:* checkpoints are generated but not stored;
2. *Centralized repository:* checkpoints are stored in a centralized repository;
3. *Replication:* one copy of the checkpoint is stored locally and another in a remote repository.

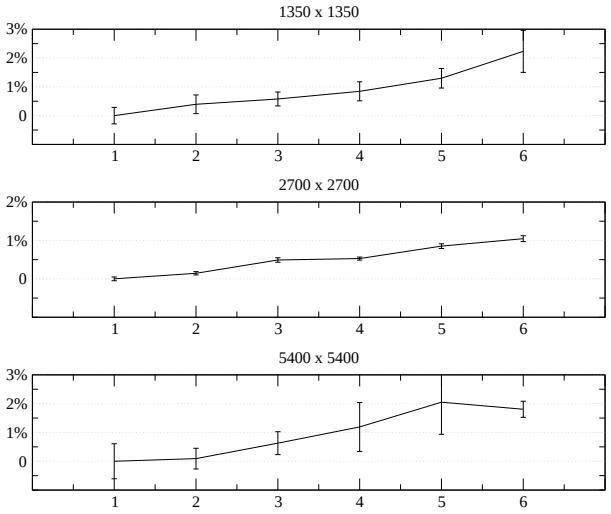

Figure 3: Checkpointing storage overhead for the matrix multiplication application.

4. *Parity over local checkpoints:* the checkpoint is broken into 10 slices, with one of them containing parity information. The slices are then stored in distributed repositories;
5. *IDA(m=9,k=1):* checkpoint is coded into 10 slices, from which 9 are sufficient to recover from a failure. The slices are then stored in distributed repositories;
6. *IDA(m=8,k=2):* checkpoint is coded into 10 slices, from which 8 are sufficient to recover from a failure. The slices are then stored in distributed repositories.

When using replication, checkpoints stored remotely are distributed through the 9 nodes executing the application. For scenarios 4, 5, and 6, which generate 10 slices, we used an additional node to store the remaining slice. We decided to store the checkpoints in the machines executing application processes because this allowed us to evaluate the impact of checkpoint storage in application execution time.

Table 1: Execution parameters.

matrix size	t_{total}	t_{seg}	n_{ckp}	$size_{local}$	$size_{global}$
1350x1350	908.5s	54.8s	17	7.3MB	65.6MB
2700x2700	2117.0s	303.1s	7	29.2MB	262.4MB
5400x5400	4281.6s	616.1s	7	116.6MB	1047.7MB

Table 1 contains the execution parameters of the experiment. We used 9 nodes to perform the matrix multiplication and 3 matrix sizes: 1350x1350, 2700x2700, and 5400x5400. To perform the benchmark, we divided the total execution time (t_{total}) in segments (t_{seg}) bounded by the checkpoint generation times. n_{ckp} represents the number of generated checkpoints, which are separated by a mean interval of t_{seg}. We represent the size of local and global checkpoints by $size_{local}$ and $size_{global}$ respectively.

In Figure 3, we show the overhead of storing checkpoints in comparison to the case where the application generates

but does not store checkpoints. The x-axis contains the 6 storage scenarios presented on this section. The y-axis has the normalized execution time.

The results show that using IDA causes the highest overhead. This was expected since it is necessary to perform data encoding. But the extra overhead is almost always below 2%, a very small overhead, especially when considering the large checkpoint sizes. Although for 5400x5400 matrices the checkpoint interval was 10 minutes, we could reduce this value to 5 minutes or less and still get a reasonable overhead.

Using parity, replication, or centralized storage it is possible to get smaller overheads, but with a lower degree of fault-tolerance. But IDA seems to provide a better trade-off between speed, resource usage, and level of fault-tolerance. Moreover, we could manipulate the IDA parameters m and k to match the desired degree of fault-tolerance.

6. CONCLUSIONS

In this work, we described the implementation of a distributed storage system for checkpoints and output files of applications for the InteGrade middleware. We analyzed several storage strategies and compared their overhead over application execution, the recovery time, and the computational time used in coding.

We showed that the overhead for global checkpoints of over 1GB is still small when using typical checkpoint intervals of a few minutes. More importantly, the time spent encoding data with IDA is not high for data segments of a few hundred Megabytes. IDA allows achieving different degrees of fault-tolerance with optimal space overhead, which will contribute to minimize the amount of network traffic.

We are now working on ways to deal with applications that generate even larger amounts of data, in the order of tenths of Gigabytes. Under these conditions, the encoding time and network usage grow very large, possibly implying that some sort of data transfer scheduling or positioning technique must be employed.

Our next step is to explore the usage of multiple InteGrade clusters. When there are multiple clusters, some interesting options become available, such as storing checkpoint data locally versus scattering IDA slices throughout the Grid. This can make checkpoint recovery reliable even in the presence cluster disconnections.

7. REFERENCES

[1] Alon, N., Kaplan, H., Krivelevich, M., Malkhi, D., and Stern, J. P. Scalable secure storage when half the system is faulty. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming* (London, UK, 2000), Springer-Verlag, pp. 576–587.

[2] Androutsellis-Theotokis, S., and Spinellis, D. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys 36*, 4 (2004), 335–371.

[3] Berman, F., Fox, G., and Hey, T. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, 2003.

[4] de Camargo, R. Y., Goldchleger, A., Carneiro, M., and Kon, F. The Grid architectural pattern: Leveraging distributed processing capabilities. In *Pattern Languages of Program Design 5* (2005), Addison-Wesley Publishing Company. Accepted.

[5] de Camargo, R. Y., Kon, F., and Goldman, A. Portable checkpointing and communication for BSP applications on dynamic heterogeneous Grid environments. In *SBAC-PAD'05: The 17th International Symposium on Computer Architecture and High Performance Computing* (Rio de Janeiro, Brazil, October 2005).

[6] Ellard, D., and Megquier, J. DISP: Practical, efficient, secure and fault-tolerant distributed data storage. *IEEE Transactions on Storage 1*, 1 (2005), 71–94.

[7] Elnozahy, M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (May 2002), 375–408.

[8] Foster, I., and Kesselman, C. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[9] Garay, J. A., Gennaro, R., Jutla, C. S., and Rabin, T. Secure distributed storage and retrieval. In *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms* (London, UK, 1997), Springer-Verlag, pp. 275–289.

[10] Goldchleger, A., Kon, F., Goldman, A., Finger, M., and Bezerra, G. C. InteGrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience 16* (March 2004), 449–459.

[11] Litzkow, M., Livny, M., and Mutka, M. Condor - A hunter of idle workstations. In *ICDCS '88: Proceedings of the 8th Int. Conference of Distributed Computing Systems* (June 1988), pp. 104–111.

[12] Long, D., Muir, A., and Golding, R. A longitudinal survey of internet host reliability. In *SRDS '95: Proceedings of the 14TH Symposium on Reliable Distributed Systems* (Washington, DC, USA, 1995), IEEE Computer Society, p. 2.

[13] Malluhi, Q. M., and Johnston, W. E. Coding for high availability of a distributed-parallel storage system. *IEEE Transactions Parallel Distributed Systems 9*, 12 (1998), 1237–1252.

[14] Plank, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software, Practice and Experience 27*, 9 (1997), 995–1012.

[15] Plank, J. S., Li, K., and Puening, M. A. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems 9*, 10 (1998), 972–986.

[16] Pruyne, J., and Livny, M. Managing checkpoints for parallel programs. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing* (London, UK, 1996), Springer-Verlag, pp. 140–154.

[17] Rabin, M. O. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36*, 2 (1989), 335–348.

[18] Sobe, P. Stable checkpointing in distributed systems without shared disks. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 214.2.