Using two-level stable storage for efficient checkpointing

L.M. Silva
J.G. Silva

Indexing terms: Checkpointing, Fault tolerance, Parallel computing systems

Abstract: Checkpointing and rollback recovery is a very effective technique to tolerate the occurrence of failures. Usually, checkpoint data is saved on disk, however, in some situations the time to write the data to disk can represent a considerable performance overhead. Alternative solutions would make use of main memory to maintain the checkpoint data. The paper starts by presenting two main memory checkpointing schemes: neighbour based and parity checkpointing. Both schemes have been implemented and evaluated in a commercial parallel machine. The results show that neighbour based checkpointing presents a very low performance overhead and assures a fast recovery for partial failures. However, it is not able to tolerate multiple and total failures of the system. To solve this shortcoming the authors propose a two-level stable storage integrating the use of neighbour based with disk based checkpointing. This approach combines the advantages of the two schemes: the efficiency of diskless checkpointing with the high reliability of disk based checkpointing.

1 Introduction

The mean time between failures (MTBF) of a parallel machine is considerably lower than a normal workstation, which increases the probability of failures and preventive shutdowns occurring during the execution of a parallel program. Some fault tolerance support is thereby required. Checkpointing allows long-running applications to save their state at regular intervals so that they may be restarted after interruptions without unduly retarding their progress. It is a feasible technique to tolerate transient failures and to avoid total loss of work. Each checkpoint is saved in some medium that is designated by stable storage [1]. By definition, the stable storage should be resilient to hardware crashes and software failures and should be immune to the phenomenon of memory decay. The write operations in stable storage should be atomic to the occurrence of failures. This means that every write operation is made completely or not at all; partial writes are not allowed to occur.

Usually, stable storage is implemented on disk. The main advantages of this approach are the simplicity, the increased level of reliability and the portability. To tolerate system crashes two checkpoint files have to be maintained: the last established file and the working checkpoint file. If there is a failure while saving the new checkpoint, there is always the chance to recover using the old established checkpoint file. There are several ways to implement stable storage on disk [1–4].

Disk based stable storage is the approach generally used. However, in some applications that need to be checkpointed very frequently the use of the disk may result in a serious performance bottleneck. In fact, several experimental studies [5–8] have shown that the main source of performance overhead is the time used in writing the checkpoints to disk.

For this reason, some researchers have developed alternative solutions for stable storage based on the use of RAM. The schemes [9–11] use special memory boards or additional hardware mechanisms to prevent erroneous accesses to stable storage. These RAM based stable storage schemes provide much faster access than those schemes that made use of the disk. Unfortunately, they require changes in hardware, undermining their portability across commercial computing systems. An alternative solution is to use the available memory from other processors to save the checkpoint data. This approach would not require additional hardware and can be implemented in any parallel machine provided there is enough spare memory among the different processors. It is not as reliable as disk based stable storage since it cannot tolerate the occurrence of a global failure of the machine. Its main use would be to tolerate single processor failures. At first sight, this approach for stable storage is faster than using the disk.

The goal of our study was to evaluate the feasibility of this approach for implementing stable storage in parallel machines and to compare the latency and the access bandwidth with a disk based stable storage approach. Finally, we integrate both stable storage approaches (diskless and disk based) and achieve a mixed solution that provides the best of these approaches: the efficiency of main memory checkpointing and the reliability of disk based stable storage. All the schemes described in this paper assume a crash failure model. Byzantine failures are not covered by these techniques.

IEE Proceedings online no. 19982440

Paper first received 20th July and in revised form 3rd November 1998

The authors are with Departamento Engenharia Informática, Universidade de Coimbra – POLO II, 3030 Coimbra, Portugal

2 Neighbour based checkpointing

A technique that avoids checkpoint writing to disk is to use the main memory of neighbour processors. Processors of the network are organised in a virtual ring. Each processor saves its checkpoint into its physical memory (*snapshot area*) and into the neighbour processor that follows on the ring. The degree of replication is only one ($k = 1$), thus the scheme can tolerate only single failures. In practice it is able to tolerate more than one failure provided the failures do not occur in adjacent processors on the virtual ring.

This scheme is not robust against failures that occur during the checkpointing protocol. To tolerate these failures each processor has to allocate two checkpoint areas in its physical memory: one to keep its own checkpoint and another to maintain the checkpoint of its preceding neighbour. The first step is to save the application into the local *snapshot area* of each processor. Then, it sends the checkpoint to the next processor on the ring. During the first step the application process is blocked, while the second step can be done concurrently with the computation. At the end of each checkpoint operation the system swaps the identity of the memory areas. The extra memory space that is required by neighbour based checkpointing can be considerable since it represents twice the size of the application's state.

This neighbour based checkpointing scheme should not be used alone, since it is not able to recover from total failures of the system. In our opinion it would be more interesting to integrate this approach with a disk based checkpointing scheme. Thus, from time to time the system should take a global checkpoint to disk (we call these 'hard' checkpoints), and in between, the application can be checkpointed in a distributed way to processor memory (these are called 'soft' checkpoints). Assuming this hybrid approach it is possible to checkpoint the application more often, and the application is able to tolerate single failures with a minor overhead and total failures with a higher recovery latency. If there is a failure during the neighbour based checkpointing protocol and it is not possible to recover from the 'soft' checkpoint then the application can be restarted from the previous 'hard' checkpoint, that is kept on disk.

We have implemented this simple neighbour based scheme and a parity based checkpointing that will be described in the next Section.

3 Parity based checkpointing

Another possible way to implement diskless checkpointing is to use a parity based approach. This was originally proposed in [12] and evolves from the use of parity schemes in the development of reliable disk arrays [13]. In our case, it is not used to provide disk reliability, rather, it is used as a compressed way to save distributed checkpoints in the main memory of the processors. The basic idea is to avoid disk writing and to maintain enough redundant information about the checkpoint data to enable it to tolerate a single processor failure. As a result, the application should be able to checkpoint far more frequently than when checkpoints are saved on disk.

To tolerate one single failure we should use a $(N + 1)$ parity technique. One processor in the network, the parity processor (PP), keeps a *parity checkpoint* of each global distributed checkpoint that is taken by the application. Each of the other processors saves its checkpoint into a local *snapshot area*. The checkpoint size of processor P_i is S_i. This means that every processor should have an amount of unused memory at least of the same size as the local checkpoint.

After this local operation, all the checkpoint contents are XORed and saved in the *parity checkpoint*. The size of the *parity checkpoint* is calculated as:

$$S_{parity\_chkp} = \max(S_i), \quad i = 0, \ldots, N-1$$

The *parity checkpoint* is computed using the XOR operator. Let us assume that b_{ij} corresponds to the jth byte of P_i's checkpoint. If j is higher than S_i (but lower than $S_{\text{parity\_chkp}}$) then it is set to 0. Then each byte of the *parity checkpoint* (B) is computed in the following way:

$$B_j = b_{1j} \oplus b_{2j} \oplus \cdots \oplus b_{nj}; \quad 1 \leq j \leq S_{parity\_chkp}$$

This checkpoint is then saved on that parity processor, while every other processor maintains a copy of its own checkpoint. If a processor P_i fails, the application can be recovered from the previous checkpoint. All the nonfailed processors restore their state from their local checkpoints, while the checkpoint of P_i can be retrieved from all the others and the *parity checkpoint*, in the following way:

$$b_{ij} = b_{1j} \oplus \cdots \oplus b_{i-1j} \oplus b_{i+1j} \oplus \cdots \oplus b_{nj} \oplus B_j; \; 1 \leq j \leq S_i$$

If the parity processor fails then it can restore its state from the backup copy (kept on disk or in main memory) or by recalculating the parity checkpoint from scratch. We have used a basic scheme with one checkpoint per processor and one parity checkpoint. Although this scheme does not assure checkpoint atomicity it requires the minimum amount of extra memory.

4 The Parix CHK-LIB

Previous parity and neighbour based checkpointing schemes were implemented in a checkpointing library, `CHK-LIB`. `CHK-LIB` is a system library that runs on top of the Parix Operating System [14]. It works primarily as a communication library and provides support for checkpointing. Any user that is not interested in the fault tolerance facilities can use `CHK-LIB` as a normal communication library instead of using the Parix system interface. The programming interface of `CHK-LIB` was inspired by the MPI standard [15] to facilitate the porting of existing MPI programs to Parix. However, it was not a full implementation of MPI: only a small subset of the numerous MPI routines can be found in `CHK-LIB`.

The library implements several checkpointing and message logging mechanisms. It was not meant to be a commercial or production tool; rather it was developed to provide support for our study into checkpointing in parallel systems available in our department (i.e. Parsytec machines).

The use of parity and neighbour based schemes requires a semitransparent approach having the programming interface presented in Fig. 1. The `CHK_Pack_chkp()` routine is used to specify the critical data of the application. The checkpoint routine, `CHK_Checkpoint()`, saves the relevant application data, that is, those variables and data structures indicated by the programmer through use of the previous routine. The

placement of checkpoints is under control of the user: he/she can make use of the global synchronisation points already existing in the application. It is the programmer's responsibility to place the checkpoint routines at points of the application that correspond to a consistent global state. Finally, the `CHK_Restart()` routine is used at the beginning of the application: if it is a restart from a previous checkpoint the program can skip the initialisation part, since the library will restore the values of the critical data structure from the last checkpoint.

```
int CHK_Pack_chkp (void *ptr, int size);
int CHK_Restart (void);
int CHK_Checkpoint (void);
```

Fig. 1 *Fault tolerant primitives of the CHK-LIB*

5 Implementation results

In our experiments, we used a Xplorer Parsytec machine with eight transputers (T805). Each processor had 4 Mbytes of main memory. All the processors can read and write directly to the file system of the host machine, a Sun Sparc 2 Workstation. This I/O system may introduce a bottleneck during the checkpoint operation, but each processor was able to write into a different file without requiring collective synchronisation.

5.1 Applications

To evaluate the checkpointing schemes we used the following application benchmarks:

- **ISING**: this program simulates the behaviour of spin-glasses.
- **SOR**: successive overrelaxation is an iterative method to solve Laplace's equation on a regular grid.
- **ASP**: solves the all-pairs shortest paths problem.
- **GAUSS**: solves a system of linear equations using the method of Gauss elimination.
- **NBODY**: this program simulates the evolution of a system of bodies under the influence of gravitational forces.

5.2 Parity versus neighbour based checkpointing

In this Section we compare these two checkpointing approaches and evaluate their performance overhead. Neighbour based checkpointing is termed **NBC**, while parity checkpointing approach is termed **PBC**.

The NBC technique, with a single degree of replication ($k = 1$), is able to tolerate single processor failures. In some cases it can tolerate more than one failure provided they occur in nonadjacent processors of the virtual ring. On the other hand, the PBC approach is able to tolerate only single processor failures. In order to tolerate total or multiple failures PBC or NBC schemes should be integrated with a disk based checkpointing mechanism. This two-level stable storage approach is described in the next subsection.

If we compare the two approaches, we can say that PBC always presents a lower memory overhead than NBC. However, it remains to be seen what the performance overhead is of both approaches. Next, we present a quantitative comparison between the NBC and PBC techniques. Five applications have been used and the overhead per checkpoint is presented in Table 1.

Table 1: Overhead per checkpoint in seconds (NBC versus PBC)

Application	Size of Chkp (Kbytes)	Overhead per Chkp NBC	Overhead per Chkp PBC	PBC/NBC
SOR 256×256	540	0.123	2.923	23.7
SOR 512×512	2104	0.832	12.625	15.1
SOR 768×768	4692	2.207	29.108	13.1
SOR 1024×1024	8304	3.761	52.008	13.8
ISING 256×256	269	0.050	1.430	28.6
ISING 512×512	1049	0.111	5.497	49.5
ISING 768×768	2341	0.156	12.216	78.3
ISING 1024×1024	4145	0.430	21.711	50.4
ISING 1280×1280	6461	0.670	34.216	51.0
ASP 512	1024	0.202	5.820	28.8
ASP 1024	4096	0.584	24.298	41.6
GAUSS 512	2052	0.437	10.065	23.0
GAUSS 1024	8200	1.447	44.247	30.5
NBODY 4000	312	0.057	1.817	31.8

The overhead per checkpoint is presented in seconds. As can be seen, the overhead introduced by the parity checkpointing scheme is much higher than that incurred by neighbour based checkpointing. The last column represents the relationship between the overhead of PBC over NBC. If we take the mean average, we can see that parity checkpointing performs 34 times worse than neighbour based checkpointing.

Parity checkpointing performed even worse than disk based checkpointing. In Fig. 2 we present a comparison between PBC, NBC and disk based checkpointing (DBC). This last scheme uses the central disk to save checkpoint data but the remote write operations are done concurrently with the execution of the application. The state of the application is first saved into a memory buffer; after that, there is a checkpointer thread that sends this buffer to a remote disk file.

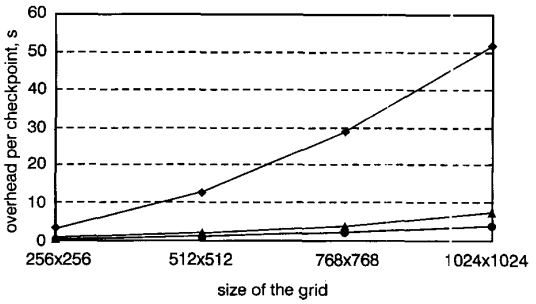

Fig. 2 *Overhead per checkpoint of NBC (—●—), PBC (—◆—) and DBC (—▲—) (SOR application)*

Neighbour based checkpointing presents a lower overhead per checkpoint than PBC, but the DBC scheme incurs a comparable overhead. The overhead of the PBC technique is much higher than the other two schemes. Take, for instance, the case of a grid size of 1024 × 1024: while the overhead per checkpoint introduced by NBC and DBC was 3.7 and 7.2 seconds, respectively, the overhead of PBC was around 52 seconds. The main reason why PBC performs a lot worse

than NBC is due to the bottleneck caused by the centralised parity operation. Processor 0 takes the role of the parity processor, but it also runs a process from the application. The execution of the PBC scheme leads to high congestion on this particular processor which results in a slower parity computation. This is the clear disadvantage of the centralised approach over the distributed approach taken by NBC.

Disk based checkpointing can have a similar overhead to NBC but presents a much higher checkpoint latency. However, the NBC scheme is not able to tolerate total failures, and in our particular implementation, it does not provide checkpoint atomicity if a failure occurs during the checkpointing protocol. To tolerate these types of failures we have to use this scheme together with checkpoints on disk. In the following Section, we present the performance results of a mixed scheme based on two-level stable storage, where we combine DBC with NBC.

5.3 Two-level stable storage

One of the advantages of neighbour based checkpointing is the potential for taking checkpoints more frequently. We can see in Fig. 3 that the checkpoint overhead is in fact very small. It represents the total performance overhead introduced by the NBC scheme when executing the ISING application for about 4 hours. With an interval between checkpoints of more than 1 minute the overhead was always lower than 1%. Reducing the interval to 30 and 10 seconds results in a natural increase in the total overhead to 2.2% and 6.7%, respectively. Even with this very high frequency of checkpointing the resulting overhead is still acceptable. With the most conservative checkpointing interval (10s) the total execution time with a grid of 1280 × 1280 is increased by less than 7%. If there is a processor failure, at most 10 seconds of computation are lost, which represents a very fast recovery.

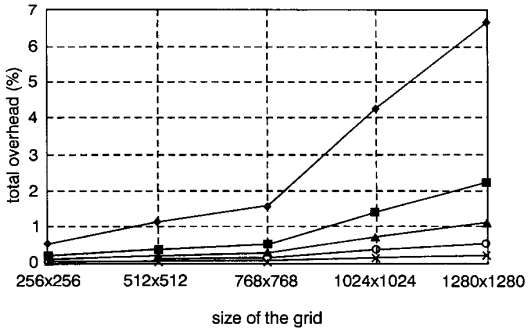

Fig. 3 *Total overhead of NBC with different checkpoint intervals (ISING)*
—◆— 10s —□— 30s —▲— 1 min —○— 2 min —×— 5 min

However, if there is a total or multiple failure the application has to be restarted from scratch, rendering completely useless all the 'soft' checkpoints that have been taken so far. Thus, the best solution is to use this scheme (NBC) with disk based checkpointing (DBC): in the event of total/multiple failure, the application can be recovered from a 'hard' checkpoint kept on disk.

NBC can be used to checkpoint at short intervals while the DBC method is used to checkpoint at long intervals. In this way, a single processor failure can be restarted from a 'soft' checkpoint while all other failures can be overcome using previous 'hard' checkpoint.

As seen earlier, the overhead per checkpoint introduced by DBC is fairly close to the NBC overhead, although it takes a bit more time to complete a checkpoint. In Fig. 4 we present the total performance overhead when using two-level stable storage, that is, combined NBC and DBC. The interval between 'soft' checkpoints is termed *IBSC* while the interval between 'hard' checkpoints is termed *IBHC*. Two values were chosen for *IBSC*: 10 and 30 seconds. With these two intervals we have chosen four possible combinations with DBC: no use of hard checkpoints, or the use of it with *IBHC* equal to 1, 2 and 5 minutes.

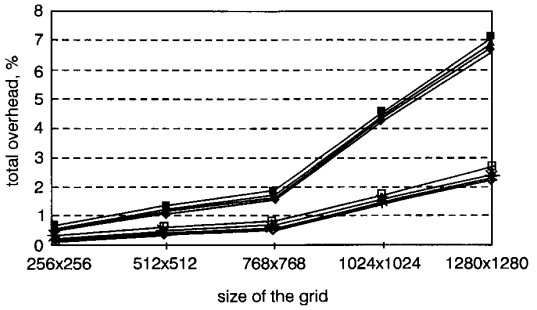

Fig. 4 *Total overhead of two-level stable storage (IBSC and IBHC)*
—◆— IBSC 10s, no hard checkpoint
—■— IBSC 10s, IBHC 1 min
—▲— IBSC 10s, IBHC 2 min
—✳— IBSC 30s, IBHC 5 min
—◇— IBSC 30s, no hard checkpoint
—□— IBSC 30s, IBHC 1 min
—△— IBSC 30s, IBHC 2 min
—×— IBSC 30s, IBHC 5 min

It can clearly be seen that the total overhead is not significantly affected by the integration of 'hard' checkpoints with the NBC scheme. The maximum difference observed was 0.45%, which is negligible. Thus, we can use this hybrid approach of NBC and DBC without additional cost. It is able to tolerate any number of failures, allows checkpoints to be taken more frequently and provides a faster recovery.

6 Comparison with related work

Parity checkpointing was first presented by Jim Plank in [12]. The first real implementation was reported in [16]. Parity checkpointing was later integrated in four subroutines of ScaLAPACK [17].

Recently, an implementation of neighbour based and parity checkpointing was reported [18]. These schemes were implemented in a different way to that used in our study: both schemes stop the application during the whole checkpoint operation and do not make use of any concurrent execution as in our case. Even so, it was shown that neighbour based checkpointing was an order of magnitude faster than parity checkpointing, but takes twice as much storage overhead.

An analytic model was presented in [19] to describe multilevel checkpoint storage schemes. A multilevel recovery scheme is one that can tolerate different numbers of failures at different costs, where the tolerance of a larger number of failures requires a larger overhead. A two-level scheme was then discussed and included 1-checkpoints and N-checkpoints: 1-checkpoints are saved in volatile memory and thus present a smaller cost, albeit they tolerate only single process failures; N-checkpoints are saved in stable storage and are able to tolerate multiple failures at the expense of a higher performance overhead. It was shown that to minimise

the average overhead it may be advantageous to take both kinds of checkpoints. We have reached a similar conclusion based on an experimental study, rather than a probabilistic mathematical model.

7 Conclusions

In this study we have presented and evaluated two schemes for diskless checkpointing: parity and neighbour based checkpointing. While the parity based scheme introduces less memory overhead it incurs a higher performance overhead than the second technique. Another interesting result was that it is possible to implement disk based checkpointing with a similar overhead to neighbour based checkpointing, although with a higher checkpoint latency.

Although neighbour based checkpointing is a very efficient technique it does not provide a high level of reliability since it is not able to tolerate multiple and total failures. To increase the reliability of this scheme the best approach is to rely on two-level stable storage where neighbour based checkpointing (NBC) is integrated with disk based checkpointing (DBC). From time to time the application saves its state to a disk file ('hard' checkpoint). During that interval the application can checkpoint its state across the memory of the neighbour processors ('soft' checkpoint). This last scheme is efficient and provides fast recovery. If there is a partial failure in the system the application can be recovered from the 'soft' checkpoint. If there is a total failure the application is restarted from a checkpoint saved on disk. This approach of two-level stable storage combines the advantages of the two schemes: the efficiency of diskless checkpointing with the reliability of disk based checkpointing.

8 Acknowledgments

We would like to thank the anonymous referees for their thoughtful comments. This work was partially supported by the Portuguese Ministry of Science and Technology (MCT), the European Union through the R&D Unit 326/94 (CISUC) and the project PRAXIS XXI 2/2.1/TIT/1625/95.

9 References

1 LAMPSON, B.W., and STURGIS, H.E.: 'Crash discovery in a distributed data storage system'. Technical Report XEROX Parc, April 1979
2 BARTLETT, J., GRAY, J., and HORST, B.: 'Fault tolerance in tandem computing systems' *in* 'Dependable computing and fault-tolerance systems' (Springer-Verlag, 1987)
3 JOHNSON, D.B.: 'Distributed system fault-tolerance using message logging and checkpointing'. PhD Thesis, TR-89-101, Rice University, Houston, Texas, December 1989
4 WILKES, J., and STATA, R.: 'Specifying data availability in multi-device file systems', *Operating Syst. Rev.*, January 1991, **25**, (1), pp. 56–59
5 ELNOZAHY, E.N., JOHNSON, D.B., and ZWAENEPOEL, W.: 'The performance of consistent checkpointing'. Proceedings of 11th symposium on *Reliable distributed systems*, 1992, pp. 39–47
6 ELNOZAHY, E.N., and ZWAENEPOEL, W.: 'On the use and implementation of message logging'. Proceedings of 24th *Fault-tolerant computing* symposium, FTCS-24, June 1994, pp. 298–307
7 PLANK, J.S., and LI, K.: 'ickp - A consistent checkpointer for multicomputers', *IEEE Parallel Distrib. Technol.*, 1994, **2**, (2), pp. 62–67
8 SILVA, L.M.: 'Checkpointing mechanisms for scientific parallel applications'. PhD Thesis, University of Coimbra, March 1997
9 BANATRE, M., MULLER, G., and BANATRE, J.P.: 'Ensuring data security and integrity with a fast stable storage'. Proceedings of 4th conference on *Data engineering*, February 1988, pp. 285–293
10 HORN, C., COGHLAN, B., HARRIS, N., and JONES, J.: 'Stable memory - another look'. International workshop on *Operating systems of the 90's and beyond*, 1991, pp. 171–177 (Lecture Notes on Computer Science, 563)
11 BAKER, M., and SULLIVAN, M.: 'The recovery box: using fast recovery to provide high availability in the UNIX environment'. Proceedings of Summer'92 USENIX, June 1992, pp. 31–42
12 PLANK, J.S.: 'Efficient checkpointing on MIMD architectures'. PhD Thesis, Department of Computer Science, Princeton University, June 1993
13 GIBSON, G.A.: 'Redundant disk arrays: reliable, parallel secondary storage'. PhD Thesis, University of California at Berkeley, December 1990
14 Parix 1.2: Software documentation, Parsytec Computer GmbH, 1993
15 MPI Forum, Message passing interface standard, March 1994, available at: http://www.netlib.org/mpi/
16 PLANK, J.S., and LI, K.: 'Faster checkpointing with N+1 parity'. Proceedings of 24th *Fault-tolerant computing* symposium, FTCS-24, June 1994, pp. 288–297
17 PLANK, J.S., KIM, Y., and DONGARRA, J.: 'Algorithm-based diskless checkpointing for fault-tolerant matrix computations'. Proceedings of 25th *Fault-tolerant computing* symposium, FTCS-25, June 1995, pp. 351–360
18 CHIUEH, T., and DENG, P.: 'Evaluation of checkpoint mechanisms for massively parallel machines'. Proceedings of 26th *Fault-tolerant computing* symposium, FTCS-26, Japan, June 1996, pp. 370–379
19 VAIDYA, N.H.: 'A case for multi-level distributed recovery schemes'. Technical Report 94-043, Dept. Computer Science, Texas A&M University, 1994