An Experimental Study about Diskless Checkpointing

Luís M. Silva João Gabriel Silva

Departamento Engenharia Informática
Universidade de Coimbra - POLO II
Vila Franca - 3030 Coimbra
PORTUGAL
Email: luis@dei.uc.pt

Abstract

Checkpointing and rollback-recovery is a very effective technique to tolerate the occurrence of failures. Usually, the checkpoint data is saved in some disk files. However, in some situations the disk operation may result in a considerable performance overhead. Alternative solutions would make use of main-memory to maintain the checkpoint data.

This paper presents two main-memory checkpointing schemes that can be used in any parallel machine without requiring any change to the hardware: one scheme saves the checkpoints in the memory of other processors, while the other is based on a parity approach. Both techniques have been implemented and evaluated in a commercial parallel machine. Some conclusions have been taken that clearly show the superiority of one of those schemes.

1. Introduction

The execution of long-running programs in a parallel machine has two sides of the same coin: hopefully, the execution time of the programs will be reduced. However, the MTBF of a parallel machine is considerably lower than a normal workstation, thereby increasing the probability of occurring failures and preventive shutdowns 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 for tolerating 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, 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, since 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. In order 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 from the old established checkpoint file. There are several ways to implement stable storage on disk [1][2][3][4].

Disk-based stable storage is the mostly used approach but in some applications that need to be checkpointed more frequently the use of the disk may result in a performance bottleneck. In fact, several experimental studies presented in [5][6][7][8] have shown that the main source of performance overhead is the time of writing the checkpoints to disk.

For this reason, other researchers have developed alternative solutions for stable storage based on the use of RAM. The schemes described in [9][10][11] use some special memory boards or some additional hardware mechanisms to prevent the erroneous accesses to stable storage. These RAM-based stable storage schemes provide a much faster access than those schemes that made use of the disk. Unfortunately, they require some 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 any additional hardware and can be implemented in any

1089-6503/98 $10.00 © 1998 IEEE

parallel machine provided there is enough spare memory among the different processors. It is not that reliable as disk-based stable storage since it cannot tolerate the occurrence of a global failure of the machine. Rather, it can be mainly used 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.

The rest of the paper is organized as follows: section 2 describes one scheme for main-memory stable storage by writing the checkpoints across the memory of neighbour processors. Section 3 presents an alternative solution based on the notion of parity block calculation to merge the checkpoints of all processors in some dedicated processor. Section 4 briefly describes the system where these schemes have been implemented while section 5 presents the experimental results. Section 6 compares this study with related work and finally section 7 concludes the paper.

2. Neighbour-based Checkpointing

A possible technique that avoids the checkpoint writing to disk is to use the main-memory of neighbour processors. Processors of the network are organized 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 only tolerate single failures. In practice, it is able to tolerate more than one failure, provided the failures do not occur in adjacent processors of the virtual ring.

Although being simple, this scheme is not robust against failures that occur during the checkpointing protocol. Two failure scenarios should be considered:

(i) processor P_i fails and its failure is detected after processor P_{i+1} has written its checkpoint to its local *snapshot area*. Recovery is not possible since processor P_{i+1} has kept the old checkpoint of P_i but has lost its own old checkpoint.

(ii) processor P_i fails while it was sending the checkpoint to processor P_{i+1}. Considering that this has partially overwritten the checkpoint of P_i that is kept in P_{i+1}, then there is no valid checkpoint for that process and recovery would not be possible.

In order to tolerate the occurrence of failures during the checkpointing protocol, the old checkpoints should also be kept. 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 of 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 this solution requires extra memory twice the size of the application 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, the system should take from time to time a global checkpoint to disk (we call these as "hard" checkpoints), and in the between, the application can be checkpointed in a distributed way to the memory of the processors (these are called as "soft" checkpoints). Assuming this hybrid approach it is possible to checkpoint the application more often, being 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 a neighbour-based scheme to compare with 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. It 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 reliability on disks, 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 maintain enough redundant information about the checkpoint data able 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.

In order to tolerate one single failure we should use a (N+1) parity technique. There is one processor in the network that we call parity processor (PP). It keeps a *parity checkpoint* of each global distributed checkpoint that is taken by the application.

Each of the other processors will save its checkpoint into a local *snapshot area*. The checkpoint size of processor P_i is called by S_i. This means that every processor should have at least an amount of unused memory of the same size of 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:

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

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

$$B_j = b_{1j} \text{ XOR } b_{2j} \text{ XOR } .... \text{ XOR } b_{nj};$$
$$1 \le j \le S_{parity\_chkp}$$

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 non-failed processors restore their state from their local checkpoints, while the checkpoint of P_i can be retrieved from all the other ones and the *parity checkpoint*, in the following way:

$$bij = b_{1j} \text{ XOR } ... \text{ XOR } b_{i-1j} \text{ XOR } b_{i+1j} \text{ XOR } ...$$
$$\text{XOR } b_{nj} \text{ XOR } B_j; \qquad 1 \le j \le S_i$$

If the parity processor fails then it can restore its state from the backup copy (that can be 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 any checkpoint atomicity it requires the minimum amount of extra memory.

Next section describes the system where we have implemented these two checkpointing schemes.

4. The Parix CHK-LIB

Those previous schemes of parity and neighbour-based checkpointing were implemented in a checkpointing library, called CHK-LIB. The 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 in 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 sub-set of the numerous MPI routines can be found in CHK-LIB.

The library implements several mechanisms of checkpointing and message logging. It was not meant to be a commercial or production tool. It was rather developed to provide the support for our study about checkpointing in the parallel systems that were available in our Department (i.e. Parsytec machines).

The use of parity and neighbour-based requires a semi-transparent approach that has the following programming interface:

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

Figure 1: Fault-Tolerant Primitives of the CHK-LIB.

The CHK_Pack_chkp() routine is used to specify the critical data of the application. The checkpoint routine -CHK_Checkpoint()- saves the relevant data of the application. That is, those variables and data-structures that have been indicated by the programmer through the use of the previous routine. The placement of checkpoints is under control of the user: she can make use of the global synchronization points already existing in the application. It is the programmer's responsibility to place the checkpoint routines in some 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 initialization part, since the library will restore the values of the critical data structure from the last checkpoint.

5. Implementation Results

In our experiments, we used an Xplorer Parsytec machine with 8 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, that is a Sun Sparc 2 Workstation. This I/O system may introduce some bottleneck during the checkpoint operation, but each processor was able to write into a different file without requiring collective synchronization.

5.1 Applications

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

- **ISING**: This program simulates the behaviour of Spin-glasses. Each particle has a spin, and it can change its spin from time to time depending on the state of its direct 4 neighbours and the temperature of the system. Above a critical temperature the system is in complete disarray. Below this temperature the system has the tendency to establish clusters of particles with the same spin. Each element of the grid is represented by an integer , and we executed this application for several grid sizes.
- **SOR**: successive overrelaxation is an iterative method to solve Laplace's equation on a regular grid. The grid is partitioned into regions, each containing a band of rows of the global grid. Each region is assigned to a process. The update of the points in the grid is done by a Red/Black scheme. This requires two phases per iteration: one for black points and other for red points. During each iteration the slave processes have to exchange the boundaries of their data blocks with two other neighbours, and at the end of the iteration all the processes perform a global synchronization and evaluate a global test of convergence. Each element of the grid is represented in double precision, and we executed this application for several grid sizes.
- **ASP**: solves the All-pairs Shortest Paths problem i.e. it finds the length of the shortest path from any node i to any other node j in a given graph with N nodes by using Floyd's algorithm. The distances between the nodes of the graph are represented in a matrix and each worker computes part of the matrix. It is an iterative algorithm. In each iteration there is one of the workers that has the pivot row. It broadcasts its value to all the other slaves. We will solve the problem with two graphs of 512 and 1024 nodes.
- **NBODY**: this program simulates the evolution of a system of bodies under the influence of gravitational forces. Every body is modelled as a point mass that exerts forces on all other bodies in the system and the algorithm calculates the forces in a three-dimensional dimension. We ran this application for 4000 particles.
- **GAUSS**: solves a system of linear equations using the method of Gauss-elimination. The algorithm uses partial pivoting and distributes the columns of the input matrix among the processes in an interleaved way to avoid imbalance problems. In each iteration, one of the processes finds the pivot element and sends the pivot column to all the other processes. We will solve two systems of 512 and 1024 equations.
- **TSP**: solves the travelling salesman problem for a dense map of 16 cities, using a branch and bound algorithm. The jobs were divided by using the possible combinations of the first 3 cities.
- **NQUEENS**: counts the number of solutions to the N-queens problem. The problem is distributed by several jobs assigning to each job a possible placement of the first two queens. We solved this algorithm with 13 queens.

5.2 Parity versus Neighbour-based Checkpointing

In this section, we compare these two checkpointing approaches and evaluate their performance overhead. The neighbour-based checkpointing is herein referred as **NBC**, while the parity checkpointing approach is referred by **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 non-adjacent processors of the virtual ring. On the other hand, the PBC approach is only able to tolerate single processor failures. In order to tolerate total or multiple failures any of those schemes (PBC or NBC) should be integrated with a disk-based checkpointing mechanism.

If we compare the two approaches in a pair basis, we can say that PBC always presents a

lower memory overhead than NBC. However, it remains to be seen what is the performance overhead of both approaches. Next, we will present a quantitative comparison between the NBC and PBC techniques. The five applications have been used, and the overhead per checkpoint is presented in the Table 1.

Application	Size of Chkp (Kbytes)	Overhead per Chkp NBC	Overhead per Chkp PBC	PBC / NBC
SOR 256x256	540	0.123	2.923	23.7
SOR 512x512	2104	0.832	12.625	15.1
SOR 768x768	4692	2.207	29.108	13.1
SOR 1024x1024	8304	3.761	52.008	13.8
ISING 256x256	269	0.050	1.430	28.6
ISING 512x512	1049	0.111	5.497	49.5
ISING 768x768	2341	0.156	12.216	78.3
ISING 1024x1024	4145	0.430	21.711	50.4
ISING 1280x1280	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

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

The overhead per checkpoint is presented in seconds. As can be seen, the overhead introduced by the parity checkpointing scheme is much higher than the one 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 say that parity checkpointing performs 34 times worse than neighbour-based checkpointing.

Parity checkpointing performed even worse than disk-based checkpointing. In Figure 2 we present a comparison between PBC, NBC, and disk-based checkpointing (DBC). This last scheme refers to the use of non-blocking disk-based checkpointing scheme: it uses the central disk to save the checkpoint data but the remote write operation is done concurrently with the execution of the application. The state of the application is firstly saved into a memory buffer and then there is a checkpointer thread that sends this buffer to a remote disk file.

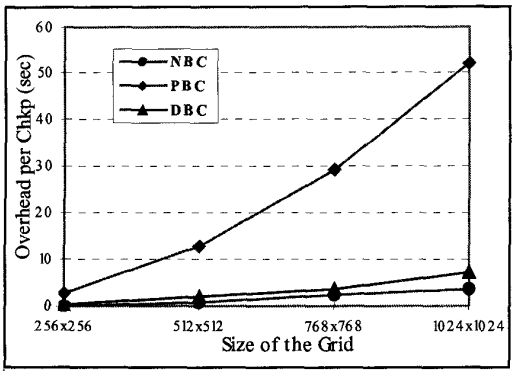

Figure 2: Overhead per checkpoint of NBC, PBC and DBC (SOR 1024x1024).

Neighbour-based checkpoint presents the lower overhead per checkpoint, but the DBC scheme incurs in a comparable overhead. On the other hand, the overhead of the PBC technique is much higher than the two other schemes. Take for instance, the case with a grid size of 1024x1024: 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.

For a grid size of 1024x1024, the time to complete a checkpoint with the PBC technique was around 109 seconds. Fortunately, the parity checkpoint is taken in background otherwise the overhead per checkpoint would be even higher. This is the advantage of using the concurrent checkpointing technique.

The main reason why PBC performs really worse than NBC is due to the bottleneck that is caused by the centralized 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 a high congestion on that processor which results in a slower parity computation. This is the clear disadvantage of a centralized approach over a distributed one that is taken by the NBC.

To improve the performance of parity checkpointing we could allocate a dedicated processor for the parity computation. It would certainly reduce the congestion on the parity processor.

We have an experiment in this line: only 7 processors of the machine were used by the application, while the 8^{th} one was allocated exclusively as the parity processor. The application data have to be distributed by the 7 available processors, which means that each individual processor will take a bigger block of data to compute with, than with 8 processors. Consequently, it should be expected some increase in the total execution of the application.

The performance results with this new configuration are presented in Table 2. They refer to the SOR benchmark with a grid size of 768x768, and 1024x1024. The NBC scheme was not able to run with a grid size of 1024x1024 for lack of memory. All the timing values are presented in seconds. We compare the overhead per checkpoint and the checkpoint latency when using NBC, PBC or DBC. In the last two rows, we also compared

two versions of the PBC scheme: non-blocking (PBC) and blocking (PBC-B).

Grid Size	Scheme	Overhead with 7 proc	Overhead with 8 proc	Latency with 7 proc	Latency with 8 proc
768x768	NBC	1.15	2.20	1.47	2.29
768x768	PBC	1.75	29.10	36.73	61.39
768x768	DBC	3.17	3.65	9.31	8.93
1024x1024	PBC	0.27	52.0	64.94	109.10
1024x1024	PBC-B	58.05	56.43	59.74	54.47

Table 2: Comparing the checkpointing schemes with 7 and 8 processors.

This Table presents some interesting results. First of all, lets examine the results with a size of 768x768. With 8 processors the overhead per checkpoint introduced by the PBC scheme was 13 times worse than with the NBC scheme (i.e. 29.1 against 2.2 seconds). It is even worse than the disk-based checkpointing scheme that presented an overhead of 3.65 seconds.

With a dedicated parity processor and 7 processors for the application the performance of PBC was really improved: an overhead of 1.75 seconds per checkpoint. Compared with the overhead of the NBC scheme (1.15 seconds) we see the difference is quite small. This time, the overhead of PBC is better than the overhead of DBC, that is 3.17 seconds.

The price to pay for this improvement is the allocation of a dedicated processor that cannot be used to do any useful computation. If the user is willing to give up of one processor for the sake of fault-tolerance, then the parity checkpointing scheme performs better.

Nevertheless, even in this configuration the NBC scheme incurred in a smaller overhead and presented a lower checkpoint latency than the PBC technique (1.47 against 36.73 seconds). An interesting aspect is the fact that the checkpoint latency of parity checkpointing is always higher than the time taken by the disk-based checkpointing scheme, regardless the number of processors: 36.73 seconds with 7 processors and 61.39 seconds with 8 processors, against 9.31 and 8.93 seconds, respectively. This is clearly an unexpected result: it takes less time to complete a checkpoint that is saved on disk than to compute and save a parity checkpoint in the memory of some processor.

In the two last rows, we present results for a grid size of 1024x1024 and we compare the performance of non-blocking with blocking parity checkpointing. As expected, with 8 or 7 processors the non-blocking PBC scheme takes always more time to complete a checkpoint than the blocking PBC technique. Since the checkpoint operation is distributed over the time it is always expected that a non-blocking algorithm take more time to complete than a blocking one. However, what is more relevant is the overhead per checkpoint. With 8 processors, the difference in the overhead is only marginal: 52 seconds for the non-blocking approach and 56.43 seconds for the blocking one. With a dedicated parity processor the difference in the overhead is quite significant: 0.27 seconds for the non-blocking algorithm against 58.05 for the blocking approach. This result emphasizes once more the importance of exploiting the concurrent checkpointing principle.

The PBC scheme is only able to tolerate single processor failures. In our particular implementation, it does not assure any checkpoint atomicity: if there is a failure during the checkpoint operation the parity checkpoint can be partially overwritten which is not valid for recovery. One way is to keep a copy of the old checkpoint in main memory or in disk. We have done some experiments with an extended version of the PBC scheme: after computing the current parity checkpoint it is saved into a disk file. This disk writing operation is done in a "lazy" way, concurrently with the execution of the application. Compared with the previous PBC scheme, the difference in the overhead per checkpoint was absolutely negligible. Thus, we can provide a backup copy of the parity checkpoint with no additional overhead.

As we have seen before, the NBC and DBC schemes present a similar overhead per checkpoint. *Per se*, this is a very impressive result. However, we have to measure as well the time that is taken to complete a single checkpoint. In Figure 3 we compare the overhead and the checkpoint latency of both schemes for the ISING application. The overhead per checkpoint is represented by columns and is relative to the left y-axis, while the checkpoint latency is depicted by lines and relative to the right y-axis.

The NBC scheme always presented a lower overhead than disk-based checkpointing, but the difference is lower than 0.3 seconds. On the other side, the checkpoint latency of DBC is much higher than the latency of NBC, and the difference can be more than 10 seconds.

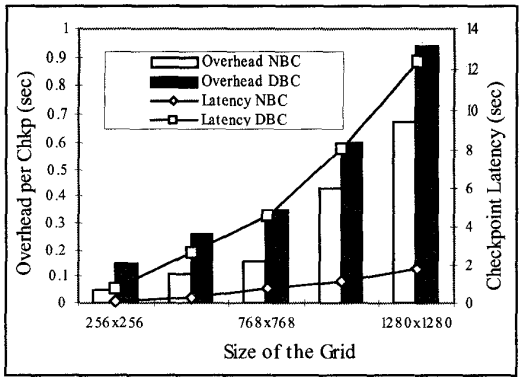

Figure 3: Checkpoint overhead and latency for the ISING application (NBC vs DBC).

Similar results were taken with the SOR benchmark (1024x1024): the difference in the overhead of NBC and DBC was almost insignificant, but there was some clear difference in the checkpoint latency: the time to complete a checkpoint in NBC was 2.3 seconds while the DBC scheme took around 16 seconds.

Disk-based checkpointing can have a similar overhead than NBC but presents a higher checkpoint latency. However, the NBC scheme does not provide checkpoint atomicity if some failure occurs during the checkpointing protocol. Thus, we recommend the use of the NBC scheme together with disk checkpointing (DBC) in order to tolerate this sort of failures.

6. Comparison with Related Work

Parity checkpointing was firstly presented by Jim Plank in [12]. The first real implementation was reported in [16]. It required two extra processors: the parity checkpoint processor and a backup processor. These ones were dedicated to the checkpointing mechanism and did not run any application process. The backup processor was used to keep a copy of the parity checkpoint when the checkpoint processor needs to update its copy. To assure the atomicity in the occurrence of failures, each processor has to maintain two in-memory copies of the local checkpoint (the current and the old one). This scheme used a hybrid approach that combined parity with incremental checkpoint.

Parity checkpointing was later integrated in four subroutines of ScaLAPACK [17]: Cholesky factorization, LU decomposition, QR factorization and Pre-conditioned Conjugate Gradient. The checkpointing scheme was "hard-wired" into any of those programs. Depending on the size of the program and the frequency of checkpointing, the total overhead reported could go from less than 1% up to 220%.

Recently, it was reported in [18] an implementation of neighbour-based, parity, and partial parity checkpointing. Partial parity checkpointing refers to the combined use of parity with incremental checkpointing. That implementation was performed on a SIMD machine (DECmpp 12000). The schemes of neighbour and parity-based checkpointing were implemented in a different way than with our study. Both schemes stop the application during the whole checkpoint operation and do not make use of any concurrent activity.

It was shown that neighbour-based checkpointing was an order of magnitude faster than parity checkpointing, but takes twice as much storage overhead. Partial parity checkpointing is able to reduce the storage overhead but can lead to unpredictable execution performance. This can be also concluded from the results of Jim Plank.

7. Conclusions and Future Work

In this study, we have presented and evaluated two schemes for diskless checkpointing: parity and neighbour-based checkpointing. While the first scheme introduces less memory overhead it incurs in a higher performance overhead than the second technique.

The overhead of parity checkpointing is only comparable to the other scheme if we use a dedicated processor of the network to compute and maintain the parity checkpoint. Otherwise, if we allocate the parity checkpoint into some application processor then the overhead of parity checkpointing is about 30 times worse than neighbour-based checkpointing and much higher than disk-based checkpointing, which makes that scheme completely useless. It only performs in an acceptable way if the user is willing to give up of one processor to be used as a dedicated parity processor. Neighbour checkpointing tolerates multiple failures as long as they do not occur in adjacent processors, while parity checkpointing is only able to tolerate single processor failures.

Another interesting result we have achieved was that it is possible to implement disk-based checkpointing with a similar overhead of

neighbour-based checkpointing, although with a higher checkpoint latency.

As future work, it would be interesting to repeat all these experiments in a cluster of workstations, connected by a 10/100 Mbit/sec Ethernet switch and then compare the overall results.

References

[1] B.W.Lampson, H.E.Sturgis. "Crash Discovery in a Distributed Data Storage System", Technical Report XEROX Parc, April 1979

[2] J.Bartlett, J.Gray, B.Horst. "Fault Tolerance in Tandem Computing Systems", in Dependable Computing and Fault-Tolerance Systems, Springer-Verlag, 1987

[3] D.B.Johnson. "Distributed System Fault-Tolerance using Message Logging and Checkpointing", PhD Thesis, TR-89-101, Rice University, Houston, Texas, December 1989

[4] J.Wilkes, R.Stata. "Specifying Data Availability in Multi-Device File Systems", Operating Systems Review, Vol. 25 (1), pp. 56-59, January 1991

[5] E.N.Elnozahy, D.B.Johnson, W.Zwaenepoel. "The Performance of Consistent Checkpointing", Proc. 11th Symposium on Reliable Distributed Systems, pp. 39-47, 1992

[6] E.N.Elnozahy, W.Zwaenepoel. "On the Use and Implementation of Message Logging", Proc. 24th Fault-Tolerant Computing Symposium, FTCS-24, pp. 298-307, June 1994

[7] J.S.Plank, K.Li. "ickp - A Consistent Checkpointer for Multicomputers", IEEE Parallel and Distributed Technology, Vol. 2 (2), pp. 62-67, 1994

[8] L.M.Silva. "Checkpointing Mechanisms for Scientific Parallel Applications", PhD Thesis, University of Coimbra, March 1997

[9] M.Banatre, G.Muller, J.P.Banatre. "Ensuring Data Security and Integrity with a Fast Stable Storage", Proc. 4th Conference on Data Engineering, pp. 285-293, February 1988

[10] C.Horn, B.Coghlan, N.Harris, J.Jones. "Stable Memory - Another Look", Int. Workshop on Operating Systems of the 90's and Beyond, Lecture Notes on Computer Science, 563, pp. 171-177, 1991

[11] M.Baker, M.Sullivan. "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment", Proc. Summer'92 USENIX, pp. 31-42, June 1992

[12] J.S.Plank. "Efficient Checkpointing on MIMD Architectures", PhD Thesis, Department of Computer Science, Princeton University, June 1993

[13] G.A.Gibson. "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] J.S.Plank, K.Li. "Faster Checkpointing with N+1 Parity", Proc. 24th Fault-Tolerant Computing Symposium, FTCS-24, pp. 288-297, June 1994

[17] J.S.Plank, Y.Kim, J.Dongarra. "Algorithm-Based Diskless Checkpointing for Fault-Tolerant Matrix Computations", Proc. 25th Fault-Tolerant Computing Symposium, FTCS-25, pp. 351-360, June 1995

[18] T.Chiueh, P.Deng. "Evaluation of Checkpoint Mechanisms for Massively Parallel Machines", Proc. 26th Fault-Tolerant Computing Symposium, FTCS-26, Japan, pp. 370-379, June 1996