A Faster Checkpointing and Recovery Algorithm with a Hierarchical Storage Approach

Wen GAO
Institute of Computing Technology, Chinese Academy of Sciences
Email: gw@ncic.ac.cn

Mingyu CHEN
Institute of Computing Technology, Chinese Academy of Sciences
Email: cmy@ncic.ac.cn

Takashi NANYA
Research Center for Advanced Science and Technology, The University of Tokyo
Email: nanya@hal.rcast.u-tokyo.ac.jp

Abstract

Fault tolerant is an inevitable part of cluster operating system. In SCore cluster system, it provides coordinated checkpointing, rollback recovery mechanism and watch-dog timer detector for fault tolerance. In the checkpointing algorithm in Score, disk write is the bottleneck. To eliminate disk write overhead, this paper proposes a new diskless checkpointing and rollback recovery algorithm. Since the proposed algorithm does not need to calculate parity and write the checkpointing data into disk, it is analyzed to be a faster checkpointing algorithm than the original one. Based on comparison, the recovery time of the proposed algorithm is also less. However, the cluster can not tolerant multiple transient failure using this diskless checkpointing algorithm. To compensate this drawback, a hierarchical storage strategy is adopted. An experimental result will be shown that this diskless algorithm with a hierarchical storage approach is fast and effective.

1. Introduction[1]

SCore cluster system software [1], which was developed by the Japan Real World Computing Partnership, is a parallel programming environment for High Performance Computing (HPC). It consists of a communication facility called PM, a MPI implementation on top of PM library called MPICH-SCore, a global operating system called SCore-D, and so on. The Score-D provides checkpointing, rollback recovery mechanism and watch-dog timer detector for fault tolerance in cluster system.

Checkpointing and rollback recovery mechanism for cluster systems have been well studied so far. They can be classified into two categories: checkpointing based, such as Cocheck[4], and log based, such as MPICH-V[5]. Checkpointing based protocols rely on checkpointing for system state restoration. It can be coordinated, uncoordinated, or communication-induced[3]. Since garbage collection and recovery involve calculating a recovery line, they can be performed by simple procedures under coordinated checkpointing. Score adopts coordinated checkpointing mechanism, and in Score-D inter process communications are flushed before checkpointing. Therefore, the system does not need to save the in_transit message, and the maximum recoverable state is the last checkpointing.

In previous work [2], Score's checkpointing and rollback recovery mechanism was evaluated and revealed that disk write is the bottleneck for checkpointing. In this paper, to eliminate disk write overhead, a faster diskless checkpointing and rollback recovery algorithm is proposed and analyzed. The contributions of this paper are summarized as follows:

- propose a new faster checkpointing and rollback recovery algorithm with a hierarchical storage approach;
- analyze the proposed checkpointing and recovery overhead and compare with the original algorithm in Score;
- discuss the advantage and disadvantage of this proposed algorithm.

[1] This research is supported by the Japan Society for the Promotion of Science and National '863' high-tech program of China

The paper is organized as follows: In section 2, checkpointing and recovery mechanism in SCore is presented. In the following section 3, the faster checkpointing and rollback recovery algorithm is demonstrated in detail. In section 4, the analysis and an experimental result of the proposed algorithm is given. Finally, the conclusion of this paper and the future work are described in section 5.

2. SCore Checkpointing and Recovery mechanism

In Score's checkpointing algorithm, each node stores its checkpoint data into the local disk in parallel. In addition, redundant data is saved to ensure the reliability for non-transient failures. This redundancy is realized through parity generation. Figure 1 show the parity generation for four nodes. Checkpointing is performed in the following steps:

1. Each node divides its checkpoint data into small blocks.
2. Each node sends the initial block to the next node in parallel.
3. Each node computes parity of the received block and its own second block in parallel.
4. Each node sends the generated parity block to the next node in parallel.
5. Step 3 and Step 4 are repeated (N-2) times, assuming the number of nodes is N.
6. Each node stores the received block into its local disk.
7. Step 2 to Step 6 are repeated until all of the data is checkpointed

In this algorithm, one parity block is generated from (N-1) data blocks after (N-2) times parity calculations. Note that the total data amount that one node sends to the next node is independent from the number of nodes and equal to the original data size in each node without parity.

When a failure is detected and execution of user program is stopped, SCore-D tries recovery from the latest checkpoint. When the recovery begins, all the computing nodes are checked.

If it is a transient failure, user program is simply restarted from the most recent checkpoint data without using parity data. Restarting is performed in each node by restoring memory image of user program from checkpoint data.

If it is a permanent failure, the node must be replaced with a spare node. To restart user program on the spare node, memory image of user program must be recovered from both checkpoint data and parity data stored in the active nodes. The recovery is performed as follows: First, the spare node tentatively prepares checkpoint data where all the data is 0. Second, Step 2 to Step 5 of checkpointing processes are performed including the spare node. After Step 5, each node executes parity calculation between the received block and the parity block stored in its local disk. Finally, the block is transferred to the spare node from non-faulty node.

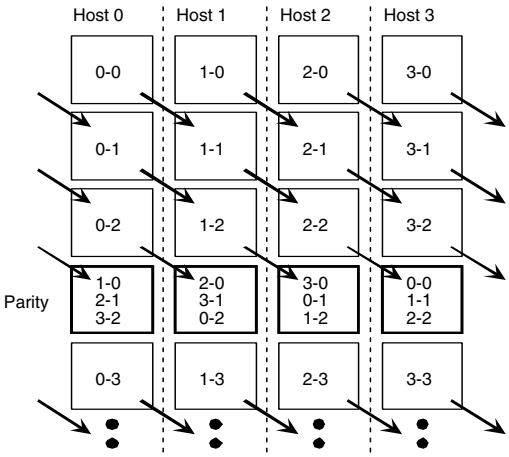

Figure 1. Parity generation

3. Diskless Checkpointing and Recovery Algorithm

Previous experimental result showed that the major source of overhead in Score checkpointing systems is the time that it takes to write checkpointings to stable storage (i.e. disks). On the other hand, the state-of-the-art network bandwidth is much higher than the disk write bandwidth. Therefore, we adopt the philosophy of Diskless Checkpointing technique which eliminates stable storage as the bottleneck in checkpointing, and places the burden on the network. Instead of using extra processors [6], we employ neighboring node's memory to provide redundancy, because we can provide parallel checkpointing in this approach

The basic idea of the checkpointing algorithm is as follows: when the checkpointing i takes place in time i, each node sends the checkpointing data to its neighboring node. After each node receives its neighboring node's checkpointing data and stores it in its memory, the application resumes running. When the application generates a page-fault while running, the original page will be copied to a local buffer. This is for the use of later recovery of checkpointing data in time i. Figure 2 shows an example of the proposed

checkpointing algorithm for four node's cluster. M0, M1, M2, and M3 indicate the checkpointing data of node0, node1, node2 and node3 respectively. Dm0, Dm1, Dm2, and Dm3 indicate the original pages in local buffer..

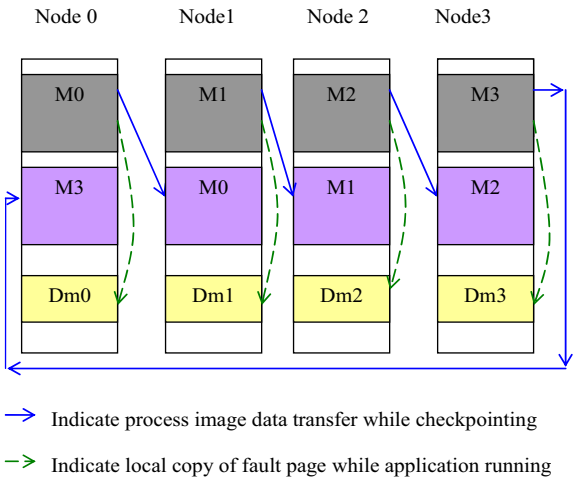

Figure 2. An example of checkpointing algorithm

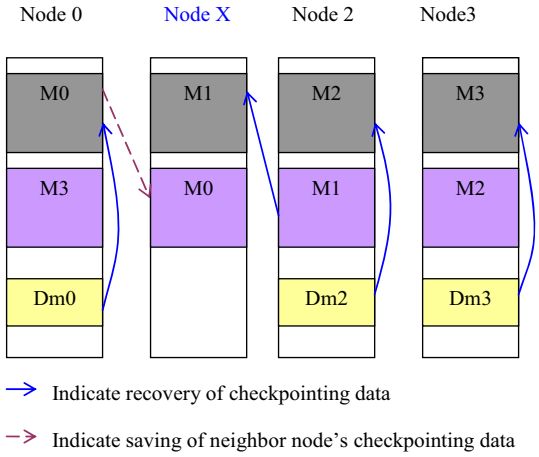

Figure 3. An example of recovery algorithm

If any node fails during the application execution, the system may be restored to the most recent checkpointing i in time i. The basic idea of the recovery algorithm is as follows: First, each active node recovers the checkpoint data from its own memory and the original pages in local buffer which will be copied back. Whether a transient failure or a permanent failure occurs, the faulty node will recover the checkpoint data from its neighboring node's memory. After that, if a transient failure occurs, the faulty node will restart again. If a permanent failure occurs, the faulty node will be replaced by a spare node in cluster. The next step is to recover the neighboring node's checkpointing data originally saved on the faulty node. This is for the next recovery use when other node fails during the application restarts but next checkpointing i+1 in time i+1 has not been made. Figure3 shows an example of the proposed recovery algorithm for four node's cluster. If node1 fails, it will restart or replaced by an extra node named node X. The node X will recover the checkpointing data M1 in time i from node2. Node0, node2 and node3 will recover the checkpointing data M0, M2, and M3 in time i from its local buffer. Then the checkpointing data M0 in time i will transfer to node X for the safe of later use.

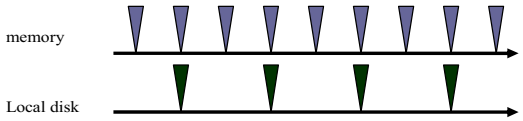

Figure 4. Hierarchical checkpointing

However, the failure coverage of the diskless algorithm is less than that of the original one, because the cluster can not tolerant multiple transient failure using the diskless checkpointing algorithm. In original algorithm, each node stores its checkpoint data into the local disk in parallel. When a transient failure occurs, user program is simply restarted from the most recent checkpoint data by restoring memory image of user program from local disk. Therefore, the number of transient failure that cluster can tolerant at the same time is N. In contrast, in the diskless algorithm, whether a transient failure or a permanent failure occurs, the faulty node will recover the checkpoint data from its neighboring node's memory. If a node and its neighbor node have transient failure at the same time, cluster can not recover correctly. Therefore, the number of transient failure that cluster can tolerant is less in the proposed algorithm.

To compensate this drawback, a hierarchical storage strategy, which is showed in Fig 4, is adopted. The diskless checkpointing is taken after every checkpointing interval. After every two checkpointing interval, checkpointing data will be also stored in local disk. Memory checkpointing is frequent and disk checkpointing is infrequent. Therefore, if more than

one transient failure occurs at the same time, the system can be restored to the most recent local disk checkpointing.

4. Analysis of Proposed Algorithm

The checkpointing and recovery overhead in Score can be estimated as follows:

$$T\_overhead = CP\_time * \left\lfloor \frac{Run\_time}{CP\_itvl} \right\rfloor + \sum_{n=1}^{\infty} n * (\mathrm{Re}c\_time + \frac{CP\_itvl}{2}) * Pn$$

Here, CP_time indicates checkpointing time, Run_time indicates application running time, CP_itvl indicates checkpointing interval, Rec_time indicates recovery time, CP_itvl / 2 indicates the average lost computation due to rollback, Pn indicates the probability of n times failure during application running. The different T_overhead caused by the proposed algorithm and the original one in Score system is decided by the CP_time and Rec_time if the same application runs on the same environment. We compare the checkpointing time and recovery time respectively in the following tables.

Table1 shows an example comparison of checkpointing time among original Score system, modified approach of previous work, and proposed algorithm of this paper when the checkpointing data size is 500MB. The cluster has 8 nodes and the network is Gigabit Ethernet. Bandwidth of ping-pong transfer of Gigabit Ethernet is nearly 100MB/s when data size is 500MB in the experiment.

As seen from Table 1, after modification of transfer block size from 1400B to 14000B, the network transfer time will decrease from 55 second to 11 second. Since the proposed algorithm does not need to calculate parity, it is not need to divide its checkpoint data into small blocks. The whole 500MB checkpointing data can transfer at one time. The network transfer time will be 5 second. Since the proposed algorithm does not need to write the checkpointing data into disk, the total checkpointing time will be only 5 second compare with 30 second in the Modified approach.

Table2 shows an analysis of recovery time between the original one in Score system and the proposed algorithm. The recovery time includes factors as network transfer time, disk read time, parity operation time, data gathering time and memory copy time. We can see from the table2 that in the Original Score system, recovering from transient failure will need less time than from permanent failure. Here, we compare Tt (the recovery time from transient failure in Original Score system) with Tp (the recovery time of proposed algorithm) as follows:

$$T_{diff} = T_t - T_p = \frac{D_c + \frac{1}{N-1} D_c}{B_{dr}} - (\frac{D_c}{B_n} + \frac{D_c}{B_m})$$

$$= D_c * (\frac{1}{B_{dr}} - \frac{1}{B_n}) + D_c * (\frac{1}{(N-1) * B_{dr}} - \frac{1}{B_m})$$

From this formula, if Bdr < Bn and (N-1) *Bdr < Bm, then Tdiff > 0. This is to say that if bandwidth of disk read is less than bandwidth of network and N-1 times of disk read bandwidth is less than memory copy in node bandwidth, the recovery time from transient failure in Original Score system will be more than the recovery time of the proposed algorithm. Therefore, under this reasonable assumption the recovery time of proposed algorithm is less than that of the original Score system.

Table 1 Comparison of checkpointing time

Recovery time factors	Original Score		Proposed Algorithm
	Permanent	Transient	
Network transfer	Dc/Bn	0	Dc/Bn
Disk read	N*Dc/(N-1)*Bdr	N*Dc/(N-1)*Bdr	0
Parity operation	Dc/Bx	0	0
Gather data	Dc/Bn	0	0
memory copy	0	0	Dc/Bm

Table 2 Analysis of recovery time

Dc: checkpointing data size. Bn: network bandwidth
Bx: parity throughput Bdr: disk read bandwidth
Bm: memory copy bandwidth

	Original (block size is 1400B)		Modified (block size is 14000B)		Proposed Algorithm
Network transfer	74%	55 sec	37%	11 sec	5 sec
Disk write	26%	19 sec	63%	19 sec	0 sec
Parity calculation	negligible		negligible		0 sec
Total	74 sec		30 sec		5 sec

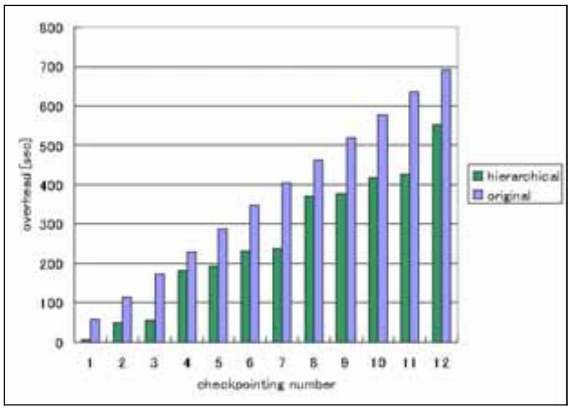

Figure 5. Checkpointing time experiment

To analyze the hierarchical storage strategy, the checkpoiting time is experimented, which is shown in Figure 5. The experimental environment is a cluster which has 8 nodes. Each node has a Xeon 2.4GHz CPU, DDR SDRAM 2048MB Memory, Ultra-160 SCSI HDD Interface and network is Gigabit Ethernet with NETGEAR GS524T Switch. Operating system is Linux kernel 2.4.18-2SCORE SCore 5.2.0. We can see that the accumulated checkpointing overhead of original algorithm is always larger than that of the proposed algorithm. Therefore, this diskless algorithm with hierarchical storage strategy is effective. Author names and affiliations are to be centered beneath the title and printed in Times 12-point, non-boldface type. Multiple authors may be shown in a two- or three-column format, with their affiliations italicized and centered below their respective names. Include e-mail addresses if possible. Author information should be followed by two 12-point blank lines.

5. Conclusion and Future Work

This paper proposes a faster diskless checkpointing and rollback recovery algorithm with a hierarchical storage approach. Compare with the original Score checkpointing and rollback algorithm and other diskless checkpointing algorithm with N+1 parity[6][7], this algorithm has the following advantage: first, since the proposed algorithm does not need to calculate parity and write the checkpointing data into disk, it is a faster checkpointing algorithm than the original one in the Score. Second, based on the comparison, the recovery time of the proposed algorithm is also less. Therefore, the total overhead of the new algorithm is lower. Third, this algorithm does not need extra processors to save parity data as the traditional proposed algorithm. We also extend this diskless checkpointing algorithm with hierarchical storage approach.

However, the disadvantage of this algorithm is that more memory space is required. In the worst condition, the requirement will be 3 times of the checkpointing data size. At the same time, Application running time increases due to the memory writing of faulty page. Therefore, application running time should be experimented and different coding techniques should be used to reduce the extra memory requirements in the future. In addition, the fault coverage assumption of this algorithm is that it only tolerant single fault at one time. We are continuing to develop this mechanism to add resilience to multiple failures in future work.

6. References

[1]Y. Ishikawa, H. Tezuka, A Hori, S. Sumimoto, T. Takahashi, F. O'Carroll, and H. Harada, "RWC PC Cluster II and SCore Cluster System Software -- High Performance Linux Cluster", In Proc. of the 5th Annual Linux Expo, pp.55--62, 1999.

[2]M.Kondo, T. Hayashida, M.Imai, H.Nakamura, T.Nanya, A.Hori, "Evaluation of Checkpointing Mechanism on Score Cluster System", IEICE Trans. Inf. & Syst. Vol.E86-D, No.1 January 2003

[3]E.N. Elnozahy, L. Alvisi, Y. Wang, and D.B. Johson, "A survey of Rollback-Recovery Protocols in Message-Passing Systems", ACM Computing Surveys, Vol.34, No.3, pp.375-408, Sep. 2002

[4]G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI", In Proc. of the 10th International Parallel Processing Symposium, pp.526--531, April 1996.

[5]G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes", In proc. of the IEEE/ACM SuperComputing 2002, Nov. 2002.

[6]J.S.Plank, kai Li, "Faster Checkpointing with N+1 Parity", 24th International Symposium on Fault-Tolerant Computing, Austin, TX, June, 1994

[7]C.Engelmann and A.Geist, "A Diskless Checkpointing Algorithm for Super-scale architectures to the Fast Fourier Transform", Proceedings of the International workshop on challenges of large applications in Distributed environments IEEE, June 2003.