Mutual-Aid: Diskless Checkpointing Scheme for Tolerating Double Faults

Jane-Ferng Chiu
Department of Information Technology and Communication
Tungnan University, ShenKeng, Taipei, 22202
Taiwan, R.O.C.
jfchiu@mail.tnu.edu.tw

Wei-Hua Hao
Department of Computer Science and Information Engineering
Tamkang University, Tamsui, Taipei, 25137
Taiwan, R.O.C.
117168@mail.tku.edu.tw

Abstract

Tolerating double faults is an important issue for diskless checkpointing due to the size and increase of the executing time. This is why Mutual-Aid checkpointing has become the first scheme to achieve the goal. Mutual-aid checkpointing combines the advantages of neighbor-based, with parity-based diskless approaches. This also tolerates all double processor faults by bitwising exclusive-or snapshots from its neighbor processors in its virtual assistant ring. In view of the fact that checkpointing and recovery of mutual-aid are so simple and efficient, this increases the performance, reduces application running time, and allows more frequent checkpoints. Moreover, it could be employed towards a very large-scale and high performance computing field because of its distributed methods as well as localized operations. The degree of fault tolerance has achieved higher success than other schemes.

Keywords fault tolerance, diskless checkpointing, tolerate double faults, mutual-aid, parity.

1. Introduction

During the last several decades, there has been rapid growth of large scale distributed and parallel computing due to the development of computers and networks. Fault tolerance is necessary for a long-running or a large scale system. To achieve the important goal, *checkpointing* is a way that system regularly saves current states on to stable storages. Once a failure occurs, the system is able to recover from the previous states that saved in a stable storage; this operation is called *rollback recovery* [4].

Stable storage [17] is a data storage device that is robust against some hardware and power failures, but the latency of writing onto a hard disk is a performance bottleneck [8]. Therefore, taking checkpoints more frequently for shorten rollback time incurs much time overhead during normal operation. That is not to operate in an acceptable way in a real-time system. In addition, some computing environments that each processor does not have its own hard disk storage, such as Single-Instruction-Multiple-Data-stream (SIMD) machine [18], massively parallel processors(MPPs), mesh, hypercube, diskless processors [3], mobile computing and opportunistic grids [13],etc. Obviously, a centralized disk-based checkpointing method in distributed or parallel computing will be a considerable overhead along with processors increase.

Diskless checkpointing is a viable approach [1-3, 6-10, 13, 18, 19] that provides alternative solutions for stable storage based on the use of main memory. There are two main types of schemes, namely, neighbor-based [10,16,18] and parity-based [6-10, 16, 19].

Neighbor-based approach is using the main memory of neighbor processor to store the states of checkpointing. When a processor fails, the checkpoint data can be recovered from its neighbor processors. Three different neighbor-based schemes had been proposed for diskless checkpoint: mirroring scheme [18], pair neighbor scheme [19] and ring neighbor scheme [18].

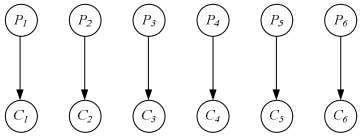

Figure 1. Mirroring scheme

Mirroring scheme [18] means that each application processor has a dedicated checkpointing processor, as shown in Figure 1. For example, application processor P_3 has a checkpoint processor C_3. This scheme is vulnerable to the failures of both related processors. For example, if P_4 and C_4 fail, the whole system is not able to recover from this kind of failure. The integrity and consistence is not guaranteed under this situation.

978-0-7695-3352-0/08 $25.00 © 2008 IEEE
DOI 10.1109/HPCC.2008.123

Pair neighbor scheme [18] is two processors in a pair and each processor sends its local checkpoint data to its partner each other, as shown in Figure 2. No additional dedicated checkpointing processor is needed. However, it is also not able to tolerate the failures occurring on both processors in the same pair. For example, if both P_3 and P_4 fail, no other processor could provide checkpoint data to recover them.

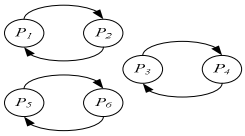

Figure 2. Pair neighbor scheme

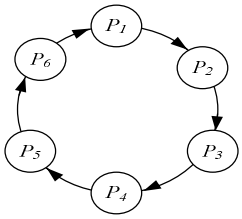

Figure 3. Ring neighbor scheme

Ring neighbor scheme [19] does not require additional processors, as shown in Figure 3. All computation processors are organized in a virtual ring. Each processor sends a copy of its local checkpoint to the neighbor processor next to it of the virtual ring. Therefore, each processor has a checkpoint maintained in memory of its neighbor. But the system could not recover from the checkpoint if a processor and its neighbor fail at the same time. For example, no checkpoint will be available for P_4 if both P_4 and P_5 fail.

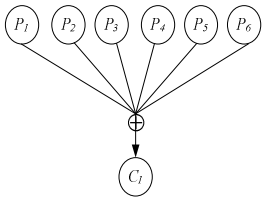

Figure 4. Parity scheme

On the other hand, parity-based approach[6, 8], as shown in Figure 4, is that all application processors P_1, P_2, P_3, P_4, P_5, and P_6 coordinate to take checkpoint with their redundancy codes (parity bits or Reed-Solomon codes) into the main memory of an extra checkpointing processor C_1 simultaneously. When a processor fails, the extra processor C_1 with other still-alive processors will be able to decode the last checkpoint of the failed processor. As a result, parity-based approach is able to reduce the size of diskless checkpoints. However, the weakness is that all processors must coordinate with each other when a checkpointing requests or a processor fails. Besides, this approach could not tolerate double faults since the redundancy code is only decoded for a single fault situation. For example, if both P_3 and C_1 fail or both P_3 and P_4 fail, the recovery information will not be decoded from other alive processor.

In neighbor-based checkpointing approach, the time overhead that incurred by checkpointing and recovery can be reduced by both checkpointing and recovering operations, because each processor only involves related processors, on the other hand, each processor with parity technique can reduces checkpointing space with Exclusive-OR(⊕) operation in parity-based approach. In general, the neighbor-based approach tends to provide better performance, whereas the parity-based approach incurs less memory overhead.

However, due to the scale of processors and the executing time increase, it is possible that two processors fail simultaneously. The next generation DOE ASCI computers (IBM Blue Gene L) are being designed with 131,000 processors [11]. The failure of some nodes or links for such a large system is likely to happen just a few minutes away. Moreover, the systems that tolerate a single processor fault may not be recovered if there is another failure during recovery [15]. Although diskless checkpointing techniques have been studied [1-3, 6-10, 13, 18, 19], to the author's knowledge none of the previous studies does tolerate double processors simultaneous faults [13]. Consequently, it is necessary to develop a new scheme to meet the needs of tolerating double faults in a large scale or a long-running distributed system.

Mutual-aid checkpointing scheme is combining the advantages of neighbor-based with parity-based diskless checkpointing to tolerate double faults.

The rest of this paper is organized as follows. In section 2, it describes mutual-aid checkpointing and recovery mechanism. Section 3 presents checkpointing and failure recovery schemes. Section 4 proves the correctness. Section 5 compares this study with related work. Finally, section 6 draws the conclusion.

2. Mutual-Aid checkpointing and recovery mechanism

We consider a distributed system which consists of a collection of n diskless processors (or nodes): P_1, P_2, ..., P_n. Because mutual-aid checkpointing achieves for tolerating double faults, the number of processors should be at least five ($n \geq 5$). Each processor contains its own physical memory and communication devices.

A computing task is partitioned into such n subtasks that each subtask is executed on a distinct processor P_i, where $1 \le i \le n$, in an asynchronous manner. These subtasks communicate with each other by passing messages via the underlying interconnection network. Processors are fail-stop [12], and communication channels are reliable.

Processors of the network are organized in a *Virtual Assistant Ring* (VAR), as shown in Figure 5.

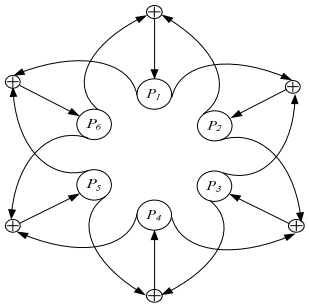

Figure 5. **Virtual Assistant Ring (VAR)**

In VAR, each processor P_i, $1 \le i \le n$, has its own two correlate neighbor processors. For example, P_1 and P_3 are correlate neighbor processors of P_2.

In VAR, the preceding and following ends of the processors are joined. For example, as shown in Figure 5, P_6 is the preceding processor of P_1, as well as P_1 is the following processor of P_6.

Each processor sends its checkpoint into its two neighbor processors' main memory as neighbor-based checkpointing approach. Besides, each processor is employed to reduce checkpointing space. Exclusive-OR ($\oplus$) operation is used to reclaim memory space of checkpoint into one half as parity-based checkpointing. However, unlike the previous method, our checkpointing operation is localized from the parity update operation.

The main memory of each processor P_i, $1 \le i \le n$, is divided into four sections: *application section*(AS_i), *parity section*(PS_i), *backup section*(BS_i) and *reserving section*(RS_i), as shown in Figure 6.

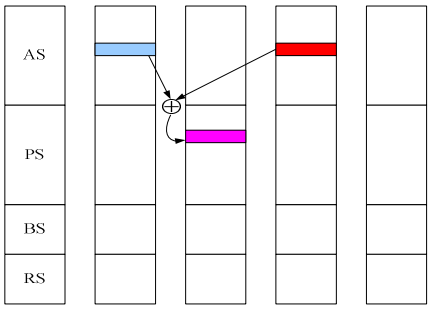

Figure 6. **Sections of main memory**

AS_i is for application execution and checkpoint area of itself. Each processor P_i will save its checkpoint from AS_i. PS_i is an area for taking the mutual-aid parity checkpoint for neighbor processors. During normal operation, each processor P_i will modify the content of AS_i, and BS_i is a temporary space to backup previous data of AS_i which will be correct decoded the last checkpoint if its neighbor processor fails. The section of RS_i is also a temporary space for reserving previous checkpoint during each processor taking a new checkpoint. Memory management unit (MMU) of each processor is enabled to protect pages of memory as read-only or read-write. Mutual-aid checkpointing is able to access MMU and catch the resulting page faults. Each processor must have adequate memory space for its own and other checkpointing.

3. Checkpointing and recovery scheme

The key of tolerating double faults is the ability to take two copies of a checkpoint by combining with the techniques of parity-based and neighbor-based. In order to reduce checkpoint size, we can make each processor take a mutual-aid checkpoint by parity-based method. In addtion, in order to speed up these operations, each processor only sends its checkpoint to its adjacent neighbors.

3.1 Basic idea

The basic idea of mutual-aid checkpointing is a group work that exchanges critical data wherein the processors is act as both the provider and the recipient in service to achieve the goal of fault tolerance . This method has two phases of neighbor-based phase and parity-based phase.

In neighbor-based phase as shown in Figure 5, processors in a system are arranged as a VAR. Not only does each processor send two copies of its recovery information to its two adjacent neighbor processors but it also receives two copies from its two adjacent neighbor processors. For example as shown in Figure 5, P_2 sends its checkpoint to P_1 and P_3.

In parity-based phase as shown in Figure 5, mutual-aid checkpoint is a redundancy that is a result of calculating an exclusive-or of two received snapshot from its neighbor processors. For example, P_2 and P_4 send their checkpoints to P_3. After receiving each page of checkpoint from P_2 and P_4, P_3 bitwises with exclusive-or and saves into parity section (PS_i). For the sake of reducing memory space, each processor reduces the size of recovery information from its adjacent neighbor processors into its parity section

(PS_i). Consequently, a VAR has double assistant chains against double faults.

Incremental checkpointing technique[12, 15] has been used to reduce the size of information that needs to be saved for a checkpointing request. With incremental checkpointing the data that need to be stored for a new checkpoint are limited to only the memory pages that have been modified after making the last checkpoint. Essentially, checkpointing of a processor P_i only requires that the lately modified pages, not whole states data, be sent to the adjacent neighbor processors. When a page of a processor is to be modified, the original content of the page must be maintained in order to help recover the corresponding page of its neighbor processors. Asynchronous page-by-page updating of the parity data reclaims the spaces in its two neighbors. Hence, the mutual-aid checkpointing only operate with its neighbor processors so that computation and checkpointing can be minimized and localized.

As the time needed to take a checkpoint is reduced, mutual-aid checkpointing is able to adopt a checkpointing scheme which may have frequent checkpointing requests. When a processor or processors fail, the state of the failed processors can be reconstructed by using the contents of the neighbors' memories. Essentially, its mutual-aid checkpoints on the adjacent neighbor processors store the most recently updated pages of its latest checkpoint.

However, due to communication between processors, a failure at a processor may cause other failure-free processors to roll back in order to maintain global consistency of the computing task with uncoordinated checkpointing. Such rollback propagation results in a waste of computing resources. To avoid this drawback, mutual-aid checkpointing mechanism requires that a processor takes a checkpoint right before it transmits a message to another processor or right before it delivers a message from other process [4, 14]. In other words, the sender of a message must commit to the transmission of the message. This obviously prevents rollback propagation.

Mutual-aid checkpointing scheme is split into four operations: initial action, normal operation checkpointing and fail recovery.

3.2 Initial action

The parity section (PS_i) size of processor P_i should be larger than or equal to the maximum checkpoint size of its neighbor processors, the preceding processor P_a, and the following processor P_c. The jth bytes of PS_i holds the parity as a result of exclusive-oring two of jth bytes of AS_a of P_a and AS_c of P_c, that is

$b_{PS,\ i,\ j}=b_{AS,\ a,\ j}\oplus b_{AS,\ c,\ j}$, for a=mod(i+n-2,n)+1; c=mod(i+n,n)+1, where $1\le i \le n$.

The access mode of each memory page can be in either read only or read write state. If a processor attempts to write to a page with read only state, a page fault occurs.

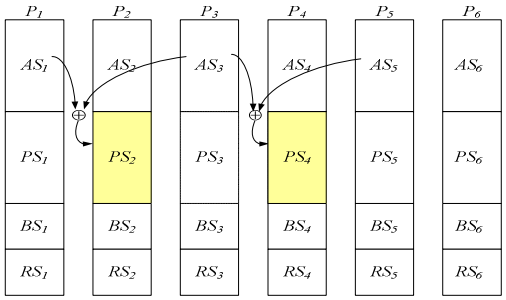

Figure 7. Initial action

Initially, each processor takes a mutual-aid checkpoint by simply setting the access modes of all of the memory pages to read only state. That is let AS_i be read only with memory protect. Moreover, each processor clears of both its BS and RS. P_i sends it's AS_i to the preceding processor P_a and the following processor P_c. At the same time, when P_i receives AS_a and AS_c from P_a and P_c (a=mod(i+n-2,n)+1; c=mod(i+n,n)+1 where $1\le i\le n$), P_i calculates parities by bitwising exclusive-or and stores the result in PS_i. Each processor P_i ={ AS_i, PS_i, BS_i, RS_i}, PS_i= $AS_a\oplus AS_c$. For instance as shown in Figure 7, P_3 sends its AS_3 to P_2 and P_4, then P_2 stores the result of the exclusive-or operation on AS_1 and AS_3 on PS_2, PS_2= $AS_1\oplus AS_3$. That is P_2 takes the first mutual-aid checkpoint of P_1 and P_3. P_4 is similar to P_2 that takes the first mutual-aid checkpoint of P_3 and P_5, PS_4= $AS_3\oplus AS_5$.

3.3 Normal operation

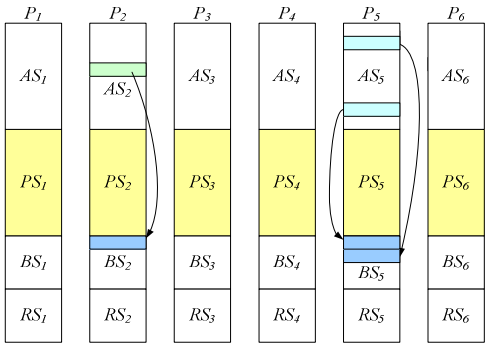

Figure 8. Normal action

Processors start executing their programs after the initialization phase is completed. Read operations proceed as usual. However, a page fault is generated when a processor attempts to write to a read only page. Once a page fault occurs, the content of the accessed page is copied to its BS_i, and the page's access mode is set to read-write so that it can be written. Note that the parity datas of the pages in BS_i are stored in PS_a of its following processor and PS_c of preceding processor respectively. Hence, the content of PS_a and PS_c, will be used for crash recovery in case of another neighbor processor P_k or P_l, , k=mod(i+n+1,n)+1; l=mod(i+n-3,n)+1; where $1 \le i \le n$ fails later. As shown in Figure 8, P_2 and P_5 copy an old page or pages from AS_2 and AS_5 into BS_2 and BS_5 respectively.

3.4 Checkpointing

Processor P_i has to take a new checkpoint by application programmed when communication is induced or backup section BS_i, is full. Processor P_i senses the new page flag status, calculates the differences between new page and the pervious page in backup section, and sends the difference to its two adjacent neighbor processors P_a and P_c.; a=mod(i+n-2,n)+1; c=mod(i+n,n)+1 for $1 \le i \le n$. When processor P_j receives the difference $diff_i$ of new page pg_d, P_j is ready to take a new mutual-aid checkpoint into parity section PS_i.. P_j copy the content of the relative address of pg_d in PS_j to its RS_j. After copying those pages completely, P_j takes its new mutual-aid checkpoint by exclusive-or pg_d with $diff_i$. After all pages $diff_i$ have been processed with exclusive-or, then P_j takes this new mutual-aid checkpoint completely, each P_j removes the content of RS_j. For example, as shown in Figure 9, P_2 takes two mutual-aid checkpoints into PS_1 and PS_3 individually at the same instant in time. In the meantime, P_5 takes two mutual-aid checkpoints into PS_4 and PS_6.

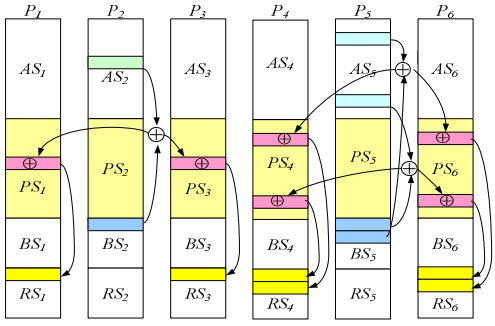

Figure 9. Checkpointing

Since above operations are limited to communicate and to exclusive-or the modified pages after last checkpoints of related processors, the operations of checkpointing is localized and asynchronous. Therefore, the time overhead of checkpointing is relatively small. This advantage let each processor could take mutual-aid checkpoints frequently even if there are more checkpoints induced by application communication.

3.5 Failure recovery

Let P_f be the fail processor, when processor P_f restarts or a new processor replace P_f, processor P_f sends the messages to its four neighbor processors, two are its preceding processors P_{a1} and P_{a2}; $a1$=mod(f+n-2,n)+1; $a2$=mod(f+n-3,n)+1; for $1 \le f \le n$., the other two are its following processors P_{c1} and P_{c2} . $c1$=mod(f+n,n)+1; $c2$=mod(f+n+1,n)+1

.

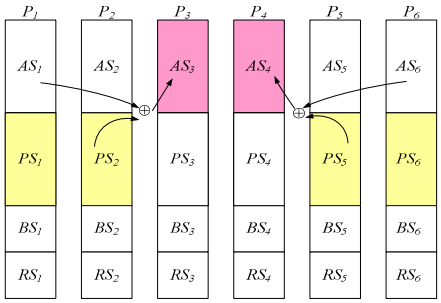

Figure 10. Direct failure recovery

There are four cases described as follows.

Case I. If both preceding processors P_{a1} and P_{a2} are alive, then failure processor P_f could use $AS_f = PS_{a1} \oplus AS_{a2}$ to recover. For example, as shown in Figure 10, failure processors P_3 could be recovered by $AS_3 = PS_2 \oplus AS_1$.

Case II. If both following processors P_{c1} and P_{c2} are alive, then P_f could utilizes $AS_f = PS_{c1} \oplus AS_{c2}$ to recover. For example, as shown in Figure 10, failure processors P_4 could be recovered by $AS_4 = PS_5 \oplus AS_6$. Case I and case II are direct failure recovery. The recovery operation affects only three processors without global coordination. That is a small overhead in system during recovering.

Case III. If related processor only P_{c1} is alive but P_{c2} is failed. For example, as shown in Figure 11, failure processors P_2 and P_4 *are* failed. However, if the number of processors is larger than four and only double faults in systems, the following processor P_{e1} and P_{e2} of P_{c2} should be alive, $e1$=mod($c2$+n,n)+1 ; $e2$=mod($c2$+n+1,n)+1. In Figure 11, P_5 and P_6 are alive. The failure processor P_{c2} is able to recover by $AS_{c2} = PS_{e1} \oplus AS_{e2}$, then processor P_f is able to recover by $AS_f = PS_{c1} \oplus AS_{c2}$ indirectly. For example as shown

in Figure 11, processor P_4 could be recovered by $AS_4 = PS_5 \oplus AS_6$. After processor P_4 rolls back to its last checkpoint. Processor P_2 could be restarted by $AS_2 = PS_3 \oplus AS_4$. Case III is indirect failure recovery.

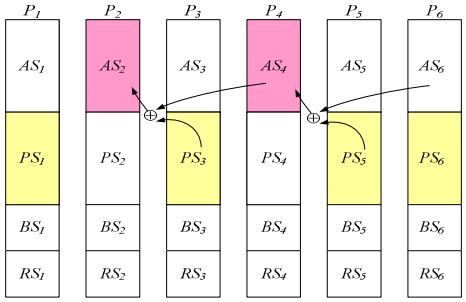

Figure 11. Indirect failure recovery

Case IV. If those related processors P_{c1}, and P_{a1} of processor P_f are failed, then there is no information to recover P_f. For example, as shown in Figure 12, processor P_2, P_3, and P_4 fail at the same time, P_3's recovery information in both P_2 and P_4 are lost, so P_3 could not be recovered. However, in this condition, there are at least more than two faults.

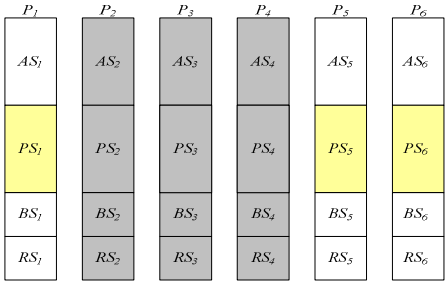

Figure 12. More than double faults

Another point to stress, the following processor P_d of processor P_f may has changed some pages after processor P_f has taken its own last mutual-aid checkpoint. In this condition, processor P_d should roll back to its last mutual-aid checkpoint by copy BS_d to AS_d for consistent with PS_c that is able to assist processor P_f recover correctly. That is $AS_f = PS_c \oplus AS_d$ and AS_d is the state of last mutual-aid checkpoint. Consequently, processor P_f could recover during normal operation even if some pages had been changed after last checkpoint.

If processor P_f 's following processor P_c didn't take the last mutual-aid checkpoint complete before processor P_f had been failed. Although, current PS_c is not complete, the previous mutual-aid checkpoint is succeeded to get by copying the difference from reserving section RS_c. Therefore, PS_c after modified is consistent with the state of its neighbor processors P_f and P_d. However, P_d may modify some pages after last checkpoint, so AS_d is also modified by copying from reserving section RS_d. After changing PS_c and AS_d, by $AS_f = PS_c \oplus AS_d$, processor P_f is able to recover during its checkpoint period. For example as shown in Figure 13, processor P_3 fails while P_3 is taking its xth mutual-aid checkpoint and processor P_4 has a page changed after the $(x-1)$th mutual-aid checkpoint. P_4 rolls back $(x-1)$th mutual-aid checkpoint by copying the previous page with reserving session RS_4, then AS_4 is consistent with $(x-1)$th mutual-aid checkpoint. Processor P_5 copies the difference pages from BS_5, hence AS_5 is also consistent with $(x-1)$th mutual-aid checkpoint. Processor P_3 is able to recover $(x-1)$th checkpoint by $AS_3 = PS_4 \oplus AS_5$.

As described above, the operations for recovery is so simple and asynchronous that only needs minor number of processors (often only three). As a result, the overhead for recovery is small and is easy to implement to a large scale system.

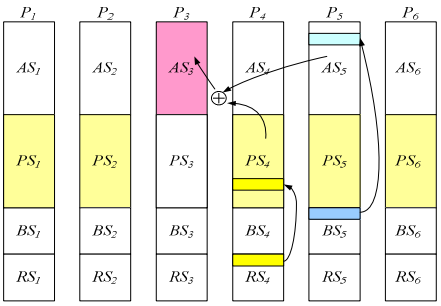

Figure 13. Failure recovery of system during checkpointing

4. Correctness

Processor would take a checkpoint right before it sends the message and receive the message according to mutual-aid checkpointing scheme. Therefore, orphan messages do not exist, and thus consistency of the system is ensured.

For each processor P_i, $1 \le i \le n$, has its correlate processors, two preceding processors, P_a and P_b; $a = \text{mod}(i+n-2,n)+1$; $b = \text{mod}(i+n-3,n)+1$ for $1 \le i \le n$, and two following processors P_c and P_d.; $c = \text{mod}(i+n,n)+1$; $d = \text{mod}(i+n+1,n)+1$ for $1 \le i \le n$. In VAR, the preceding and following ends of the processors are joined.

Theorem 4.1 Double failure processors P_x and P_y can be recovered to a consistent state by its own mutual-aid checkpoint.

Proof: Since the number of processors in the assistant virtual ring is equal to or larger than four $(n \ge 5)$, it is easy to know that the second following

processor P_d of processor P_i, is not the second preceding processor P_b . That is $d \neq b$, for $n \geq 5$. There are two relation cases between processors P_x and P_y.

case I: Failure processors P_x and P_y are adjacent. Assume P_x is the preceding processor of P_y , that is $y=x+1$. There are two following processors P_u and P_v of P_y are alive because P_u or P_v is not P_x , for $n \geq 5$. In such condition, according to section 3, processors P_y is direct to recover by its two following processors P_u and P_v of P_y, $AS_y = PS_u \oplus AS_v$.
Namely, there are two preceding processors are alive. In the same way, P_x is enable to recover by its the preceding processors directly.

case II: Failure processors P_x and P_y are not adjacent.It means that at least one processor P_u is alive between P_x and P_y. At least the other two processor $P_{v'}$ and $P_{v''}$ are alive if only P_u between P_x and P_y , since the number of processors in a system surpass four. According to section 3, processors P_x and P_y is enabled to recover by the assistant of their neighbor $P_{v'}$ and $P_{v''}$, if there are more than one processor $P_{u'}$ and $P_{u''}$ between P_x and P_y. Due to the fact that $P_{u'}$ and $P_{u''}$ are the neighbors of P_x and P_y, they are able to assist P_x and P_y directly. Consequently, mutual-aid checkpoints always tolerate double faults. Q.E.D.

As described above, those checkpoints are consistent with other processors.

5. Comparison with Related Work

To compare with other diskless checkpointing schemes, the degree of fault tolerance is an important measurement to evaluate [18]. The degree of fault tolerance is defined the probability of tolerate k faults and this paper assumes that the failure of each processors are independent and identically distributed.

In mutual-aid checkpointing, up to $n/2$ processor failures may be tolerated. The probability to survive from k faults by mutual-aid checkpointing can be calculated as in Table 1. #P is denoted the number of processors. The detail is as follows. The number of cases for recovery is denoted S. The symbol ${}_nC_k$ denotes the number of ways of picking k unordered outcomes from n possibilities.

#P	10	20	30	40	50	100
k=2	1	1	1	1	1	1
k=3	0.917	0.982	0.993	0.996	0.997	0.999
k=4	0.67	0.930	0.970	0.984	0.990	0.997
k=5	0.262	0.833	0.928	0.960	0.975	0.994

Table 1. The probability of tolerate *k* faults by mutual-aid checkpoint

In mutual-aid checkpointing, $S = {}_nC_k$ if $k=2$, $S = {}_nC_k - n$ if $k=3$, $S = {}_nC_k - 2n - 2({}_{n-k}C_2 + {}_{n-k-1}C_{k-1}) - 2(n-k-1)$ if $k=4$, $S = {}_{n-k}C_k + {}_{n-k-1}C_{k-1} + 4\,{}_{n-k}C_{k-2} + 3\,{}_{n-k-1}C_{k-3} + 4\,({}_{n-k}C_4 + {}_{n-k-1}C_3) - 6(n-k-2) - n$ if $k=5$.

The degree of fault tolerance is $S/({}_nC_k)$.

The degree of fault tolerance shows that mutual-aid checkpointing scheme is not only to tolerate double faults completely but also to nearly land one hundred percent for large scale systems.

Mutual-aid check compares with related work in the probability of tolerate k faults in Table 2. The word "Mirror" denotes mirror checkpointing [17] that shown in Figure 1. In this comparison, the number of application processors is 100, so there are 200 processors in the mirroring scheme. The probability that the mirroring scheme survives k processor failures is $({}_nC_k 2^k)/({}_{2n}C_k)$ .The word "Ring" denotes ring neighbor scheme [18] , as shown in Figure 3. The probability that the ring scheme survives k processor failures is $({}_{n-k}C_k + {}_{n-k-1}C_{k-1})/({}_nC_k)$ .The word "Pair" denotes pair neighbor scheme [17] , as shown in Figure 2. The probability that the pair neighbor scheme survives k failures in an n processor job is $({}_{n/2}C_k 2^k)/({}_nC_k)$. The word "Parity" denotes the parity scheme as shown in Figure 4, that is adopted one dimensional checksum scheme [17] and there are 10 sub-groups and each sub-group has 10 processors. That is $n=100$ and $m=10$. The failure of each processor is independent and identically distributed, then the probability of such scheme survives k($k<m$) failures is $({}_{m+1}C_k (n+1)^k)/({}_{(n+1)m}C_k)$.

Finally, "MA" denotes mutual-aid checkpointing scheme and the number is also 100 processors. From Table 2, the degree of fault tolerance is higher than other schemes.

	Mirror	Ring	Pair	Parity	MA
k=2	0.995	0.980	0.990	0.91	1
k=3	0.984	0.94	0.970	0.74	0.999
k=4	0.980	0.882	0.968	0.53	0.997
k=5	0.95	0.81	0.93	0.32	0.994

Table 2. Comparison of the probability of tolerate *k* faults

Furthermore, comparison of the overhead with related works is in Table 3. The memory redundancy of mutual-aid checkpointing is not the best but equal to other neighbor-based for tolerating double faults.

Compared with parity-based, the number of exclusive-or operations for each checkpointing and recovering, mutual-aid checkpointing is bounded. Beside, the advantage of distributed and asynchronous way is able to scalable. The high fault tolerant degree, fast checkpointing, efficient recovery and scalability are of great worth.

	Parity	Mirror	Ring	Pair	MA
Add # pro.	1/group	n	0	0	0
Ckp memory	n+1	2n	2n	2n	2n
XOR/ckp	n+1	0	0	0	3
XOR/recover	n+1	0	0	0	3
Asynchronous	no	yes	yes	yes	yes

Table 3. Comparison of the overhead

6. Conclusion

Six terms of mutual-aid checkpointing are summarized as follows:

1. It is a novel scheme, to our knowledge, for tolerating double processors faults in diskless checkpointing.
2. Like neighbor-based method, it simplifies operation processes without additional processors. It also compresses checkpoints to reduce memory size, which is the function as parity-based method.
3. Both checkpointing and recovery operations are simple and localized that only a minority of the processor is needed.
4. To reduce the checkpoint size, the incremental checkpoint technique is adopted. Communication induced checkpointing (CIC) allows processes in a distributed computation to take independent checkpoints and to avoid the domino effect.
5. It is possible to tolerate up to the half the number of processors. According to the results of our calculations, the survival rate of mutual-aid checkpoint is greater than that of the other diskless checkpointing.
6. With the benefits of independent distributed models and less operational steps, it increases the number of multiple operating systems and accomplishes the computing needs of high-volume and high-efficiency.

7. References

[1] C. Engelmann and A. Geist, "A diskless checkpointing algorithm for super-scale architectures applied to the fast fourier transform", *Proceedings of CLADE 2003*, pp. 47–52.

[2] C. Engelmann and S. Scott, "High availability for ultrascale high-end scientific computing",. *Proceedings of COSET-2*, 2005.

[3] Chao-Tung Yang, Ping-I Chen, and Ya-Ling Chen, "Performance Evaluation of SLIM and DRBL Diskless PC Clusters on Fedora Core 3", *PDCAT 2005*, pp. 479 – 482.

[4] E.N. Elnozahy, L. Alvisi, Y. Wang, and D.B. Johson, "A survey of Rollback-Recovery Protocols in Message-Passing Systems", *ACM Computing Surveys*, Sep. 2002, Vol.34, Po.3, pp.375-408.

[5] Gengbin Zheng; Lixia Shi and Kale L."V., "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI", 2004 *IEEE International Conference on Cluster Computing*, 20-23 Sept. 2004, pp.93 – 103.

[6] Jane-Ferng Chiu and Ge-Ming Chiu, "Hardware-supported asynchronous checkpointing scheme", *Computers and Digital Techniques, IEE Proceedings, Volume 145, Issue 2*, March 1998, pp. 109–115.

[7] J.S. Plank and K. Li, "Faster Checkpointing with N + 1 Parity," *Proc. 24th Int'l Symp. Fault-Tolerant Computing*, Austin, Tex., June 1994,. pp. 288-297.

[8] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing", IEEE Transaction on Parallel Distributed Systems, 1998, pp.972-986.

[9] J. S. Plank, Y. Kim, and J. J. Dongarra,. "Fault-tolerant matrix operations for networks of workstations using diskless checkpointing", *Journal of Parallel and Distributed Computing*, JUN 1997, pp. 125 – 38.

[10] L. M. Silva and J. G. Silva. "An experimental study about diskless checkpointing", In EUROMICRO'98, pp. 395–402.

[11] N. R. Adiga and et al. "An overview of the BlueGene/L Supercomputer", Proceedings of the Supercomputing Conference (SC'2002), 2002Baltimore MD, USA, pp.1–22

[12] P.R. Wilson and T.G Moher, "Demonic Memory for Process Histories", *Proc. SIGPLAN '89 Conf. Programming Language Design and Implementation*, pp. 330-343.

[13] R. Y. de Camargo, R. Cerqueira, and F. Kon. "Strategies for checkpoint storage on opportunistic grids", IEEE Distributed Systems Online, September 2006, pp. 1 – 11.

[14] Russell, D. L. "State restoration in systems of communicating processes", *IEEE Transactions on Software Engineering, Volume SE-6, Issue 2*, pp. 183-194.

[15] S.I. Feldman and C.B. Brown, "Igor: A System for Program Debugging via Reversible Execution," *ACM SIGPLAN Notices, Workshop Parallel and Distributed Debugging*, vol. 24, no. 1, Jan. 1989, pp. 112-123.

[16] Silva, L.M. and Silva, J.G.; "Using two-level stable storage for efficient checkpointing", Software, IEE Proceedings-Volume 145, Issue 6, Dec. 1998, pp. 198 – 202

[17] Sobe, P., "Stable checkpointing in distributed systems without shared disks", *International Parallel and Distributed Processing Symposium, 2003 Proceedings*, 22-26 April 2003 Page(s):8 pp.

[18] Tzi-Cker Chiueh and Peitao Deng; "Evaluation of checkpoint mechanisms for massively parallel machines", *Proceedings of Annual Symposium on Fault Tolerant Computing, 1996*., 25-27 June 1996, pp. 370 – 379.

[19] Zizhong Chen, and et al. , "Testing and fault tolerance: Fault tolerant high performance computing by a coding approach", *Proceedings of the tenth ACM SIGPLAN symposium on PPoPP '05*, pp. 93 – 103.