INFORMATION TECHNOLOGY JOURNAL

Information Technology Journal 4 (4): 367-376, 2005
ISSN 1812-5638
© 2005 Asian Network for Scientific Information

On Disk-based and Diskless Checkpointing for Parallel and Distributed Systems: An Empirical Analysis

[1]Najib A. Kofahi, [2]Said Al-Bokhitan and [3]Ahmed Al-Nazer
[1]Department of Computer Sciences, Faculty of Information Technology and Computer Sciences, Yarmouk University, Irbid, Jordan
[2]Hadeed, Jubail 31961, Saudi Arabia
[3]Saudi Aramco, Dhahran 31311, Saudi Arabia

Abstract: Checkpointing is the act of saving the state of a running program so that it may be reconstructed later in time. It is an important basic functionality in computing systems that paves the way for powerful tools in many fields of computer science. This study provides a comprehensive overview of two main checkpointing approaches in parallel and distributed systems: Disk-based and diskless checkpointing approaches. The two approaches are discussed and compared and an experimental study was conducted to lend support to the arguments presented in the study. We found that each approach has advantages over the other and they are not truly alternative to each other. We conclude that a combined approach of both will have the advantages of the two approaches and therefore is a desirable option.

Key words: Checkpointing, disk-based checkpointing, diskless checkpointing, distributed systems, parallel systems, fault-tolerance, rollback recovery

INTRODUCTION

Distributed systems consisting of a network of workstations or personal computers are an attractive way to speed up large computations. These systems have a much higher performance-to-price ratio than large parallel computers and they are also more widely available. The computing nodes in a distributed system may fail. As some applications may require hours to execute, it is important to be able to continue computation in the presence of node failure. Two main classes of solutions to the problem of node failure are checkpoint-based and log-based rollback-recovery schemes as suggested by Bouteiller *et al.*[1]. Recovery from failures becomes more important for large systems, since the possibility of a node failure increases with the number of computing nodes. A rollback-recovery mechanism consists of three parts: checkpointing, fault detection and failure recovery. During checkpointing the states of the participating processes are periodically saved. The saved process state is called a checkpoint. When a node failure occurs, the recovery mechanism uses saved checkpoints to recover the system to the consistent system state and continue execution from that state. The number of processes that have to be rolled back to the previous checkpoint varies, depending on the recovery algorithm[2-4]. It may be necessary for one, some, or all processes to rollback to the previous checkpoint. Checkpointing is so important that recent research[5] includes investigating the possibility of implementing an incremental checkpointing system that is completely automatic and user transparent, minimally intrusive, scalable and feasible with current and foreseeable I/O technology. They try in their study to demonstrate that frequent, user-transparent, automatic incremental checkpointing is a viable technique. There are several applications of checkpointing[6] including: rollback recovery, playback debugging, process migration, job swapping and load balancing[7]. In the next three paragraphs we briefly discuss some of the issues to consider with checkpointing[8].

The first issue is the frequency of checkpointing. The number of checkpoints initiated should be such that the cost of information loss due to failure is small and the overhead due to checkpointing is not significant. This number depends on the failure probability and the importance of the computation. For example, in a transaction processing system where every transaction is important and information loss is not permitted, a checkpoint may be taken after every transaction, increasing the checkpointing overhead significantly.

Corresponding Author: Najib A. Kofahi, Associate Professor, Department of Computer Sciences, Faculty of Information Technology and Computer Sciences, Yarmouk University, Irbid, Jordan
Tel: +962 2 7211111/2677 Fax: +962 2 7211128

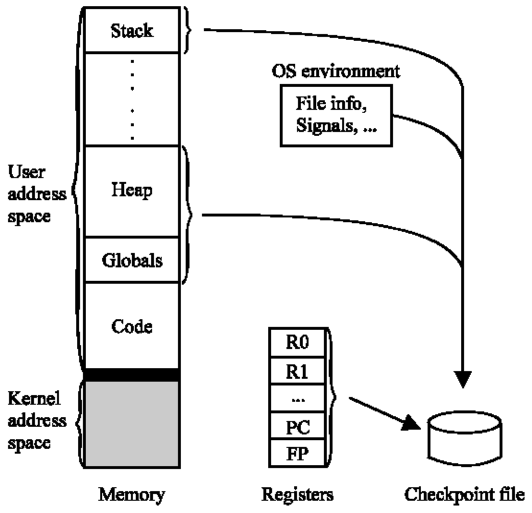

Fig. 1: Contents of a checkpoint

The second issue is the contents of the checkpoint as shown in Fig. 1. The minimal required information of the state should be saved so that the process can be restarted in case of an error.

The third issue is the location of checkpointing. It is either static or dynamic. While dynamic is determined at run time, static is predefined in design time.

Finally, the last issue is the methodology used for checkpointing. It depends on the architecture of the system. Methods used in multiprocessor systems should incorporate explicit coordination. For message-passing systems[5], the messages should be monitored and if necessary saved as part of the global context. The reason is that the messages introduce dependencies among the processors. On the other hand, a shared memory system communicates through shared variables which introduce dependency among the nodes and thus, at the time of checkpointing, the memory has to be in a consistent state to obtain a set of concurrent checkpoints[9].

RELATED WORK

Although the literature is full of proposals on checkpointing approaches, up to the knowledge of the authors there is little has been done regarding the comparative study between the two approaches (Table 1).

Table 1: Summary of related work listed by topic

Subject	Reference
Application	[6]
Diskless	[15,16,19,20,22,23]
Disk-based	[2,17]
DSM	[2,11]
Message Passing	[2,5,8,11,24]
Coordinated Ckpt	[1,10-12,14,22]
Independent Ckpt	[7,14,17]
Two level recovery	[4,10,13]
Storage location	[8]
Checkpointers	[7,8,18,24,25,27]
Optimization	[8,22]
Consistency	[3,10,17,22]
Experiment	[5,14,17,19,20,23,26]
Ckpt library	[6]
Algorithm analysis	[10,12]

DISK-BASED APPROACH

Disk-Based checkpointing for parallel and distributed applications[5,10-13] is the process of saving a global application state during execution on a stable storage. In case of a single system failure, the application could be brought back to a consistent state before the failure from the last consistent checkpoint taken on all nodes. A state is consistent if no orphan messages are recorded in that state. For example, if the receiving event of a message is recorded in a state s, then the sending event of the message is also given by Silva and Silva[14].

The methodology used for checkpointing depends on the architecture of the parallel and distributed systems, which could be classified into two major categories[11]: Message Passing (PM) Systems[15] and Distributed Shared Memory (DSM) Systems[16].

A distributed system consists of several processes executing on different nodes that communicate with each other via message passing[5]. For Message Passing systems (MP), the major difference in checkpointing algorithms is based on whether coordination is done at runtime or at recovery time.

Coordinated approaches[1,14] coordinate the nodes and form a consistent state at runtime, whereas independent algorithms form a consistent state only at recovery time. A third algorithm, which is a mix of the two to avoid rollback propagation, was suggested by Zambonelli[17], Kalaiselvi and Rajaraman[11].

In the independent approach, application processes are allowed to establish checkpoints in an independent way and no synchronization is enforced between their checkpoint operations. When there is a failure, the system will search in stable storage and will try to find some set of local checkpoints that, taken together, correspond to a consistent state of the application. This requires each process to keep several checkpoints in stable storage and there is no certainty that a global consistent state can be built. This approach has the following two main advantages:

a. There is no need to exchange any protocol messages during a checkpoint operation.

b. If the execution of the processes is completely asynchronous, the system can checkpoint one process at a time and this may reduce the bandwidth produced by the checkpoint I/O data.

The two main disadvantages of this approach are:

a. The possibility of occurrence of domino effect during the rollback operation. That is the rolling-back of one process causes another process to rollback, which in turn causes another process to rollback and ultimately rolling-back to the initial state of computation.
b. The storage overhead, since several checkpoints have to be maintained in stable storage[14].

In the coordinated checkpointing approach, a global checkpoint is taken periodically and the processes have to synchronize between themselves to assure that their set of local checkpoints corresponds to a consistent state of the application. A consistent state is achieved during run-time, while in the independent schemes the determination of a consistent recovery line was left to the recovery phase, which could result in some rollback propagation. In coordinated checkpointing the recovery phase is simple and quite predictable since all the processes rollback to their last committed checkpoints. This approach presents a lower overhead in stable storage. Some overhead is introduced during the checkpoint operation due to the synchronization of processes and some people argue that independent checkpointing is much better than coordinated checkpointing because of the overhead that is caused by this synchronization.

In the mixed independent-coordinated checkpointing approach (quasi-synchronous), a global checkpoint is similar to the approach of coordinated checkpointing while rollback propagation can be avoided by forcing additional un-coordinated local checkpoint in processes[17]. In between two coordinated checkpoints, messages are logged like the independent approach so that the rollback recovery is restricted to just the faulty processor. Periodically, checkpoints are initiated to maintain a consistent global state in stable storage at all times. As in the coordinated approach, a consistent recovery line is always maintained in stable storage by selectively logging messages and initiating checkpoints when necessary. Instead of the usual garbage collection techniques followed in independent approaches, this algorithm always maintains the latest checkpoint and makes sure that the rollback will not go beyond the latest checkpoint of a node[11].

Most available algorithms are based on Chandy and Lamport[18] algorithm. This algorithm uses some assumptions. One of these basic assumptions is that the global state of the system includes the local states of the processors and the state of the communication channels. A general description of Chandy and Lamport[18] algorithm is given below.

In this algorithm, the global state is constructed by coordinating all processors and logging the channel states at the time of checkpointing. Special messages called markers are used for coordination and for identifying the messages originating at different checkpoint intervals. The algorithm is initiated by a centralized node. The steps followed after a checkpoint initiation, however, are the same in all the nodes except that a centralized node initiates checkpoint on its own and the other nodes initiate checkpoints as soon as they receive a marker[11]. The steps are as follows:

1. Save the local context in stable storage;
2. For I = 1 to all outgoing channels do send markers along channel I;
3. Continue regular computation;
4. for I = 1 to all incoming channels do save incoming messages in channel I until a marker is received along that channel.

A large number of algorithms have been published in this area by relaxing some assumptions made in this Chandy and Lamport algorithm and by extending it to minimize the overheads of coordination and context saving[11].

The main advantages of MP approach[19] are: simplicity, increasing the level of reliability, portability and toleration of wholesale failure and little memory and network overhead[20].

The main disadvantages of MP approach[19] are performance bottleneck in writing checkpoint to disk and increase of computation time of application.

Distributed Shared Memory (DSM) provides a shared memory programming abstraction on a network of machines for which it is generally considered easier to write programs than for message passing models. If several processes in a message passing system need to share data, the programmer must explicitly send the data to each process. Writing a program to efficiently distribute the data among processes and keep each process updated can be difficult and tedious. In a DSM system, processes access shared data the same way they access regular memory. Changes to shared data are propagated to the processes that need them by the DSM[2,16].

Distributed Shared Memory systems have global address space where the memory is distributed across all the nodes. It is a software layer that provides the appearance of a shared-memory system to the user and internally communicates through messages. Caches are present to minimize latency in data access and the programming paradigm is a shared-memory model. DSM systems should concentrate on making the memory consistent at the time of checkpointing. To tolerate node failures, checkpoints should be maintained in stable storage like MP systems[11].

The goal of checkpointing a distributed system is to save a consistent global state from which the computation can be restored in the event of a failure. For checkpointing DSM systems, the meaning of a consistent global state changes depending on the memory consistency model of the system. As with message passing systems, the two main approaches to checkpointing DSM systems are coordinated checkpointing and independent checkpointing.

Coordinated checkpointing can be used for DSM in a way similar to coordinated checkpointing for message passing. All processes synchronize and record their states as well as in transit messages. As with message passing systems, only one checkpoint must be stored and there is no extra overhead for logging messages. To recover from a failure, all processes rollback to the most recent checkpoint. This type of coordinated checkpointing can be improved by taking advantage of the nature of DSM systems to reduce checkpointing overhead. One of the overheads of checkpointing is the time required to write data to stable storage. Rather than storing the state of shared memory to stable storage, which can be time consuming, one coordinated checkpointing scheme guarantees that each page is replicated on at least two processors during a checkpoint. If one processor fails, the pages stored on that processor will have been replicated in at least one other processor's memory. This technique will not handle more than one processor failure at a time. Experimental results show that as the number of processors in the system is increased, the overhead due to replicating pages is decreased. In some cases extra replication of a page can improve an application's performance because some pages will be fetched before they are needed. However, this depends on the program's behaviour. Unfortunately, as with message passing systems, coordinating all of the processors to take a checkpoint can incur considerable overhead. For programs that frequently use barriers across all processors for synchronization, the checkpointing system can wait until a barrier to take a checkpoint. While all processes are waiting at a barrier, but before any of them leave, the state of each process is saved. The state of the system is guaranteed to be consistent because all processes are waiting at the barrier. Though this works well for programs that use barriers regularly, not all programs use barriers and barriers are expensive in large-scale systems.

Independent checkpointing techniques have the advantage that no costly coordination is required. Given that distributed shared memory systems are built on an underlying message passing facility, one approach to implement independent checkpointing for DSM systems is to directly use one of the previously described message passing checkpointing methods on the underlying message passing. However, DSM systems tend to send significantly more messages than message passing systems and many of the messages do not cause dependencies between processors. Recoverable DSM systems have been developed that reduce the amount of tracing by further reducing the number of messages that constitute data dependencies. This system uses a variation of the fixed distributed manager protocol, which prevents request messages to the page manager and messages from the page manager to the owner of a page from becoming dependencies. Reducing the number of dependencies reduces not only the amount of information that needs to be tracked but also the probability of rollback propagation. In message passing systems, rollback propagation can be eliminated by logging messages. During recovery, messages are replayed from the message log when the program executes a receive call. However, there are no explicit receive calls in a DSM system. Page update and invalidate messages arrive at a processor at unpredictable times. A straight-forward solution to the problem of unpredictable updates or invalidate message arrivals in a sequentially consistent system is to checkpoint when any write to shared data is available to another processor. No processor will see any shared data, which is not stored in the most recent checkpoint of the processor that wrote it.

Considering the facts mentioned above, a basic DSM checkpointing algorithm is as follows:

1. At the time of checkpointing, make the distributed main memory consistent through the memory management protocols.
2. Save the process contexts in the memory.
3. Save the global state in secondary storage.

Techniques to overlap the context-saving process with computation are possible to reduce the overhead of storing context in stable storage. The advantage of DSM approach is that it could overlap context saving with computation. On the other hand, the disadvantages of DSM approach are large memory and network overhead and the required system software support.

DISKLESS APPROACH

Checkpointing operates by saving all the information needed to restart a process. Disk-based checkpointing saves the information on stable storage like a disk. This information includes all variables, the environment, control information and register values. So, it consumes a lot of time and it is a performance bottleneck. The problem has more effect in systems that have many more processors than disks[15,20,21] .

Disk-based checkpointing has high cost and this limits the number of checkpoints that can be established. Also, it has network and disk overhead and this makes a computation take more time than the normal case. So, the goal of diskless checkpointing is to make computation faster, eliminate the overhead of saving on disk and finally utilize the available memory[15,20,21].

Diskless checkpointing is a technique of checkpointing that is based on main memory. It is based on coordinated checkpointing, a collection of processors with memories coordinate to have checkpoint of the process state[15,20,21].

There are two main memory checkpointing schemes that can be used without any hardware changes: neighbor-based checkpointing and parity-based checkpointing. While the neighbor-based checkpointing scheme saves the checkpoints in the main memory of other processors, parity-based checkpointing is based on a parity approach[19].

The main advantages of diskless checkpointing are: It has less checkpoint latency and recovery time; it reduces the usage of shared resources; it uses and utilizes the available memory to save checkpoints; it does not require any additional hardware and it is faster than the disk-based approach. Also, it provides a better expected running time[19,21,22].

The main disadvantages of diskless checkpointing are: it has encoding, memory, CPU and network overhead and it cannot tolerate the occurrence of a global failure of the machine. Rather, it can be mainly used to tolerate single processor failure. So, it has less failure coverage than checkpointing to stable disk, since none of the components in a diskless checkpointing system can survive a wholesale failure[15,19,20].

Neighbor-based checkpointing: It is a checkpointing scheme that avoids using disk to write checkpoint. Rather, it uses the main memory of neighbor processors. Processors are organized in a virtual ring. Each processor saves its checkpoint into its physical memory, snapshot area and into the neighbor processor that follows on the ring. So, this scheme can tolerate single failures. Actually, it can tolerate more than one failure provided that the failures do not occur in adjacent processors of the virtual ring[19].

Although this scheme is simple, there are some scenarios that show that this scheme is not robust against failures that occur during the checkpointing protocol[19].

So, the old checkpoints should be kept in order to tolerate the occurrence of failures during the checkpointing protocol. 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 neighbor. The following steps are required to have a good tolerance of failures: save the checkpoint into the local snapshot area of each processor and then send the checkpoint to the next processor of the ring. Although the application process is blocked during the first step, 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. So, we can consider that this solution requires extra memory twice the size of the application state[19].

Neighbor-based checkpointing scheme should not be used alone, since it is not able to recover from total failures of the system. Thus, the system should take from time to time a global checkpoint to disk. (Back to disk-based). Also, it costs higher memory overhead although it has lower overhead per checkpoint[19].

Parity-based checkpointing: The basic idea is to avoid disk writing and maintain enough redundant information about the checkpoint data able to tolerate a single processor failure. So, the application should be able to checkpoint far more frequently than when checkpoints are saved on disk[19]. It consists of two parts: locally checkpointing by each processor to its memory and encoding checkpoints and storing the encoding in checkpoint processors dedicated for storing checkpoints[22].

The steps when the failure occurs are: the non-failed application processors rollback to the checkpoint stored in their memory. Then, replacement processors are chosen to replace the failed processors. Finally, there comes the rollback to the checkpoint calculated from the checkpoints of non-failed processors and encoding in checkpoint processors.

After that, the application can restart. If there are not enough spare processors left, checkpoint processors can be used as replacement processors. It is assumed that the memories in processors are disjoined and communication is by message passing only. The application needs n application processors and m extra processors are given as checkpoint processors. When there are less than n processors available, the application has to be terminated[22]. The components of parity-based checkpointing include local checkpointing, encoding checkpoints and integration of local and encoding checkpointing.

a- Local checkpointing: Each processor only needs to store its checkpoint in memory rather than on disk. There are three methods:

Simple checkpointing: The simplest form is to have a copy of the address space and registers. In its simplest form diskless checkpointing requires an in-memory copy of the address space and registers. If a rollback is required, the contents of the address space and registers are restored from the in-memory checkpoint.

Note that this checkpoint will not tolerate the failure of the application processor itself. It simply enables the processor to rollback to the most recent checkpoint if another processor fails.

One drawback of simple diskless checkpointing is memory usage. A complete copy of the application must be retained in the memory of each application processor. A solution to this problem is to use incremental checkpointing[15,20,22].

Incremental checkpointing: Another method is incremental diskless checkpointing. Virtual memory protection bits of all pages are set to read-only and will be set to read-write after page fault caused by a write by the application. Then the checkpointing system stores a copy of the faulty page. Thus, the processor's checkpoint consists of the read-only pages in the address space and the stored faulty pages[15,22].

Forked checkpointing: The last method is forked diskless checkpointing. To checkpoint, the application clones itself (with, for example, the fork() system call in Unix). The clone is the diskless checkpoint. The checkpoint stored in the memory is only for the non-failed processor to rollback when another processor fails, but not for a failed processor to recover, i.e. it is not tolerant of the failure of the application processor itself[15,22].

b- Encoding checkpoints: The m extra processors are used for encoding the checkpoints from all application processors and storing useful information so that the checkpoints of the failed processors can be re-calculated. There are a number of methods of encoding. The main methods are parity, mirroring, one-dimensional parity, two-dimensional parity and Reed-solomon.

Parity (RAID level 5): The simplest method is parity (RAID level 5). Only one checkpoint processor is needed to encode bitwise parity of all processors' checkpoints. The jth byte of the checkpoint processor is the result of exclusive-OR operation on all jth bytes of all application processors. When a processor fails, the checkpoint of the failed processor can be obtained from the exclusive-OR operation on checkpoints stored in checkpoint processor and non-failed application processors. This is the same recovery method used in RAID level 5 in disk array technology[15,22].

Mirroring: Another method is checkpoint mirroring. With m=n extra checkpoint processors, the checkpoint of each application processor is copied to a corresponding checkpoint processor. Failure of an application processor can be recovered by copying the checkpoint stored in the corresponding checkpoint processor. This scheme can tolerate n processor failures, but not failure of any pair of application processor and its corresponding checkpoint processor[15,22].

One-dimensional parity: With one-dimensional parity, there are $1 \le m \le n$ checkpoint processors. The application processors are partitioned into m groups, $g_1, \ldots, g_m$ of roughly equal size. Checkpoint processor I then calculates the parity of the checkpoints in group I. This increases the failure coverage, because now one processor failure per group may be tolerated. Moreover, the calculation of the checkpoint encoding should be more efficient because there is no longer a single bottleneck (the checkpoint processor). Note that 1-dimensional parity reduces to RAID level 5 when m=1 and to mirroring when m=n[15,20].

Two-dimensional parity: Two-dimensional parity is an extension of one-dimensional parity. With two-dimensional parity, the application processors are arranged logically in a two-dimensional grid and there is a checkpoint processor for each row and column of the grid. Each checkpoint processor calculates the parity of the application processors in its row or column. Two-dimensional parity requires $m \ge 2\sqrt{n}$ checkpoint processors and can tolerate the failure of any one processor in each row and column. This means that any two processor failures may be tolerated[15,20].

Reed-solomon: Finally, the most general-purpose encoding technique is Reed-solomon coding, which requires m checkpoint processors to use Galois Field arithmetic to encode the checkpoints in such a way that any m processor failure can be tolerated. This coding is more complicated and the overhead is larger, but its failure coverage per checkpoint processor is the largest[15,22].

c- Integration of local and encoding checkpointing: One of the most important things in checkpointing is performance[14,23]. The checkpoint overhead comes from the sending and calculating of encoding. This subsection considers how to reduce the overhead caused by inter-network communication for the checkpointing system. In some encoding schemes mentioned previously, there is a bottleneck caused by the fact that checkpoint processors are the destinations of all checkpoint messages and so they have to receive the checkpoint information sent by all processors. Besides, they have to do all the encoding calculations. One solution is FAN-IN method. Application processors do the encoding in log n steps and send the final result to the checkpoint processors. FAN-IN method is usually preferable except that the network supports multicast (e.g. Ethernet).

Another approach to reduce the network communication is to reduce the message size. Using an approach similar to incremental checkpointing, only pages are modified since the most recent checkpoint is sent to checkpoint processors. The changes above are called as diff, which is bitwise exclusive-OR (XOR) of the current copy of the page and the copy of the page in the previous checkpoint. The diff is sent to checkpoint processor, which XOR's it into its checkpoint. The message size can be reduced further by compressing diff before sending it[22].

RESULTS AND DISCUSSION

In present experimental study, compared the performance of the coordinated disk-based checkpointing algorithm with parity diskless checkpointing. Four PIII machines running at 700 MHZ were used in the experiment to implement the disk-based checkpointing algorithm. Each machine has 256 MB RAM and 40 GB disk space. The four machines are connected to each other through 100 MB Ethernet network. A fifth identical machine was used as diskless parity processor.

Several checkpointing packages do exist. Examples of such systems include: MIST[9], CLIP[24], Dynamite[25], MigThread[26], Fail-Safe PVM[7] and CoCheck[8,27].

Dynamite[25], a transparent checkpointing system, is used for disk-based experiment and a modified version for diskless checkpointing. Dynamite is a user-level checkpointing system. It is part of dynamic load-balancing environment that provides support for parallel processes running applications under parallel virtual machine (PVM) and message passing interface (MPI). It supports migration of tasks between nodes in a manner transparent to both the programmer as well as the user. It provides support for applications running under Solaris and Intel Linux operating systems. In present experiment we used Linux operating system environment. Dynamite checkpointing system runs on top of PVM. It does not require any re-compilation or re-linking of the application. Dynamite consists of the following three main components:

- A scheduler to manage workload distribution.
- A monitor to monitor the workload on workstations.
- A checkpointing/migration mechanism.

Two parallel applications are used to test the performance of the checkpointing systems[14,23]. The first application is called Nbody, which computes n-body interaction between particles in a system. 20,000 particles and 20 interactions were used. The second one is called MAT that multiplies two square floating-point matrices. A matrix size of 4096x4096 was used.

The experiment was carried out six times per application for the two approaches. Table 2 shows the latency in seconds along with the mean and standard deviation.

Checkpoint latency is the time needed to save the checkpoint. It observed that there is a big difference between the latency in the two approaches. Diskless approach is superior to disk-based in latency point of view. We think this is due to the time of saving checkpoints to the disk in disk-based approach.

Table 3 shows the overhead in seconds as well as the mean and Standard Deviation. The results are for the whole application.

Checkpoint overhead refers to the time required to record the checkpoint. The two approaches have comparable overhead because it is independent of the way of saving the checkpoint. It is also important noting that the recovery time of diskless checkpointing is roughly the same as its checkpoint latency.

Compared with disk-based approach, the overhead of diskless checkpointing is comparable to that of disk-based checkpointing, while its checkpoint latency and recovery time is much lower than disk-based checkpointing. It is observed that disk-based checkpoint latency and recovery time is much greater than its counterpart diskless checkpointing. This is due to the slow disk speed that limits the transfer rate of checkpoint to the disk during checkpointing and from the disk during

Table 2: Latency (sec) when running the two applications under disk-based and diskless approaches

	Disk-based		Diskless	
Trial No.	Nbody	MAT	Nbody	MAT
1	1010.000	1950.00	14.00	25.00000
2	1092.000	1890.00	15.00	23.00000
3	1101.000	1972.00	13.00	24.00000
4	1005.000	1961.00	16.00	26.00000
5	1000.000	1890.00	12.00	25.00000
6	1015.000	1917.00	17.00	22.00000
Mean	1037.167	1930.00	14.50	24.17000
SD	46.32	36.04	01.87	1.47196

Table 3: Overhead (sec) when running the two applications under disk-based and diskless approaches

	Overhead (sec)			
	Disk-based		Diskless	
Trial No.	Nbody	MAT	Nbody	MAT
1	26.00	20.00	20.00	25.00
2	25.00	21.00	21.00	27.00
3	23.00	20.00	23.00	26.00
4	27.00	19.00	22.00	28.00
5	30.00	21.00	21.00	24.00
6	24.00	20.00	20.00	19.00
Mean	25.83	20.17	21.17	24.83
SD	2.48	0.75	1.17	3.19

Table 4: Comparison between disk-based and diskless checkpointing systems

Parameter	Disk-based	Diskless
Latency time	High	Low
CPU overhead	High	High
Memory requirement	Low	High
Stable storage requirements	High	Low
Toleration of wholesale failure	Yes	No
Reliability	High	Low
Efficiency	Low	High
Addition hardware	Not required	Additional processors
Portability	High	Low

recovery. The expected running time of the application depends heavily on the checkpoint overhead, latency and recovery time. Thus, it is obvious that diskless checkpointing has a much shorter expected running time. Another disadvantage of the large checkpoint latency is degradation of performance of shared resource.

In the case of disk-based checkpointing, the shared resource generally refers to the shared disk, whereas it is the network for diskless checkpointing. In the experimental results, the performance of the stable storage degrades by 87%. With much less checkpoint latency, diskless checkpointing gives slight degradation on network performance[22].

As a result of several studies, a summary of the main differences between the disk-based and diskless checkpointing systems is shown in Table 4.

CONCLUSIONS

Two approaches of checkpointing for parallel and distributed applications are empirically studied and analyzed. From the present study it may be concluded that the latency of disk-based approach is a bottleneck while the overhead of both approaches is comparable.

The use of optimization techniques, like main memory checkpointing and checkpoint staggering, reduces the performance overhead of our coordinated algorithm considerably. Whenever, possible the system should try to perform the checkpoint concurrently with the computation and use an ordering technique to reduce the congestion on stable storage.

Choosing between the two approaches studied in this research depends on whether one can sacrifice the additional memory and processors to get better performance and less reliable using diskless approach, or sacrifice the performance to get reliable system in the case of disk-based approach.

If more frequent checkpoints are required, then it is more efficient to choose the diskless approach since it takes less time to checkpoint. For high critical applications, where the reliability is more important, the disk-based approach is preferable since it saves the checkpoints on a stable storage.

FUTURE WORK

In this research work, an experimental study is made between coordinated disk-based and parity diskless checkpointing algorithms. As a future study it is possible to extend this study to include independent disk-based approach, mixed independent coordinated approach and neighbor-based checkpointing.

Another future possibility is to take the advantages of both approaches and use a two-level stable storage approach. This approach combines the advantages of the two approaches. It combines the reliability of the disk-based approach with the efficiency of the diskless approach. In the two-level stable storage approach, the saving of the state of the application to a stable storage can be overlapped with the process of checkpointing of the application state to main memory. This scheme is efficient and provides fast recovery. If there is a partial failure in the system, the application can recover from the main memory checkpoint. On the other hand, if there is a total failure, the application is restarted from a checkpoint saved on stable storage.

ACKNOWLEDGMENT

The publication of this study was supported by Yarmouk University Research Council.

REFERENCES

1. Bouteiller, A., P. Lemarinier, G. Krawezik and F. Cappello, 2003. Coordinated checkpoint versus message log for fault tolerant MPI. IEEE International Conference on Cluster Computing (CLUSTER'03), Hong Kong, December 01-04, 2003, pp: 242-250.
2. Dieter, W.R. and J.E. Lumpp, 1997. Fault recovery for distributed shared memory systems. Proceeding, 1997 IEEE-Aerospace Conference.
3. Helary, J.M., R.H.B. Netzer and M. Raynal, 1999. Consistency issues in distributed checkpoints. IEEE Trans. Software Eng., 25: 274-281.
4. Vaidya, N.H., 1998. A case for two-level recovery schemes. IEEE Transactions on Computers, 47: 656-666.
5. Sancho, J.C. *et al.,* 2004. On the feasibility of incremental checkpointing for scientific computing. Proceeding International Parallel and Distributed Processing Symposium IPDPS'04, Santa Fe, NM, USA. http://www.cs.huji.ac.il/~etcs/pubs/papers/ipdps04.pdf
6. Wang, Y.M. *et al.,* 1995. Checkpointing and its applications. IEEE Fault-tolerant Computing Symposium, FTCS-25, June 1995, pp: 22-31.
7. Kofahi, N.A. and Q.A. Rahman, 2004. Empirical study of Variable Granularity and Global Centralized Load Balancing Algorithms. Proceeding International Conference Parallel and Distributed Processing Techniques and Applications (PDPTA'04), Las Vegas, Nevada, USA, June 21-24, 2004, CSREA Press, 1: 283-288.
8. Pruyne, J. and M. Livny, 1996. Managing checkpoints for parallel programs. Proceeding 2nd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'96), LNCS, of Lecture Notes in Computer Science, 1162: 140-154.
9. Casas, J. *et al.,* 1995. MIST: PVM with transparent migration and checkpointing. In: 3rd Annual PVM Users' Group Meeting, Pittsburgh, PA.
10. Gendelman, E., L.F. Bic and M.B. Dillencourt, 1999. An efficient checkpointing algorithm for distributed systems implementing reliable communication channels. Proceeding of 18th IEEE Symposium Reliable Distributed System, Switzerland, pp: 290-291.
11. Kalaiselvi, S. and V. Rajaraman, 2000. A survey of checkpointing algorithms for parallel and distributed computers. In: SaÅdhanaÅ, 25: 489-510.
12. Neogy, S., A. Sinha and P.K. Das, 2002. Distributed checkpointing using synchronized clocks. Proceeding 26th Annual International Computer Software and Applications Conference, COMPSAC'02, pp: 199-206.
13. Silva, L.M. and J.G. Silva, 1998. Using two-level stable storage for efficient checkpointing. IEE Proceeding Software Engineering, Volume 145 No. 6, December 1998.
14. Silva, L.M. and J.G. Silva, 1999. The performance of coordinated and independent checkpointing. Proceeding 3rd International Parallel Processing Symposium and 10th Symposium Parallel and Distributed Processing. San Juan, Puerto Rico, April 12-16, pp: 280-284.
15. Plank, J.S., K. Li and M.A. Puening, 2001. Diskless checkpointing. presentation, November 2001, http://www.cs.cornell.edu/Courses/cs717/2001fa/lectures/Diskless.ppt
16. Carothers, C.D. and B.K. Szymanski, 2002. Linux support for transparent checkpointing of multithreaded programs. http://www.cs.rpi.edu/~szymansk/genesis/ch.pdf
17. Zambonelli, F., 1998. Distributed checkpoint algorithms to avoid roll-back propagation. 24th Euromicro Conference Proceedings, 1: 403-410.
18. Chandy, K.M. and L. Lamport, 1985. Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comp. Sys., 3: 63-75.
19. Silva, L.M. and J.G. Silva, 1998. An experimental study about diskless checkpointing. 24th Euromicro Conference Proceeding, August 25-27, 1998, 1: 395-402.
20. Plank, J.S., K. Li and M.A. Puening, 1998. Diskless checkpointing. IEEE Trans. Parallel and Distributed System, 9: 972-986.
21. Bowman, I.T., 1998. Diskless checkpointing, presentation, CS756B. http://plg.uwaterloo.ca/~itbowman/CS756B/P2/
22. Hung, E., 1998. Fault tolerance and checkpointing schemes for clusters of workstations. ELEC6062 Scalable Parallel Computing, Project Report: http://www.cs.umd.edu/~ehung/research.htm.
23. Chiueh, T. and P. Deng, 1996. Evaluation of checkpoint mechanisms for massively parallel machines. Proceeding of 26th Fault-Tolerance Computer Symposium, FTCS-26, Japan, pp: 370-379.
24. Chen, Y. *et al.*, 1997. CLIP: A checkpointing tool for message-passing parallel programs. In: SC97: High Perf. Networking and Comp.

25. Iskra, K.A. *et al.*, 2000. The implementation of Dynamite: An environment for migrating PVM tasks. Association for computing machinery. Special Interest Group on Operating Systems. Operating Systems Review, 34: 40-55.
26. Jiang, H. and V. Chaudhary, 2004. Process/Thread Migration and checkpointing in heterogeneous distributed systems. Proceeding of 37th Hawaii International Conference System Sciences. http://csdl.computer.org/comp/proceedings/hicss/2004/2056/09/205690282b.pdf
27. Stellner, G., 1996. CoCheck: Checkpointing and process migration for MPI. In: 10th International Parallel Processing Symposium, pp: 526-531.