On Staggered Checkpointing*

Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
Phone: 409-845-0512
Fax: 409-847-8578
E-mail: vaidya@cs.tamu.edu

Key words: Consistent checkpointing, staggered checkpointing, checkpoint overhead, rollback recovery, experimental evaluation.

Abstract

A "consistent checkpointing" algorithm saves a consistent view of the distributed system state on stable storage. The loss of computation upon a failure can be bounded by taking consistent checkpoints with adequate frequency.

The traditional consistent checkpointing algorithms require the different processes to save their state at about the same time. This causes contention for the stable storage, potentially resulting in large overheads. *Staggering* the checkpoints taken by various processes can reduce the overhead. Some techniques for *staggering* the checkpoints have been proposed previously [11], however, these techniques result in "limited staggering" in that not all processes' checkpoints can be staggered. Ideally, one would like to stagger the checkpoints *arbitrarily.*

This paper presents a simple approach to *arbitrarily stagger* the checkpoints. Our approach requires that the processes take consistent *logical* checkpoints, as compared to consistent *physical* checkpoints enforced by existing algorithms. This paper presents experimental results using an implementation on a nCube-2 multicomputer.

1 Introduction

Applications executed on a large number of processors, either in a distributed environment, or on multicomputers such as nCube, are subject to processor failures. Unless some recovery techniques are utilized, a processor failure will require a restart of the application, resulting in significant loss of performance.

*This research is supported in part by National Science Foundation grant MIP-9502563 and Texas Advanced Technology Program grant 009741-052-C.

Consistent checkpointing is a commonly used technique to prevent complete loss of computation upon a failure [1, 3, 5, 8, 10, 11, 13, 15]. A "consistent checkpointing" algorithm saves a consistent view of the distributed system state on a stable storage. The loss of computation upon a failure is bounded by taking consistent checkpoints with adequate frequency.

The traditional consistent checkpointing algorithms require the different processes to save their state at about the same time. This causes contention for the stable storage, potentially resulting in significant performance degradation [11]. *Staggering* the checkpoints taken by various processes can reduce the overhead of consistent checkpointing by reducing stable storage contention. Some techniques for *staggering* the checkpoints have been previously proposed [11], however, these techniques result in "limited" staggering in that not all processes' checkpoints can be staggered. Ideally, one would like to stagger the checkpoints *arbitrarily.* If processors are able to make an "in-memory" copy of entire process state, then checkpoint staggering is trivial. This paper considers systems where it is not feasible to make an "in-memory" copy of entire process state. This situation may occur because: (i) memory size is small, or (ii) the memory may be shared by processes of multiple applications – making in-memory copy of a process from one application may cause processes from other applications to be swapped out (degrading their performance).

This paper presents a simple approach to "arbitrarily stagger" the checkpoints. Our approach requires that the processes take consistent *logical* checkpoints, as compared to consistent *physical* checkpoints enforced by existing algorithms for *staggering.* This paper discusses the proposed approach and presents experimental results (on nCube-2 multicomputer).

The paper is organized as follows. Section 2 discusses the related work. Section 3 discusses the notion of a *logical checkpoint.* Section 4 presents a consistent checkpointing algorithm proposed by Chandy and Lamport [1] and Plank [11]. Section 5 presents the proposed algorithm. Section 6 presents experimental results. Some variations of the proposed scheme are discussed in Section 7. Section 8 concludes the paper.

2 Related Work

The algorithm presented here is closely related to [1, 10, 11, 14]. As discussed later in the paper, the proposed algorithm combines two different techniques: "staggered" checkpoints and consistent "logical" checkpoints to obtain an algorithm with better performance than existing

schemes. These terms will be explained later – the reader may want to revisit this section after reading the rest of this paper.

Although the proposed scheme uses message logging, our work differs from past work in that our scheme only logs those messages that are needed to make the "staggered" checkpoints consistent. In contrast, previous message logging schemes (e.g., [5]) log all messages, and determine a consistent state after a failure occurs.

Plank [11] was the first to observe that stable storage contention can be serious problem for consistent checkpointing, and suggested checkpoint staggering as a solution. Plank presented a consistent checkpointing algorithms similar to Chandy-Lamport [1] that attempts to stagger the checkpoints. However, often some checkpoints taken by this algorithm are not staggered. In contrast, our algorithm allows arbitrary staggering of the checkpoints. Plank [11] also presents another approach for staggering checkpoints, that is applicable to wormhole routed networks. (This algorithm also does not permit arbitrary staggering.) As our algorithm applies to all networks, we do not consider Plank's second algorithm for comparison purposes.

Long et al. [10] discuss an interesting approach, named *evolutionary* checkpointing, that is similar to *logical checkpointing*. The fundamental difference between the two approaches is that our algorithm *staggers* the checkpoints, while the scheme in [10] does not allow staggering. By enforcing staggering, our approach can perform better. Long et al. also assume *synchronized communication* and an upper bounds on communication delays; no such assumptions are made in the proposed approach.

Wang et al. [14, 15] introduced the notion of a *logical* checkpoint. They present an algorithm to determine a recovery line consisting of consistent logical checkpoints, *after* a failure occurs. This recovery line is used to recover from the failure. Their goal is to determine the "latest" consistent recovery line using the information saved on the stable storage. During failure-free operation each process is allowed to independently take checkpoints and log messages. On the other hand, our scheme *coordinates* logical checkpoints *before* a failure occurs. These logical checkpoints are used to recover from a *future* failure. One consequence of this is that we do not need to log all messages, only those message are logged which make the staggered checkpoints consistent.

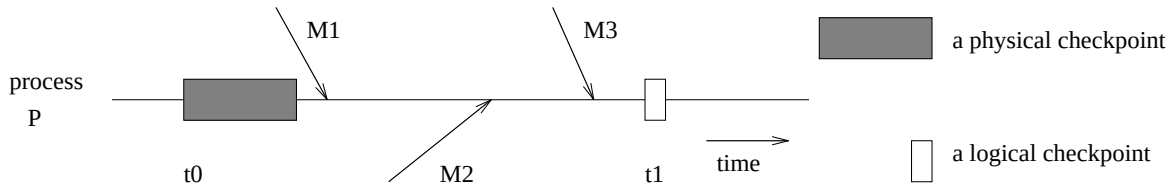

Figure 1: Physical checkpoint + message log = logical checkpoint

3 A *Logical* Checkpoint

A process is said to be *deterministic* if its state depends only on its initial state and the messages delivered to it [5, 12]. A *deterministic* process can take two types of checkpoints: a *physical* checkpoint or a *logical* checkpoint. A process is said to have taken a *physical* checkpoint at some time t_1, if the process state at time t_1 is saved on the stable storage. A process is said to have taken a *logical* checkpoint at time t_1, if *adequate* information is saved on the stable storage to allow the process state at time t_1 to be recovered.

To the best of our knowledge, the term *logical* checkpoint was first introduced by Wang et al. [14, 15], who also presented one approach for taking a logical checkpoint. Now we summarize three approaches for taking a logical checkpoint at time t_1. Although the three approaches are equivalent, each approach may be more attractive for some applications than the other approaches. Not all approaches will be feasible on all systems.

- One approach for establishing a logical checkpoint at time t_1 is to take a *physical* checkpoint at some time $t_0 \leq t_1$ and log (on stable storage) all messages delivered to the process between time t_0 and t_1. (For each message, the message log contains the *receive sequence number* for the message as well as the entire message.) This approach is essentially identical to that presented by Wang et al. [14].

  Figure 1 presents an example wherein process P takes a physical checkpoint at time t_0. Messages M1, M2 and M3 are delivered to process P by time t_1. To establish a logical checkpoint of process P at time t_1, messages M1, M2 and M3 are logged on the stable storage. As process P is deterministic, the state of process P at time t_1 can be recovered using the information on the stable storage (i.e., physical checkpoint at t_0 and messages M1, M2 and M3).

  We summarize this approach as:

  physical checkpoint + *message log* = *logical checkpoint*

- The essential purpose behind saving the messages above is to be able to recreate the state at time t_1, or to be able to "re-perform" the incremental changes made in process state by each of these messages. This may be achieved simply by taking a *physical* checkpoint at time t_0 and taking an *incremental* checkpoint at time t_1. The incremental checkpoint is taken by logging[1] the changes made to process state between time t_0 and t_1. We summarize this approach as:

  physical checkpoint + *incremental checkpoint* = *logical checkpoint*

  The scheme presented by Long et al. [10] takes checkpoints similar to above procedure, although they do not use the term *logical* checkpoint.

- The above two approaches take a physical checkpoint *prior* to the desired logical checkpoint, *followed* by logging of additional information (either messages or incremental state change).

  The third approach is the *converse* of the above two approaches. Here, the *physical* checkpoint is taken at a time t_2, where $t_2 > t_1$. In addition, enough information is saved to *un-do* the effect of messages received between time t_1 and t_2. For each relevant message (whose effect must be undone), an *anti-message* is saved on the stable storage. The notion of an *anti-message* here is similar to that used in time warp mechanism [4] or that of UNDO records [2] in database systems. Anti-message M* corresponding to a message M can be used to undo the state change caused by message M.

  Figure 2 illustrate this approach. A *logical* checkpoint of process P is to be established at time t_1. Process P delivers messages M4 and M5 between time t_1 and t_2. A *physical* checkpoint is taken at time t_2, and *anti-messages* corresponding to messages M4 and M5 are logged on the stable storage. The anti-messages are named M4* and M5*, respectively.

  To recover the state, say S1, of process P at time t_1, the process is initialized to the physical checkpoint taken at time t_2 and then anti-messages M5* and M4* are sent to the process. The order in which the anti-messages are delivered is reverse the order in which the messages were delivered. As shown in Figure 3, the final state of process P is identical to the state (or logical checkpoint) at time t_1.

  We summarize this approach as:

  anti-message log + *physical checkpoint* = *logical checkpoint*

[1]The term *logging* is used to mean "saving on the stable storage".

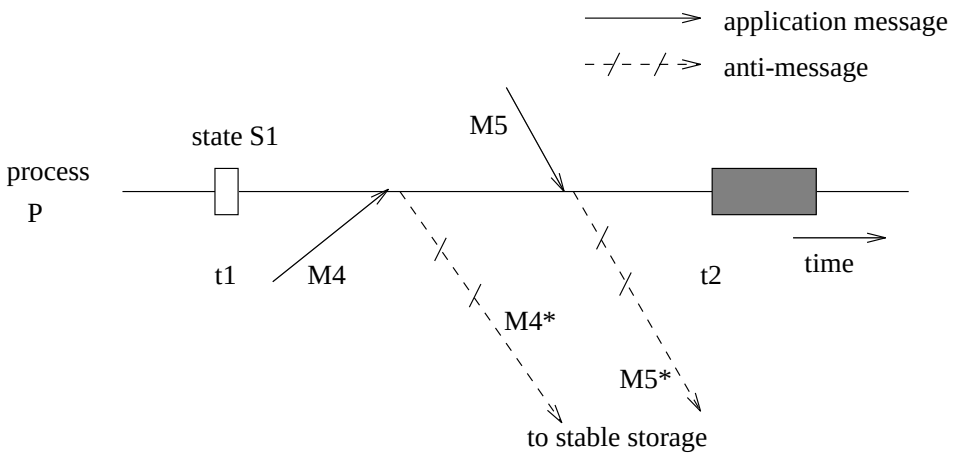

Figure 2: Anti-message log + physical checkpoint = logical checkpoint

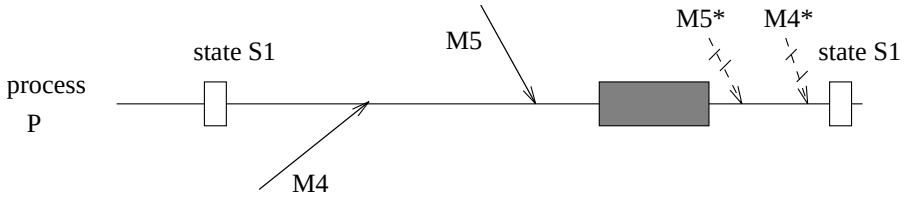

Figure 3: Recovering a logical checkpoint using anti-messages

An important issue is that of forming the "anti-messages". The anti-messages can possibly be formed by the application itself, or they may consist of a copy of the (old) process state *modified* by the message (similar to copy-on-write [9]). We have, as yet, not experimented with anti-messages. Therefore, practicality of this idea is open to debate.

Note that a physical checkpoint is trivially a logical checkpoint, however, the converse is not true.

4 Chandy-Lamport Algorithm [1] and Plank's Scheme [11]

Chandy and Lamport [1] presented an algorithm for taking a consistent checkpoint of a distributed system. Although the proposed approach can potentially be used with any consistent checkpointing algorithm, for brevity, we limit our discussion to the Chandy-Lamport algorithm.

Assume that the processes communicate with each other using unidirectional communication *channels*; a bidirectional channel can be modeled as two unidirectional channels. The communication graph is assumed to be strongly connected. The algorithm presented next is

essentially identical to Chandy-Lamport [1] and assumes that a certain process (named P_0) is designated as the *checkpoint coordinator*. This algorithm is also presented in [11].

Algorithm: The coordinator process P_0 initiates the consistent checkpointing algorithm by sending *marker* messages on each channel, incident on, and directed away from P_0 and immediately takes a *checkpoint*. (This is a *physical* checkpoint.)

A process, say Q, on receiving a *marker* message along a channel c takes the following steps:

if Q has not taken a checkpoint **then**
begin
Q sends a marker on each channel, incident on, and directed away from Q.
Q takes a checkpoint.
Q records the state of channel c as being empty.
end
else Q records the state of channel c as the sequence of messages received along c,
after Q had taken a checkpoint and before Q received the marker along c.

4.1 Checkpoint Staggering with Chandy-Lamport Algorithm

Plank [11] suggested that the processes should send markers *after* taking their checkpoints, rather than before taking the checkpoint (unlike the algorithm above). This simple modification introduces some staggering of checkpoints. However, not all checkpoints can be staggered.

In our experiments, we use the Chandy-Lamport algorithm that incorporates Plank's modification. In the rest of this paper, this modified algorithm will be referred to as Chandy-Lamport/Plank algorithm, or CL/P for brevity.

Observations: Plank [11] observed that his staggering schemes work better than the original "non-staggered" algorithm when (i) degree of synchronization amongst the processes is relatively small, and (ii) the message volume is relatively small.

5 Staggered Consistent Checkpointing

The extent of checkpoint staggering using CL/P algorithm is dependent on the application's communication pattern, and also on how the algorithm is implemented (e.g., whether the markers are sent synchronously or asynchronously). On the other hand, the proposed algorithm can stagger the checkpoints in any manner desired. Many variations are possible, depending on which checkpoints are desired to be staggered. As an illustration, we assume that the objective is to stagger *all* checkpoints, i.e., no two checkpoints should overlap in time. Later, we will illustrate a situation where some overlap in checkpointing is desired (when multiple stable storages are available).

The proposed algorithm (named STAGGER) can be summarized as follows:

staggered physical checkpoints + consistent logical checkpoints = staggered consistent checkpoints

The basic idea is to coordinate *logical* checkpoints rather than *physical* checkpoints. In this section, we assume that the first approach, described in Section 3, for taking logical checkpoints is being used. Thus, a logical checkpoint is taken by logging all the messages delivered to a process since its most recent physical checkpoint.

For the purpose of this discussion, assume that the *checkpoint coordinator* is named P_0, and other processes are named P_1 through P_{n-1}. (n is the number of processes.)

We now present the proposed algorithm (consisting of two phases), followed by an illustration. Presently, we assume that all processors share a single stable storage; Section 7 considers the situation where multiple stable storages are available.

Algorithm STAGGER

1. *Physical checkpointing phase:* Checkpoint coordinator P_0 takes a *physical checkpoint* and then sends a *take_checkpoint* message to process P_1.

   When a process P_i, $i > 0$, receives a *take_checkpoint* message, it takes a *physical checkpoint* and then sends a *take_checkpoint* message to process P_j, where $j = (i + 1) \bmod n$.

   When process P_0 receives a *take_checkpoint* message from process P_{n-1}, it initiates the second phase of the algorithm (named *consistent logical checkpointing* phase).

   After a process takes the physical checkpoint, it continues execution. Each message delivered to the process, after taking the physical checkpoint (but before the completion of the next phase), is logged in the stable storage.

The above procedure ensures that physical checkpoints taken by the processes are *staggered* because only one process takes a physical checkpoint at any time. The physical checkpoints taken by the processes are not, in general, necessarily consistent. (No attempt is made to ensure consistency of physical checkpoints.)

2. *Consistent logical checkpointing phase:* This phase is very similar to the Chandy-Lamport algorithm. The difference between Chandy-Lamport algorithm and this phase is that when the original Chandy-Lamport algorithm requires a process to take a "checkpoint", our processes takes a *logical* checkpoint (not a physical checkpoint as in the Chandy-Lamport algorithm). A logical checkpoint is taken by ensuring that the messages delivered since the physical checkpoint (taken in the previous phase) are logged on stable storage. The exact algorithm for this phase is provided below:

   Initiation: The coordinator P_0 initiates this phase on receipt of the *take_checkpoint* message from process P_{n-1}. Process P_0 sends *marker* message on each channel, incident on, and directed away from P_0. Also, P_0 takes a logical checkpoint by ensuring that all messages delivered to it since its physical checkpoint are logged.

   A process, say Q, on receiving a *marker* message along a channel c takes the following steps:

   if Q has not taken a *logical* checkpoint **then**
   begin
   Q sends a marker on each channel, incident on, and directed away from Q.
   Q takes a *logical* checkpoint by ensuring that all messages delivered to it
   (on any channel) after Q's recent physical checkpoint have been logged.
   end
   else Q ensures that all messages received on channel c since its recent
   logical checkpoint are logged.

   Messages received on channel c after a marker is received on that channel are not logged.

The above algorithm establishes a consistent recovery line consisting of one *logical* checkpoint per process. This algorithm reduces the contention for the stable storage by completely staggering the physical checkpoints. However, contention is now introduced in the second phase of the algorithm when the processes log messages. Our scheme will perform well if message volume is relatively small compared to checkpoint sizes. A few variations to the above algorithm are possible, as discussed in Section 7.

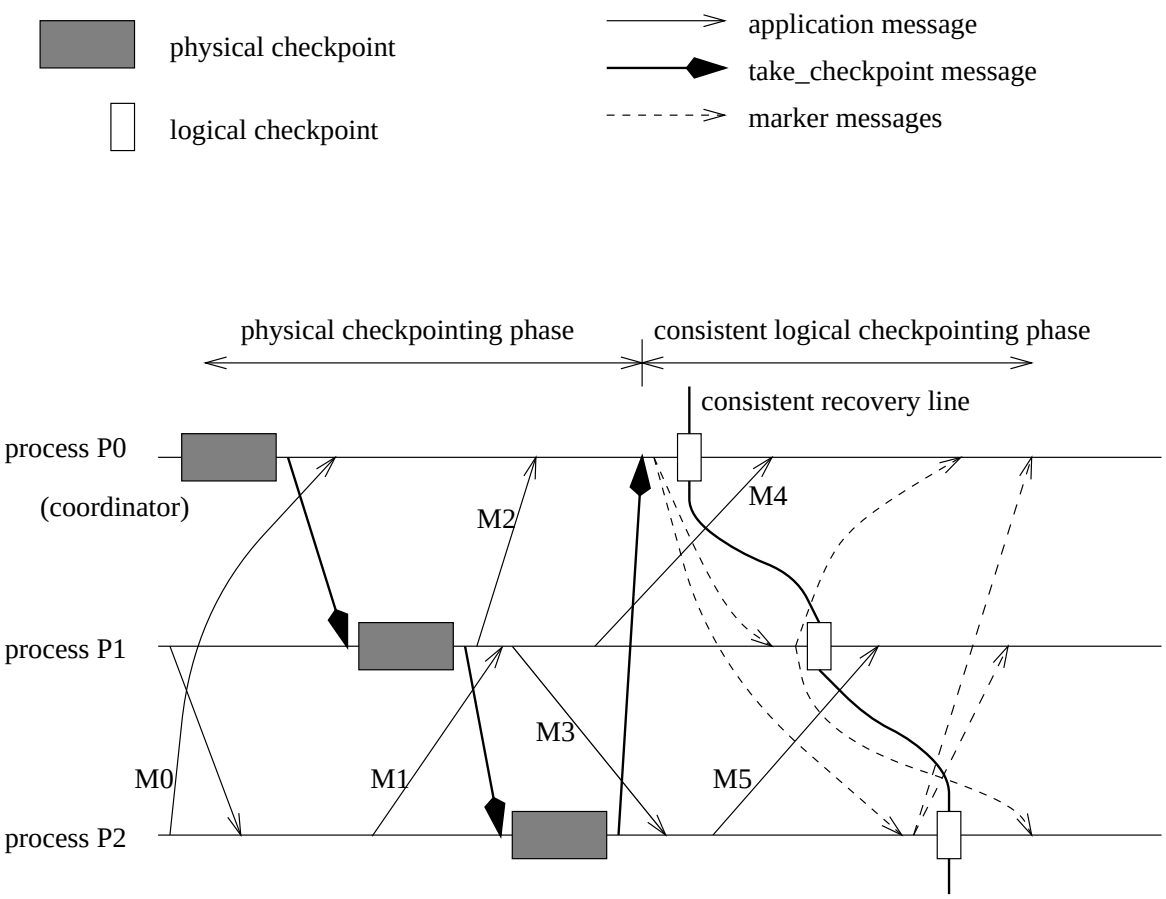

Figure 4: An example

Figure 4 illustrates the algorithm assuming that the system consists of three processes. Process P_0 acts as the coordinator and initiates the checkpointing phase by taking a physical checkpoint and sending a *take_checkpoint* message to P_1. Processes P_0, P_1 and P_2 take staggered checkpoints during the first phase. When process P_0 receives *take_checkpoint* message from process P_2, it initiates the *consistent logical checkpointing* phase. Process P_0 sends marker messages to P_1 and P_2 and then takes a *logical* checkpoint by logging messages M0 and M2 on the stable storage. When process P_1 receives the marker message from process P_0, it sends markers to P_0 and P_2 and then takes a logical checkpoint by logging message M1 on the stable storage. Similarly, process P_2 takes a logical checkpoint by logging message M3 on the stable storage. Messages M4 and M5 are also logged during the second phase (as they represent the channel "state").

Recovery: After a failure, each process rolls back to its recent physical checkpoint and re-executes (using the logged messages) to restore the process state to the logical checkpoint that

belongs to the most recent consistent recovery line.

Proof of correctness: The correctness follows directly from the proof of correctness for the Chandy-Lamport algorithm [1]. The details are omitted here for brevity.

Variations: In Section 7 we present some variations on the above algorithm, including an algorithm to stagger checkpoints when multiple stable storages are available.

6 Performance Evaluation

We implemented the proposed algorithm STAGGER and the Chandy-Lamport/Plank scheme (abbreviated as CL/P) on a nCube-2 multicomputer. It should be noted that performance of each scheme is closely dependent on the underlying hardware and behavior of the application program. Clearly, no single scheme can perform well on all applications. Our objective here is to demonstrate that the proposed algorithm can perform better than the previously proposed algorithm for staggering, and to identify the circumstances where it performs better.

In our implementation of CL/P, the markers are sent asynchronously using interrupts – sufficient care is taken to ensure that the markers appear in FIFO order with respect to other messages even though they are sent asynchronously. The other alternative is to send the markers without using interrupts – the drawback of this approach is that the checkpointing algorithm may not make progress in the cases where the synchronization is very infrequent. As staggering can be useful primarily under these circumstances, it is necessary to ensure that the algorithm progresses without any explicit communication by application processes.

The application used for evaluation is a synthetic program, named `sync-loop`, similar to a program used by Plank [11]. The `pseudo-code` for the program is presented below. (Although C-like syntax is used, the program does not follow C grammar.)

```
sync-loop(num_iteration, state_size, compute_size) {
   state_array = malloc(state_size);  /* create state     */
   initialize (state_array);          /* initialize state */

   repeat (num_iteration) times {
       perform (compute_size) floating-point multiplications;  /* compute     */
```

```
      synchronize with all other processes;                    /* synchronize */
    }
  }
```

Process state size (and checkpoint size) is controlled by the `state_size` parameter. Each process repeats a loop in which it performs some computation (the amount of computation controlled by `compute_size` parameter). The loop is repeated `num_iteration` times.

By choosing a very large value for `compute_size` the degree of synchronization in the program is minimized. A small `compute_size`, on the other hand, implies that processes synchronize very frequently. Synchronization is achieved by means of an all-to-all message exchange.

Figure 5 presents the experimental results for the STAGGER and CL/P schemes. Synchronization interval is the time between two consecutive synchronizations of the processes – thus, synchronization interval is approximately equal to the time required to perform the computation (i.e., the `compute_size` multiplications) in each iteration of the loop. The checkpoint size for each process is approximately 2.1 Mbyte. Checkpoint overhead is obtained by using the formula

$$\frac{(\text{execution time with } S \text{ consistent checkpoints}) - (\text{execution time without any checkpoints})}{S}$$

Figure 5 presents overhead measurements for experiments on a cube of dimension 1, 2, 3 and 4. (Curve labeled $d = n$ in the figure is for experiments on n-dimensional cube.) All processes shared a single disk to store the checkpoints. Observe that, for a fixed dimension, as the synchronization interval becomes smaller, the checkpoint overhead grows for both schemes. For very small synchronization intervals, the STAGGER scheme does not perform much better than the Chandy-Lamport/Plank scheme. However, when synchronization interval is large, the proposed scheme achieves significant improvements. (For dimension $d = 1$, the two schemes achieve essentially identical performance.)

Observe in Figure 5 that, for a given instance of the application, as the dimension is increased the overhead for STAGGER as well as CL/P increases. However, the increase in the overhead of CL/P is much greater than that of STAGGER.

The stable storage contention tends to increase with an increase in the number of application processes. To better understand the impact of stable storage contention, in Figure 6,

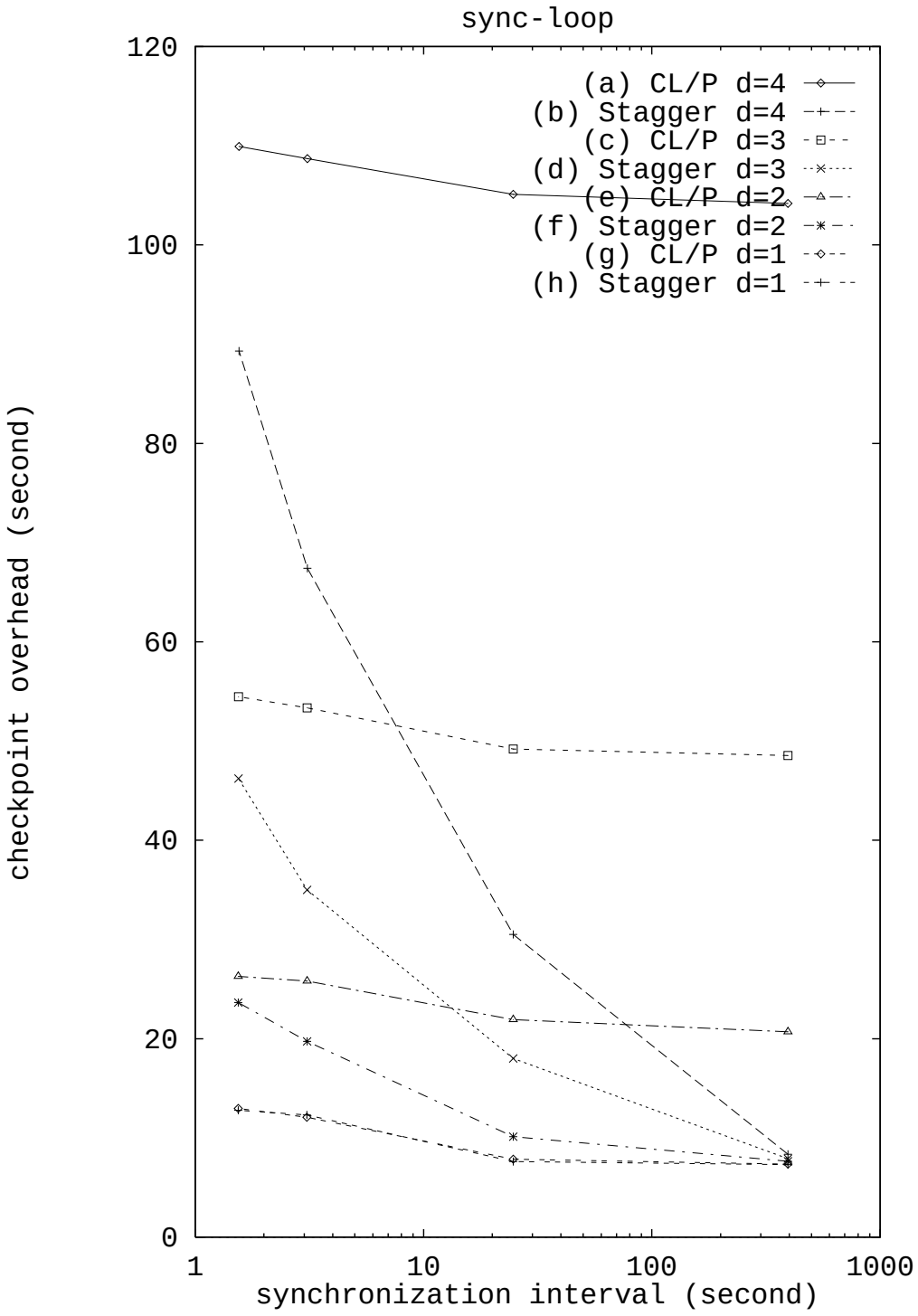

Figure 5: Checkpoint overhead for `sync-loop` program

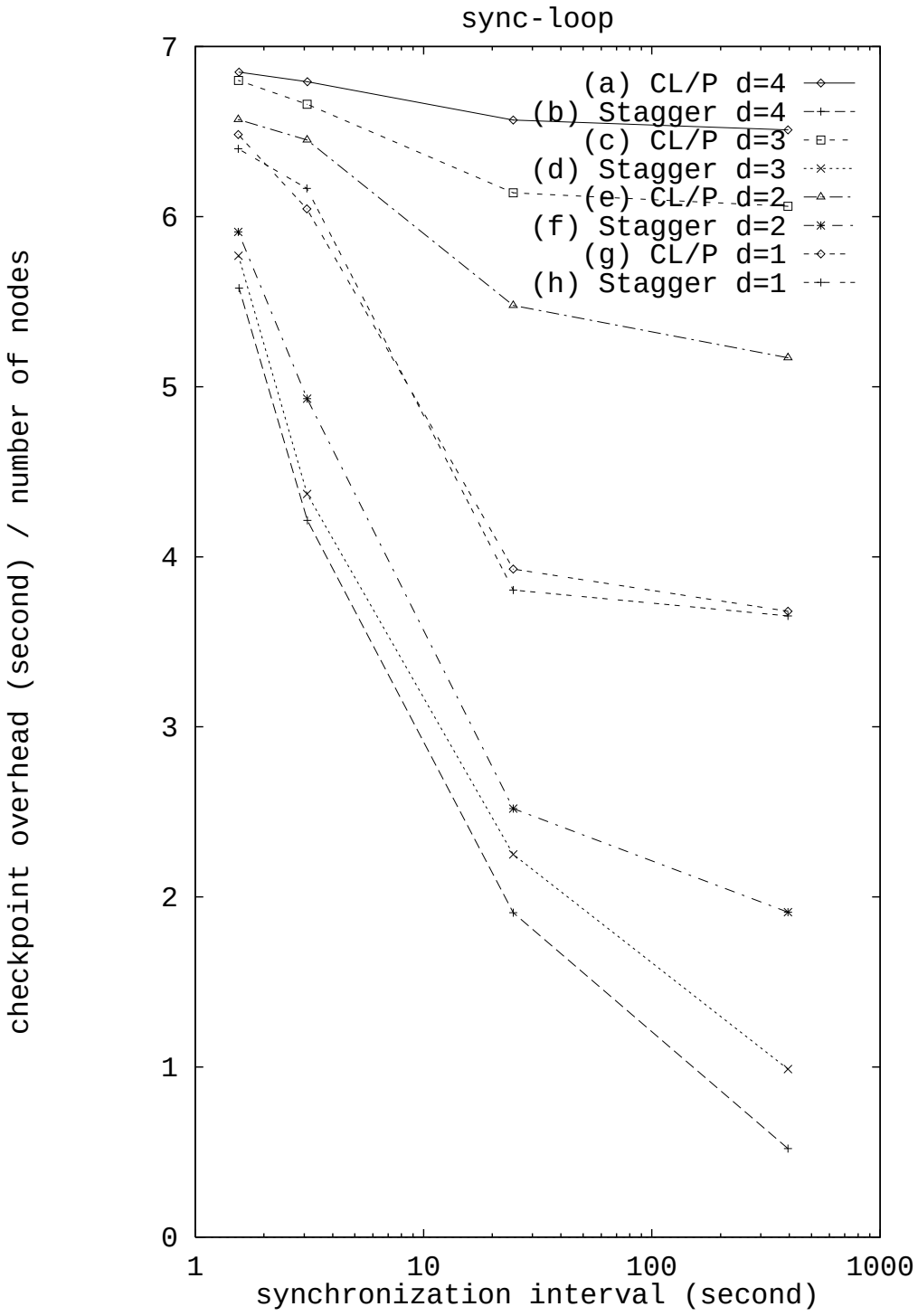

Figure 6: (Checkpoint overhead/number of nodes) for `sync-loop` program

we plot the ratio (checkpoint overhead/number of nodes). Observe that, for a given instance of the application, the ratio is higher for larger dimension when using the CL/P scheme – on the other hand, the ratio is smaller for larger dimension when using the STAGGER scheme. The reason being that the increase in the overhead of STAGGER, with increasing dimension, is relatively small as compared to CL/P.

The measurements presented above imply that when the parallel application has a large granularity (thus, requiring infrequent communication or synchronization), the proposed STAGGER algorithm can perform better than Plank's version of the Chandy-Lamport algorithm [11]. As an example of an application with coarse-grain parallelism we present measurements for a simulation program (SIM), in Figure 7. The simulation program evaluates the expected execution time of a task when using rollback recovery. The simulation program is completely parallelized, and the processes synchronize only at the beginning and at the completion of the simulation. This synchronization pattern represents the best possible scenario for staggered checkpointing. As seen from Figure 7, the checkpoint overhead for STAGGER remains constant independent of the dimension, as synchronization is very infrequent. On the other hand, the overhead for CL/P increases with the dimension.

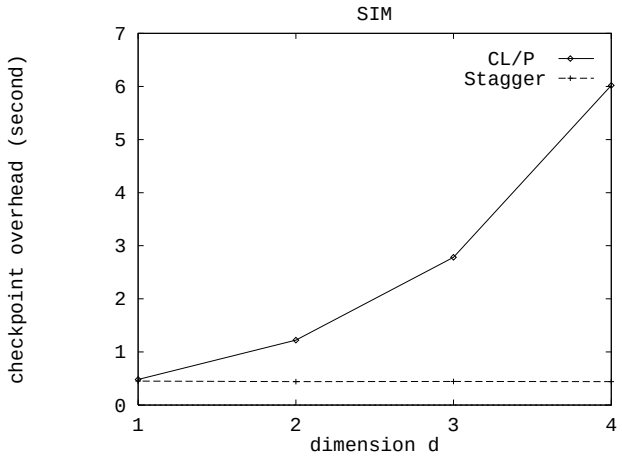

Figure 7: Measurements for SIM application

Impact of message size on performance: Plank [11] observed that his staggered checkpointing schemes log more messages than non-staggered checkpointing schemes. Therefore, his schemes do not perform well compared to non-staggering schemes, when message sizes are large. Similarly, as the STAGGER algorithm staggers checkpoints more than Plank's algorithm, it tends to log more messages than Plank's algorithm. Therefore, STAGGER will not perform

well when message sizes are large. This observation follows directly from that made previously by Plank.

7 Variations on the Theme

Many variations of the algorithm presented earlier are possible. Utility of these variations depends on the nature of the application and the execution environment. In the following, we discuss some variations.

(a) Process clustering to exploit multiple stable storages: The algorithm STAGGER presented above assumes that all processes share a single stable storage. However, in some systems, the processes may share multiple stable storages. For instance, number of processes may be 16 and the number of stable storages may be 4. For such systems, we modify the proposed STAGGER algorithm to make use of all stable storages while minimizing contention for each stable storage. To achieve this we partition the processes into *clusters*, the number of clusters being identical to the number of stable storages. Each cluster is associated with a unique stable storage; processes within the cluster access only the associated stable storage [7].

The algorithm STAGGER modified to use multiple stable storages differs from the original STAGGER algorithm only in the first phase (i.e., staggered checkpointing phase). We illustrate the modified staggered checkpointing phase with an example. Consider a system consisting of 6 processes, and 2 stable storages. The processes are now named P_{ij}, where i denotes the cluster number and j denotes the process number within the cluster. As 2 stable storages are available, the processes are divided into 2 clusters containing 3 processes each. Cluster i ($i = 0, 1$) contains processes P_{i0}, P_{i1} and P_{i2}. Process P_{i0} in cluster i is identified as the *checkpoint coordinator* for cluster i, and process P_{00} is also identified as the *global* checkpoint coordinator. Figure 8 depicts the first phase of the modified algorithm.

The global checkpoint coordinator P_{00} initiates phase 1 of the algorithm by sending *take_checkpoint* messages to the checkpoint coordinators in all other clusters. Process P_{00} then takes a physical checkpoint and sends a *take_checkpoint* message to process P_{01}.

When a process P_{ij} ($ij \neq 00$) receives a *take_checkpoint* message, it takes a physical checkpoint and sends a *take_checkpoint* message to process P_{km} where

$$m = (j + 1) \text{ modulo (cluster size)}$$

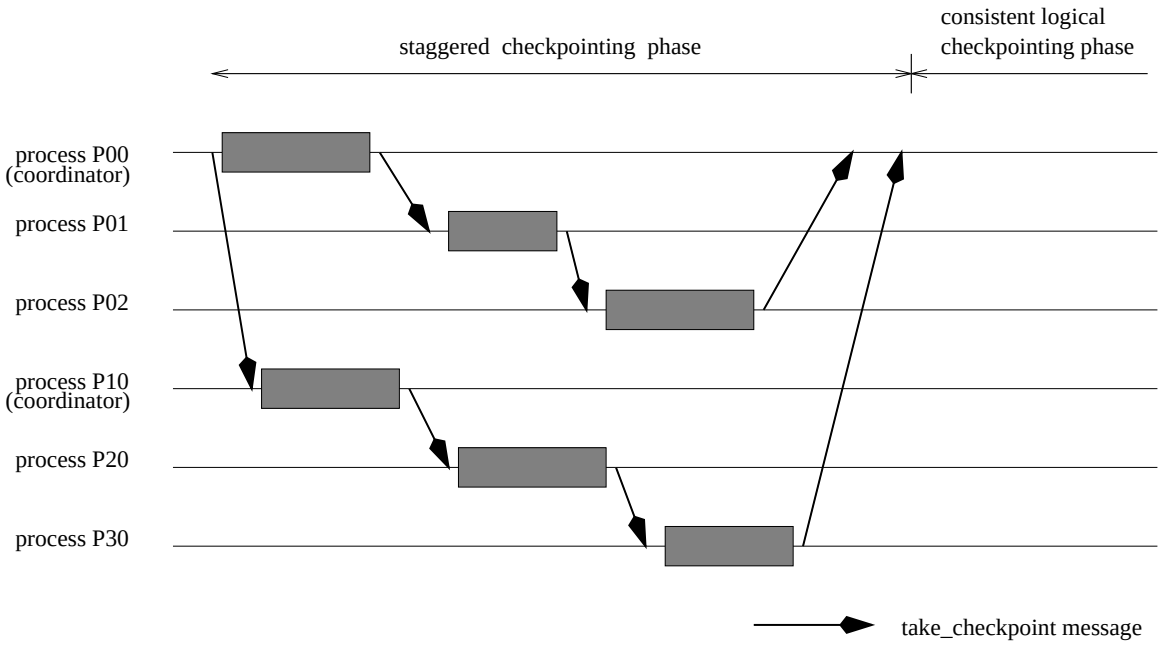

Figure 8: Process clustering to utilize multiple stable storages

$$k = \begin{cases} 0 & \text{if } m = 0 \\ i & \text{otherwise} \end{cases}$$

When the global coordinator P_{00} receives one *take_checkpoint* message from a process in *each* cluster, it initiates the second phase of the algorithm (this phase is identical to the original STAGGER algorithm).

Essentially, the above procedure guarantees that at most one process accesses each stable storage at any time during the *first* phase, and that all stable storages are used for saving physical checkpoints.

(b) Approach for taking a logical checkpoint: The discussion so far assumed that a logical checkpoint is taken by taking a physical checkpoint and logging subsequently received messages. It is easy to see that the proposed algorithm can be modified to allow a process to use any of the three approaches presented earlier (in Section 3) for establishing a logical checkpoint. In fact, different processes may simultaneously use different approaches for establishing a logical checkpoint.

(c) Checkpointing versus message logging: As staggering tends to increase the number of messages logged, the following variations will be beneficial for some applications.

- A process may decide to not take the physical checkpoint in the first phase, if it a priori knows that its message log will be large. In this case, the process would take a physical checkpoint in the second phase.[2]

- If a process receives too many messages after taking the physical checkpoint in the first phase of the algorithm, then it may decide to take a physical checkpoint in the second phase (rather than logging messages). This makes the physical checkpoint taken by the process in the first phase redundant. However, this modification may reduce the overhead when checkpoint size is smaller than what the message log would be.

- The coordinator may initiate the *consistent logical checkpointing* phase even before all processes have taken the physical checkpoint. In this case, consider a process Q that receives a marker message before Q has taken the physical checkpoint (in the first phase). Then, process Q can take a physical checkpoint in the second phase rather than logging messages to establish a logical checkpoint (essentially, process P can pretend that it decided to not take a physical checkpoint in the first phase).[3]

A future goal of our research is to design an *adaptive* algorithm that can, at run-time, determine if staggering is beneficial or not.

7.1 Improving performance of CL/P:

The performance of the CL/P algorithm is sensitive to the manner in which the markers are sent – asynchronously using interrupts, or without interrupts. Both approaches have their benefits and disadvantages. One possible approach for improving performance of CL/P algorithm would be to send some markers asynchronously (to ensure that the algorithm proceeds even with infrequent application messages), and send the other markers without using interrupts. For instance, a process i may send an asynchronous marker only to process $i+1$. This will ensure that each process receives at least one asynchronous marker, thus, guaranteeing that the algorithm will make progress in tha absence of application messages.

As the proposed STAGGER algorithm encompasses the CL/P algorithm, we have not

[2] Johnson [6] suggested a scheme where each process uses a similar heuristic to decide whether to log messages or not.

[3] Recollect that a *physical* checkpoint is also trivially a *logical* checkpoint. So the process here is actually taking a logical checkpoint, but not by logging messages.

studied variations on CL/P. Our future work will deal with variations on the STAGGER algorithm.

8 Summary

This paper presents an algorithm for taking consistent *logical* checkpoints. The proposed algorithm ensures that the *physical* checkpoints taken by various processes are completely staggered to minimize the contention in accessing the stable storage. Experimental results on nCube-2 suggest that the proposed scheme can improve performance as compared to existing staggering techniques, particularly when processes synchronize infrequently and message sizes are not very large. The paper also suggests a few variations of the proposed scheme, including an approach for staggering checkpoints when multiple stable storages are available.

Acknowledgements

Thanks are due to James Plank and Yi-Min Wang for their comments on an earlier draft of the paper. Vidya Iyer wrote parts of the checkpointing layer used for experiments.

References

[1] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states in distributed systems," *ACM Trans. Comp. Syst.*, vol. 3, pp. 63–75, February 1985.

[2] C. J. Date, *An Introduction to Database Systems*. Addison-Wesley, 1986.

[3] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Symposium on Reliable Distributed Systems*, 1992.

[4] D. Jefferson, "Virtual time," *ACM Trans. Prog. Lang. Syst.*, vol. 3, pp. 404–425, July 1985.

[5] D. B. Johnson, *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Computer Science, Rice University, December 1989.

[6] D. B. Johnson, "Efficient transparent optimistic rollback recovery for distributed application programs," in *Symposium on Reliable Distributed Systems*, pp. 86–95, October 1993.

[7] S. Kaul (Advisor: N. Vaidya), "Evaluation of consistent logical checkpointing." M.S. Thesis, Dept. of Computer Science, Texas A&M University, May 1995.

[8] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Softw. Eng.*, vol. 13, pp. 23–31, January 1987.

[9] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Trans. Par. Distr. Syst.*, vol. 5, pp. 874–879, August 1994.

[10] J. Long, B. Janssens, and W. K. Fuchs, "An evolutionary approach to concurrent checkpointing," manuscript submitted for publication, 1994.

[11] J. S. Plank, *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Dept. of Computer Science, Princeton University, June 1993.

[12] R. E. Strom and S. A. Yemini, "Optimistic recovery: An asynchronous approach to fault-tolerance in distributed systems," *Digest of papers: The* 14^{th} *Int. Symp. Fault-Tolerant Comp.*, pp. 374–379, 1984.

[13] Y. M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," in *Symposium on Reliable Distributed Systems*, pp. 78–85, October 1993.

[14] Y. M. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems," in *Digest of papers: The* 23^{rd} *Int. Symp. Fault-Tolerant Comp.*, pp. 138–144, 1993.

[15] Y. M. Wang, A. Lowry, and W. K. Fuchs, "Consistent global checkpoints based on direct dependency tracking." To appear in *Inform. Process. Lett.*