# Staggered Consistent 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
Web: http://www.cs.tamu.edu/faculty/vaidya

September 17, 1996

## Abstract

*A consistent checkpointing algorithm saves a consistent view of a distributed application's state on stable storage. The traditional consistent checkpointing algorithms require 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 checkpoint overhead [13]. 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. Experimental results on nCube-2 are presented.*

**Key words:** Staggered checkpoints, consistent recovery line, rollback recovery, stable storage contention, fault tolerance.

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

[†]To be presented in part at the IEEE Symposium on Parallel and Distributed Processing (SPDP), October 1996, New Orleans.

## 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. *Consistent checkpointing* is a commonly used technique to prevent complete loss of computation upon a failure [1, 2, 4, 5, 8, 11, 13, 17]. A consistent checkpointing algorithm saves a consistent view of a distributed application's state on a *stable storage* (often, a disk is used as 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 different application processes to save their state at about the same time. This causes contention for the stable storage when multiple processors share a stable storage, potentially resulting in significant performance degradation. Clearly, if each processor has access to a separate stable storage, such contention will not occur.[1] However, many installations of multicomputers and distributed systems require multiple processors to share a stable storage.

*Staggering* the checkpoints taken by various processes can reduce the overhead of consistent checkpointing by reducing stable storage contention, as observed by Plank [13]. Plank proposed some techniques for *staggering* the checkpoints [13], however, these techniques result in "limited" staggering in that not all processes' checkpoints can be staggered. Moreover, the previous algorithms do not have much control on which checkpoints are staggered. Ideally, one would like to be able to stagger the checkpoints in a manner most appropriate for a given system.

In systems where processors are able to make an "in-memory" copy of entire process state, checkpoint staggering is trivial. In this case, the checkpoints can be first taken in-memory, and then written to the stable storage one at a time. This paper considers systems where it is not feasible to make an in-memory copy of entire process state. This situation may occur because, either (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

[1]If different stable storages are accessed over the same network, network contention can become a bottleneck.

approach requires that the processes take consistent *logical* checkpoints, as compared to consistent *physical* checkpoints enforced by existing algorithms for *staggering*. As elaborated later, a *physical* checkpoint is a copy of a process' state, and a *logical* checkpoint is obtained by saving *sufficient* information (e.g., messages) to recover a process' state. The objective of this paper is to show how checkpoints can be staggered in a controlled manner, independent of the application's communication patterns, and to present different variations of the algorithm. To illustrate that our approach can be of interest in practice, experimental results for one version of the algorithm on nCube-2 multicomputer are presented.

The paper is organized as follows. Section 2 discusses the related work. Section 3 discusses the notion of a *logical checkpoint*. Section 4 presents consistent checkpointing algorithms proposed by Chandy and Lamport [2] and Plank [13]. Section 5 presents the proposed algorithm. Section 6 presents experimental results. Some variations of the proposed scheme are discussed in Section 7. Section 8 summarizes the paper.

## 2 Related Work

Plank [13] was the first to observe that stable storage contention can be a problem for consistent checkpointing, and suggested checkpoint staggering as a solution. The degree of staggering with Plank's algorithm (based on the Chandy-Lamport algorithm [2]) is *limited* in that checkpoints of many processes are not staggered. In contrast, our algorithm allows arbitrary and controlled staggering of checkpoints. Plank [13] also presents another approach for staggering checkpoints, that is applicable to wormhole routed networks. This algorithm also does not permit arbitrary/controlled staggering.

Fowler and Zwaenepoel [6] present an algorithm for determining causal breakpoints (for the purpose of debugging). As a part of the breakpoint algorithm, they establish consistent recovery lines using an algorithm similar to ours. Our approach can be considered to be a modification of the algorithm in [6] to facilitate checkpoint staggering. Because the algorithm in [6] was designed for debugging purposes, various possibilities for checkpoint staggering, and different approaches for establishing checkpoints were not considered.

Long et al. [11] discuss an *evolutionary* checkpointing approach, that is similar to *logical checkpointing*. Our algorithm *staggers* the checkpoints, while the scheme in [11] does not allow staggering. [11] also assumes *synchronized communication* and an upper bound on communication delays; no such assumptions are made in the proposed scheme.

Wang et al. [18] introduced the term *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. Message logging and independent checkpointing schemes, such as [8], also, effectively, determine a recovery line consisting of consistent logical checkpoints after a failure occurs. In these schemes, 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 approach is that our scheme does not log all messages; only those messages which make the logical checkpoints consistent are logged.

Staggering the checkpoints taken by various processes tends to increase the elapsed time (sometimes called checkpoint "latency" [15]) while the checkpointing algorithm is in progress. Our previous work [15] shows that a *large increase* in checkpoint latency is acceptable if it is accompanied by even a *small decrease* in checkpoint overhead. Therefore, techniques such as staggering are of interest even though they may result in greater checkpoint latency.

## 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 [8, 14]. 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 available 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. A physical checkpoint is trivially a logical checkpoint, however, the converse is not true.

Physical checkpoint itself can be taken in two different ways: One possibility is to save the entire process state on the stable storage. The second possibility is to take an *incremental* checkpoint [12]. (That is, only the difference between the current state and the state at the previous physical checkpoint needs to be saved.) We will return to incremental

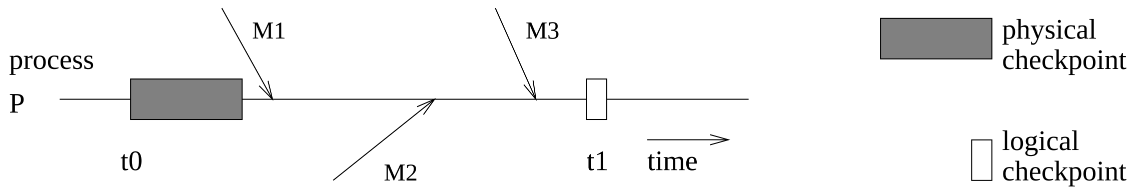


Figure 1: Physical checkpoint + message log = logical checkpoint

checkpointing soon again.

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.

**Approach 1:** 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 $t_1$. This approach is essentially identical to that presented by Wang et al. [18]. 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. We summarize this approach as:

$$\textit{physical checkpoint} \quad + \quad \textit{message log} \quad = \quad \textit{logical checkpoint}$$

**Approach 2:** The essential purpose behind saving the messages above is to be able to recreate the state at time $t_1$. This may also be achieved by taking a *physical* checkpoint at time $t_0$ and taking an *incremental* checkpoint at time $t_1$. The incremental checkpoint is taken by saving (on the stable storage) the changes made to process state between time $t_0$ and $t_1$. We summarize this approach as:

$$\textit{physical checkpoint} \quad + \quad \textit{incremental checkpoint} \quad = \quad \textit{logical checkpoint}$$

As noted earlier, the physical checkpoint itself may be taken using the incremental checkpointing method. Therefore, it is possible to completely eliminate the physical checkpoint. However, it is not necessarily desirable. Figure 2 illustrates this. Assume that physical checkpoint P2 at time $t_0$ is taken as the incremental change from the state at the previous physical checkpoint P1. Also, the logical checkpoint at time $t_1$ is taken as the incremental change from the state at time $t_0$ until time $t_1$. The time interval between P1 and P2 is

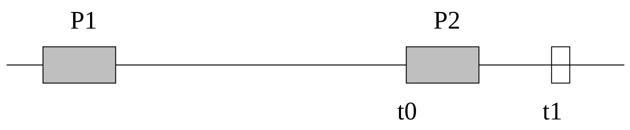


Figure 2: Incremental checkpointing in approach 2 for logical checkpointing

much larger than time interval $t_1 - t_0$. Therefore, the size of incremental state saved to establish the physical checkpoint P2 is likely to be much larger than that saved to establish the logical checkpoint at $t_1$. Our algorithm staggers the physical checkpoints, whereas the logical checkpoints contend for the stable storage. Now consider the situation where physical checkpoint P2 is not taken at all. In this case, the incremental state saved at $t_1$ will consist of the modifications made to the state, from the time when checkpoint P1 is taken, until time $t_1$. Therefore, the size of this incremental state will be at least as large as that saved above to establish P2. As will be apparent later, this would defeat the staggering algorithm by introducing significant stable storage contention when taking the logical checkpoints. Therefore, it will often be desirable to take a physical checkpoint first (possibly an incremental checkpoint), followed by an incremental logical checkpoint.

The *evolutionary* checkpointing scheme by Long et al. [11] also takes incremental checkpoints similar to the above procedure.

**Approach 3:** 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 [7] or that of UNDO records [3] in database systems. Anti-message $M^*$ corresponding to a message M can be used to undo the state change caused by message M.

Figure 3 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.

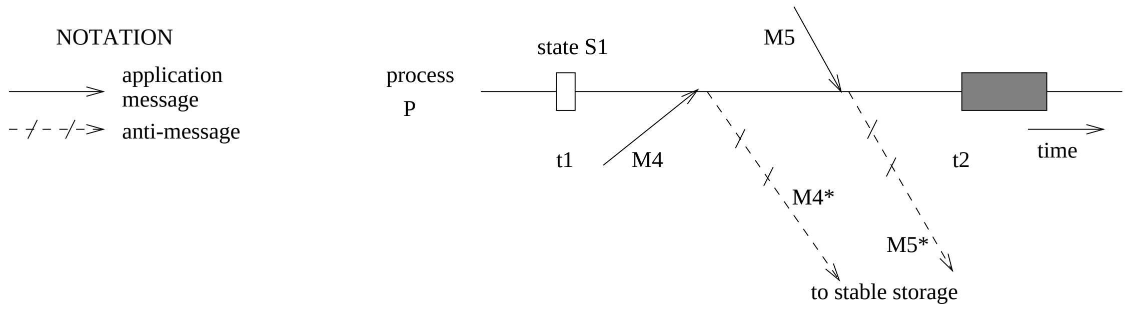


Figure 3: Anti-message log + physical checkpoint = logical checkpoint

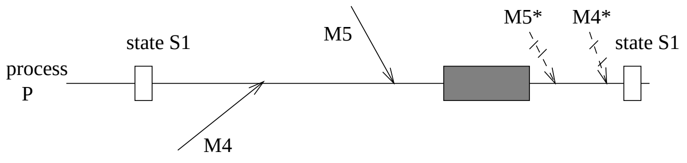


Figure 4: Recovering a logical checkpoint using anti-messages

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 4, 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*

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 [10]).

## 4 Chandy-Lamport Algorithm [2]

Chandy and Lamport [2] presented an algorithm for taking a consistent checkpoint of a distributed system. Assume that the processes communicate with each other using first-in-first-out (FIFO) unidirectional communication *channels*; a bidirectional channel can be modeled as two unidirectional channels. For simplicity, we assume that the communication graph is fully connected.[2] The algorithm presented next is essentially identical to

[2]Note that Chandy-Lamport algorithm is applicable to any strongly connected graph. Our algorithm can also be generalized to strongly connected graphs.

Chandy-Lamport [2, 13] and assumes that a certain process (named $P_0$) is designated as the *checkpoint coordinator*.

**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 Plank's Staggering Scheme

Plank [13] 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 [13] observed that his staggering scheme works better than the original "non-staggered" algorithm when (i) degree of synchronization (or communication) amongst the processes is relatively small, and (ii) the *message volume* is relatively small (*message volume* is the amount of information communicated by messages). In Figure 5, the horizontal axis indicates degree of *synchronization* in an application, and the vertical axis indicates the *message volume*. As shown in the figure, when synchronization is very frequent and/or message volume is large, it is better to avoid staggering checkpoints [13]. Extrapolating Plank's results, it follows that, the region where a given staggering algorithm works best shrinks with the degree of staggering. Greater staggering is beneficial for applications with less synchronization and small message volume. This paper does not alter the above conclusions. Our work provides an user the ability to choose the *degree of staggering*. Our approach can achieve completely controlled staggering of checkpoints, unlike Plank [13].

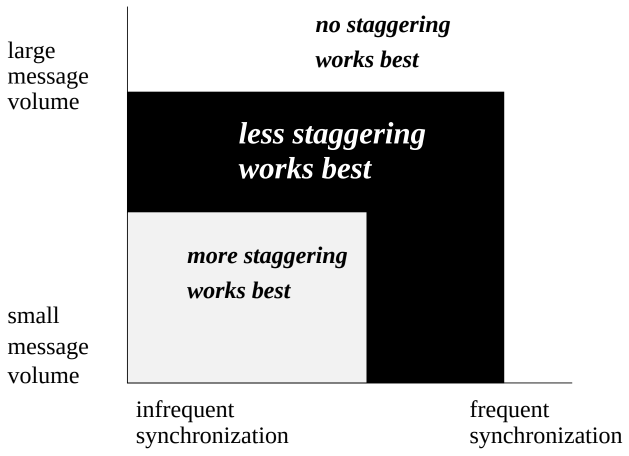


Figure 5: Checkpoint staggering and performance: Qualitative observations

## 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 asynchronously or not). 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 [16]. 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. The proposed algorithm (named STAGGER) can be summarized as follows:

$$\textit{staggered physical checkpoints} + \textit{consistent logical checkpoints} = \textit{staggered consistent checkpoints}$$

The proposed algorithm coordinates *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 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 necessarily consistent.

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 take 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. (The number of messages logged can be somewhat reduced, as discussed later.)

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. Similar to the Chandy-Lamport algorithm, messages sent by a process before its logical checkpoint, but not received before the receiver's logical checkpoint are logged as part of the *channel state*. Note that a message M that is logged to establish a logical checkpoint may be logged any time from the instant it is received until the time when the logical checkpoint is to be established. In our implementation, due to insufficient memory on nCube-2, such messages were logged immediately on receipt.

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.[3] Our scheme will perform well if message volume is relatively small compared to checkpoint sizes.

Figure 6 illustrates the algorithm assuming that the system consists of three processes. Process $P_0$ acts as the coordinator and initiates the physical 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 $P_0$

[3]This contention can potentially be reduced by taking a logical checkpoint before sending markers in the *consistent logical checkpointing* phase.

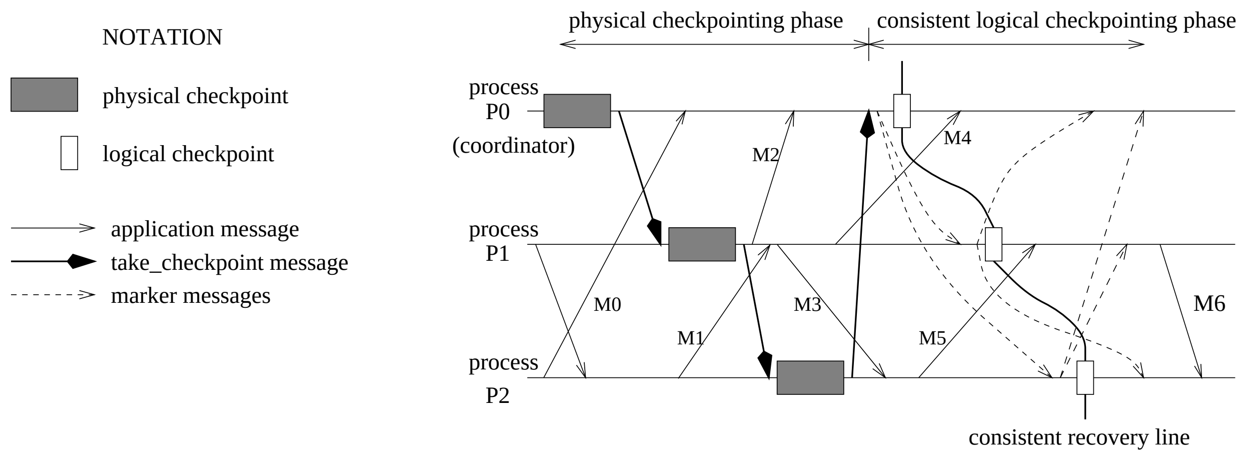


Figure 6: An example

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). Message M6 is not logged.

**Proof of correctness:** The correctness follows directly from the proof of correctness of the Chandy-Lamport algorithm [2].

**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.

Note that, the above STAGGER algorithm was designed assuming that it is desirable to stagger *all* checkpoints. If some other pattern of staggering is more desirable, the above algorithm can be easily modified to achieve that pattern. Section 7 illustrates this with an example.

## 6 Performance Evaluation

We implemented the proposed algorithm STAGGER and the Chandy-Lamport/Plank (CL/P) algorithm on a nCube-2 multicomputer with a single disk (stable storage). It should be noted that performance of each scheme is closely dependent on underlying hardware, software implementation of the scheme, and nature of the application program. Clearly, no single scheme can perform well for all applications. Our objective here is to demonstrate that the proposed scheme can perform well under certain circumstances.

In our implementation of CL/P and STAGGER, the markers sent by process 0 are sent asynchronously using interrupts (or signals) – sufficient care is taken to ensure that the markers appear in first-in-first-out (FIFO) order with respect to other messages even though the markers are sent asynchronously. Markers sent by other processes are sent without using interrupts. If no markers are sent asynchronously, the checkpointing algorithm may not make progress in the cases where synchronization (or communication) is infrequent. As staggering is most beneficial under these circumstances, it is necessary to ensure that the algorithm progresses without any explicit communication by application processes. Therefore, process 0 sends asynchronous markers. We will return to the issue of using asynchronous markers later in Section 7.

The first application used for evaluation of STAGGER is a synthetic program, named `sync-loop`, similar to a program used by Plank [13]. The `pseudo-code` for the program is presented below using a C-like syntax.

```
sync-loop(iter, size, M) {
 char state[size];
 initialize (state);

 repeat (iter) times {
   perform M floating-point multiplications;
   synchronize with all other processes;
  }
}
```

Process state size (and checkpoint size) is controlled by the `size` parameter. For the `size` chosen for our experiments, checkpoint size for each process of `sync-loop` is approximately 2.1 Mbyte. Each process repeats a loop in which it performs some computation (the amount of computation being controlled by the `M` parameter). The loop is repeated `iter` times.

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

Figure 7 presents experimental results for STAGGER and CL/P schemes. *Synchronization interval* in this figure 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 `M` multiplications) in each iteration of the loop. The `synchronization interval` on the horizontal axis in Figure 7 is determined by dividing by `iter` the execution time of `sync-loop` without taking any checkpoints. Checkpoint overhead is obtained as:

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

For our measurements, $S = 5$ (that is, five checkpoints per execution of the program). Each instance of the `sync-loop` application was executed five times, and checkpoint overhead was averaged over these five executions.

Figure 7 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 consisting of $2^N$ processors. (Labels (a) through (h) in Figure 7 can be used to match the curves with the corresponding legend in top right corner of the figure.) In Figure 7, 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 $d > 1$. For dimension $d = 1$ (that is, 2 processors), the two schemes achieve essentially identical performance.

Observe in Figure 7 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.

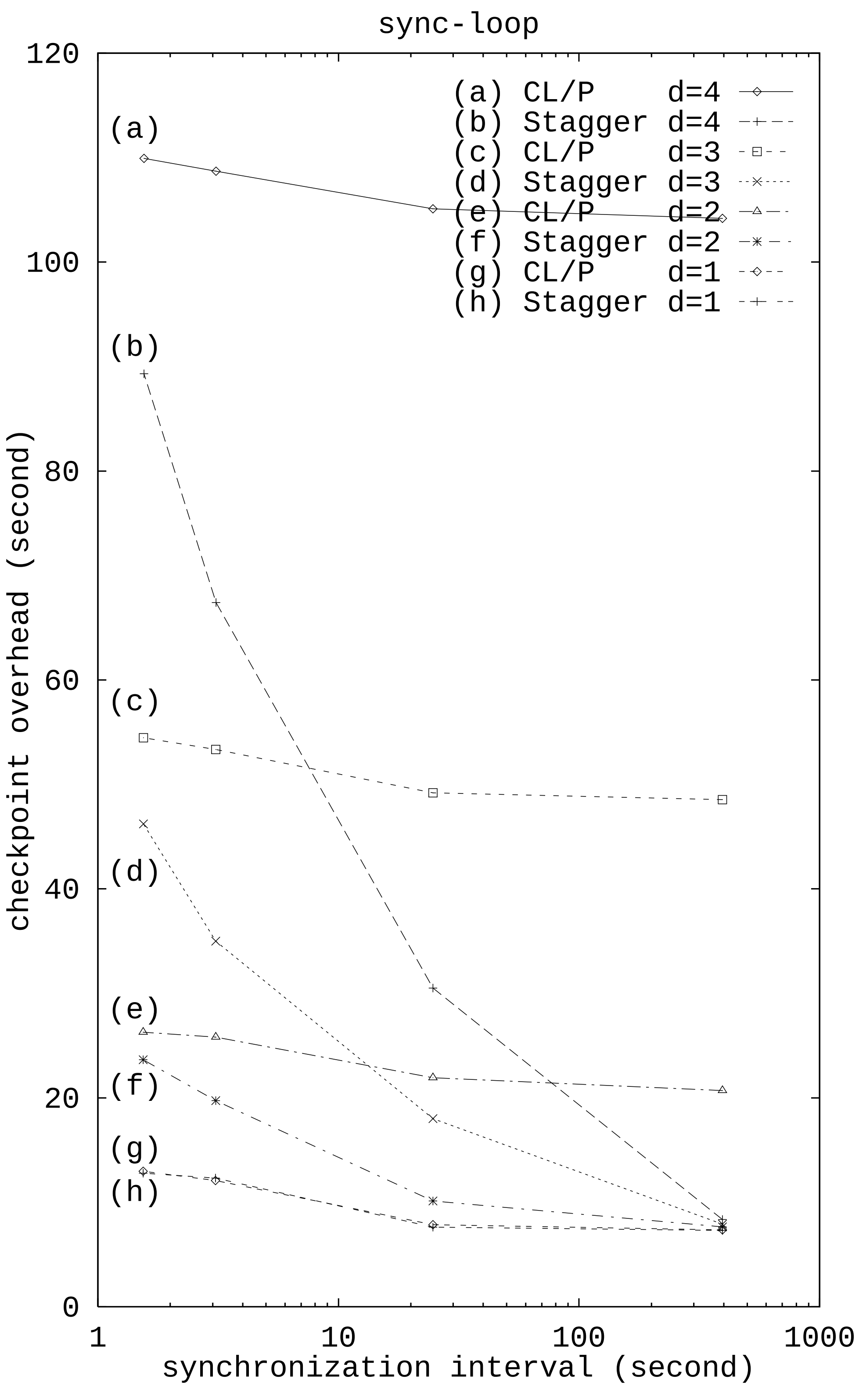


Figure 7: Checkpoint overhead for `sync-loop`. Labels (a) through (h) can be used to match the curves with the corresponding legend in top right corner of the figure.

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 well. As an example of an application with coarse-grain parallelism, Figure 8 presents measurements for a simulation program (SIM). Simulation program SIM evaluates the expected execution time of a task when using rollback recovery. State size for each process in SIM is approximately 34 Kbyte. 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 8, 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.

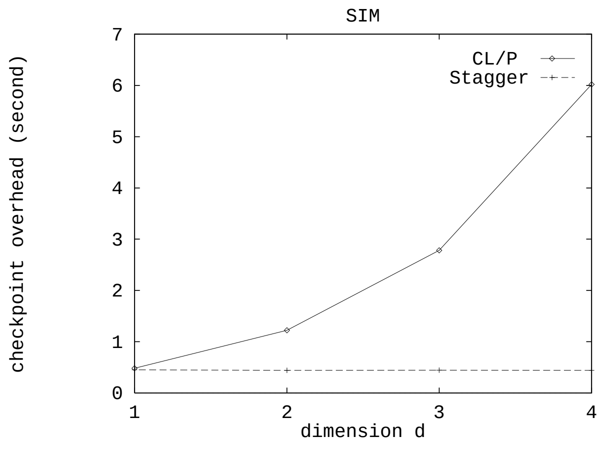
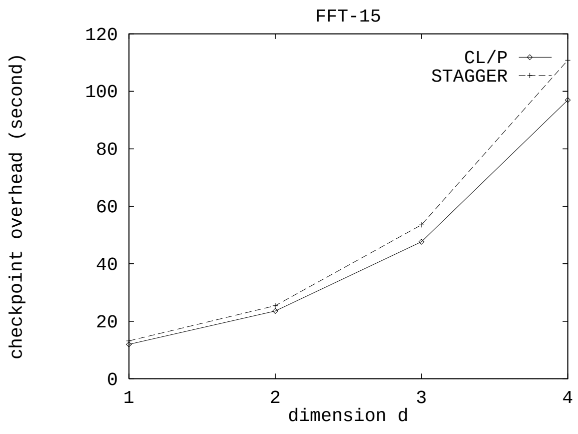


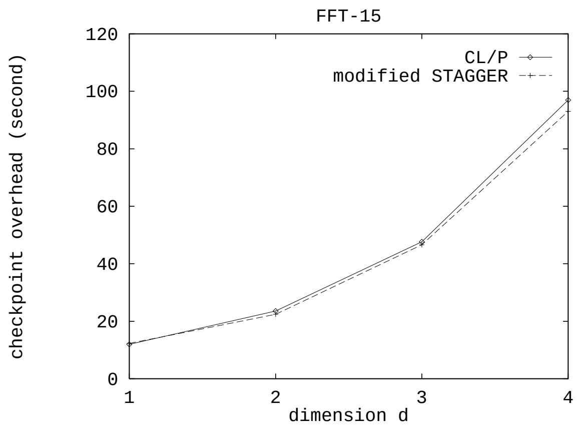
Figure 8: Measurements for SIM application

To be fair, we should note that STAGGER does not always outperform CL/P. As noted in Figure 5, an algorithm that staggers more tends to perform poorly when degree of synchronization and/or message volume is large. To illustrate this, Figure 9(a) presents measurements for a program named FFT-15 that repeatedly evaluates fast Fourier transform of $2^{15}$ data points, and has frequent interaction between processes. Checkpoint size for each process is approximately 1.85 Mbyte. For this application, the overhead of STAGGER is larger than that of CL/P.

The performance of STAGGER can be improved by reducing the amount of information logged, using an optimization similar to that in [6]. Unlike in the original STAGGER algorithm, it is not necessary to log a message's data content if it was sent by a process *after*



(a) Using STAGGER algorithm



(b) Using modified STAGGER algorithm

Figure 9: Measurements for FFT-15 application

taking its *physical* checkpoint – for such a message, it is sufficient to log its order information (i.e., send and receive sequence numbers, and sender and receiver identifiers). During recovery, such a message is always reproduced by the sender process. Therefore, logging of order information is sufficient. Figure 9(b) plots overhead of the STAGGER algorithm modified to implement the above optimization. The overhead of the modified algorithm is lower than the original STAGGER algorithm (see Figure 9(a)), however, the overhead is still not much better than CL/P. As the FFT-15 application performs frequent communication, it is hard to achieve overhead better than CL/P.

## 7 Variations on the Theme

**(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 a cluster access only the associated stable storage [9].

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 cluster number and $j$ denotes 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 10 depicts the first phase of the modified algorithm.

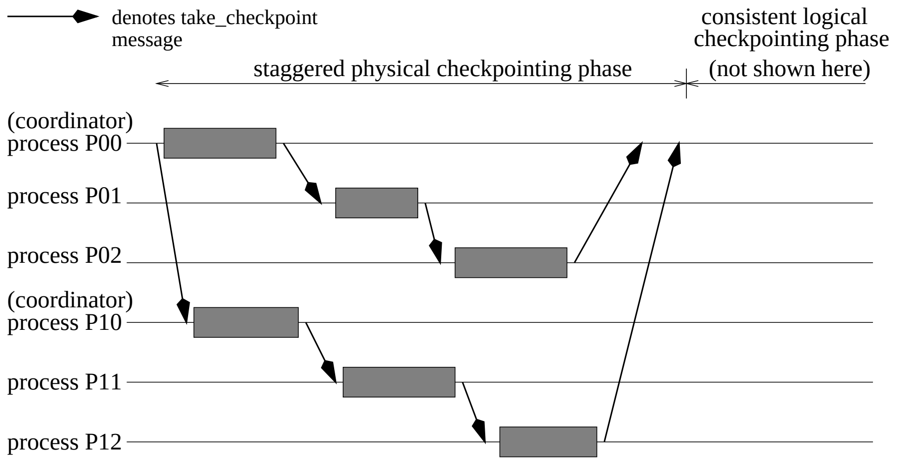

denotes take_checkpoint message
consistent logical checkpointing phase (not shown here)
staggered physical checkpointing phase
(coordinator) process P00
process P01
process P02
(coordinator) process P10
process P11
process P12

Figure 10: Process clustering to utilize multiple stable storages

The global checkpoint coordinator $P_{00}$ initiates phase 1 of the algorithm (i.e., staggered physical checkpointing phase) 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

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

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

When the global coordinator $P_{00}$ receives one *take_checkpoint* message from a process in *each* cluster, it initiates the *consistent logical checkpointing* phase (this phase is identical to the second phase of 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. The proposed algorithm can be easily 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) Asynchronous Markers:**
Arrival of an asynchronous marker is informed to the destination process by means of an interrupt (or signal). In spite of the asynchronous nature, a marker should appear in its appropriate position on the FIFO channel on which it is sent. We call a marker that is not sent with an interrupt a "synchronous" marker (for the lack of a better terminology). While an asynchronous marker can be processed as soon as it arrives, a synchronous marker may not be processed for a long time – particularly, if the destination process does not need any messages on the corresponding channel.

Which markers (if any) are sent asynchronously can affect performance of STAGGER and CL/P algorithms. As noted previously, in our implementation, markers sent by process 0 are asynchronous, other markers are synchronous.

Plank [13] does not address the distinction between *asynchronous* and *synchronous* markers. One variation that can make CL/P imitate STAGGER, particularly for applications with infrequent synchronization (communication), is as follows: In CL/P algorithm, ensure that the marker sent by process $i$ to process $j$ is asynchronous if and only if $j = i + 1$ (modulo number of processes). Thus, each process will take checkpoint, and the algorithm will make progress, even if the processes are not communicating with each other. Also, as each process sends only one asynchronous marker, the algorithm would tend to reduce contention for the stable storage. With infrequent synchronization (communication), the above rule will tend to stagger checkpoints by different processes (i.e., the algorithm becomes similar to STAGGER).

The above variation could also be used to reduce stable storage contention during the *consistent logical checkpointing* phase of STAGGER algorithm.

## 8 Summary

This paper presents an algorithm for taking consistent *logical* checkpoints. The proposed algorithm can ensure that *physical* checkpoints taken by various processes are staggered to minimize contention in accessing the stable storage. Experimental results on nCube-2 suggest that the proposed scheme can improve performance as compared to an existing staggering technique, 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**

We thank the referees of the *conference version* of this paper, and James Plank and Yi-Min Wang, for their comments on an earlier draft of this paper. The FFT program used for experiments was written by Akhilesh Kumar. Vidya Iyer wrote parts of the checkpointing layer used for experiments. This research is supported in part by National Science Foundation grant MIP-9502563 and Texas Advanced Technology Program grant 009741-052-C.

## References

[1] G. Cabillic, G. Muller, and I. Puaut, "The performance of consistent checkpointing in distributed shared memory systems," in *Int. Symp. Reliable Distr. Systems (SRDS)*, pp. 96–105, September 1995.

[2] 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.

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

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

[5] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *IEEE Trans. Computers*, vol. 41, May 1992.

[6] J. Fowler and W. Zwaenepoel, "Causal distributed breakpoints," in *International Conf. Distributed Computing Systems*, pp. 134–141, May 1990.

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

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

[9] S. Kaul, "Evaluation of consistent logical checkpointing." M.S. Thesis, Dept. of Computer Science, Texas A&M University, May 1995.

[10] 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.

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

[12] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Usenix Winter 1995 Technical Conference, New Orleans*, January 1995.

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

[14] 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.

[15] N. H. Vaidya, "On checkpoint letency," in *Pacific Rim International Conference on Fault-Tolerant Systems*, December 1995.

[16] N. H. Vaidya, "On staggered checkpointing," in *Eighth IEEE Symposium on Parallel and Distributed Processing (SPDP)*, October 1996.

[17] 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.

[18] 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.