A Symmetric $O(n \log n)$ Message Distributed
Snapshot Algorithm for Large-Scale Systems

Ajay D. Kshemkalyani
Computer Science Department, University of Illinois at Chicago
Chicago, IL 60607, USA
ajayk@cs.uic.edu

Abstract—This paper presents a $O(n \log n)$ message distributed snapshot algorithm for a system with non-FIFO channels, where $n$ is the number of processors. The algorithm finds applications for checkpointing in large scale supercomputers and distributed systems that have a fully connected logical topology over a large number of processors. Each processor sends $\log n$ messages in the algorithm. The sizes of the messages are geometrically distributed, and the sum of the sizes of the messages sent by any processor is $n$. The response time of the algorithm is $O(\log n)$. The algorithm is fully distributed and the role of each processor is symmetric, unlike tree-based, ring-based, and centralized algorithms.

I. INTRODUCTION AND PROBLEM DEFINITION

Consider a distributed system that is modeled as a directed graph $(N, L)$, where $N$ is the set of processors and $L$ is the set of non-FIFO links connecting the processors in a logical application layer overlay. Let $n = |N|$. The logical overlay is typically fully connected, hence the all-to-all logical overlay gives $n(n-1)/2$ logical channels. Typically, the rich interconnectivity of the underlying graph (such as a torus, hypercube, and other regular topologies) allows for multiple logical paths among any pair of processors. Such a logical path can be modeled as a non-FIFO channel in the overlay.

A snapshot of a distributed system represents a consistent global state of the system [3]. A snapshot consists of $\langle \bigcup_i \{LS_i\}, \bigcup_{i,j} \{SC_{i,j}\} \rangle$, where $LS_i$ is the local state of processor $P_i$ and $SC_{i,j}$ is the state of channel $C_{i,j}$. In a system with non-FIFO channels, $SC_{i,j} = \{ \text{messages sent up to } LS_i \} \setminus \{ \text{messages received up to } LS_j \}$. Recording distributed snapshots of an execution is a fundamental problem in asynchronous distributed systems [3], and is used for observing various properties of interest [6].

The seminal algorithm by Chandy and Lamport [3] requires sending a special control message called the marker message on each of the logical channels in the system. In the typical case where there exists a fully connected overlay on the network graph, this amounts to a $O(n^2)$ message overhead. Many variants of the Chandy-Lamport algorithm have been proposed. However, in the traditional literature, the best known bound on the number of messages in a distributed algorithm in systems assuming either FIFO or non-FIFO channels is $O(n^2)$ because a marker is sent on each logical channel.

Present day supercomputing machines based on the MIMD architecture have hundreds of thousands of processors [11]. Examples of such machines include the BlueGene supercomputer. Such machines are distributed systems as they are often used for solving complex tasks and communicate by message passing. Checkpointing (or recording global snapshots) is therefore an important problem in such systems [1], [2], [4], [5], [7], [10]. A message overhead of $O(n^2)$ messages per snapshot becomes too expensive and is not scalable as the number of processors increases. Recent work has focused on reducing the snapshot complexity in such systems [5].

In this paper, we give a distributed snapshot algorithm with message complexity $O(n \log n)$ messages. Each processor sends $\log n$ messages. The sizes of the messages are geometrically distributed, and the sum of the message sizes sent by any processor is $O(n)$. The response time of the algorithm is $O(\log n)$. The role of each processor in the algorithm is fully symmetric. We compare this algorithm with the literature in Section III.

The Chandy-Lamport algorithm for a FIFO system, and its variant by Mattern for a non-FIFO system [9], use a marker per logical channel. The role of a marker is three-fold.

1) To inform processors that some processor has initiated the snapshot execution.
2) To distinguish white (prerecording) messages from red (postrecording) messages.
3) To mark the end of the white messages. In a system with non-FIFO channels, the computation messages are explicitly colored. To determine the number of white messages to be expected, Mattern’s variant of the Chandy-Lamport algorithm works as follows [9]. It piggybacks the number of white messages sent along the channel on the corresponding marker sent on that channel. This allows the receiver to know how many white messages to expect before termination. We name this algorithm as piggyback, in contrast to the deficiency counting and vector counter algorithms also introduced by Mattern [9].

II. SNAPSHOT ALGORITHM

We assume a hypercube overlay topology on the distributed system. Let $n = 2^d$. A hypercube overlay has a one-time cost, and can be easily implemented. For convenience, we assume a pre-established spanning tree, which can be set up at a one-time cost of $O(n \log n)$. We also assume that a single process runs at each processor as part of the distributed application.
Logically, a process can be in one of two states: white (prerecording) or red (postrecording). All processes are initially white. Application messages sent by a white process are colored white (prerecording messages). When some process records its local state, the algorithm is initiated. To inform other processes of this, a broadcast is done using RECORD control messages on a precomputed spanning tree. On receiving a RECORD message or a red computation message, a (white) process atomically records its local state (if it has not already done so) and turns red. Application messages sent by a red process are colored red (postrecording messages). The use of RECORD and red messages fulfills the first role of the marker. The coloring of messages fulfills the second role of the marker.

The third role of the marker is fulfilled by letting each process know the number of white messages sent to it. Rather than using a marker, this is achieved indirectly based on the following observation [9]: it is sufficient to know the total number of white messages sent to a process by all other processes. This number can be conveyed to a process using less than \( n \) messages, i.e., by not requiring a dedicated message from every other process. The proposed distributed algorithm can achieve this in \( n \log n \) messages, wherein each process sends \( \log n \) messages. Specifically, we use the hypercube overlay and perform \( n \) reductions concurrently in \( \log n \) iterations.

There are three steps in the algorithm which is shown in Figure 1.

1) Snapshot initiation: The snapshot initiator triggers a one-to-all broadcast of RECORD control messages. The RECORD messages can be sent along a pre-established spanning tree.

2) On receiving the RECORD message or a red colored message, the process records the local state and turns from white to red. It initializes the states of all the incoming channels to the empty set. (Henceforth, a red process sends red-labeled computation messages.) The algorithm then conveys the sum of the number of all white messages sent by all the processes to \( x \), to that \( x \), for every process \( x \). The symmetrical manner in which this is achieved is the main innovation in this paper.

Each process \( P_i \) maintains \( \text{white} \_\text{sent}_i[1..n] \) to count the number of white messages it sent to \( P_i \). \( \text{SENT}_i[1..n] \) is initialized to \( \text{white} \_\text{sent}_i[1..n] \). Using a hypercube overlay, the algorithm performs \( n \) all-to-one reductions concurrently in \( \log n \) iterations. Each concurrent reduction is an in-network aggregation of the number of messages sent to a particular destination \( P_i \). The in-network aggregation for \( P_i \) happens on a logical convergecast tree rooted at \( P_i \) and based on the order of the dimensions in the hypercube, from the MSB dimension to the LSB dimension. With respect to any destination \( P_i \), the partial sum of the count of white messages sent to \( P_i \) exists in a hypercube that keeps halving in size in each of the \( \log n \) iterations. In iteration \( \text{count} \), where \( \text{count} \) ranges from \( d-1 \) to 0, \( P_i \) communicates to \( P_{i \oplus 2^{d-1}} \), the entries \( \text{SENT}_i[j] \), for all \( j \) satisfying the following. Process \( j \) lies in the half-hypercube where \( j \)'s label differs from \( i \)'s label in the \( (\text{count} + 1) \)th LSB and the \( d - \text{count} - 1 \) MSBs match those of \( i \)'s label. At the end of \( \log n \) iterations, the sum of the number of white messages sent to \( P_i \) is accumulated in \( \text{SENT}_i[i] \), i.e., \( \sum_{x \in N} \text{white} \_\text{sent}_x[i] = \text{SENT}_i[i] \).

3) Recording channel states: When a white message is received from \( P_j \) by a red process \( P_i \), it is added to the state of channel \( C_{ji} \) and the count \( \text{white} \_\text{recd}_i \) is incremented. When step (2) is completed and \( \text{white} \_\text{recd}_i \) equals \( \text{SENT}_i[i] \), all white messages have been received, and the algorithm can terminate locally.

Optionally, if the snapshot needs to be assembled, a convergecast on a spanning tree can be performed after the termination of the local snapshot recording at each process. For checkpointing in large-scale systems, the checkpoints may be stored locally.

**A. Correctness**

The correctness of the local state recording is evident because we adapt Mattern’s algorithm [9]. We only need to show that the channel states correctly record the in-transit white messages. For all \( j \in N \), consider the \( n \) initial entries \( \text{white} \_\text{sent}_j[i] \) and the logical convergecast tree rooted at \( P_i \). The sum of these \( n \) initial entries represents the number of white messages sent to process \( P_i \). In iteration \( \text{count} \) (0 \( \leq \) \( \text{count} \) \( \leq \) \( d - 1 \) ) of the main loop of step (2), \( 2^{\text{count}} \) entries of this form get added concurrently at various processes along the convergecast tree rooted at \( P_i \). The total number of additions after all the rounds is

\[
\sum_{0}^{d-1} 2^{\text{count}} = 2^d - 1 = n - 1,
\]

yielding the desired sum of the \( n \) numbers. \( \sum_{j} \text{white} \_\text{sent}_j[i] \) is thus correctly computed. The channel recording terminates when \( \sum_{j} \text{white} \_\text{sent}_j[i] = \text{white} \_\text{recd}_i \).

**III. DISCUSSION**

This paper presented the first \( n \log n \) message distributed snapshot algorithm for a system with non-FIFO channels. The algorithm finds direct application in large scale distributed systems such as the MIMD supercomputers which have a fully connected topology of a large number of processors.

Table I compares the proposed algorithm, denoted as the hypercube algorithm, with other non-inhibitory algorithms for non-FIFO channels. We compare the deficiency counting,
int white_recvd;
int white_senti[1..n], SENTi[1..n];
state LS;
set of messages SCj,i for all j;

(1) When a white process \( P_i \) wants to initiate snapshot recording:
Broadcast a RECORD control message along a pre-established spanning tree and to \( P_i \) itself.

(2) When a white process \( P_i \) receives a RECORD control message or a red computation message:
   turn red;
   if a RECORD control message was received then
      propagate the RECORD control message along the spanning tree;
      record local state LS;
   white_recvd is number of (white) messages received until now;
   for \( j = 1 \) to \( n \) do
      \( SENT_i[j] \leftarrow white_sent_i[j] \);
   initialize \( SC_j,i \) (for all \( j \in N \)) to \( \emptyset \);
   for \( count = d - 1 \) down to \( 0 \) do
      send \( SENT_i[j] \) to \( P_i \oplus 2^{count} \), for all \( j \) such that
      \( j = d - count - 1 \) MSBs of \( i \cdot (count + 1) \)th LSB of \( i \cdot * * * \);
      receive \( 2^{count} \) entries of the form \( SENT_i[k] \) from \( P_i \oplus 2^{count} \);
      for all received entries of the form \( SENT_i[k] \) do
         \( SENT_i[k] = SENT_i[k] + SENT_i[k] \);
      if white_recvd = \( SENT_i[i] \) then
         local snapshot recording is complete.

(3) When a red process \( P_i \) receives a white message \( M \) along \( C_{j,i} \):
record \( M \) in \( SC_{j,i} \) as \( SC_{j,i} \leftarrow SC_{j,i} \cup \{M\} \);
white_recvd + 1;
if white_recvd = \( SENT_i[i] \) and Step (2) is completed then
local snapshot recording is complete.

Fig. 1. Snapshot recording algorithm at processor \( P_i \). \( \oplus \) is the XOR operator.

vector counter, and piggyback algorithms [9], and the two-dimensional grid-based, tree-based, and centralized algorithms by Garg et al. [5]. We also compare the following two algorithms: Simple_Ring and Simple_Tree.

Simple_Ring: The processes are arranged in a logical ring, with \( P_0 \) as the initiator process. \( P_0 \) circulates a token around the ring once. The receipt of the token triggers recording the local snapshot and turning red. The token also carries the accumulated count of the vector \( white_sent \), and is initialized to the vector \( white_sent_0 \). When \( P_i \) (\( i > 0 \)) receives the token, it adds its vector to the vector in the token. When \( P_0 \) receives the token back, \( white_sent[j] \) in the token contains the count of white messages sent to process \( P_j \). A second pass of the token around the ring distributes the values of \( white_sent \) to the processes.

Simple_Tree: The processes are arranged in a logical tree. A tree broadcast initiates the recording of local states and turning red. After the broadcast completes, a convergecast (initiated by the leaves) accumulates the vector \( white_sent_j \), for all \( j \), at the root. After the convergecast completes, a tree broadcast initiated by the root distributes the accumulated values of \( white_sent \) to the processes.

All the algorithms are compared against the following metrics: number of messages, total message space, local storage, whether the roles of the processes are symmetrical, response time (or latency), and parallel communication time. We define the roles of the processes to be symmetrical if the processes execute identical code. In a symmetrical algorithm, there is perfectly uniform distribution of workload, no bandwidth and processing bottlenecks, and greater elegance. Response time is defined as the net parallel time of the control messages (to record the local states and enable detection of in-transit messages) in the parallel algorithm, counting the processing time for a message as one unit. An alternate version of the response time metric is the parallel communication time. Here, the time for a message is \( t_s + t_w \cdot x \), where \( t_s \) is the local processing overhead per message (at the sender and the receiver), \( t_w \) is the transmission time per word, and \( x \) is the number of words in the message. The parallel communication time is the net parallel message time in the parallel algorithm.
The tree-based, grid-based, and the centralized algorithms [5] all have varying degrees of asymmetry among the processes. Specifically, the grid-based algorithm performs accumulation of the white sent vectors along the grid diagonal, and the diagonal elements then distribute the values to non-diagonal elements. The tree algorithms (Simple Tree and tree-based) have asymmetrical roles among leaf nodes, internal nodes, and the root node. Note that in vector counter and Simple_Ring, the initiator plays the additional role of changing the phases of the algorithm; hence we classify them as being asymmetric. The only distributed algorithms that have perfectly symmetric roles for the processes are the piggyback and hypercube algorithms. Observe from Table I that the proposed hypercube algorithm has the lowest number of messages from among algorithms in which the roles of all the processes are completely symmetrical. Note however, that the vector counter algorithm, Simple_Ring, and Simple_Tree use fewer messages than the hypercube algorithm.

Notwithstanding the asymmetry of roles in the grid-based and tree-based algorithms [5], the hypercube algorithm is also superior to the grid-based and tree-based algorithms in terms of the number of messages and in terms of response time. As the system scales up in terms of the number of processors, the number of messages becomes very important. It is more efficient to send few large messages than more small messages.

The response time of the hypercube algorithm is \(\log n\) because the messages in step (2) are immediately pipelined after the RECORD messages. Hence there is no latency for the initiation phase. Compared to the Simple_Ring algorithm, the hypercube algorithm has lower response time. Simple_Ring is asymmetric, as it requires a leader process to change the phases of the algorithm. Compared to the Simple_Tree algorithm, the hypercube algorithm has lower response time: \(\log n\), as against \(4 \log n\) for the sequential convergecast and broadcast that follow the initiation phase in Simple_Tree. Simple_Tree is asymmetric as it requires different roles to be played by leaf nodes, internal nodes, and the root node.

The response time and parallel communication time of the hypercube are lowest among all algorithms.

We make the following conjectures, based on the properties of the hypercube architecture.

**Conjecture 1:** Among distributed snapshot recording algorithms that are perfectly symmetrical, i.e., identical code is executed by the processes, the hypercube algorithm in Figure 1 is optimal in terms of the number of messages used and in terms of response time and parallel communication time.

**Conjecture 2:** Among distributed snapshot recording algorithms, the hypercube algorithm in Figure 1 is optimal in terms of response time and parallel communication time. Note, however, that the hypercube overlay may contain multiple edges compared to the tree and ring overlays; this may impact the response time and parallel communication time.

### References

11. [Top 500 Supercomputer Sites](http://www.top500.org)